AN ESTELLE-C COMPILER FOR AUTOMATIC PROTOCOL IMPLEMENTATION

by

ROBIN ISAAC MAN-HANG CHAN

B.Sc., The University of British Columbia, 1980

M.Sc., The University of British Columbia, 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

THE UNIVERSITY OF BRITISH COLUMBIA

October 1987

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date October 13, 1987

Abstract

Over the past few years, much experience has been gained in semi-automatic protocol implementation using an existing Estelle-C compiler developed at the University of British Columbia. However, with the continual evolution of the Estelle language, that compiler is now obsolete. The present study found substantial syntactic and semantic differences between the Estelle language as implemented by the existing compiler and that specified in the latest ISO document to warrant the construction of a new Estelle-C compiler. The result is a new compiler which translates Estelle as defined in the second version of the ISO Draft Proposal 9074 into the programming language C. The new Estelle-C compiler addresses issues such as dynamic reconfiguration of modules and maintenance of priority relationships among nested modules. A run-time environment capable of supporting the new Estelle features is also presented. The implementation strategy used in the new Estelle-C compiler is illustrated by using the alternating bit protocol found in the ISO Draft Proposal 9074 document.

Contents

List of Figures

Acknowledgement

The author wishes to express his thanks to his supervisor, Dr. Son Vuong, for his guidance and to Dr. Samuel Chanson for his careful reading of the thesis.

He also wishes to thank his wife, Silvian, for her many helpful suggestions in the preparation of this manuscript and for her patience and support throughout the course of this work.

Chapter 1

Introduction

Estelle is a formal description technique (FDT) developed to be used by ISO standards committees for the specification of communication protocols and services destined to become international standards. The use of formal methods for protocol specification reduces the risks of erroneous or incompatible implementations of these protocols. In addition, the availability of precise and unambiguous descriptions of protocols allows automatic tools to be built for generating protocol implementations directly from the formal specifications.

In response to the challenge of realizing automatic implementation of protocols from Estelle specifications, the first Estelle compiler was developed at the University of Montreal [Gerb83]. This compiler accepts Estelle specifications and generates implementation codes in Pascal. Currently, several Estelle compilers, interpreters and simulators have already been developed [Ansa87,Cour86,Garg87].

1.1 Motivations for a New Estelle Compiler

At the University of British Columbia (UBC), an Estelle-C compiler was developed by Daniel Ford in 1984 [Ford85]. The compiler accepts Estelle as defined by the 1984 Estelle working document [Este84] and generates target codes in the programming language C. The

original compiler was found to be erroneous and was subsequently improved by Alan Lau in 1986 [Lau86]. The improved compiler was successfully used by Lau in a comparative study on semi-automatic versus manual implementation [Vuon87,Vuon88] of the ISO class 2 transport protocol [ISO82a,ISO82b].

However, the Estelle language has undergone two major changes since 1984 and is currently in the second draft proposal stage [Este85,Este86]. The UBC Estelle-C compiler is now obsolete due to the substantial differences between the current Estelle specification and the Estelle language as implemented by Ford. Therefore, it is necessary to build a new UBC Estelle-C compiler that will conform to the new standards [Este86] and, thus, allow further works in automatic protocol implementations.

1.2 Thesis Outline

This thesis describes the implementation of a new Estelle-C compiler. Chapter 2 presents the changes made to the Estelle language since 1984 and the justification for the reimplementation of a new compiler instead of the modification of the old compiler to conform to the new standards. Chapter 3 describes the translation scheme used in the new compiler and compares it with the scheme used in the old compiler. The implementation strategy used in the new Estelle-C compiler is illustrated by using the alternating bit protocol. Chapter 4 discusses the run-time environment used in the new compiler. Chapter 5 concludes the thesis with some insights gained from this project.

Chapter 2

Estelle Evolution

Estelle is a hybrid formal protocol description technique which combines an underlying extended finite state machine model with the use of a programming language notation. Syntactically, Estelle is based on the programming language Pascal with additional features borrowed from Ada and Modula-2. An Estelle specification describes a complex protocol specification as a hierarchical structure of increasingly refined communicating finite state machines called **modules**. The syntax provides constructs necessary to specify state transitions within the modules as well as the means to interconnect the various specified modules. Semantically, these modules are allowed to be executed in parallel.

The modules communicate with each other through abstract interfaces called **interaction points**. A bidirectional communication path between two interacting modules, called a **channel**, is formed when two interaction points, one from each module, are connected together. After a channel is established between two modules, the modules can interact by transmitting units of information, called **interactions**, through the channel.

The dynamics of a channel is modeled abstractly as a pair of first-in-first-out queues located at the two linked interaction points. Each interaction signaled between two modules is routed

from the interaction point at the sending module to the queue in the interaction point at the receiving module.

Each channel is associated with a channel type. For each channel type, a set of parameterized interaction primitives can be specified for generating interaction instances which are to be transmitted through the channel. Because of the bidirectional nature of the channels, two interaction role identifiers must be specified for each channel in order to distinguish the two directions. Each of the allowable interaction primitives may be associated with either one or both of the defined roles. The use of the role identifier allows each interaction primitive to be specified as either unidirectional or bidirectional. The two corresponding interaction points for each channel must have opposite roles so that they can be used to send and to receive interactions of the opposite type.

The abstraction provided by the Estelle channel can be used naturally for modeling the set of service primitives allowable at the boundaries between two adjacent protocol layers. When a protocol specification is refined into submodules, the same abstraction can also be used to specify interactions between any two submodules.

Excellent descriptions of the Estelle features and facilities can be found in Linn [Linn86] and Courtiat et al. [Cour86]. This chapter only describes the changes to the Estelle language from the 1984 working document [Este84] to its present 1986 draft proposal form [Este86] and concludes with a justification for building a new Estelle-C compiler from scratch instead of modifying the current UBC Estelle-C compiler.

2.1 Module Hierarchy

Some of the major differences between the Estelle language as defined in the original working document [Este84] and that defined in the resulting draft proposal documents [Este85,Este86] are the changes made to the hierarchical structuring of the modules. Since a module is the basic unit of protocol specification in Estelle, the changes have profound effects on the run-time environment that the new Estelle-C compiler must support.

2.1.1 Static Organization

A protocol specification is originally defined as a hierarchy of modules of two different types [Este84]. At the bottom of the module hierarchy are **processes**. A process defines an atomic unit of protocol specification as an extended finite state machine which cannot be further subdivided. The behavior of a process is specified as a list of possible transitions. All processes are specified to be executed in parallel. The modules at the higher layers in the module hierarchy are called **refinements**. Refinements may be further divided into submodules, each of which may be either a process or another refinement. Refinements, however, may not contain any transition specification. The sole purpose of the refinement modules is to impose a structure on the set of defined processes during system initialization time; these modules are inactive during protocol execution.

The implication of this modular organization is that a protocol specified in this manner has a static structure. Since only the bottom layer of the hierarchy contains active modules, the structure cannot be changed during run-time. Another consequence of this organization is that the structure is linear. The set of active modules can be linked together into a linear list. Simple round-robin scheduling over this list will suffice during run-time to simulate parallelism

[Ford85].

2.1.2 Dynamic Organization

In the first draft proposal for Estelle [Este85], transition execution is allowed in the higher level modules. In addition, provisions are made to allow modules to share common variables. In order to ensure mutual exclusion between the shared variables, two restrictions are imposed on the structuring principles of an Estelle protocol specification.

First, two types of modules with different execution semantics are defined. An **activity** is a module which is considered to be atomic and cannot be substructured. A **process** is a module which may be substructured into either child processes or child activities. Processes at the same level may be run in parallel, but activities may be run only in an interleaved fashion. Second, a parent/child priority relation is imposed on the module hierarchy. If a transition of a parent module is enabled, no child may begin a transition.

The first restriction will ensure mutual exclusion among modules at the same level in the module hierarchy if they are specified as activities. The second restriction will ensure mutual exclusion among modules at different levels in the module hierarchy.

In the second draft proposal for Estelle [Este86], the major change to the Estelle language specification is to forbid the use of shared variables among modules at the same level in the module hierarchy. The inclusion of this restriction eliminates the concern of mutual exclusion among modules at the same level. Consequently, there is no further need to distinguish between processes and activities. However, the concepts of processes and activities are retained to distinguish the two possible forms of module execution semantics. The process abstraction represents a synchronous parallel execution while the activity abstraction represents a nondeterministic sequential execution. Because of these new semantics, activity modules may now

be further substructured into other activities.

The new module synchronization semantics defined in the two Estelle draft proposals [Este85,Este86] imply a more complicated run-time environment than is necessary for Estelle as defined in the working document [Este84]. The new run-time scheduler must differentiate between modules at different levels in order to enforce the parent/child priority relationships. The scheduler used in the new Estelle-C compiler is discussed in Section 4.1.

2.2 Module Configuration

Module configuration is the process of instantiating and interconnecting the modules defined in an Estelle specification. Module configuration can only be performed by a parent module on its immediate child modules. In the original version of Estelle [Este84], module configuration can only be performed at system initialization time. Since all modules above the bottom level are inactive, the number and the types of modules will remain unchanged for the life-time of the specified system. In the later versions [Este85,Este86] where active modules are present in the higher levels, module configuration may be carried out any time. The potential result of this enhancement is a protocol specification with a dynamically varying module organization.

2.2.1 Dynamic Module Instantiation

The process of module instantiation includes the declaration of a module variable, the initialization of a module instance, and the binding of the initialized module instance to the module variable.

In the working document version of Estelle [Este84], module instantiation is an implied operation associated with the declaration of a module variable. In the later versions [Este85,Este86], modules are explicitly created and initialized by using the INIT statement. Module termination

is possible using the RELEASE statement. These two special Estelle statements may be used either at system initialization or within transition execution. The use of explicit statements to perform these two operations in an Estelle protocol specification provides the power to change the number and the type of modules within the specification dynamically. The need to support dynamic module creation and termination results in a run-time environment which must maintain the complete hierarchical module organization at all times. In contrast, under the old Estelle environment, this information may be discarded after the system has been initialized [Ford85].

2.2.2 Dynamic Module Interconnection

With the addition of dynamic module instantiation, it becomes necessary to provide explicit module interconnection statements. These operations are provided in Estelle by the special statements: CONNECT, DISCONNECT, ATTACH and DETACH. The CONNECT and DISCONNECT statements are used to alter the interconnections between modules at the same level; and the ATTACH and DETACH statements are used to alter the interconnections between modules at adjacent levels.

With the added capabilities for dynamic reconfiguration, the immediate parent of a set of modules can be specified to act as a supervisory manager. However, these provisions for dynamic reconfiguration of the various entities in a protocol specification result in an Estelle run-time environment that is more complex than one which simply maintains a complete module hierarchy.

2.3 Justification for a New Estelle-C Compiler

With the basic ideas underlying the semi-automatic approach to protocol implementation well understood and demonstrated by the many existing Estelle compilers [Boch87a], the major motivation for developing a new compiler for Estelle is to upgrade the UBC Estelle-C compiler to support the latest Estelle language specification [Este86]. However, it is apparent that there are many issues which must be addressed in changing from a static run-time environment to a dynamic one. This section describes the justifications for a complete rewrite of the Estelle-C compiler instead of modifying the existing compiler.

2.3.1 Syntactic Issues

One of most important reasons for rewriting the new Estelle-C compiler is that the syntax of Estelle has changed so significantly that building a new parser is desirable. The old compiler was written with the aid of the UNIX utilities *lex* and *yacc*. A significant omission in the old compiler was the lack of syntax error recovery mechanisms. The building of a new parser offers an opportunity to incorporate this important compiler feature into the new compiler. With added error recovery as part of the design goal, the Estelle grammar is rewritten into a LL(1) form. Then, the parser for the new Estelle-C compiler is hand-coded in C using recursive descent techniques. Syntax error recovery is carried out using the panic mode technique with a dynamic stop symbol set. An unrelated advantage gained from rewriting the compiler without using *lex* and *yacc* is the possibility for further development of the new compiler in non-UNIX environments.

2.3.2 Semantics Issues

Another important reason for rewriting the new compiler relates to semantics issues. Being a formal description technique for protocol specification, the Estelle language must have precise meaning; otherwise, protocols specified in Estelle will not have a sound foundation and may be opened to different interpretations. In order to satisfy this requirement, the second draft proposal for Estelle is published with a new section on formal semantics [Este86]. When the implementation of the old Estelle-C compiler is compared with the new formal semantics, several features are found to be incompatible.

The major area of incompatibility has to do with the scoping rules for variables. The old Estelle compiler did not pay particular attention to the scoping of many variables. Variables local to individual transitions were not supported. Module parameters were not made available to the module transitions. Procedures and functions defined within a module were not allowed access to global variables declared within the same module. Furthermore, the old Estelle-C compiler is still erroneous despite the improvements made by Alan Lau [Lau86]. In particular, the old compiler lacks some important Estelle features, such as the data types SET and multi-dimensional ARRAY. Solutions to these and other problems are all part of the redesign of the new Estelle-C compiler.

Other semantics issues deal with error checking. The old Estelle-C compiler has no provision for checking semantic errors besides Estelle specific semantic errors. Since the code generated by the Estelle compiler would have to be further compiled by the C compiler, the rationale was that the C compiler can be used for most of the semantic checks. However, by placing most of semantic checking burden on the C compiler, the error messages from the C compiler become cryptic. Users of the compiler without firm understanding of the organization of the

generated C-codes frequently have trouble understanding errors detected during the subsequent C compilation. In order to build a more "user-friendly" Estelle compiler, more emphasis is placed on semantic checking in the new Estelle-C compiler.

2.3.3 User Issues

Building extensive error checking facilities into the new Estelle-C compiler is not adequate to make the new compiler user-friendly. The old Estelle-C compiler produces a C program which is not readily compilable by the C compiler without extensive user modifications. Also, the old Estelle run-time support routines contain specification dependent details which must be modified for each Estelle specification. In order to generate an executable implementation from an Estelle specification, the user must recompile the run-time support routines using the C compiler along with the C-code generated by the Estelle-C compiler. In the new Estelle-C compiler, it is no longer necessary for the user to modify any of the generated codes. Besides improving and extending the run-time routines to support the new Estelle features, all specification dependent details have been extracted from these run-time support routines. The specification independent routines have been precompiled into a single object library. After the generated C-codes have been compiled by the C compiler, they can be easily linked to this object library to form the final executable program.

In summary, the new Estelle-C compiler is written to incorporate the features in the latest version of the Estelle language and to improve the user-friendliness of the compiler. The user-friendliness aspect of the improvement includes the use of effective error diagnostics for the user and the freeing of the user from the need to know the details in the underlying run-time environment. The goal is to increase the degree of automation than that achieved in the previous semi-automatic implementations of protocols from formal specifications. With regards to the

shortcomings in the old Estelle-C compiler, a complete rewrite of the compiler is desirable as well as necessary.

Chapter 3

Estelle to C Translation

The translation of Estelle to C in the new Estelle-C compiler follows the implementation strategy used by Gerber [Gerb83], a strategy which was also adopted by Ford [Ford85]. Each Estelle module is translated into two separate C routines. One routine is used for transition execution while the other is used for module initialization. The **transition routine** implements the finite state machine specified for the module. The **initialization routine** sets up the internal states of the module before it is ready for the subsequent transition execution.

Besides generating executable codes to implement the modules, the Estelle-C compiler also generates two sets of global declaration structures. One structure, called the **signal parameter block** (*FDTSVAR*), is used for storing the parameter information carried in the interactions passed between modules. The other structure, called the **module variable block** (*FDTLVAR*), is used by each module for storing local variables. After these four sets of generated C codes are compiled and then linked together with a set of pre-compiled run-time support routines, an executable protocol implementation results.

The Estelle run-time environment is constructed from three major control blocks representing the three major abstractions defined in the Estelle language. The interactions which are

signaled between modules are represented by **signal control blocks** (*FDTSCBs*). The channels through which the interactions are transmitted are represented by **channel control blocks** (*FDTCCBs*). Finally, the modules which send and receive the interactions are represented by **process control blocks** (*FDTPCBs*).

The following sections present the Alternating Bit Protocol as an example in Estelle to C translation. The complete Estelle specification for the Alternating Bit Protocol and the C program generated from it are included in Appendices A and B, respectively. The sections begin with the explanation on the generation of the two global declaration blocks followed by a description of the three control blocks and the ways in which they are combined with the generated declaration codes and with each other to form the run-time environment. Subsequently, the translation of an Estelle module into an initialization and a transition routine is discussed.

3.1 Global Declaration Blocks

Two global declaration blocks are generated to represent all of the specification dependent variables needed during run-time. To facilitate the ease of understanding the generated code, the identifiers used in the generated C code retain their Estelle names. The elaborate variant structures described below is necessary to protect the Estelle names from identifier conflicts when used within a C program.

3.1.1 Signal Parameter Block

The **signal parameter block** (*FDTSVAR*) is a three-level variant record structure representing the combination of all specified parameters in all interaction primitives for all channel types within an Estelle specification. The *FDTSVAR* generated for the alternating bit protocol is shown in Figure 3.1.

```
typedef union {

  /* CHANNEL U_access_point primitives and their identity numbers */

  union {
    struct {
      int udata ;           /* 1.  SEND_REQ (udata : udata_type); */
    } SEND_REQ ;
    int RECV_REQ ;          /* 2.  RECV_REQ; */
    struct {
      int udata ;           /* 3.  RECV_RSP (udata : udata_type); */
    } RECV_RSP ;
  } U_access_point ;

  /* CHANNEL N_access_point primitives and their identifiers */

  union {
    struct {
      ndata_type ndata ;    /* 1.  DATA_REQ (ndata : ndata_type); */
    } DATA_REQ ;
    struct {
      ndata_type ndata ;    /* 2.  DATA_RSP (ndata : ndata_type); */
    } DATA_RSP ;
  } N_access_point ;

  /* CHANNEL S_access_point primitives and their identifiers */

  union {
    int TIMER_REQ ;         /* 1.  TIMER_REQ; */
    int TIMER_RSP ;         /* 2.  TIMER_RSP; */
  } S_access_point ;

} FDTSVAR;
```

Figure 3.1: Signal Parameter Block Structure

The first level variant structures identify the channel types while the second level variant structures identify the interaction primitives within the channels. For easy identification, the interaction primitives defined for each channel type are numbered. The identity numbers assigned by the Estelle-C compiler for the interaction primitives are shown in Figure 3.1. The innermost structures represent an enumeration of the parameters for the interaction primitives. These innermost structures are absent for interaction primitives without parameters.

3.1.2 Module Variable Block

The **module variable block** (*FDTLVAR*) is a two-level variant record structure that contains the complete global state variables for all module bodies. The module variable block generated for the alternating bit protocol is presented in Figure 3.2.

The outer level variant structures identify the module bodies in the Estelle specification. The inner structures store all major and minor state variables declared for the module bodies. The variables are collected from the various Estelle declaration sections. The first set of variables is extracted from the module parameter declaration and the exported variable declaration sections in the associated module header declarations. The rest of the variables are derived from the STATE, STATESET, VAR and MODVAR declaration sections in the module bodies. The origins of the various variables in the *FDTLVAR* are shown in Figure 3.2 as comments. Module bodies without any variable declarations are not represented in the *FDTLVAR* structure.

3.2 Run-time Control Blocks

Three control blocks are used to represent all of the specification independent bookkeeping information during run-time. The following sections describe the three control blocks and conclude with a discussion on the improvements made with respect to the old Estelle-C run-

```
typedef union {
          struct {                          /* MODULE user_body */
            int cep_id ;                    /* Parameter */
            int flag ;                      /* Variable  */
            int data ;                      /* Variable  */
          } user_body;
          struct {                          /* MODULE network_body */
            int count ;                     /* Variable  */
          } network_body;
          struct {                          /* MODULE timer_body */
            int time ;                      /* Parameter */
            int FDT3 ;                      /* Temporary */
            int stop ;                      /* Variable  */
            int stop_bis ;                  /* Variable  */
          } timer_body;
          struct {                          /* MODULE datax_body */
            int cep_id ;                    /* Parameter */
            set_type EITHER ;               /* Stateset  */
            int STATE ;                     /* State     */
            ndata_type buf ;                /* Variable  */
            msg_type recv_msg ;             /* Variable  */
            msg_type send_msg ;             /* Variable  */
            buf_type recv_buf ;             /* Variable  */
            buf_type send_buf ;             /* Variable  */
            int recv_seq ;                  /* Variable  */
            int send_seq ;                  /* Variable  */
          } datax_body;
          struct {                          /* MODULE abit_body */
            int cep_id ;                    /* Parameter */
            FDTPCB *timer_module ;          /* Modvar    */
            FDTPCB *datax_module ;          /* Modvar    */
          } abit_body;
          struct {                          /* SPECIFICATION abit_spec */
            FDTPCB *abit_module [ 2 ];      /* Modvar    */
            FDTPCB *user_module [ 2 ];      /* Modvar    */
            FDTPCB *network_module ;        /* Modvar    */
          } SPECIFICATION ;
        } FDTLVAR;
```

Figure 3.2: Module Variable Block Structure

time control blocks.

3.2.1 Signal Control Block

Each unit of interaction (or signal) sent through a channel is represented by a **signal control block** (*FDTSCB*) in conjunction with a **signal parameter block** (*FDTSVAR*). As described in Section 3.1.1, the *FDTSVAR* is a specification dependent structure generated from the channel type declaration sections in an Estelle specification. In contrast, the *FDTSCB* is a specification independent run-time control block (Figure 3.3). The two blocks are linked

```
struct FDTSCB_struct
  {
    struct FDTSCB_struct   *next;   /* Next signal  */
    int                    cid;     /* Channel id   */
    int                    sid;     /* Primitive id */
    int                    *svar;   /* Parameters   */
  };
typedef struct FDTSCB_struct FDTSCB;
```

Figure 3.3: Signal Control Block Structure

together by the pointer `svar` located in the *FDTSCB*. Since the *FDTSVAR* contains only interaction parameter fields (see Figure 3.1), additional information must be provided within the *FDTSCB* for the identification of interaction primitives. With the *FDTSVAR* implemented as a three-level variant record structure, two identifiers are necessary to uniquely identify the interaction being conveyed. The first identifier, encoded in the field `cid`, indicates the index number of the target interaction point within the target module. Since each interaction point is associated with only one channel type in Estelle, this number also identifies the channel type. The second identifier, stored in the field `sid`, is used to specify the interaction primitive within

the channel identified by cid. The next field is used for queue manipulation at the target interaction point.

3.2.2 Channel Control Block

Each interaction point associated with a channel is represented by a **channel control block** (*FDTCCB*). The number of *FDTCCBs* necessary to completely specify an Estelle channel at run-time depends on the number of CONNECT and ATTACH statements used to build the channel. Every CONNECT or ATTACH operation involves two *FDTCCBs*.

The bookkeeping for interaction point binding is maintained by three pairs of variables within each *FDTCCB* (See Figure 3.4). The pair (targeta, channela) is used to specify the

```
struct FDTCCB_struct
  {
    struct FDTSCB_struct  *head, *tail;
    struct FDTPCB_struct  *targeta, *targetc, *targete;
    int                   channela, channelc, channele;
    int                   queue_kind;
  };
typedef struct FDTCCB_struct FDTCCB;
```

Figure 3.4: Channel Control Block Structure

target interaction point of an ATTACH operation when the target interaction point is located at a child module. The field targeta indicates the target module while the field channela specifies the index number of the target channel within the target module. Similarly, the pair (targetc, channelc) is used to represent the target interaction point of a CONNECT operation or the target interaction point of an ATTACH operation when the target interaction point is located at a parent module. Because connected interaction points may be further attached to

other interaction points, in order to be efficient in determining the real target interaction point when interaction are sent between module, the pair (`targete`, `channele`) is used to specify the effective target interaction point directly. This tremendous amount of bookkeeping is necessary to keep track of the channel binding between modules so that channels may be DISCONNECTed and DETACHed afterwards. In contrast, under the old run-time environment for Estelle, where channels are prohibited from unbinding, only the effective target interaction point needs to be maintained in each *FDTCCB* [Ford85].

The other fields with the *FDTCCB* depicted in Figure 3.4 are used to implement the interaction queue. When an interaction, represented by a *FDTSCB* is sent through one *FDTCCB*, it will be queued at the opposite *FDTCCB* indicated by the (`targete`, `channele`) pair. The `head` and `tail` fields are pointers to the first and last *FDTSCBs* in this queue, respectively.

In Estelle, interaction points may be specified with either COMMON or INDIVIDUAL queueing discipline. The `queue_kind` field is used to indicate this queueing discipline for the interaction point.

3.2.3 Process Control Block

Each module instance at run-time is represented by a **process control block** (*FDTPCB*) in conjunction with a **module variable block** (*FDTLVAR*). While the *FDTLVAR*, as described in Section 3.1.2, is a specification dependent structure generated from the various variable declaration sections in an Estelle specification, the *FDTPCB* is a specification independent run-time control block (Figure 3.5). This control block is used to store bookkeeping information for each module instance for the duration of its existence at run-time.

The fields `parent`, `sib` and `ref` are pointers used to maintain the static module hierarchical structure at run-time. They point to the parent module, the next sibling at the same hierarchical

```
struct FDTPCB_struct
  {
    struct FDTPCB_struct   *parent;     /* Parent FDTPCB          */
    struct FDTPCB_struct   *sib;        /* Next sibling FDTPCB    */
    struct FDTPCB_struct   *ref;        /* First child FDTPCB     */
    struct FDTCCB_struct   *chan;       /* FDTCCB array           */
    int                    ipnum;       /* Size of FDTCCB array   */
    int                    ipnext;      /* Next ip to search      */
    int                    prio;        /* Hierarchical level     */
    int                    sigcnt;      /* Pending signal count   */
    int                    delay;       /* Delay clause id        */
    int                    t0, t1;      /* Delay time limits      */
    int                    spont;       /* Spontaneous present?   */
    int                    export;      /* Export variables?      */
    int                    (*trans)();  /* Transition routine     */
    int                    *lvar;       /* Module variable block */
  };
typedef struct FDTPCB_struct   FDTPCB;
```

Figure 3.5: Process Control Block Structure

level and the head of the list of child modules at the next hierarchical level, respectively.

The address of the transition routine which implements the extended finite state machine specified for the module is stored at the field `trans`. The field `lvar` is a pointer to the *FDTLVAR* block.

Since the number of interaction points in a module can be statically determined, all channel control blocks *FDTCCBs* needed for a module are placed into one common array. The field `chan` is used to point to this *FDTCCB* array while the field `ipnum` is used to store the size of this array. Each interaction point is assigned an index number into this array starting at the index value '1'. The *FDTCCB* with the index value of '0' is an extra channel control block reserved for the COMMON queue. Interactions destined for a target interaction point specified with COMMON queueing discipline are queued in this COMMON *FDTCCB*. Interactions destined for a target interaction point specified with INDIVIDUAL queueing discipline are queued in the specified target *FDTCCB*. The field `sigcnt` indicates the sum of all pending interactions in the queues.

The three fields `delay`, `t0` and `t1` are used to implement DELAYed transitions (See Section 4.2.3 for detail). The remaining fields identify the hierarchical level of the module (`prio`), the next interaction queue to examine for the pending interactions (`ipnext`), and whether or not the module has spontaneous transitions (`spont`) and export variables (`export`).

3.2.4 Improvement over past Estelle Compilers

The run-time data structures used in the new Estelle-C compiler represent some of the major improvements made to the new compiler as compared to those used in the old compilers [Gerb83,Ford85].

In the past, the signal parameter block is placed within the signal control block; and the

module variable block is placed within the process control block. Since, these two combined control blocks contain specification dependent information, they are different for different Estelle specifications. Furthermore, since the run-time routines must have access to the control blocks, it is necessary for the old run-time routines to be recompiled for different specifications. With the arrangement used in the new Estelle-C compiler, the specification dependent details are isolated into the *FDTSVAR* and *FDTLVAR* structures while the control blocks remain the same for all specifications. Consequently, the new run-time support routines are specification independent and they no longer require recompilation.

A second improvement is made in the structuring of the *FDTCCBs*. In the old Estelle compiler, the *FDTCCBs* are linked together into a linear list. Therefore, every channel access must involve a linear search along this list for the required *FDTCCB*. The new Estelle-C compiler takes advantage of the fact that the number of channel is fixed for each module and assigns a unique index number to each channel. Channel access in the new Estelle-C compiler is thus performed by array indexing rather than by linear searching.

3.3 Run-time Support Routines

The control blocks described so far are constructed into a run-time data structure that reflects the module organization defined in the Estelle specification. This structure is built using a set of pre-written support routines. The calls to these routines are made by codes incorporated within the two sets of generated module routines. The support routines can be divided into three groups, each of which manipulates one type of control blocks. The following sections describe these three groups of support routines.

3.3.1 Process Control Block Routines

There are two process control block routines used to create and destroy process control blocks. The codes for these routines are depicted in Appendix C. The routine `FDTPCBinit()` creates and initializes an *FDTPCB*. This routine is used to instantiate a new Estelle module and it implements all of the specification independent operations required for the INIT operation defined in Estelle. The first part of the initialization process results in the linking of the newly created *FDTPCB* to the *FDTPCB* of its parent module and to the *FDTPCBs* of its other sibling modules. Afterwards, the appropriate number of *FDTCCBs* are created for the module being instantiated. Finally, other bookkeeping variables are initialized. The specification dependent operations are individually implemented in the initialization routines generated for each defined Estelle module (See Section 3.4.1 for detail).

The routine `FDTPCBterm()` destroys a specified module and all its child modules recursively. This routine implements the RELEASE operation defined in Estelle.

3.3.2 Channel Control Block Routines

The channel control blocks are manipulated by a set of seven run-time routines shown in Appendix D. These routines can be divided into two functional groups.

One group consists of the two routines, `FDTCCBinit()` and `FDTCCBterm()`, are used to allocate and to release the appropriate number of channel control blocks for a module. These two routines are in turn used by the routines, `FDTPCBinit()` and `FDTPCBterm()`, respectively, when modules are created and destroyed.

The other group is made up of five routines used to bind and unbind pairs of communicating channel control blocks. The routines `FDTCCBconnect()` and `FDTCCBdisconn()` are used

to bind and unbind *FDTCCBs* of modules at the same level in the module hierarchy. The routines `FDTCCBattach()`, `FDTCCBdetach1()` and `FDTCCBdetach2()` are used to bind and unbind *FDTCCBs* of modules at adjacent levels in the hierarchy. Essentially, they implement the Estelle operations CONNECT, DISCONNECT, ATTACH, and DETACH respectively.

3.3.3 Signal Control Block Routines

Interactions queued at a *FDTCCB* are manipulated by four pre-written library routines shown in Appendix E. In this version of the Estelle-C run-time environment, the interaction queues are implemented as singly-linked circular queues.

The routine `FDTSCBsignal()` is used to dispatch an interaction through a specified channel. A call to this routine is generated as the last step in the translation of an OUTPUT statement (See also Section 3.5.2). The newly dispatched interaction is placed in either the COMMON channel or the specified target channel of the target module depending on the queueing discipline of the target interaction point (See also Section 3.2.3).

The routine `FDTSCBspont()` is used when it is necessary to generate a spontaneous signal. A call to this routine is issued, when appropriate, after a transition is completed.

The routine `FDTSCBdispose()` is used after a transition has been completed in order to disposed of a received input interaction or a spontaneous interaction (See also Section 3.4.2 for details).

Finally, the function `FDTSCBpending()` is used by the global scheduler to search for a pending signal destined for a particular process.

3.4 Module Translation

The module abstraction in an Estelle specification represents the basic unit of protocol specification. A **module type** is declared as an abstract data type. The external visibility of the module is defined in a **module header** while the internal behavior is specified in a **module body**. The Estelle language definition allows several different module bodies to be specified for each module header.

As described in Section 3.1.2, the module head and the global declarations within the module body are used to generate declaration structures within the module variable block (*FDTLVAR*). The two C routines that are generated from each module are translated from the initialization parts and the transition parts within the module body. The convention used in the Estelle-C compiler is to name these routines after their corresponding module body. In order to distinguish the initialization routine from the transition routine, the prefix FDT is added to the name of the initialization routine. There is one exception to this naming convention. The two routines translated from the outermost SPECIFICATION module are always named `FDTSPECIFICATION()` and `SPECIFICATION()`, respectively.

The following sections describe the general structure of the two generated implementation routines. The discussion on the translation of the transitions themselves is deferred until Section 3.5.

3.4.1 Initialization Routine

The **initialization routine** for a module is generated from the initialization parts within a module body. This routine is used to set up the initial states of the extended finite state machine representing the module. The initialization routine for a module is executed whenever

an INIT statement referencing the module is executed by its parent module.

The general structure for an initialization routine is depicted in Figure 3.6. The first part of the routine allocates a control block for the module. Initialization begins by calling the run-time support routine `FDTPCBinit()` to create a *FDTPCB*. Afterwards, an *FDTLVAR* is created and linked to the *FDTPCB*. The routine `FDTPCBinit()` implements the specification independent aspects for the INIT statement (See Section 3.3.1 for detail). The rest of the initialization routine represents the specification dependent portion of the initialization process.

After a *FDTPCB* and a *FDTLVAR* has been allocated for the module, the initialization routine begins with module parameter initialization. All the module parameters declared in the module header section for the module are passed to the initialization routine. These parameters are copied into the *FDTLVAR* by assignment statements.

The next section in the routine contains the code for the initialization transitions. These transitions are usually responsible for calling other initialization routines which, in turns, instantiate the underlying submodules and then interconnect these submodules to the module being initialized. Other activities performed by the initialization transitions includes initializing the various global variables and setting the module into the proper state before the subsequent transition execution.

If the module contain spontaneous transitions, a spontaneous signal is generated in the next section. The last step in the initialization routine returns the address of the created *FDTPCB* to the parent module which calls this initialization routine. This pointer is stored within the *FDTLVAR* of the parent module for subsequent references to this particular child module.

```
struct FDTPCB *FDTbody (parent, arg1, arg2, ...)
FDTPCB *parent;
...   /* type declarations for arg1, arg2, ... */
  {
    FDTPCB *pcb;
    FDTLVAR *lvar;

    /* Creates control blocks */
    pcb = FDTPCBinit (parent, ipnum, SPONTbody, XPORTbody, TRANSbody);
    pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));


    /* Copying arguments to module variable block */
    lvar->body.arg1 = arg1;
    lvar->body.arg2 = arg2;
    ...


    /* Initialization transitions (See Section 3.5) */
    ...


    /* Generates a spontaneous interaction if possible */
    trans_end :
    if (pcb->spont)
        FDTSCBspont (pcb);
    return (pcb);
  }
```

Figure 3.6: Initialization Routine

3.4.2 Transition Routine

The **transition routine** for a module is generated from the transition parts within a module body. This part of the Estelle specification is used to define the state transitions that constitute the extended finite state machine representing the module. The routine itself is called whenever the run-time scheduler selects the corresponding module for execution.

The general structure of a transition routine is depicted in Figure 3.7. Two parameters are passed to each transition routine when the module is executed by the scheduler. The parameter `process` supplies the routine with the correct *FDTPCB* for the module while the parameter `signal` indicates the interaction selected by the scheduler to be processed by the module. Within the routine, the local variables `lvar` and `svar` are used to facilitate access to the *FDTLVAR* and *FDTSVAR* control blocks, respectively.

The codes in the first part of the routine implements the module transitions. The details for these codes are described in Section 3.5. The final section in the routine contains the exit sequence for the module. The details for these codes are explained in Section 4.2.

3.5 Transition Translation

In Estelle, the transitions for an extended finite state machine may be described in either an initialization part or a transition part within the module bodies. There may be zero or more of these transition description sections within a module body. A module without any initialization part will be initialized with the creation of its *FDTPCB* and *FDTLVAR* blocks and the copying of its module parameters, if any, into its *FDTLVAR*. A module without any transition part are inactive after its initialization. Inactive modules are generally used as structuring devices which impose an hierarchical organization to their underlying child modules.

```
body (process, signal)
FDTPCB *process;
FDTSCB *signal;
  {
    FDTLVAR *lvar = process->lvar;
    FDTSVAR *svar = signal->svar;

    /* Code for transitions (See Section 3.5) */
    ...


    /* Exit code when no transition was triggered */
    if (signal->cid == 0)
      FDTSCBdispose (process, signal);
    return;

    /* Exit code when a transition was triggered */
    trans_end :
    FDTSCBdispose (process, signal);
    if (process->spont)
      FDTSCBspont (process);

    /* Exit code for spontaneous transition */
    spont_end :
    process->delay = 0;
  }
```

Figure 3.7: Transition Routine

Each transition is specified in two parts. The actions to be performed by the transition are defined by a Pascal style statement block. This transition block is preceded by zero or more transition clauses. The transition clauses are used to specify the enabling conditions which must be satisfied before the transition block can be executed. This section describes the translation of the transition clauses followed by the translation of the transition blocks.

3.5.1 Transition Clauses

In Estelle, transition clauses may be used to specify the enabling conditions of a transition in terms of the present state (FROM clause), the input signal (WHEN clause), an enabling predicate (PROVIDED clause) or a transition priority (PRIORITY clause). Other clauses may be used to specify actions such as going to a specified next state (TO clause) or delaying the action for a specified time (DELAY clause). Finally, there is also a clause that can be used as a shorthand notation for a sequence of transition (ANY clause). The translation scheme for generating codes for the enabling transition clauses is essentially one of substituting a corresponding boolean expression for the clause. The strategy for translating the action transition clauses is to place statements that perform the indicated action within the enclosing transition block. Compare Appendices A and B for illustrations.

The translation of most of the transition clauses are straight forward. A FROM clause is translated into a boolean expression testing for the current state (Figure 3.8). As described in Section 3.2, the current state for a module is stored as the variable STATE in the module variable block (*FDTLVAR*) for the module. The WHEN clause is also translated into a boolean expression (Figure 3.9). Two tests must be made. First the specified interaction point is tested against the incoming signal (`signal->cid`). Then the incoming interaction primitive type (`signal->sid`) must match the primitive specified in the clause.

```
if (lvar->body.STATE == <FROM state>)
  {
     ... /* Nested transitions */
  }
```

Figure 3.8: Codes generated for a FROM clause

```
if ((signal->cid == <channel>) && (signal->sid == <primitive>))
  {
    ... /* Nested transitions */
  }
```

Figure 3.9: Codes generated for a WHEN clause

Although the natural translation for a PROVIDED clause is also a boolean expression, but because of the possible presence of an OTHERWISE condition, its translation is not straight forward (Figure 3.10). The strategy taken in this implementation is to declare and initialize a boolean flag to TRUE before the execution of the boolean expressions. After every PROVIDED clause where a boolean expression is specified, a statement is added to set the flag to FALSE. If the boolean expression is evaluated to TRUE, then the flag will be set to FALSE, otherwise the flag will remain TRUE. If an OTHERWISE condition is specified in the final PROVIDED clause, a boolean expression is generated to test the boolean flag for the TRUE condition. In this way, the OTHERWISE transition is executed if and only if none of the previous boolean expressions are satisfied.

The DELAY clause is the most complicated clause to translate (Figure 3.11). Three auxiliary variables (`delay`, `t0`, `t1`) located in the *FDTPCB* are used in order to implement this clause. Each DELAY clause specified is assigned an unique number. The variable `delay` is always set to

```
{
  int flag = 1 /* TRUE */;

  if (boolean expression 1)   /* PROVIDED clause 1 */
    {
      flag = 0 /* FALSE */;
        {
          ...  /* Nested transitions */
        }
    }

  if (boolean expression 2)   /* PROVIDED clause 2 */
    {
      flag = 0 /* FALSE */;
        {
          ...  /* Nested transitions */
        }
    }

  ...

  if (flag == 1 /* TRUE */)   /* OTHERWISE clause */
    {
      ... /* Nested transitions */
    }
}
```

Figure 3.10: Codes generated for a PROVIDED clause

```
{
  process->t0 = time(0);

  /* Set timer if not already set */

  if (process->delay != <delay id>)
    {
      process->t1 = process->t0 + <delay time>;
      process->delay = <delay id>;
    }

  /* Tests for timer expiration */

  if (process->t0 >= process->t1)
    {
      ....  /* Perform action */
    }
  else return;
}
```

Figure 3.11: Codes generated for a DELAY clause

the number assigned to the DELAY clause currently in effect. If none is in effect, the variable is set to the value '0'. The variable `t0` is used to store the current time while the variable `t1` is used to store the expiration time for the delay. See Section 4.2.3 for the run-time effect of this clause.

The TO clause is translated into a statement in the enclosing transition block which changes the module state variable to the indicated state in the TO clause. If there are several transition block nested under a TO clause, then the state change statement is replicated in all of the enclosing transition blocks.

The ANY clause is translated into a simple `for` statement which steps through all values in the specified scalar domain. If more than one domain is specified, a set of nested `for` statements are used.

Within a transition routine, the transitions are layed out in the same sequence that they are defined in the Estelle specification. Therefore, the transition clauses will be evaluated in the order in which that they are specified. The transition that will be executed will be the first transition which enabling clauses are all satisfied. The use of this scheme implies that the order in which the transitions are specified is significant. Consequently, the PRIORITY clause is not implemented in the Estelle-C compiler. The user can always rearrange the transitions in the order of their priority.

Although the scheme used is deterministic, the Estelle definition does allow the protocol implementer to make this choice [Este86]. If non-deterministic transition is to be supported, all of the transition clauses must be evaluated to determine the enabled set. From the enabled set of transitions, the ones with the highest priority and which also satisfy the delay criteria are selected. Finally, from this fireable set, a transition must be offered to be executed non-

deterministically. This three step selection process will only make for an inefficient protocol implementation.

3.5.2 Transition Blocks

The translation of the Pascal style statements in a transition block into equivalent C style statements is generally done by straight substitution. The problems encountered are already noted by Ford and Lau [Ford85,Lau86]. Most of these difficulties have to do with Pascal constructs which have no equivalence in C.

Most of the special statements provided in Estelle are translated into subroutine calls to the appropriate run-time support routines which implement the corresponding functions. These routines are described in Sections 3.3.1 and 3.3.2.

The OUTPUT statement is translated into a C block containing a local pointer variable newsvar (Figure 3.12). This local variable is used to allocate a signal parameter block *FDTSVAR*

```
{
  FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

  newsvar-><channel>.<primitive>.<parameter1> = <value1>;
  newsvar-><channel>.<primitive>.<parameter2> = <value2>;
  ...

  FDTSCBsignal (process, <channel id>, <primitive id>, newsvar);
}
```

Figure 3.12: Codes generated for an OUTPUT statement

within the block. Then, the parameters supplied to the OUTPUT statement are copied into the *FDTSVAR*. Finally, a signal control block (*FDTSCB*) is constructed and appended to the destination queue specified in the OUTPUT statement using the run-time routine FDTSCBsignal().

Chapter 4

Estelle Run-time Execution

An executable program generated from the Estelle specification described in Chapter 3 still only represents a static description of the Estelle specification. It is only when this C program is executed that a dynamic entity will result. This chapter describes the inner working of the generated program during execution.

4.1 Run-time Organization

The execution of an Estelle specification is implemented as a two-stage process driven by the main driver routine supplied in the Estelle run-time support package (Figure 4.1). First,

```
main()
  {
    FDTPCB *root;

    root = FDTSPECIFICATION(NULL);
    FDTSCHexec(root);
  }
```

Figure 4.1: Main Driver Routine

the driver constructs a run-time structure to represent the initial module hierarchy for the

specification by calling the initialization routine, FDTSPECIFICATION(). Then, the driver calls the run-time scheduler routine, FDTSCHexec(), to take over the execution of the protocol.

The following sections explain how two dependent run-time structures are generated from the same set of control blocks and how these structures are used by the run-time scheduler to execute an Estelle specification.

4.1.1 Initialization Routines

The initialization routines generated from the Estelle module body declarations are invoked in a sequence which reflects the nested module organization defined in the Estelle specification. As noted in Section 3.4, the initialization routine of the specification module is always named FDTSPECIFICATION(). This routine is the only specification dependent routine that is directly invoked by the run-time support system. Consequently, the use of a fixed name for this routine is one of the reasons why the new Estelle run-time support system needs not be recompiled for every different Estelle specification.

The function of an initialization routine for a module is to create the two control blocks, *FDTPCB* and *FDTLVAR*, which when taken together, represent an instance of the module, and to execute the initialization routines of all its child modules. The result of each invocation of an initialization routine is the simultaneous construction of two tree structures which represent the initialized module and all its child modules in two different ways.

In one system, each tree structure constructed by a child module is stored in a module variable located in the *FDTLVAR* of the parent (See Figure 3.2). After the initialization process, a parent module can refer to any of its child modules by name through the use of these module variables in its *FDTLVAR*. This system is used within the implementation routines generated from the Estelle specification. The second system is generated automatically when

the initialization routines invoke `FDTPCBinit()` to create the *FDTPCBs*. This system is used by the run-time scheduler to refer to the modules anonymously and in a specification independent fashion.

4.1.2 Scheduler Routine

The function of the scheduler is to repeatedly select an interaction from the pool of pending interactions and to execute the appropriate module to process the selected interaction. The scheduling algorithm used is, in essence, a pre-order traversal of the module hierarchy tree in a round-robin manner. However, the scheduling algorithm is not straightly round-robin because of the parent/child priority relationship which exists in the module hierarchy. The scheduler routine is shown in Figure 4.2.

The scheduler keeps track of a current module for each level in the hierarchy. At any one time, the current module at one of these level is the current module in the system. The scheduler first checks if there are any pending interactions for the module. If a pending interaction exists, then the current module is selected to be executed. There is also the concept of a next level. If no pending interaction exists, the next level will be one level down. But, if a pending interaction exists and the execution of current module affects some of its ancestor modules, then the level of the ancestor module closest to the specification module will become the next level. Otherwise, the next level is still the current level. In any case, the next module to be selected at the current level will be the next sibling module of the current module.

To summarize, the next level stays at the current level or goes up if a pending interaction exists for the current level. Otherwise, the next level becomes one level down. This scheduler algorithm ensured that when a module has the potential to execute transitions, none of its child modules can execute. The algorithm also ensures against module starvation because it

```
FDTSCHexec (root)
FDTPCB *root;
  {
    CurrLevel = (FDTSCH *) malloc(sizeof(FDTSCH));
    CurrLevel->prev = CurrLevel;
    CurrLevel->next = NULL;
    CurrLevel->pcb  = root;
    while (1)
      {
        CurrBlock  = CurrLevel->pcb;
        CurrSignal = FDTSCBpending(CurrBlock);
        if (CurrSignal != NULL)
          {
            if (CurrBlock->export)
              NextLevel = CurrLevel->prev;
            else
              NextLevel = CurrLevel;
            /* Transition routine may change NextLevel */
            CurrBlock->trans (CurrBlock, CurrSignal);
          }
        else if (CurrBlock->ref != NULL)
          {
            if (CurrLevel->next == NULL) {
                NextLevel = (FDTSCH *) malloc(sizeof(FDTSCH));
                NextLevel->prev = CurrLevel;
                NextLevel->next = NULL;
                NextLevel->pcb  = CurrBlock->ref;
            } else {
                NextLevel = CurrLevel->next;
                if (NextLevel->pcb == NULL)
                    NextLevel->pcb = CurrBlock->ref;
            }
          }
        CurrLevel->pcb = CurrBlock->sib;
        CurrLevel = NextLevel;
        while (CurrLevel->pcb == NULL)
            CurrLevel = CurrLevel->prev;
      }
  }
```

Figure 4.2: Run-time Scheduler Routine

is not possible for a module to execute two transitions in a row even if it has several pending interactions enqueued at the same time.

4.2 Transition Processing

After the scheduler executes the transition routine of the current module, the next step is for the transition routine to search for the first transition within the module which satisfies all its enabling conditions. This transition is then executed to process the input interaction. The following sections elaborate on the procedures for processing transitions in various situations.

4.2.1 Input Transitions

When all the enabling condition of an input transition is satisfied, the actions specified in its transition block is executed. After the completion of the transition actions, the module performs an exit sequence which is common to all input transitions (See Figure 3.7). First, because the interaction which caused the transition has already been processed, it is disposed. Second, because the actions of the just completed transition may have enabled one of the spontaneous transitions in the module, actions must be taken to ensure that the scheduler will execute the module once more in order to check for this situation. If a pending interaction exists for the module, nothing needs to be done. However, if none exists then a spontaneous interaction is generated for the module by the use of the routine `FDTSCBspont()`. Finally, because the just completed transition would have nullified any pending delayed transition, the variable `delay` in the *FDTPCB* is cleared.

4.2.2 Spontaneous Transitions

The exit sequence after the completion of a spontaneous transition is a lot simpler than that for an input transition. There are two reasons for this difference. Firstly, if the interaction selected for the module is an input interaction, then it will not be processed by the spontaneous transition. Therefore, the interaction selected for the module by the scheduler should not be disposed. Secondly, if the interaction is a spontaneous interaction, then another spontaneous interaction must be needed to check for more enabled spontaneous transitions. Again, there is no need to dispose the interaction. The only action necessary after a spontaneous interaction is to nullify any pending delayed transition (See Figure 3.7).

4.2.3 Delayed Transitions

Delayed transitions are spontaneous transitions with additional timing constraints. However, unlike other transitions, delay transition also has a entry code sequence (See Figure 3.11). If the transition is newly enabled, then the delay parameters are saved by this entry sequence into the *FDTPCB* of the module. A desirable side effect of this action is that it will also nullify another pending delayed transition in the module. In the next step, the delay timer is checked. If the delay constraints are satisfied, the transition is executed. In this case, the exit sequence for it is identical to that for a regular spontaneous transition. Otherwise, control is immediately returned to the scheduler. This bypassing of the exit sequence will cause the selected interaction for the module to remain pending and to cause the scheduler to execute the module at a future time.

4.2.4 No Enabled Transitions

The final situation which must be taken into account is the situation in which the interaction selected by the scheduler does not trigger any transitions. This situation can occur because the scheduler has no knowledge of the transitions within the transition routines. Therefore, the availability of a pending interaction does not guarantee its processing and disposal. This situation is most likely caused by a spontaneous interaction which is used only to check for enabled spontaneous transitions. The actions needed to handle this situation is to dispose the interaction if it is spontaneous, and to leave it pending in the queue if it is not. In order to help prevent deadlock, when the module is again selected for execution, the scheduler will search for pending interactions at other interaction points first.

Chapter 5

Conclusions

The present study found substantial syntactic and semantic differences between the Estelle language as implemented by the existing UBC Estelle-C compiler and that specified in the latest ISO document [Este86] and cumulated in the construction of a new Estelle-C compiler. The new compiler supports a large subset of the latest Estelle language specification. The following sections summarize the present work and suggest possible future studies.

5.1 Thesis Summary

As stated in Section 2.3, the major motivation for developing the new compiler is to upgrade an existing UBC Estelle-C compiler to support the latest Estelle language specification. The new compiler fulfills this goal by supporting dynamic reconfiguration of the various entities in a protocol specification. However, there are several Estelle features not included in this new compiler. The PRIORITY clause is not supported due to the reasons given in Section 3.5.1. Additional Estelle features missing are the ALL statement, the FORONE statement and the EXIST expression which are used for implicit access to module instances by types rather than by names. A general solution that implements these constructs would require a module directory service to associate module types with module names. The module directory would be further

complicated by the presence of indexed module names. In view of the fact that these constructs can usually be replaced by a mixture of iterative and conditional statements designed for specific situations, their general support would not be cost effective in terms of run-time overhead.

Other issues resolved in the new Estelle-C compiler include problems cited by Lau in regards to the old compiler [Lau86]. The old compiler handles module parameter passing very clumsily. In the new compiler, module parameters are automatically accessible in the module initialization routines, the module transition routines and the subroutines nested within the Estelle module. Global variables are also supported as defined by the formal semantics in the new Estelle language specification. The Pascal multi-dimensional ARRAY type is now available for user defined variables as well as MODULE and IP types. However, the data type SET is only partially supported in the form of STATESET. Other Pascal-to-C translation problems that remained are in the areas of SET expressions and nested procedures and functions. These limitations are certainly not unsolvable but their solutions are beyond the scope of protocol implementation.

The new compiler is hand-coded in C without using the UNIX utilities *lex* and *yacc*. It consists of approximately 300 subroutines totaling to just under 11,000 lines of source code and just under 11K bytes of object code. The size of the new compiler compares favorably with that of the old compiler which contains over 14,000 lines of source code and just over 14K bytes of object code.

The new Estelle run-time support routines are also implemented in C. There are 40 routines in the package with approximately 1,500 lines of source code and 7K bytes of object code. In contrast, there are only 17 routines in the old run-time support routines with 1,400 lines of source code and 2K bytes of object code.

As part of the redesign of the Estelle-C compiler, the user operation of the compiler has been

greatly simplified. In the old compiler, the user is required to modify certain sections of the generated C code as well as the run-time support routines. Consequently, the old compiler was labeled with the term "semi-automatic." The new compiler translates an Estelle specification directly into a compilable C program. Modification of any of the generated C code is no longer necessary. In addition, the new run-time support routines have been rewritten to contain only specification independent details and they can be directly linked to the compiled C program without the need for recompilation. With these two user-friendly improvements, "automatic" implementation of protocols from formal protocol specifications is now realized.

5.2 Future Work

Although an executable program can be generated automatically from a formal protocol specification using the new Estelle-C compiler, formal protocol specifications are usually incomplete. It is the nature of formal specifications to be "completely independent of methods of implementation, so that the technique itself does not provide any undue constraints on implementers" [Este86]. Examples of implementation dependent properties that are almost always left unspecified are functions such as user data management routines, timer management routines and protocol data unit encoding and decoding routines. The user must provide customized versions of these functions manually for the actual implementations in different machine environments. However, it may be possible to define generic interfaces for these functions to be usable from within a formal Estelle specification. The user data management example in the Estelle language specification [Este86] presents one possibility. Another direction for research may be in the area of incorporating ASN.1 [CCI86] into Estelle for the specification of protocol data unit encoding and decoding functions.

The usefulness of Estelle compilers has already been demonstrated for semi-automatic generation of communication protocols [Boch86,Lau86,Boch87b] and for automatic generation of test skeletons from protocol specifications [Favr87]. The possible enhancements of these Estelle compilers for the production of emulators for protocol testing and monitors for protocol performance evaluation would be invaluable. Therefore, the further development of novel applications using these compilers in other areas of protocol development should be encouraged.

Bibliography

[Ansa87] Ansart, J.P., Amer, P., Chari, V., Lenotre, J.F., Lumbroso, L., Mariani, E. and Mattera, E., "Software Tools for Estelle," *Protocol Specification, Testing, and Verification, VI,* (IFIP/WG6.1), B. Sarikaya and G.V. Bochmann, Eds., North Holland, pp. 55–61, 1987.

[Boch86] Bochmann, G.v., "Semi-Automatic Implementation of Transport and Session Protocols," *Computer Standards & Interfaces,* 5:343–349, 1986.

[Boch87a] Bochmann, G.v., "Usage of Protocol Development Tools: The Results of a Survey," *Protocol Specification, Testing, and Verification, VII,* (IFIP/WG6.1), H. Rudin and C.H. West, Eds., North Holland, 1987.

[Boch87b] Bochmann, G.v., Gerber, G.W. and Serre, J.M., "Semiautomatic Implementation of Communication Protocols," *IEEE Trans. on Software Engineering,* SE-13:989–1000, 1987.

[CCI86] CCITT, "Data Communication Networks — Message Handling Systems — Recommendations X.400-X.430," Red Book, Vol. VIII, Fascicle VIII.7, Ganeva, 1986.

[Cour86] Courtiat, J.P., Pedroza, A. and Ayache, J.M., "A Simulation Environment for Protocol Specifications Described in Estelle," *Protocol Specification, Testing, and Verification, V,* (IFIP/WG6.1), M. Diaz, Ed., North Holland, pp. 297–312, 1986.

[Este84] ISO/TC97/SC21/WG1/Subgroup B, "A Formal Description Technique Based on an Extended State Transition Model," Working Document, 1984.

[Este85] ISO/TC97/SC21/WG1/Subgroup B, "Estelle — A Formal Description Technique Based on an Extended State Transition Model," DP 9074, 1985.

[Este86] ISO/TC97/SC21/WG1/Subgroup B, "Estelle — A Formal Description Technique Based on an Extended State Transition Model," 2nd DP 9074, 1986.

[Favr87] Favreau, J.P. and Linn, R.J., "Automatic Generation of Test Skeletons from Protocol Specification Written in Estelle," *Protocol Specification, Testing and Verification, VI* (IFIP/WG 6.1), B. Sarikaya and Bochmann, G.v., Eds., North Holland (1987).

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

[Garg87] Garguilo, J., Fauneau, J.P., Hobbs, M. and Linn, R.J. "Automated Protocol Development Through Use of the NBS Prototype Estelle Compiler," ICST/APM-87-2, National Bureau of Standards, Guithersburg, 1987.

[Gerb83] Gerber, G.W., "Une Methode D'Implantation Automatique de Systemes Specifies Formellement," Master Thesis, University of Montreal, 1983.

[ISO82a] ISO/TC97/SC16, "Transport Protocol Specification," DP 8073, 1982.

[ISO82b] ISO/TC97/SC16, "Transport Service Definition," DP 8072, 1982.

[Lau86] Lau, A.C. "A Semi-Automatic Approach to Protocol Implementation — The ISO Class 2 Transport Protocol as an Example," Master Thesis, University of British Columbia, 1986.

[Linn86] Linn, R.J., Jr., "The Features and Facilities of Estelle," *Protocol Specification, Testing, and Verification, V,* (IFIP/WG6.1), M. Diaz, Ed., North Holland, pp. 271–296, 1986.

[Vuon87] Vuong, S.T. and Lau, A.C., "A Semi-Automatic Approach to Protocol Implementation — The ISO Class 2 Transport Protocol as an Example," INFOCOM '87, San Francisco, 1987.

[Vuon88] Vuong, S.T., Lau, A.C. and Chan, R.I., "Semi-Automatic Implementation of Protocols using an Estelle-C Compiler," *IEEE Trans. on Software Engineering,*, to be published, 1988.

Appendix A

Alternating Bit Protocol — Estelle Specification

```
SPECIFICATION abp_spec SYSTEMPROCESS;  TIMESCALE seconds;

CONST
  LOW_CEP    = 1;  { Minimum cep subscript }
  HIGH_CEP   = 2;  { Maximum cep subscript }

TYPE
  cep_type   = LOW_CEP..HIGH_CEP;  { Connection end point }
  seq_type   = 0..1;               { Sequence number      }
  pid_type   = (DATA, ACKM);       { Packet type          }
  udata_type = INTEGER;
  ndata_type = RECORD
                 pid : pid_type;   { Type of message }
                 cid : cep_type;   { Cep of sender   }
                 seq : seq_type;   { Sequence number }
                 dat : udata_type  { User data       }
               END;


{ Channel between user and alternating bit protocol provider }

CHANNEL U_access_point (user, provider);
  BY user :
    SEND_REQ  (udata : udata_type);
    RECV_REQ;
  BY provider :
    RECV_RSP (udata : udata_type);


{ Channel between alternating bit protocol provider and the network }

CHANNEL N_access_point (user, provider);
  BY user :
    DATA_REQ  (ndata : ndata_type);
  BY provider :
    DATA_RSP (ndata : ndata_type);


MODULE user_type PROCESS (cep_id : cep_type); IP
  U : U_access_point(user) INDIVIDUAL QUEUE;
END;  { MODULE user_type }

BODY user_body FOR user_type;

  VAR
    data : udata_type;
    flag : BOOLEAN;
```

```
  INITIALIZE

    BEGIN
      data := 0;
      flag := TRUE
    END;  { INITIALIZE }

  TRANS

    WHEN U.RECV_RSP

    { Received data from peer and proceeds to send next data to peer }

    NAME user1 : BEGIN
      data := data + 1;
      OUTPUT U.SEND_REQ (data);
      OUTPUT U.RECV_REQ
    END;  { user1 }

  TRANS

    PROVIDED flag

    { Spontaneous transition to send initial data }

    NAME user2 : BEGIN
      flag := FALSE;
      OUTPUT U.SEND_REQ (data);
      OUTPUT U.RECV_REQ
    END;  { user2 }

END;  { BODY user_body }


MODULE network_type PROCESS; IP
  N : ARRAY [cep_type] OF N_access_point (provider) COMMON QUEUE;
END;  { MODULE network_type }

BODY network_body FOR network_type;

  VAR
    count : INTEGER;

  TRANS

    ANY i : cep_type DO
```

```
      WHEN N[i].DATA_REQ

    NAME network1 : BEGIN
      count := count + 1;
      IF count <> 4 THEN
        OUTPUT N[HIGH_CEP-i+1].DATA_RSP (ndata)
    END;  { network1 }

END;  { BODY network_body }


MODULE abit_type PROCESS (cep_id : cep_type); IP
  U : U_access_point (provider) INDIVIDUAL QUEUE;
  N : N_access_point (user)     INDIVIDUAL QUEUE;
END;  { MODULE abit_type }

BODY abit_body FOR abit_type;

  CONST
    RETRAN_TIME = 30;  { Retransmission time }

  CHANNEL S_access_point (user, provider);
    BY user :
      TIMER_REQ;
    BY provider :
      TIMER_RSP;

  MODULE timer_type ACTIVITY (time : INTEGER); IP
    S : S_access_point (provider) INDIVIDUAL QUEUE;
  END;  { MODULE timer_type }

  BODY timer_body FOR timer_type;

    VAR
      stop, stop_bis : BOOLEAN;

    INITIALIZE

      BEGIN
        stop     := TRUE;
        stop_bis := TRUE
      END;  { INITIALIZE }

    TRANS

      WHEN S.TIMER_REQ
```

```
    NAME timer1 : BEGIN
      stop     := TRUE;
      stop_bis := FALSE
    END;  { timer1 }

  TRANS

    PROVIDED NOT stop_bis

    NAME timer2 : BEGIN
      stop     := FALSE;
      stop_bis := TRUE
    END;  { timer2 }

    PROVIDED NOT stop
      DELAY (time, time)

    NAME timer3 : BEGIN
      stop := TRUE;
      OUTPUT S.TIMER_RSP
    END;  { timer3 }

END;  { BODY timer_body }


MODULE datax_type ACTIVITY (cep_id : cep_type); IP
  U : U_access_point (provider) INDIVIDUAL QUEUE;
  N : N_access_point (user)     INDIVIDUAL QUEUE;
  S : S_access_point (user)     INDIVIDUAL QUEUE;
END;  { MODULE datax_type }

BODY datax_body FOR datax_type;

  TYPE
    msg_type = RECORD
                 msgcid : cep_type;
                 msgseq : seq_type;
                 msgdat : udata_type
               END;
    buf_type = RECORD
                 empty   : BOOLEAN;
                 message : msg_type
               END;

  VAR
    send_seq : seq_type;    { Send sequence number    }
    recv_seq : seq_type;    { Receive sequence number }
```

```
    send_buf : buf_type;     { ACKM pending flag       }
    recv_buf : buf_type;     { DATA pending flag       }
    send_msg : msg_type;     { Message being sent      }
    recv_msg : msg_type;     { Message receive         }
    buf      : ndata_type;   { Network buffer          }

  STATE
    ACK_WAIT, ESTAB;

  STATESET
    EITHER = [ACK_WAIT, ESTAB];

  PURE FUNCTION ack_ok (buf : ndata_type) : BOOLEAN;

    { Checks ACKM message in the network buffer }

    BEGIN  { ack_ok }
      ack_ok := (buf.pid = ACKM) AND (buf.seq = send_seq)
    END;  { ack_ok }

  PROCEDURE format_data (msg : msg_type; VAR buf : ndata_type);

    { Formats a DATA message into the network buffer }

    BEGIN  { format_data }
      buf.pid := DATA;
      buf.cid := cep_id;
      buf.seq := msg.msgseq;
      buf.dat := msg.msgdat
    END;  { format_data }

  PROCEDURE format_ack (msg : msg_type; VAR buf : ndata_type);

    { Places an ACKM message into the network buffer }

    BEGIN  { format_ack }
      buf.pid := ACKM;
      buf.cid := msg.msgcid;
      buf.seq := msg.msgseq;
      buf.dat := msg.msgdat
    END;  { format_ack }

  PROCEDURE store (VAR buf : buf_type; msg : msg_type);

    { Stores message into the buffer }

    BEGIN  { store }
```

```
    buf.empty   := FALSE;
    buf.message := msg;
  END;  { store }

PROCEDURE remove (VAR buf : buf_type; msg : msg_type);

  { Empties the buffer }

  BEGIN  { remove }
    buf.empty := TRUE
  END;

FUNCTION retrieve (buf : buf_type) : msg_type;

  { Retrieves the message from the buffer }

  BEGIN  { retrieve }
    retrieve := buf.message
  END;  { retrieve }

FUNCTION buffer_empty (buf : buf_type) : BOOLEAN;

  { Checks for empty buffer }

  BEGIN  { buffer_empty }
    buffer_empty := buf.empty
  END;  { buffer_empty }

PROCEDURE inc_send_seq;

  { Increments the send sequence number }

  BEGIN  { inc_send_seq }
    send_seq := (send_seq + 1) MOD 2
  END;  { inc_send_seq }

PROCEDURE inc_recv_seq;

  { Increments the receive sequence number }

  BEGIN  { inc_recv_seq }
    recv_seq := (recv_seq + 1) MOD 2
  END;  { inc_recv_seq }


INITIALIZE  { data_body }
```

```
    TO ESTAB
    BEGIN
      send_seq       := 0;
      recv_seq       := 0;
      send_buf.empty := TRUE;
      recv_buf.empty := TRUE
    END;  { INITIALIZE }


  TRANS

    FROM ESTAB TO ACK_WAIT
      WHEN U.SEND_REQ

    { Processes user send REQ }

    NAME datax1 : BEGIN
      send_msg.msgdat := udata;
      send_msg.msgseq := send_seq;
      store (send_buf, send_msg);
      format_data (send_msg, buf);
      OUTPUT N.DATA_REQ (buf);
      OUTPUT S.TIMER_REQ
    END;  { datax1 }

    FROM EITHER TO SAME
      WHEN U.RECV_REQ
        PROVIDED NOT buffer_empty (recv_buf)

    { Retrieves received message for user if one has been received }

    NAME datax2 : BEGIN
      recv_msg := retrieve (recv_buf);
      OUTPUT U.RECV_RSP (recv_msg.msgdat);
      remove (recv_buf, recv_msg)
    END;  { datax2 }

    FROM ACK_WAIT TO ACK_WAIT
      WHEN S.TIMER_RSP

    { Resend user message after time out }

    NAME datax3 : BEGIN
      send_msg := retrieve (send_buf);
      format_data (send_msg, buf);
      OUTPUT N.DATA_REQ (buf);
      OUTPUT S.TIMER_REQ
```

```
    END;  { datax3 }

    FROM ESTAB TO ESTAB
      WHEN S.TIMER_RSP

    { The message that caused this timer to be set has been acknowledged }

    NAME datax4 : BEGIN
    END;  { datax4 }

    FROM ACK_WAIT TO ESTAB
      WHEN N.DATA_RSP
        PROVIDED ack_ok(ndata)

    { Acknowlegement for the last message sent has been received }

    NAME datax5 : BEGIN
      send_msg := retrieve (send_buf);
      remove (send_buf, send_msg);
      inc_send_seq
    END;  { datax5 }

    FROM EITHER TO SAME
      WHEN N.DATA_RSP
        PROVIDED ndata.pid = DATA

    { Processes message received from peer }

    NAME datax6 : BEGIN
      recv_msg.msgdat := ndata.dat;
      recv_msg.msgseq := ndata.seq;
      format_ack (recv_msg, buf);
      OUTPUT N.DATA_REQ (buf);
      IF ndata.seq = recv_seq THEN BEGIN
        store (recv_buf, recv_msg);
        inc_recv_seq
      END  { IF }
    END;  { datax6 }

END;  { BODY datax_body }

MODVAR
  datax_module : datax_type;
  timer_module : timer_type;

INITIALIZE  { abit_body }
  BEGIN
```

```
      INIT datax_module WITH datax_body (cep_id);
      INIT timer_module WITH timer_body (RETRAN_TIME);

      CONNECT datax_module.S TO timer_module.S;
      ATTACH U TO datax_module.U;
      ATTACH N TO datax_module.N;
    END;  { INITIALIZE }
  END;  { abit_body }


MODVAR
  network_module : network_type;
  user_module    : ARRAY [cep_type] OF user_type;
  abit_module    : ARRAY [cep_type] OF abit_type;

INITIALIZE  { abp_spec }

  BEGIN
    INIT network_module WITH network_body;
    ALL cep : cep_type DO BEGIN
      INIT user_module[cep] WITH user_body(cep);
      INIT abit_module[cep] WITH abit_body(cep);

      CONNECT user_module[cep].U TO abit_module[cep].U;
      CONNECT abit_module[cep].N TO network_module.N[cep];
    END;  { ALL }
  END;  { INITIALIZE }
END.  { abp_spec }
```

Appendix B

Alternating Bit Protocol — Generated Codes

```
#include   <stdio.h>
#include   "fdtset.h"
#include   "fdtscb.h"
#include   "fdtccb.h"
#include   "fdtpcb.h"
#include   "fdtsch.h"

/* Type declarations */

typedef int cep_type ;
typedef int seq_type ;
typedef int pid_type ;
typedef int udata_type ;
typedef struct {
          int dat ;
          int seq ;
          int cid ;
          int pid ;
        } ndata_type ;
typedef struct {
          int msgdat ;
          int msgseq ;
          int msgcid ;
        } msg_type ;
typedef struct {
          msg_type message ;
          int empty ;
        } buf_type ;


/* Signal parameter block declarations */

typedef union {
  union {
    struct {
      int udata ;
    } SEND_request ;
    int RECV_request ;
    struct {
      int udata ;
    } RECV_response ;
  } U_access_point ;
  union {
    struct {
      ndata_type ndata ;
    } DATA_request ;
    struct {
```

```
      ndata_type ndata ;
    } DATA_response ;
  } N_access_point ;
  union {
    int TIMER_request ;
    int TIMER_response ;
  } S_access_point ;
} FDTSVAR;


/* Variable block declarations */

typedef union {
  struct {
    int cep_id ;
    int flag ;
    int data ;
  } user_body ;
  struct {
    int dummy ;
  }
  struct {
    int time ;
    int FDT3 ;
    int stop ;
    int stop_bis ;
  } timer_body;
  struct {
    int cep_id ;
    set_type EITHER ;
    int STATE ;
    ndata_type buf ;
    msg_type recv_msg ;
    msg_type send_msg ;
    buf_type recv_buf ;
    buf_type send_buf ;
    int recv_seq ;
    int send_seq ;
  } datax_body;
  struct {
    int cep_id ;
    struct FDTPCB *timer_module ;
    struct FDTPCB *datax_module ;
  } abit_body;
  struct {
    struct FDTPCB *abit_module [ 2 ];
    struct FDTPCB *user_module [ 2 ];
```

```
    struct FDTPCB *network_module ;
  } SPECIFICATION ;
} FDTLVAR;


/* Miscellaneous declarations */

#define XPORTuser_body       0
#define SPONTuser_body       0
extern int user_body();
#define TRANSuser_body       user_body
#define XPORTnetwork_body    0
#define SPONTnetwork_body    0
extern int network_body();
#define TRANSnetwork_body    network_body
#define XPORTtimer_body      0
#define SPONTtimer_body      1
extern int timer_body();
#define TRANStimer_body      timer_body
#define XPORTdatax_body      0
#define SPONTdatax_body      0
extern int datax_body();
#define TRANSdatax_body      datax_body
#define XPORTabit_body       0
#define SPONTabit_body       0
#define TRANSabit_body       NULL
#define XPORTSPECIFICATION   0
#define SPONTSPECIFICATION   0
#define TRANSSPECIFICATION   NULL


/* Procedure and function declarations */

int ack_ok ( buf )
ndata_type buf ;
{
  FDTLVAR *lvar = CurrBlock->lvar;
  int FUNCTION;

  FUNCTION = ( buf .pid == 1  /* ACKM */) &&
             ( buf .seq == lvar->datax_body.send_seq ) ;
  return ( FUNCTION ) ;
}


format_data ( msg , buf )
msg_type msg ;
```

```
ndata_type *buf ;
{
  FDTLVAR *lvar = CurrBlock->lvar;

  (*buf ) .pid = 0  /* DATA */;
  (*buf ) .cid = lvar->datax_body.cep_id ;
  (*buf ) .seq = msg .msgseq ;
  (*buf ) .dat = msg .msgdat ;
}


format_ack ( msg , buf )
msg_type msg ;
ndata_type *buf ;
{
  FDTLVAR *lvar = CurrBlock->lvar;

  (*buf ) .pid = 1  /* ACKM */;
  (*buf ) .cid = msg .msgcid ;
  (*buf ) .seq = msg .msgseq ;
  (*buf ) .dat = msg .msgdat ;
}


store ( buf , msg )
buf_type *buf ;
msg_type msg ;
{
  FDTLVAR *lvar = CurrBlock->lvar;

  (*buf ) .empty = 0  /* FALSE */;
  (*buf ) .message = msg ;
}


remove ( buf , msg )
buf_type *buf ;
msg_type msg ;
{
  FDTLVAR *lvar = CurrBlock->lvar;

  (*buf ) .empty = 1  /* TRUE */;
}


msg_type retrieve ( buf )
buf_type buf ;
```

```
{
  FDTLVAR *lvar = CurrBlock->lvar;
  msg_type FUNCTION;

  FUNCTION = buf .message ;
  return ( FUNCTION ) ;
}


int buffer_empty ( buf )
buf_type buf ;
{
  FDTLVAR *lvar = CurrBlock->lvar;
  int FUNCTION;

  FUNCTION = buf .empty ;
  return ( FUNCTION ) ;
}


inc_send_seq ()
{
  FDTLVAR *lvar = (FDTLVAR *) CurrBlock->lvar;

  lvar->datax_body.send_seq = ( lvar->datax_body.send_seq + 1 ) % 2 ;
}


inc_recv_seq ()
{
  FDTLVAR *lvar = (FDTLVAR *) CurrBlock->lvar;

  lvar->datax_body.recv_seq = ( lvar->datax_body.recv_seq + 1 ) % 2 ;
}


/* Specification declarations */

FDTPCB *FDTuser_body ( parent , cep_id )
FDTPCB *parent;
int cep_id ;
{
  FDTPCB *pcb;
  FDTLVAR *lvar;

  pcb = FDTPCBinit ( parent , 1 , SPONTuser_body , XPORTuser_body , TRANSuser_body );
  pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));
```

```
  lvar->user_body.cep_id = cep_id ;
  {
    lvar->user_body.data = 0 ;
    lvar->user_body.flag = 1  /* TRUE */;
    goto trans_end ;
  }
  trans_end :
  if (pcb->spont)
    FDTSCBspont (pcb);
  return (pcb);
}


user_body ( process, signal )
FDTPCB *process;
FDTSCB *signal;
{
  FDTLVAR *lvar = (FDTLVAR *) process->lvar;
  FDTSVAR *svar = (FDTSVAR *) signal->svar;

  {
    if ( ( signal->cid == 1 ) && ( signal->sid == 3 ) )
    { /* user1 */
      lvar->user_body.data = lvar->user_body.data + 1 ;
      {
      FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

      newsvar->U_access_point.SEND_REQ.udata = lvar->user_body.data ;
      FDTSCBsignal ( process, 1 , 1 , newsvar);
      }
      {
      FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

      FDTSCBsignal ( process, 1 , 2 , newsvar);
      }
      goto trans_end ;
    }
  }
  {
    int FDT1  = 1 ;

    if (lvar->user_body.flag )
    {
      FDT1  = 0;
      { /* user2 */
        lvar->user_body.flag = 0  /* FALSE */;
```

```
        {
        FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

        newsvar->U_access_point.SEND_REQ.udata = lvar->user_body.data ;
        FDTSCBsignal ( process, 1 , 1 , newsvar);
        }
        {
        FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

        FDTSCBsignal ( process, 1 , 2 , newsvar);
        }
        goto spont_end ;
      }
    }
  }
  if (signal->cid == 0)
    FDTSCBdispose (process, signal);
  return;

  trans_end :
  FDTSCBdispose (process, signal);
  if (process->spont)
    FDTSCBspont (process);
  spont_end :
  process->delay = 0;
}


FDTPCB *FDTnetwork_body ( parent )
FDTPCB *parent;
{
  FDTPCB *pcb;
  FDTLVAR *lvar;

  pcb = FDTPCBinit ( parent , 2 , SPONTnetwork_body ,
                     XPORTnetwork_body , TRANSnetwork_body ) ;
  pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));

  trans_end :
  if (pcb->spont)
    FDTSCBspont (pcb);
  return (pcb);
}


network_body ( process, signal )
FDTPCB *process;
```

```
FDTSCB *signal;
{
  FDTLVAR *lvar = (FDTLVAR *) process->lvar;
  FDTSVAR *svar = (FDTSVAR *) signal->svar;

  {
    int i ;

    for ( i = 1  ; i <= 2  ; i++ )
    {
      if ( ( signal->cid == 1 + i - 1 ) && ( signal->sid == 1 ) )
      { /* network1 */
        {
        FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

        newsvar->N_access_point.DATA_RSP.ndata = svar->N_access_point.DATA_REQ.ndata ;
        FDTSCBsignal ( process, 1 + 2  /* HIGH_CEP */- i + 1 - 1 , 2 , newsvar);
        }
        goto trans_end ;
      }
    }
  }
  if (signal->cid == 0)
    FDTSCBdispose (process, signal);
  return;

  trans_end :
  FDTSCBdispose (process, signal);
  if (process->spont)
    FDTSCBspont (process);
  spont_end :
  process->delay = 0;
}


FDTPCB *FDTtimer_body ( parent , time )
FDTPCB *parent;
int time ;
{
  FDTPCB *pcb;
  FDTLVAR *lvar;

  pcb = FDTPCBinit ( parent , 1 , SPONTtimer_body ,
                     XPORTtimer_body , TRANStimer_body );
  pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));

  lvar->timer_body.time = time ;
```

```
  {
    lvar->timer_body.stop = 1  /* TRUE */;
    lvar->timer_body.stop_bis = 1  /* TRUE */;
    goto trans_end ;
  }
  trans_end :
  if (pcb->spont)
    FDTSCBspont (pcb);
  return (pcb);
}


timer_body ( process, signal )
FDTPCB *process;
FDTSCB *signal;
{
  FDTLVAR *lvar = (FDTLVAR *) process->lvar;
  FDTSVAR *svar = (FDTSVAR *) signal->svar;

  {
    if ( ( signal->cid == 1 ) && ( signal->sid == 1 ) )
    { /* timer1 */
      lvar->timer_body.stop = 1  /* TRUE */;
      lvar->timer_body.stop_bis = 0  /* FALSE */;
      goto trans_end ;
    }
  }
  {
    int FDT2  = 1 ;

    if (!lvar->timer_body.stop_bis )
    {
      FDT2  = 0;
      { /* timer2 */
        lvar->timer_body.stop = 0  /* FALSE */;
        lvar->timer_body.stop_bis = 1  /* TRUE */;
        goto spont_end ;
      }
    }
    if (!lvar->timer_body.stop )
    {
      FDT2  = 0;
      {
        process->t0 = time(0);

        if ( process->delay != 3 )
        {
```

```
          process->t1 = process->t0 + ( lvar->timer_body.time );
          process->delay = 3 ;
        }

        if (process->t0 >= process->t1 )
        { /* timer3 */
          lvar->timer_body.stop = 1  /* TRUE */;
          {
            FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

            FDTSCBsignal ( process, 1 , 2 , newsvar);
          }
          goto spont_end ;
        }
        else return;
      }
    }
  }
  if (signal->cid == 0)
    FDTSCBdispose (process, signal);
  return;

  trans_end :
  FDTSCBdispose (process, signal);
  if (process->spont)
    FDTSCBspont (process);
  spont_end :
  process->delay = 0;
}


FDTPCB *FDTdatax_body ( parent , cep_id )
FDTPCB *parent;
int cep_id ;
{
  FDTPCB *pcb;
  FDTLVAR *lvar;

  pcb = FDTPCBinit ( parent , 3 , SPONTdatax_body ,
                     XPORTdatax_body , TRANSdatax_body );
  pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));

  lvar->datax_body.cep_id = cep_id ;

  assign_set ( &(lvar->datax_body.EITHER) , 2 , 1  /* ESTAB */, 0  /* ACK_WAIT */);

  {
```

```
    lvar->datax_body.send_seq = 0 ;
    lvar->datax_body.recv_seq = 0 ;
    lvar->datax_body.send_buf .empty = 1  /* TRUE */;
    lvar->datax_body.recv_buf .empty = 1  /* TRUE */;
    lvar->datax_body.STATE = 1  /* ESTAB */ ;
    goto trans_end ;
  }
  trans_end :
  if (pcb->spont)
    FDTSCBspont (pcb);
  return (pcb);
}


datax_body ( process, signal )
FDTPCB *process;
FDTSCB *signal;
{
  FDTLVAR *lvar = (FDTLVAR *) process->lvar;
  FDTSVAR *svar = (FDTSVAR *) signal->svar;

  {
    if ( ( lvar->datax_body.STATE == 1  /* ESTAB */) )
    {
      if ( ( signal->cid == 1 ) && ( signal->sid == 1 ) )
      { /* datax1 */
        lvar->datax_body.send_msg .msgdat = svar->U_access_point.SEND_REQ.udata ;
        lvar->datax_body.send_msg .msgseq = lvar->datax_body.send_seq ;
        store ( &( lvar->datax_body.send_buf ) , lvar->datax_body.send_msg );
        format_data ( lvar->datax_body.send_msg , &( lvar->datax_body.buf ) );
        {
        FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

        newsvar->N_access_point.DATA_REQ.ndata = lvar->datax_body.buf ;
        FDTSCBsignal ( process, 2 , 1 , newsvar);
        }
        {
        FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

        FDTSCBsignal ( process, 3 , 1 , newsvar);
        }
        lvar->datax_body.STATE = 0  /* ACK_WAIT */ ;
        goto trans_end ;
      }
    }
    if ( ( is_set_member ( &(lvar->datax_body.EITHER) , lvar->datax_body.STATE ) ) )
    {
```

```
  if ( ( signal->cid == 1 ) && ( signal->sid == 2 ) )
  {
    int FDT4  = 1 ;

    if (!buffer_empty ( lvar->datax_body.recv_buf ) )
    {
      FDT4  = 0;
      { /* datax2 */
        lvar->datax_body.recv_msg = retrieve ( lvar->datax_body.recv_buf ) ;
        {
        FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

        newsvar->U_access_point.RECV_RSP.udata
            = lvar->datax_body.recv_msg .msgdat ;
        FDTSCBsignal ( process, 1 , 3 , newsvar);
        }
        remove ( &( lvar->datax_body.recv_buf ) , lvar->datax_body.recv_msg );
        goto trans_end ;
      }
    }
  }
}
if ( ( lvar->datax_body.STATE == 0  /* ACK_WAIT */) )
{
  if ( ( signal->cid == 3 ) && ( signal->sid == 2 ) )
  { /* datax3 */
    lvar->datax_body.send_msg = retrieve ( lvar->datax_body.send_buf ) ;
    format_data ( lvar->datax_body.send_msg , &( lvar->datax_body.buf ) );
    {
      FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

      newsvar->N_access_point.DATA_REQ.ndata = lvar->datax_body.buf ;
      FDTSCBsignal ( process, 2 , 1 , newsvar);
    }
    {
      FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

      FDTSCBsignal ( process, 3 , 1 , newsvar);
    }
    lvar->datax_body.STATE = 0  /* ACK_WAIT */ ;
    goto trans_end ;
  }
}
if ( ( lvar->datax_body.STATE == 1  /* ESTAB */) )
{
  if ( ( signal->cid == 3 ) && ( signal->sid == 2 ) )
  { /* datax4 */
```

```
      lvar->datax_body.STATE = 1  /* ESTAB */ ;
      goto trans_end ;
    }
  }
  if ( ( lvar->datax_body.STATE == 0  /* ACK_WAIT */) )
  {
    if ( ( signal->cid == 2 ) && ( signal->sid == 2 ) )
    {
      int FDT5  = 1 ;

      if (ack_ok ( svar->N_access_point.DATA_RSP.ndata ) )
      {
        FDT5  = 0;
        { /* datax5 */
          lvar->datax_body.send_msg = retrieve ( lvar->datax_body.send_buf ) ;
          remove ( &( lvar->datax_body.send_buf ) , lvar->datax_body.send_msg );
          inc_send_seq ( );
          lvar->datax_body.STATE = 1  /* ESTAB */ ;
          goto trans_end ;
        }
      }
    }
  }
  if ( ( is_set_member ( &(lvar->datax_body.EITHER) , lvar->datax_body.STATE ) ) )
  {
    if ( ( signal->cid == 2 ) && ( signal->sid == 2 ) )
    {
      int FDT6  = 1 ;

      if (svar->N_access_point.DATA_RSP.ndata .pid == 0  /* DATA */)
      {
        FDT6  = 0;
        { /* datax6 */
          lvar->datax_body.recv_msg .msgdat
              = svar->N_access_point.DATA_RSP.ndata .dat ;
          lvar->datax_body.recv_msg .msgseq
              = svar->N_access_point.DATA_RSP.ndata .seq ;
          format_ack ( lvar->datax_body.recv_msg , &( lvar->datax_body.buf ) );
          {
          FDTSVAR *newsvar = (FDTSVAR *) malloc(sizeof(FDTSVAR));

          newsvar->N_access_point.DATA_REQ.ndata = lvar->datax_body.buf ;
          FDTSCBsignal ( process, 2 , 1 , newsvar);
          }
          if ( svar->N_access_point.DATA_RSP.ndata .seq ==
               lvar->datax_body.recv_seq )
          {
```

```
            store ( &( lvar->datax_body.recv_buf ) , lvar->datax_body.recv_msg );
            inc_recv_seq ( );
          }
          goto trans_end ;
        }
      }
    }
  }
  if (signal->cid == 0)
    FDTSCBdispose (process, signal);
  return;

  trans_end :
  FDTSCBdispose (process, signal);
  if (process->spont)
    FDTSCBspont (process);
  spont_end :
  process->delay = 0;
}


FDTPCB *FDTabit_body ( parent , cep_id )
FDTPCB *parent;
int cep_id ;
{
  FDTPCB *pcb;
  FDTLVAR *lvar;

  pcb = FDTPCBinit ( parent , 2 , SPONTabit_body ,
                     XPORTabit_body , TRANSabit_body );
  pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));

  lvar->abit_body.cep_id = cep_id ;
  {
    lvar->abit_body.datax_module = FDTdatax_body( pcb , lvar->abit_body.cep_id );
    lvar->abit_body.timer_module = FDTtimer_body( pcb , 30  /* RETRAN_TIME */);
    FDTCCBconnect (lvar->abit_body.datax_module , 3 , 1 ,
                   lvar->abit_body.timer_module , 1 , 1 );
    FDTCCBattach  (pcb, 1 , 1 ,
                   lvar->abit_body.datax_module , 1 , 1 );
    FDTCCBattach  (pcb, 2 , 1 ,
                   lvar->abit_body.datax_module , 2 , 1 );
    goto trans_end ;
  }
  trans_end :
  if (pcb->spont)
```

```
    FDTSCBspont (pcb);
  return (pcb);
}


FDTPCB *FDTSPECIFICATION ( parent )
FDTPCB *parent;
{
  FDTPCB *pcb;
  FDTLVAR *lvar;

  pcb = FDTPCBinit ( parent , 0 , SPONTSPECIFICATION ,
                     XPORTSPECIFICATION , TRANSSPECIFICATION );
  pcb->lvar = (int *) (lvar = (FDTLVAR *) malloc(sizeof(FDTLVAR)));
  {
    lvar->SPECIFICATION.network_module = FDTnetwork_body( pcb );
    {
      int cep ;
      for ( cep = 1  ; cep <= 2  ; cep++ )
      {
        lvar->SPECIFICATION.user_module [ cep - 1 ] = FDTuser_body( pcb , cep );
        lvar->SPECIFICATION.abit_module [ cep - 1 ] = FDTabit_body( pcb , cep );
        FDTCCBconnect (lvar->SPECIFICATION.user_module [ cep - 1 ] , 1 , 1 ,
                       lvar->SPECIFICATION.abit_module [ cep - 1 ] , 1 , 1 );
        FDTCCBconnect (lvar->SPECIFICATION.abit_module [ cep - 1 ] , 2 , 1 ,
                       lvar->SPECIFICATION.network_module , 1 + cep - 1 , 0 );
      }
    }
    goto trans_end ;
  }
  trans_end :
  if (pcb->spont)
    FDTSCBspont (pcb);
  return (pcb);
}
```

Appendix C

Process Control Block Support Routines

```
/*
 *  Creates and initializes a new process control block (PCB)
 *  Places the PCB at the head of the sibling list
 *  If process contains transitions, inserts PCB into the scheduler's lists
 *  Returns the new PCB
 */

FDTPCB *FDTPCBinit (parent, ipnum, spont, export, transition)
FDTPCB   *parent;
int      ipnum;
int      spont;
int      export;
int      (*transition)();
  {
    FDTPCB   *newpcb = (FDTPCB *) malloc(sizeof(FDTPCB));

    /***** Links new process control block to the module hierarchy *****/

    newpcb->pid    = Pid++;
    newpcb->parent = parent;
    if (parent != NULL)
      {
        newpcb->sib  = parent->ref;
        parent->ref  = newpcb;
        newpcb->prio = parent->prio + 1;
      }
    else  /* This is the root module */
      {
        newpcb->sib  = newpcb;
        newpcb->prio = 1;
      }
    newpcb->ref    = NULL;

    /***** Allocates enough interaction points for the module *****/

    newpcb->ipnum = ipnum;
    if (ipnum > 0)
        newpcb->chan  = FDTCCBinit (newpcb->ipnum);
    else
        newpcb->chan  = NULL;

    /***** Initializes other module state variables *****/

    newpcb->ipnext = 0;
    newpcb->sigcnt = 0;
    newpcb->delay  = 0;
    newpcb->spont  = spont;
```

```
    newpcb->export = export;
    newpcb->trans  = transition;

    return (newpcb);
  }


/*
 *  Releases the specified process control block and all its descendents
 */

FDTPCBterm (pcb)
FDTPCB   *pcb;
  {
    FDTPCB *p;

    if (pcb == NULL)
       return;

    if ((p = pcb->parent) == NULL)
       FDTLIBerror ("Root module attempting to kill itself\n");

    if (p->ref == pcb)
      {
        p->ref = pcb->sib;
      }
    else
      {
        p = p->ref;
        while ((p != NULL) && (p->sib != pcb))
            p = p->sib;

        if (p == NULL)
            FDTLIBerror ("Error in link to parent\n");

        p->sib = pcb->sib;
      }

    /***** Terminates all children recursively and then deallocates itself *****/

    while (pcb->ref != NULL)
        FDTPCBterm (pcb->ref);

    if (pcb->chan != NULL)
        FDTCCBterm (pcb);
    free (pcb);
  }
```

Appendix D

Channel Control Block Support Routines

```
/*
 *  Creates and initializes a new channel control block
 */

FDTCCB *FDTCCBinit (size)
int   size;
  {
    FDTCCB   *i, *ccb = (FDTCCB *) calloc(size+1, sizeof(FDTCCB));

    for (i=ccb; i<ccb+size+1; i++)
      {
        i->head     = i->tail                    = NULL;
        i->targeta  = i->targetc  = i->targete   = NULL;
        i->channela = i->channelc = i->channele  = 0;
        i->qdispl   = COMMON;
      }

    return (ccb);
  }


/*
 *  Removes a channel list from a process control block
 */

FDTCCB *FDTCCBterm (process)
FDTPCB *process;
  {
    FDTCCB   *ccb1, *ccb2;
    int      i;

    for (i=1; i<process->ipnum+1; i++)
      {
        ccb1 = process->chan + i;
        if (ccb1->targetc != NULL)
          {
            ccb2 = ccb1->targetc->chan + ccb1->channelc;
            if ((ccb2->targetc == process) && (ccb2->channelc == i))
                FDTCCBdisconn (process, i);
            else
                FDTCCBdetach2 (process, i);
          }
      }
    free (process->chan);
  }
```

```
/*
 *  Implements the Estelle connect statement
 */

FDTCCBconnect (process1, channel1, qdispl1, process2, channel2, qdispl2)
FDTPCB       *process1, *process2;
int          channel1, channel2;
queue_kind   qdispl1, qdispl2;
  {
    FDTCCB   *ccb1, *ccb2;

    /***** Locates channel control blocks *****/

    ccb1 = process1->chan + channel1;
    ccb2 = process2->chan + channel2;

    ccb1->qdispl = qdispl1;
    ccb2->qdispl = qdispl2;

    /***** Tests for prior connections *****/

    if ((ccb1->targetc != NULL) || (ccb2->targetc != NULL))
        FDTLIBerror ("Channel is already connected");

    /***** Makes formal connections *****/

    ccb1->targetc  = process2;   ccb1->channelc = channel2;
    ccb2->targetc  = process1;   ccb2->channelc = channel1;

    /***** Finds actual target channel control blocks *****/

    if (ccb1->targeta != NULL)
      {
        process1 = ccb1->targete;
        channel1 = ccb1->channele;
        ccb1->targete  = NULL;
        ccb1->channele = 0;
        ccb1 = process1->chan + channel1;
      }

    if (ccb2->targeta != NULL)
      {
        process2 = ccb2->targete;
        channel2 = ccb2->channele;
        ccb2->targete  = NULL;
        ccb2->channele = 0;
        ccb2 = process2->chan + channel2;
```

```
    }

  /***** Makes actual connection *****/

  ccb1->targete  = process2;   ccb1->channele = channel2;
  ccb2->targete  = process1;   ccb2->channele = channel1;
}


/*
 *  Implements the Estelle ATTACH statement
 */

FDTCCBattach (process1, channel1, qdispl1, process2, channel2, qdispl2)
FDTPCB       *process1, *process2;
int          channel1, channel2;
queue_kind   qdispl1, qdispl2;
{
  FDTCCB   *ccb1, *ccb2;

  /***** Locates channel control blocks *****/

  ccb1 = process1->chan + channel1;
  ccb2 = process2->chan + channel2;

  ccb1->qdispl = qdispl1;
  ccb2->qdispl = qdispl2;

  /***** Tests for prior connections *****/

  if ((ccb1->targeta != NULL) || (ccb2->targetc != NULL))
      FDTLIBerror ("Channel is already attached");

  /***** Makes formal attachments *****/

  ccb1->targeta = process2;   ccb1->channela = channel2;  /* attach down */
  ccb2->targetc = process1;   ccb2->channelc = channel1;  /* connect up  */

  /***** Finds actual target channels *****/

  if (ccb1->targetc != NULL)
    {
      process1 = ccb1->targete;
      channel1 = ccb1->channele;
      ccb1->targete  = NULL;
      ccb1->channele = 0;
      ccb1 = process1->chan + channel1;
```

```
      }

    if (ccb2->targeta != NULL)
      {
        process2 = ccb2->targete;
        channel2 = ccb2->channele;
        ccb2->targete  = NULL;
        ccb2->channele = 0;
        ccb2 = process2->chan + channel2;
      }

    /***** Makes actual attachment *****/

    ccb1->targete = process2;   ccb1->channele = channel2;
    ccb2->targete = process1;   ccb2->channele = channel1;
  }


/*
 *  Implements the Estelle DISCONNECT statement
 */

FDTCCBdisconn (process1c, channel1c)
FDTPCB   *process1c;
int      channel1c;
  {
    FDTCCB   *ccb1c, *ccb2c;
    FDTCCB   *ccb1e, *ccb2e;
    FDTPCB   *process1e, *process2c, *process2e;
    int      channel1e, channel2c, channel2e;

    if (channel1c == 0)
      {
        int   i;

        for (i=1; i<process1c->ipnum; i++)
            FDTCCBdisconn (process1c, i);
      }
    else
      {
        /***** Locates actual channel control blocks *****/

        ccb1c = process1c->chan + channel1c;

        process2c = ccb1c->targetc;
        channel2c = ccb1c->channelc;
        if (process2c == NULL)
```

```
    FDTLIBerror ("Attempt to disconnect unbound channel");
ccb2c = process2c->chan + channel2c;

/***** Tests for prior connections *****/

if ((ccb2c->targetc != process1c) || (ccb2c->channelc != channel1c))
    FDTLIBerror ("Attempt to disconnect attached channel");

/***** Locates effective channel control blocks *****/

ccb1e = ccb1c;
while (ccb1e->targete == NULL)
  {
    process1e = ccb1e->targeta;
    channel1e = ccb1e->channela;
    ccb1e = process1e->chan + channel1e;
  }

ccb2e = ccb2c;
while (ccb2e->targete == NULL)
  {
    process2e = ccb2e->targeta;
    channel2e = ccb2e->channela;
    ccb2e = process2e->chan + channel2e;
  }

/***** Disconnects actual channels *****/

ccb1c->targetc = NULL;   ccb1c->channelc = 0;
ccb2c->targetc = NULL;   ccb2c->channelc = 0;

/***** Rebinds effective channels, if necessary *****/

if (ccb1c != ccb1e)
  {
    ccb1c->targete = process1e;  ccb1c->channele = channel1e;
    ccb1e->targete = process1c;  ccb1e->channele = channel1c;
  }
else
  {
    ccb1e->targete = NULL;   ccb1e->channele = 0;
  }

if (ccb2c != ccb2e)
  {
    ccb2c->targete = process2e;  ccb2c->channele = channel2e;
    ccb2e->targete = process2c;  ccb2e->channele = channel2c;
```

```
        }
      else
        {
          ccb2e->targete = NULL;   ccb2e->channele = 0;
        }
    }
  }


/*
 *  Implements the Estelle DETACH statement for an external interaction point
 */

FDTCCBdetach1 (process1a, channel1a)
FDTPCB   *process1a;
int       channel1a;
  {
    FDTCCB   *ccb1a, *ccb2a;    /* Formal attachments */
    FDTCCB   *ccb1e, *ccb2e;    /* Actual attachments */
    FDTPCB   *process1e, *process2a, *process2e;
    int      channel1e, channel2a, channel2e;

    /***** Locates channel control blocks for actual attachments *****/

    ccb1a = process1a->chan + channel1a;

    process2a = ccb1a->targeta;
    channel2a = ccb1a->channela;
    if (process2a == NULL)
        FDTLIBerror ("Attempt to detach unbound channel");
    ccb2a = process2a->chan + channel2a;

    /***** Tests for prior attachments *****/

    if ((ccb2a->targetc != process1a) || (ccb2a->channelc != channel1a))
        FDTLIBerror ("Attempt to detach improperly attached channel");

    /***** Locates channel control blocks for effective attachments *****/

    ccb2e = ccb2a;
    while (ccb2e->targete == NULL)
      {
        process2e = ccb2e->targeta;
        channel2e = ccb2e->channela;
        ccb2e = process2e->chan + channel2e;
      }
    process1e = ccb2e->targete;
```

```
    channel1e = ccb2e->channele;
    ccb1e = process1e->chan + channel1e;

    /***** Dettaches actual channels *****/

    ccb1a->targeta = NULL;   ccb1a->channela = 0;
    ccb2a->targetc = NULL;   ccb2a->channelc = 0;

    /***** Rebinds effective channels, if necessary *****/

    if (ccb1a != ccb1e)
      {
        ccb1a->targete = process1e;   ccb1a->channele = channel1e;
        ccb1e->targete = process1a;   ccb1e->channele = channel1a;
      }
    else
      {
        ccb1e->targete = NULL;   ccb1e->channele = 0;
      }

    if (ccb2a != ccb2e)
      {
        ccb2a->targete = process2e;   ccb2a->channele = channel2e;
        ccb2e->targete = process2a;   ccb2e->channele = channel2a;
      }
    else
      {
        ccb2e->targete = NULL;   ccb2e->channele = 0;
      }
  }


/*
 *  Implements the Estelle DETACH statement for a child's external interaction point
 */

FDTCCBdetach2 (process2a, channel2a)
FDTPCB   *process2a;
int       channel2a;
  {
    FDTCCB   *ccb1a, *ccb2a;    /* Formal attachments */
    FDTCCB   *ccb1e, *ccb2e;    /* Actual attachments */
    FDTPCB   *process1a, *process1e, *process2e;
    int      channel1a, channel1e, channel2e;

    /***** Locates channel control blocks for actual attachments *****/
```

```
ccb2a = process2a->chan + channel2a;

process1a = ccb2a->targeta;
channel1a = ccb2a->channela;
if (process1a == NULL)
    FDTLIBerror ("Attempt to detach unbound channel");
ccb1a = process1a->chan + channel1a;

/***** Tests for prior attachments *****/

if ((ccb1a->targetc != process2a) || (ccb1a->channelc != channel2a))
    FDTLIBerror ("Attempt to detach improperly attached channel");

/***** Locates channel control blocks for effective attachments *****/

ccb2e = ccb2a;
while (ccb2e->targete == NULL)
  {
    process2e = ccb2e->targeta;
    channel2e = ccb2e->channela;
    ccb2e = process2e->chan + channel2e;
  }
process1e = ccb2e->targete;
channel1e = ccb2e->channele;
ccb1e = process1e->chan + channel1e;

/***** Dettaches actual channels *****/

ccb1a->targeta = NULL;   ccb1a->channela = 0;
ccb2a->targetc = NULL;   ccb2a->channelc = 0;

/***** Rebinds effective channels, if necessary *****/

if (ccb1a != ccb1e)
  {
    ccb1a->targete = process1e;   ccb1a->channele = channel1e;
    ccb1e->targete = process1a;   ccb1e->channele = channel1a;
  }
else
  {
    ccb1e->targete = NULL;   ccb1e->channele = 0;
  }

if (ccb2a != ccb2e)
  {
    ccb2a->targete = process2e;   ccb2a->channele = channel2e;
    ccb2e->targete = process2a;   ccb2e->channele = channel2a;
```

```
    }
  else
    {
      ccb2e->targete = NULL;    ccb2e->channele = 0;
    }
}
```

Appendix E

Signal Control Block Support Routines

```
/*
 *  Creates a new signal control block on the target of the specified channel
 */

FDTSCBsignal (process, cid, sid, svar)
FDTPCB *process;
int    cid;
int    sid;
int    *svar;
  {
    FDTCCB   *ccb1, *ccb2;
    FDTSCB   *scb;
    FDTPCB   *target;

  /***** Determines the location of the target channel *****/
    ccb1   = process->chan + cid;
    target = ccb1->targete;
    ccb2   = target->chan + ccb1->channele;
    if (ccb2->qdispl == COMMON)
        ccb2 = target->chan;

  /***** Constructs an outgoing signal control block *****/
    scb = (FDTSCB *) malloc(sizeof(FDTSCB));
    scb->cid  = ccb1->channele;
    scb->sid  = sid;
    scb->svar = svar;

  /***** Queues the signal control block to the tail of the target channel *****/
    scb->next = NULL;
    if (ccb2->tail == NULL)
        ccb2->head = scb;
    else
        ccb2->tail->next = scb;
    ccb2->tail = scb;

  /***** Increments the pending signal counter *****/
    (target->sigcnt)++;
  }


/*
 *  Creates a spontaneous signal at the common channel for the process
 *  if there are no pending signals for the process
 */

FDTSCBspont (process)
FDTPCB *process;
```

```
  {
    FDTCCB   *ccb;
    FDTSCB   *scb;

  /***** Exits if there are pending signals *****/
    if (process->sigcnt > 0)
        return;

  /***** Constructs a spontaneous signal control block *****/
    scb = (FDTSCB *) malloc(sizeof(FDTSCB));
    scb->cid  = 0;
    scb->sid  = 0;
    scb->svar = NULL;

  /***** Queues the signal control block at the common channel *****/
    ccb = process->chan;
    scb->next = ccb->head;
    ccb->head = scb;
    if (ccb->tail == NULL)
        ccb->tail = scb;

  /***** Increments the pending signal counter *****/
    (process->sigcnt)++;
  }


/*
 *  Removes a signal control block from a channel
 */

FDTSCBdispose (process, signal)
FDTPCB   *process;
FDTSCB   *signal;
  {
    FDTCCB   *ccb;
    FDTSCB   *scb;

  /***** Determines the location of the signal queue *****/
    ccb = process->chan + signal->cid;
    if (ccb->qdispl == COMMON)
        ccb = process->chan;

  /***** Removes the signal control block at the head of the queue *****/
    scb = ccb->head;
    ccb->head = scb->next;
    if (ccb->head == NULL)
        ccb->tail = NULL;
```

```
    if (scb->svar != NULL)
        free (scb->svar);
    free (scb);

  /***** Decrements the pending signal counter *****/
    (process->sigcnt)--;
  }


/*
 *  Searches for a pending signal for the process
 */

FDTSCB *FDTSCBpending (process)
FDTPCB *process;
  {
    FDTCCB   *ccb;

    if (process->sigcnt == 0)
        return (NULL);

    ccb = process->chan + process->ipnext;
    while (ccb <= process->chan + process->ipnum)
        if (ccb->head != NULL)
          {
            process->ipnext = ccb - process->chan + 1;
            return (ccb->head);
          }
        else
          {
            ccb++;
          }
    ccb = process->chan;
    while (ccb < process->chan + process->ipnext)
        if (ccb->head != NULL)
          {
            process->ipnext = ccb - process->chan + 1;
            return (ccb->head);
          }
        else
          {
            ccb++;
          }

    return (NULL);
  }
```