SEMI-AUTOMATIC IMPLEMENTATION OF PROTOCOLS USING AN ESTELLE-C COMPILER

by

S.T. Vuong, A.C. Lau and R.I. Chan

Technical Report 87-6

March, 1987

Semi-Automatic Implementation of Protocols using an Estelle-C Compiler

Son T. Vuong, Allen C. Lau and R. Isaac Chan

Distributed System Research Group
Department of Computer Science
University of British Columbia
Vancouver, B.C.

February 24, 1987

Abstract

In this paper, we present the basic ideas underlying an *Estelle-C* compiler, which accepts an *Estelle* protocol specification and produces a protocol implementation in C. We discuss our experience gained from using the semi-automatic approach to implement the ISO class 2 transport protocol. A manual implementation of the protocol is performed and compared with the semi-automatic implementation. We find the semi-automatic approach to protocol implementation offers several advantages over the conventional manual one, including correctness and modularity in protocol implementation code, conformance to the specification and reduction in implementation time. Finally, we present our ongoing development of a new *Estelle-C* compiler.

1 Introduction

The development of formal description techniques (FDTs) [Boch80] for specifying protocols has proceeded hand-in-hand with research and development of formal techniques and tools for the design, validation, implementation and testing of communication protocols. Based on standard FDTs such as Estelle[Estelle85,Estelle86] and LOTOS[Lotos84,Brink85], a number of compilers and interpreters, [Ansart83], [Blum82], [Bria86], [Ford85], [Gerber83,Boch84] and [Hans84,Hans85], have been developed to automate the process of protocol implementation.

This new approach to protocol implementation has proven to be superior to traditional methods. The compiler can generate automatically, in a well-constructed way, a large portion

of the protocol implementation code in a standard target language. The system-dependent features of the protocol can be encapsulated into a few routines. The implementation is, therefore, easy to produce and to maintain; it tends to conform to the specification and is highly portable.

We have developed an Estelle-C compiler to allow automatic generation of protocol implementations C [Ford85,Lau86,Vuong86,Vuong87]. To our knowledge, our Estell-C compiler is the only complete compiler which accepts Estelle specifications and produces code in C. Other compilers either accept non-Estelle specifications [Blum82] or produce code in Pascal [Ansart83,Gerber83,Hans84]. Our compiler currently runs on a VAX 11/750 and various SUN2 and SUN3 Workstations under UNIX 4.2BSD. In order to verify the usefulness of this tool, we have specified the ISO class 2 transport protocol [CCITT85,ISO82b] in Estelle and tested the generated code on a VAX and a number of SUNs in an Ethernet environment.

In this paper, we will discuss the Estelle-C compiler and our experience with semi-automatic implementation of protocols using this compiler. After a brief overview of Estelle, we will present the basic ideas underlying the Estelle-C compiler. The discussion will then proceed to the design and implementation of the ISO class 2 transport protocol using both the semi-automatic and the manual approaches and an evaluation of the two implementations. Subsequently, the ongoing development of our compiler will be presented, followed by a number of concluding remarks.

2 The Estelle Language

Estelle (*an Extended State Transition Language*) is a formal description technique of communication protocols and services developed within the International Standard Organization (ISO) by the TC 97/SC 16/WG 1 Subgroup B [ISO84,Estelle85]. The technique is based on an extended finite state transition model, the Pascal programming language and some Ada modular constructs. The framework of an Estelle specification is a set of co-operating entities, each described as a **module**, interacting with each other by exchanging information through **channels**. The actual behaviour of a module is specified either as an integrated behaviour of a set of interacting submodules or at the innermost level, as an extended finite state machine. An

example of an Estelle specification of the ISO Class 2 transport protocol is given in Appendix A. Basically, a **channel** is a bidirectional pipe which transmits information between two connected modules. A **channel-type** definition specifies a set of interaction primitives which are grouped under two different roles, e.g. user and provider. Information is transmitted between module instances via the parameters in the interaction primitives.

A **module** is the basic component of an Estelle specification and it represents an entity in the specification. A **module-type** definition consists of a list of interaction points at which the module interacts with its environment. The interaction points are abstract interfaces used by the module to interact with other connected modules. For each interaction point, a role is specified for its associated channel-type. An interaction is then identified by the name of the interaction point at which it occurs and the name of the interaction primitive used. For a given module-type, one or more module instances can be created.

In Estelle, the actual behaviour of a module is specified either indirectly as a **refinement** or directly as a **process**. If a module is not a completely self-contained entity, i.e. a process, it is decomposed into a set of co-operating submodules, each of which may be further decomposed. The behaviour of the module is the integrated behaviour of the submodules and, thus, it is called a refinement. An Estelle **refinement** specification includes definitions of internal channel-types, module-types, and specifications of the corresponding processes and refinements. After the definition of the internal structures, module instances are created and connected accordingly. If necessary, interaction points of internal module instances may be replaced by those in their parent module instance. An Estelle **process** definition specifies a queuing discipline (**queued** or **rendezvous**) associated with each interaction point, the initial conditions and all possible transitions for the corresponding extended finite state machine.

The general specification of a transition is given in Figure 1, with the following semantics. WHEN an interaction is received, PROVIDED that a condition is satisfied, a transition will be taken to lead the process, FROM the current major state TO a new major state, through an action. The associated action of a transition is specified in terms of Pascal statements, and may include the initiation of OUTput interactions with its peer modules.

```
TRANS
    WHEN <input interaction>
    PROVIDED <enabling condition>
    FROM <from_state>
    TO <to_state>
    BEGIN
            <action>
    END
```

Figure 1: Specification of a transition

Transitions are classified into **input** and **spontaneous** transitions, depending on the presence or absence of an input interaction specification (i.e. the WHEN clause) respectively. A spontaneous transition may be executed regardless of any input interaction. It should be mentioned that the Estelle state machine is **non-deterministic**. At any given time, several different transitions may be executable. But the particular transition chosen for execution is not specified by the specification.

3 The Estelle-C Compiler

In order to experiment with the semi-automatic approach to protocol implementation, a Estelle-C compiler was developed at the University of British Columbia [Ford85] to support Estelle as defined in 1984 [ISO84]. The compiler reads an Estelle protocol specification as input and generates C code. The generated C program is subsequently made complete by the incorporation of additional system-dependent and pre-written run-time support routines. It was later substantially enhanced to support commonly used complex data structures, such as pointer and variant record, and to produce better-organized C code [Lau86]. The compiler is currently being rewritten to support the latest Estelle language specification [Estelle85].

The current Estelle-C compiler is implemented in C. The scanner and the parser for the compiler are generated by the UNIX standard utilities LEX [Lesk75] and YACC [John75], respectively. Since both LEX and YACC generate portable C code, the compiler is portable to

any system supporting C. Error handling, table management and code generation are embedded in the YACC grammar input file. Currently, the compiler does not optimize the generated C code. It completes the translation in a single pass through the source specification.

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

Since Estelle is a Pascal-based language, some restrictions are imposed due to the inherent differences between Pascal and C. The compiler allows neither global variables and nested subroutines, nor supports the WITH statement. In addition, only a subset of the Pascal I/O statements and pointer assignments are fully supported. Due to the additional Estelle scoping rules introduced by the enabling conditions for a transition-type, and the additional variables used by the run-time support routines, additional restrictions are also imposed. For example, parameter names of input interactions are not allowed to be used for local variables in any module body. Furthermore, function and procedure identifiers used in an Estelle specification should be different from the reserved identifiers generated by the Estelle-C compiler and those of the run-time support routines. These limitations are not found to be too restrictive from our experience in using the Estelle-C compiler.

In automatic implementation of protocols, a generic structure and organization for the implementation must be adopted. The implementation strategy used by our Estelle-C compiler is similar to the one used by Gerber for his Estelle-to-Pascal compiler [Gerber83]. Record structures are used to represent module instances, interaction points, and interactions among module instances. A set of pre-written generic routines is used to allocate, initialize, and link the structures according to an Estelle specification. The pre-written routines also dispatch an output interaction to a recipient module, select the next available interaction, and make non-deterministic scheduling choices. Since different systems have different global environments and

scheduling schemes, two special routines, **system_init** and **schedule**, have to be tailored according to each specification. Figure 2 depicts the procedure of semi-automatic implementation.

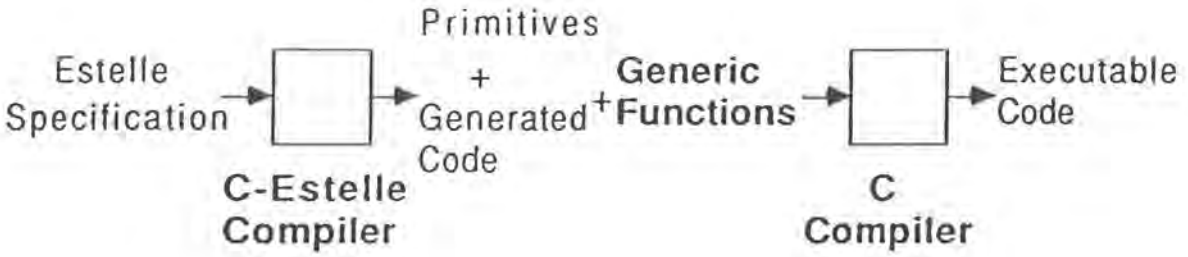

Figure 2: Procedure of Semi-Automatic Implementation

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

In Figure 3, the data structure **signal_block** represents an interaction (i.e. a signal or

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

Figure 3: Data Structure of an Interaction

message) and is comprised of three attributes, *signal_id, next* and *lvars*. For convenience, the interaction primitives specified in channel-type definitions are numbered and *signal_id* is used

to identify the primitives. The attribute *next* links data structures to implement the queuing of incoming interactions at an interaction point. The values of the parameters of an interaction are stored as a single attribute *lvars* in the data structure.

Representing a module instance, the data structure **process_block** (Figure 4) consists of

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

Figure 4: Data Structure of a Module Instance

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

Representing an interaction point, data structure **channel_block** (Figure 5) contains the following attributes: *target_proc,* and *target_channel* are entry points to data structures which represent peer module instance and its corresponding interaction point; *signal_list* points to a list of incoming interactions; *queued* is a boolean flag that indicates the queuing discipline (queued or rendezvous) of the interaction point; *c_id* identifies the interaction point and additional *index_num* is used in case of multiplexing channel; and, finally, *next* links all interaction points in a module-type.

In a given global system state, a number of different input interactions may be pending

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

Figure 5: Data Structure of an Interaction Point

for several different module instances. Within a given module instance and for a given input interaction, several different input transitions may be enabled. In addition, several spontaneous transitions may be executable irrespective of input interactions. The selection of the next available transition to be performed is made by a global scheduler. The schedule algorithm is not a part of the Estelle specification but is a part of the run-time support for the implementation. For simplicity, the first enabled transition as defined in the specification is selected for execution in our current implementation. Thus, for each cycle, the global scheduler selects a module instance with a pending interaction, and then selects the next input transition based on the interaction, and tries all possible spontaneous transitions.

For each implementation, the protocol implementors will have to manually look after the system-dependent portion of the implementation, i.e. the interface between the specified protocol machine and its working environment. For example, interactions with the operating system usually cause an undesirable blocking of the protocol machine. Solutions to such blocking problems vary largely for different machines and different operating systems. With the working environment and the operating system known apriori, the interfaces with the specified system can be well defined to simplify the system interactions. For instance, in our implementations of the transport protocol, UNIX 4.2 socket primitive **select** is used to preview the socket so that the blocking is avoided when reading a socket. For implementations of lower-level protocols such as X.25 and HDLC, we will need to interface with the corresponding lower-level

software/hardware such as the communication controller chip and its driver. Thus, output to the environment can be implemented by invoking a set of system-dependent routines, and input from the environment can be implemented by including spontaneous transitions which invoke the same set of routines. The global scheduler is fully aware of when and which spontaneous transition is to be executed.

4 Implementation Example - The ISO Transport Protocol

In order to evaluate the usefulness of the Estelle-C compiler, a complex protocol, the ISO class 2 transport protocol, has been implemented semi-automatically by using the Estelle-C compiler and subsequently re-implemented manually. Both protocol implementations run on a VAX 11/750 as well as several SUN Workstations under the UNIX 4.2BSD operating system. In this section, we present our design of the protocol implementation and discuss our experience gained from using these two different approaches. A state diagram for the ISO class 2 transport protocol is given in Figure 6. Details of the protocol can be found in the documents [CCITT85] and [ISO82a], and details of the corresponding Estelle specification are described in [ISO84] and [Lau86]. The ISO transport protocol is a connection-oriented, end-to-end protocol providing a reliable and efficient mechanism for the exchange of data between processes in different computer systems. The class 2 protocol assumes a highly reliable network service, such as X.25, and has the ability to multiplex multiple Transport connections onto a single network connection. It uses a credit allocation scheme to provide explicit flow control but does not provide mechanism for error recovery to the higher layers.

4.1 Implementation Design

The overall structure of an Estelle specification of the ISO class 2 transport entity is given in Figure 7. There are four different module types: **TS_user, ATP, System** and **RS**. The module instances of these four module types are combined to form a transport entity.

A **TS_user** module is a sub-layer which converts a Transport Service user request into a well-defined transport service primitive. A user task in the working environment can be

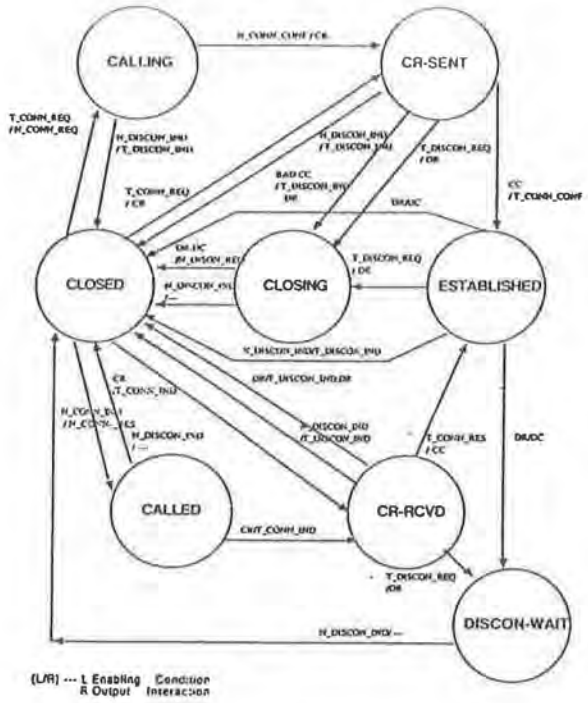

Figure 6: Transport Protocol State Diagram

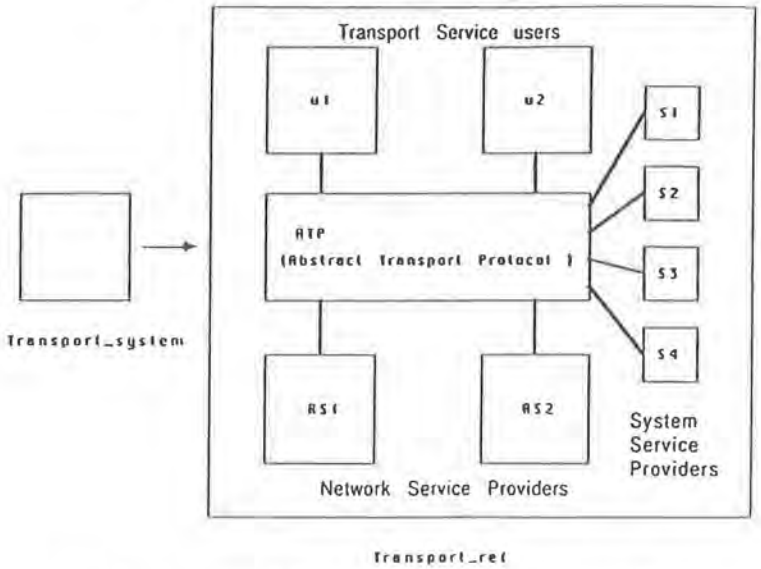

Figure 7: A Typical Refinement of a Transport System

bound to one or more **TS_user** modules, and thus to one or more transport connections. An **ATP** module is an Abstract Transport Protocol entity that establishes transport connections, transfers data, and releases connections. A **System** module simulates a system timer for an incoming network connection and for the flow control of a transport connection. Finally, a **RS** module converts the network service primitives into system calls. It also sets flags and stores data whenever an incoming network event occurs.

Since there are many unspecified properties in the protocol specification, these properties, either **implementation-defined** or **implementation-dependent**, have to be designed for each specific implementation so that the resulting implementation will best fit the working environment.

Implementation-defined properties are left unspecified and their definitions can vary from one implementation to another. For example, in the **TS_primitives** channel definition (shown in Appendix A), data type ADDR_TYPE is implementation-defined. Type ADDR_TYPE represents transport address which may be defined differently depending on the implementors. Similarly, the buffer management and data exchanged between TS_users and a TS_provider are implementation-defined.

However, some properties are defined in the specification but their implementation is left unspecified. Examples of such properties include the routines which construct the transport protocol data units (TPDUs). The format of each TPDU is specified but the method of TPDU construction is left unspecified.

4.2 Semi-Automatic Implementation

The protocol was first specified in Estelle using the description in the ISO document [CCITT85,ISO82b] and other existing specifications [ISO84,NBS83]. Next, the Estelle specification was translated by the Estelle-C compiler to generate code for a major part of the protocol implementation. Finally, the generated C code was combined with pre-written generic routines and system-dependent routines to form a complete C program.

The generated code can be classified into three types. The first type is the **typedef** and

struct declarations which represent module instance, interaction, type and variable definitions as defined in the specification. These definitions are required by the run-time executives to keep track of the state information in the protocol machines. The second type of code consists of a set of routines which creates, initializes and links data structures according to the specification. The third type is another set of routines which implements the transition processes in the protocol machines.

Initialization routines can be further subdivided into two classes. One class corresponds to the Estelle **process** definitions. These routines create and initialize the **process_block** data structure. The other class of routines corresponds to the Estelle **refinement** definitions. These routines create the sub-module instances and link the instances according to the Estelle CONNECT and REPLACE definitions. Both classes of initialization routines make use the set of pre-written generic routines to perform the creation, initialization, and integration of the various system components.

Transition functions are implemented as a series of conditional expressions and statement blocks. For each transition specified, a conditional expression is used to evaluate its enabling conditions and a statement block is used to perform the associated action. The conditional expression handles the transition clauses in the order shown in Figure 1. The WHEN clause is translated into tests for the identity (*signal_id*) of the received interaction and the identities (*c_id* and *index_num*) of the interaction point on which it is being received. Additional tests, which correspond to the PROVIDED clause and/or the FROM clause, may also appear in the conditional expression. The TO clause is translated into a statement within the statement block. At the end of each statement, a **goto dispose** statement passes control to the code which disposes the received **signal** data structure. For a spontaneous transitions, the conditional expressions does not include tests corresponding to the WHEN clause. Unless a priority is set, input transitions are always generated ahead of spontaneous transitions. In this scheme, transitions which are enabled simultaneously are executed in the order in which they are defined in the specification.

Creation and destruction of **signal** structures representing the dynamic interactions between

module instances are implemented completely within the generated transition functions. For each OUT statement a signal structure is created and initialized with the given parameters. The signal structure is then passed to a generic routine **out** together with the information of the interaction point at which the module instance interacts with the peer. If the interaction is a queued type, the signal structure is placed in the reception queue of the peer module instance, then control is returned to the initiating module instance immediately. If the interaction is a rendezvous type, control is passed to the transition function corresponding to the peer module directly. The destruction of the signal structure is handled by the recipient module instance.

4.3 Manual Implementation

Based on the experience of an initial manual implementation and the subsequent semi-automatic implementation, the protocol was finally re-implemented manually. Most principles discussed in Sections 3 and 4.1 were followed. The overall structure is similar to that of the semi-automatic implementation. The transport entity is implemented as a single task in the operating system. It communicates with user tasks and the network service provider through operating system primitives (i.e. system calls).

Instead of using a single data structure **process_block**, three different data structures, TS_MACHINE, TP_MACHINE and NP_MACHINE, are designed to effectively and economically store the state information of a transport service user, a transport connection and a network service provider respectively.

The interactions between the transport entity task and the working environment, the user tasks and the network service provider, are based on the inter-process communication primitives provided by the operating system, i.e. UNIX 4.2BSD socket primitives. Spontaneous transitions initiated by the working environment are handled in a manner similar to that in the semi-automatic implementation. Whenever an external event occurs, the corresponding module instance is selected and a proper spontaneous transition is activated. A series of input transitions, initiated after this spontaneous transition, is then executed until all module instances are in such a state that no more transition is possible. Therefore, the global scheduler can simply

be implemented as a loop which performs the processing for the incoming external events one after the other.

4.4 Results

For a comparison of the two implementation approaches, the sizes of the different parts of the resulting implementations are shown in Table 1. Both implementations use the same INET

PART OF PROGRAM	Number of Functions and Macros (A)	(B)	Number of Source Lines (A)	(B)	Program size (in bytes) (A)	(B)
INET PRIMITIVES	9		509		10969	
TSP PRIMITIVES	12		741		17073	
ESTELLE SPECIFICATION	—	20	—	1910	—	46351
GENERATED CODE	76	20	3420	1447	78621	91421
RUN-TIME SUPPORTING ROUTINES		16		770		21054
PRIMITIVE ROUTINES		82		3049		71340

(A) --- Manual Implementation

(B) --- Semi-Automatic Implementation

Table 1: Sizes of Different Parts of Implementations

primitives to interact with the network service provider. INET primitives provide an uniform access scheme which can be easily modified to suit different network service access schemes in different systems. This network service provider is usually a part of the operating system. Similarly, TSP primitives are used for the interactions between transport service user tasks and the transport entity task.

Both implementations spent a large amount of code in TPDU encoding/decoding and buffer

management. The encoding/decoding of TPDUs are almost identical in both implementations. Since all codes are implemented intermixed within each other in the manual implementation, they are not classified into separate entries in Table 1.

Forty-two additional routines are used in the semi-automatic implementation. Sixteen of which are pre-written run-time support routines and the rest are specially designed for the global scheduler to activate the specific modules.

Due to the inherent weakness in the old Estelle language[ISO84], the number of **process_block** representing the module instances must be statically allocated. The number of transport service users and network connections must also be pre-defined in the specification. The scheme used by the Estelle-C compiler is to generate code to allocate the **process_block** structures in the global initialization phase. The pre-defined number of transport service user tasks must also be created during the initialization stage so that they can be connected to the **process_block** structures.

The advantage of using the semi-automatic approach is that well-constructed code is generated. Since the code is translated directly from a formal specification, the conformance to the specification is almost guaranteed. As well, the generated code can be designed such that the system dependent properties are isolated within a few routines. Therefore, the protocol implementation can be easy to maintain and can also be easy to make portable.

Although the manual implementation is based on the same specification that produced the semi-automatic implementation, no restriction on static allocation is imposed in the global initialization phase. Any number of transport service user tasks can interact with the transport entity. The transport entity does not requires static connections in its initialization phase. Furthermore, any number of network connections can be established during the execution.

The manual implementation can take better advantage of the working environment. For instance, an interaction was implemented as simply a *function call* rather than a primitive on a channel with a complex data structure as in the case of the semi-automatic implementation. As such, we can achieve a significant reduction in the amount of implementation code and in the amount of processing overhead needed in copying data structures for module interactions.

We spent approximately one year to study, analyze and implement the ISO class 2 transport protocol manually without an Estelle specification. The protocol was subsequently specified in Estelle in about a month and implemented semi-automatically in another month with most of the effort devoted to analyzing the generated code and designing its interface to the operating system. It should be noted that most system-dependent routines and some implementation-specific primitives, such as TPDU encoding and decoding routines, were borrowed and adapted from a previous manual implementation. These routines represent about half of the total implementation code, as shown in Table 1. By this time, we have gained a profound experience on protocol (both automatic and manual) implementation and have acquired a good insight into the ISO class 2 transport protocol. Thus, in our last attempt, we have made use of the accumulated experience and the code available to produce, in just a month, the enhanced final "manual" implementation version. This version was discussed in Section 4.3 and was compared with the semi-automatic version in this section.

From our experience, we note that it is a good practice to start with the semi-automatic implementation of protocols because it saves protocol development time. The code produced is well structured, easy to maintain, and guaranteed to conform to the specification. Even if the code is not efficient, we can always attempt a manual implementation subsequently. It is worthwhile to note that in protocol implementations, whether manual or semi-automatic, a lot of time is generally required in developing the interfaces needed to work with the operating system. Additional debugging time is typically required in the manual approach as compared to the semi-automatic approach.

5 Ongoing Development of the Estelle-C Compiler

With the basic ideas underlying the semi-automatic approach to protocol implementation well understood, the motivation for developing a new Estelle-C compiler is to upgrade the compiler to support the latest Estelle language specification [Estelle85]. Because the new Estelle language is substantially different from the old specification [ISO84], a decision is made

to rewrite the compiler from scratch rather than to modify the current compiler. The new compiler will also incorporate features left out in the current compiler as well as streamlining its operations.

The new Estelle-C compiler is implemented in C but it is written without using LEX and YACC. This will allow the new compiler to be further developed in non-UNIX environments. The new compiler uses recursive descent to parse the Estelle specification and it contains the syntax error recovery mechanism described by Brinch Hansen in [Brinch85]. Consequently, this revised compiler provides better syntax error diagnostics as well as complete semantic error diagnostics. In contrast, the current compiler has no mechanism to recover from syntax errors and it will abort after the first syntax error without producing any useful error messages.

The new Estelle language has provision for dynamic reconfiguration of the various entities in the protocol specification. In order to support this feature, transitions are allowed to take place in modules other than those in the lowest level of the module hierarchy. This new feature enables a parent module in an Estelle specification to dynamically create and destroy child modules as well as to dynamically connect and disconnect channels. These features are fully supported by the new Estelle-C compiler.

In our previous versions of the Estelle-C compiler, the Pascal data type SET was left out due to the lack of its support within the C language. In the new compiler, SET is provided as a library package written in C that implement SET as an abstract data type. The compiler translates Estelle SET operations to invoke this package. Incidentally, the new compiler itself makes use of this package to implement its syntax error recovery mechanism.

As part of the redesign of the new 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 runt-time support routines. Furthermore, because the run-time support routines have to be modified for each 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 compiler, the run-time support routines have been rewritten to contain only specification independent details. Consequently, the user of the new compiler is no

longer required to modify any of generated C code and automatic implementation of protocols can be realized.

6 Conclusions

In this paper, we have proposed and discussed a semi-automatic approach to protocol implementation. A protocol specification in Estelle FDT is translated into C code by using the Estelle-C compiler. The generated code is then combined with system-dependent primitives and run-time supporting routines to form a complete C program which implements the protocol in question.

Despite the fact that an initial effort is required to learn the Estelle FDT language and the operation of the Estelle-C compiler, the new approach of protocol implementation has the following benefits:

1. Ease in maintenance due to the modularity and uniformity of the generated code.
2. Guaranteed conformance to the specification due to the direct automatic translation of the protocol specification into C code.
3. High portability because a large amount of code is generated in standard C language and system-dependent properties are easily located and modified.
4. Reduction in development time because a large amount of code is translated automatically from the specification, and the code is well-structured and easy to debug.

From our experience in implementing the ISO class 2 transport protocol semi-automatically, we find the Estelle-C compiler to be a very useful tool. The semi-automatic approach to protocol implementation is, indeed, attractive and practical. We have also used this approach to implement the alternating-bit and the LAPB protocols. We strongly recommend the following general steps to be taken in protocol implementations:

1. Implement the protocol semi-automatically using the Estelle-C compiler.
2. Optimize the generated code from the semi-automatic implementation, if necessary.
3. Re-implement the protocol manually, if necessary (e.g. for efficiency reason).

The Estelle-C compiler is currently available and is portable to any system (including non-UNIX environments) supporting C. Our new compiler will be ready by the summer of 1987. In conclusion, we would like to encourage protocol implementors to use tools similar to the Estelle-C compiler to develop protocol implementations in an effective and systematic manner.

Acknowledgment

The authors are thankful to G.v.Bochmann and J.M. Serre for access to their Estelle-Pascal compiler and many fruitful discussions, and to D. Ford for implementing the early version of the Estelle-C compiler.

References

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

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

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

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

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

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

[Brinch85] Brinch Hansen, P., "Brinch Hansen on Pascal Compilers," Prentice-Hall, 1985.

[Brink85] Brinksma, E., "A Tutotial on LOTOS," *Protocol Specification, Testing and Verification V*, (IFIP/WG 6.1), M. Diaz, eds, North Holland (1985).

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

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

[Estelle86] ISO TC 7/SC 21/WG 1 – FDT, Subgroup B, "Estelle — a formal description technique based on an extended state transition model," Oct. 1986.

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

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

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

[Hans85] Hansson, H., "Automatic Implementation of Formal Descriptions of Communication Protocols," *Protocol Specification, Testing and Verification V*, (IFIP/WG 6.1), M. Diaz, eds, North Holland (1985).

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

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

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

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

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

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

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

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

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

[Vuong87] Vuong, S.T. and Lau, A., "A Semi-Automatic Approach to Protocol Implementation — The ISO Class 2 Transport Protocol as an Example" INFOCOM '87, San Francisco, April 1987.

A Estelle Specification of the ISO Class 2 Transport Protocol

```
MODULE Transport_system;
END Transport_system;

REFINEMENT Transport_ref FOR Transport_system;
(* Constant and Type Definitions *)
.....
(* Channel Definitions *)
CHANNEL TS_primitives ( TS_user, TS_provider ) ;
  BY TS_user :
    T_CONNECT_request ( From_transport_addr : ADDR_TYPE;
                        To_transport_addr   : ADDR_TYPE;
                        Qual_of_service     : QOS_TYPE;
                        TS_user_data        : DATA_TYPE );
    T_CONNECT_response ( Qual_of_service    : QOS_TYPE;
                         TS_user_data       : DATA_TYPE );
    T_DATA_request ( TS_user_data : DATA_TYPE );
    T_XPD_request  ( TS_user_data : DATA_TYPE );
    T_DISCONNECT_request ( TS_user_data : DATA_TYPE );
  BY TS_provider :
    T_CONNECT_indication ( From_transport_addr : ADDR_TYPE;
                           To_transport_addr   : ADDR_TYPE;
                           Qual_of_service     : QOS_TYPE;
                           TS_user_data        : DATA_TYPE );
    T_CONNECT_confirm ( Qual_of_service : QOS_TYPE;
                        TS_user_data    : DATA_TYPE );
    T_DATA_indication ( TS_user_data : DATA_TYPE );
    T_XPD_indication ( TS_user_data : DATA_TYPE );
    T_DISCONNECT_indication ( Reason       : REASON_TYPE;
                              TS_user_data : DATA_TYPE );
END TS_primitives;
.....

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

MODULE TS_user_module;
  TCEP : TS_primitives ( TS_user );
END TS_user_module;

PROCESS TS_user_process( TS_index : integer ) FOR TS_user_module;
.....
END TS_user_process;

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

MODULE System_module;
```

```
  SAP : System_primitives ( S_provider );
END System_module;

PROCESS System_process ( Sys_index : integer ) FOR System_module;
.....
END System_process;

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

(* Abstract Transport Protocol module *)
MODULE ATP_module;
  TCEP : ARRAY[TSAP_TYPE] OF TS_primitives ( TS_provider );
  NSAP : ARRAY[NCEP_TYPE] OF NS_primitives ( NS_user );
  SAPT : ARRAY[TSAP_TYPE] OF System_primitives ( S_user );
  SAPN : ARRAY[NCEP_TYPE] OF System_primitives ( S_user );
END ATP_module;

PROCESS ATP_process FOR ATP_module;
QUEUED  TCEP, NSAP;
(* Variable declarations *)
.....
(* Primitive functions and procedures *)
.....
(* Initialization *)
.....
(* Transitions *)
.....
TRANS
  WHEN TCEP[tid].T_CONNECT_request    (* Transition 3 *)
    PROVIDED ( ( tc[tid].state = CLOSED )
         and ( NOT New_nc_required( nc, From_transport_addr, To_transport_addr ) )
         and ( Choose_class( Qual_of_service ) = CLASS_TWO )
         and ( Size( TS_user_data ) <= MAX_CRCC_SZ  ) )
  BEGIN
    tc[tid].state  := CR_SENT;
    tc[tid].local_addr  := From_transport_addr;
    tc[tid].remote_addr  := To_transport_addr;
    tc[tid].l_suffix  := Get_suffix( From_transport_addr );
    tc[tid].f_suffix  := Get_suffix( To_transport_addr );

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

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

```
    tc[tid].src_ref  := Alloc_ref;
    Construct_CR( data,   tc[tid].rcv_upper_edge,
         tc[tid].src_ref,
         tc[tid].l_suffix,
         tc[tid].f_suffix,
         tc[tid].max_TPDU_size,
         tc[tid].qual_of_service,
         TS_user_data  );
    Concatenate_2_NSDU ( nc[nid], data )
  END;
.....
END ATP_process;

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

MODULE RS_module;
  NSAP : NS_primitives ( NS_provider );
END RS_module;

PROCESS RS_process ( RS_index : integer ) FOR RS_module;
END RS_process;

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

(* Create the module instances *)
U1:  TS_user_module with TS_user_process(1);
U2:  TS_user_module with TS_user_process(2);
ATP: ATP_module with ATP_process;
S1: System_module with System_process(1);
S2: System_module with System_process(2);
S3: System_module with System_process(3);
S4: System_module with System_process(4);
RS1:  RS_module with RS_process(1);
RS2:  RS_module with RS_process(2);

(* Connect the module instances *)
CONNECT
U1.TSAP  TO ATP.TCEP[1];
U2.TSAP TO ATP.TCEP[2];
ATP.NSAP[1] TO RS1.NCEP;
ATP.NSAP[2] TO RS2.NCEP;
ATP.SAPT[1] TO S1.SEP;
ATP.SAPT[2] TO S2.SEP;
ATP.SAPN[1] TO S3.SEP;
ATP.SAPN[2] TO S4.SEP;

END Transport_ref; (* End of the refinement *)
```

B Sample of the Generated Code

```
/* The generated code corresponding to Transition 3 */

  if ((channel != NULL)) {
    p_block->lvars.s_ATP_process.tid =
      (channel->index_num / 1) + 1 ;
    if ((channel->c_id == 1) && (signal->signal_id == 0))
      if ((((p_block->lvars.s_ATP_process.tc
              [p_block->lvars.s_ATP_process.tid-1].state == 0)) &&
(!(New_nc_required(p_block->lvars.s_ATP_process.nc,
    &(signal->lvars.TS_primitives.T_CONNECT_request.
              From_transport_addr),
    &(signal->lvars.TS_primitives.T_CONNECT_request.
              To_transport_addr)))) &&
((Choose_class(
    &(signal->lvars.TS_primitives.T_CONNECT_request.
              Qual_of_service)) == 1)) &&
((Size(
    &(signal->lvars.TS_primitives.T_CONNECT_request.
              TS_user_data)) <= 32))))
{
      {
        p_block->lvars.s_ATP_process.tc
                [p_block->lvars.s_ATP_process.tid-1].
                  state = 3;
bcopy(
(char *)&(signal->lvars.TS_primitives.T_CONNECT_request.
                  From_transport_addr),
(char *)&(p_block->lvars.s_ATP_process.tc
                  [p_block->lvars.s_ATP_process.tid-1].
                    local_addr),
sizeof(signal->lvars.TS_primitives.T_CONNECT_request.
              From_transport_addr));

bcopy(
(char *)&(signal->lvars.TS_primitives.T_CONNECT_request.
                  To_transport_addr),
(char *)&(p_block->lvars.s_ATP_process.tc
                  [p_block->lvars.s_ATP_process.tid-1].
                    remote_addr),
sizeof(signal->lvars.TS_primitives.T_CONNECT_request.
              To_transport_addr));

p_block->lvars.s_ATP_process.tc
        [p_block->lvars.s_ATP_process.tid-1].l_suffix =
Get_suffix(
       &(signal->lvars.TS_primitives.T_CONNECT_request.
                  From_transport_addr));

p_block->lvars.s_ATP_process.tc
```

```
        [p_block->lvars.s_ATP_process.tid-1].f_suffix =
Get_suffix(
       &(signal->lvars.TS_primitives.T_CONNECT_request.
                To_transport_addr));

Get_net_addr(
p_block->lvars.s_ATP_process.tc
        [p_block->lvars.s_ATP_process.tid-1].l_net_addr,
&(signal->lvars.TS_primitives.T_CONNECT_request.
         From_transport_addr));

Get_net_addr(
p_block->lvars.s_ATP_process.tc
        [p_block->lvars.s_ATP_process.tid-1].f_net_addr,
&(signal->lvars.TS_primitives.T_CONNECT_request.
         To_transport_addr));

p_block->lvars.s_ATP_process.nid =
  Get_ncep(
    p_block->lvars.s_ATP_process.nc,
    p_block->lvars.s_ATP_process.tc
             [p_block->lvars.s_ATP_process.tid-1].
               l_net_addr,
    p_block->lvars.s_ATP_process.tc
             [p_block->lvars.s_ATP_process.tid-1].
               f_net_addr);

p_block->lvars.s_ATP_process.tc
        [p_block->lvars.s_ATP_process.tid-1].ncep_id =
  p_block->lvars.s_ATP_process.nid;

p_block->lvars.s_ATP_process.nc
        [p_block->lvars.s_ATP_process.nid-1].link =
  p_block->lvars.s_ATP_process.nc
        [p_block->lvars.s_ATP_process.nid-1].link + 1;

bcopy(
(char *)&(signal->lvars.TS_primitives.T_CONNECT_request.
                Qual_of_service),
(char *)&(p_block->lvars.s_ATP_process.tc
                 [p_block->lvars.s_ATP_process.tid-1].
                   qual_of_service),
sizeof(signal->lvars.TS_primitives.T_CONNECT_request.
             Qual_of_service));

p_block->lvars.s_ATP_process.tc
        [p_block->lvars.s_ATP_process.tid-1].src_ref
  = Alloc_ref();

Construct_CR(
  &(p_block->lvars.s_ATP_process.data),
```

```
  p_block->lvars.s_ATP_process.tc
    [p_block->lvars.s_ATP_process.tid-1].rcv_upper_edge,
  p_block->lvars.s_ATP_process.tc
    [p_block->lvars.s_ATP_process.tid-1].src_ref,
  p_block->lvars.s_ATP_process.tc
    [p_block->lvars.s_ATP_process.tid-1].l_suffix,
  p_block->lvars.s_ATP_process.tc
    [p_block->lvars.s_ATP_process.tid-1].f_suffix,
  p_block->lvars.s_ATP_process.tc
    [p_block->lvars.s_ATP_process.tid-1].max_TPDU_size,
  &(p_block->lvars.s_ATP_process.tc
    [p_block->lvars.s_ATP_process.tid-1].qual_of_service),
  &(signal->lvars.TS_primitives.T_CONNECT_request.TS_user_data));

Concatenate_2_NSDU(
  &(p_block->lvars.s_ATP_process.nc
            [p_block->lvars.s_ATP_process.nid-1]),
  &(p_block->lvars.s_ATP_process.data));

      }
      goto dispose;
    }
  }
```