Semiautomatic Implementation of Communication Protocols

GREGOR v. BOCHMANN, SENIOR MEMBER, IEEE, GEORGE WALTER GERBER, AND JEAN-MARC SERRE

Abstract—**The use of formal specifications in software development allows the use of certain automated tools during the specification and software development process. Formal description techniques have been developed for the specification of communication protocols and services. This paper describes the partial automation of the protocol implementation process based on a formal specification of the protocol to be implemented. An implementation strategy and a related software structure for the implementation of state transition oriented specifications is presented. Its application is demonstrated with a much simplified Transport protocol. The automated translation of specifications into implementation code in a high-level language is also discussed. A semiautomated implementation strategy is explained which highlights several refinement steps, part of which are automated, which lead from a formal protocol specification to an implementation. Experience with several full implementations of the OSI Transport protocol is described.**

Index Terms—**Communication protocols, Estelle, formal description techniques, formal specification, implementation methodology, protocol implementation, specification translation, transport protocol implementation.**

I. Introduction

THE use of formal specification methods has been proposed for software engineering. Such methods are of particular importance for communication software since communication software must satisfy the rules defined by the communication protocols which are used. Many different implementations are usually built for a given communication protocol. It is therefore important that these rules be specified precisely. For this reason, they are candidates for being specified with a formal specification method.

The rules for communication between different computer systems are usually based on some architectural assumptions, such as defined by the OSI reference model [21] and the service concept [30], which define a layered system structure, such as shown in Fig. 1. For each layer, the service specification defines the communication service to be provided to the user entities in the next higher layer. The protocol specification, which defines the behavior of an entity within the layer, defines how the corresponding entities in different systems communicate with one another by exchanging so-called protocol data units (PDU), using the communication service provided by the layer below.

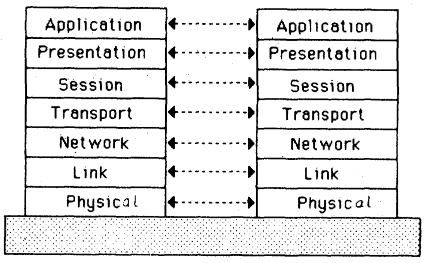


Fig. 1. OSI architecture.

Formal specifications, like informal ones, are used for the following purposes:

1) For each communication protocol and service, there is usually one specification which serves as "reference" for all other activities.

2) Protocol and service specifications are used for the validation of the design of the protocol of a given layer. For this purpose, the service provided by the protocol entities (as defined by the protocol specification) communicating through the service of the underlying layer is compared with the service defined by the service specification of the given layer.

3) The protocol specification is used for the elaboration of implementations.

4) The protocol specification is used during the validation (debugging, testing) of an implementation, and for assessing its conformance with the protocol specification.

Experiments with automated tools for the above activities have been reported in the literature (for a review, see for instance [8]). Such tools become important when formal specifications are used for real-life protocols which are usually sufficiently complex to make some automation desirable. Since such tools only become effective if they can be based on specifications written in some formal method, the development of suitable formal description techniques (FDT's) for communication protocols and services has been an area of much concern. For this reason, standardized FDT's are developed within ISO and CCITT

Manuscript received December 28, 1984; revised July 31, 1986. This work was supported by the Fond National Suisse de la Recherche Scientifique, the Department of Communications, Canada, and the Natural Sciences and Engineering Research Council of Canada.

G. v. Bochmann is with the Department of Computer Science and Operations Research, University of Montreal, C.P. 6128, Montreal, P.Q. H3C 3J7, Canada.

G. W. Gerber was with the Department of Computer Science and Operations Research, University of Montreal, C.P. 6128, Montreal, P.Q. H3C 3J7, Canada. He is now with the Centro de Pesquisas de Energia Electrica, Rio de Janeiro, Brazil.

J. M. Serre was with the Department of Computer Sciences and Operations Research, University of Montreal, C.P. 6128, Montreal, P. Q. H3C 3J7, Canada. He is now with Bell Northern Research, 3 Place du Commerce, Verdun, Ile des Soeurs, H3L 1H6, Canada.

IEEE Log Number 8716545.

[29], [12]. The results of these standardization efforts are three FDT candidates, SDL [24], Estelle [13], and Lotos [18]. SDL and Estelle are based on the model of finite state machines extended with programming language elements, while Lotos is based on CCS [20] and abstract data types [1]. A number of protocol standards for Open Systems Interconnection (OSI) have been written using these techniques.

This paper considers the automation of the protocol implementation activity (point 3) above). In Section II of this paper, general issues and design choices for protocol implementations are discussed. Also different objectives for the implementations are considered.

Section III describes a general implementation strategy assuming that a formal specification of the protocol is given written in an extended state machine formalism. For this implementation strategy, a specification compiler has been developed which translates a formal specification, written in a dialect of Estelle, into appropriate Pascal code, to be incorporated into a Pascal program implementing the protocol specification. This implementation strategy is demonstrated by a simple example for which the generated Pascal code is explained.

A complete implementation methodology including automated and nonautomated aspects is presented in Section IV. The application of this methodology to the implementation of the OSI Transport protocol class 2 is presented. The implementation obtained through the semiautomatic implementation methodology is compared with a similar implementation which was developed in a traditional manner.

II. Issues in Protocol Implementations

One advantage of formal specifications is the fact that implementations can often be obtained in a semiautomated manner, as for instance described below. However, different modes of executions, based on the formal specification, may be useful for various purposes, such as the following:

1) Traditional meaning of "implementation," where the communication protocol is executed for providing a communication service in a real operational system.

2) Simulated execution of the specification: this may be useful during the design of the protocol for analyzing the logical correctness of the protocol [16], [27], or for making performance simulations [28], [9]. Performance simulations are in particular useful for determining optimal parameters for real implementations which are expected to satisfy specific performance objectives.

3) Test trace validation during conformance testing: when a real implementation is tested for conformance with the protocol specification, the observed sequences of interactions may be "compared" with the specification, i.e., the observed trace is checked in order to determine whether it is a possible trace according to the specification [16], [27], [10].

It is important to note that for each of these different purposes, different design choices seem to be appropriate for realizing the execution of the specification. In the following we only consider case 1).

In communication software design, it seems natural to model the structure of the software system somehow along the lines of the protocol architecture. This architecture often follows the OSI Reference Model shown in Fig. 1, or a subset of the layers defined in that model. Usually several levels of protocols are involved in a given communication system. The communication software must be written in such a way that

1) all properties defined by the protocol specification are satisfied by the system, i.e., the system conforms to the protocol specification [23], and

2) other properties, not defined by the protocol specification, are chosen and implemented in such a way as to make the resulting system useful; in particular the following issues must be addressed:

• efficiency of operation (communication delays, maximum throughput, memory requirements, etc.),

• appropriate interfaces to user programs,

• appropriate interfaces to the underlying data transmission facilities, usually through the I/O facilities of the operating system.

Because of the importance of point 2), it is difficult to completely automate the protocol implementation process. However, the aspects of an implementation relating to point 1) can be obtained automatically from a formal specification, as explained below. An implementation methodology addressing both aspects is described in Section IV.

Given a formal specification of the protocol to be implemented, many properties not defined are either related to expressions, statements, functions, or procedures not explicitly defined (which are called "implementation dependent"), or to the intrinsic nondeterminism of the specification. Most specification languages allow the specification of nondeterministic systems for which several behaviors are possible in a given situation. In the case that an extended FSM model is used as specification language, nondeterminism is introduced when for a given state of the machine and given input interaction, there may be more than one transition that could be executed. Nondeterminism may also be introduced by spontaneous transitions which may be executed, without involving any input, provided that the present state satisfies a specified condition. A method for implementing such transitions is described in Section IV.

Another important implementation issue is the overall software architecture in terms of procedures, modules, and processes. This structure will usually reflect the structure of the specification. A complete protocol specification for a given layer often consists of several specification modules. As an example, Fig. 2 shows the structure of the Transport protocol specification which was used for the implementation project described in Section IV. It contains one **AP** module for each Transport connection,

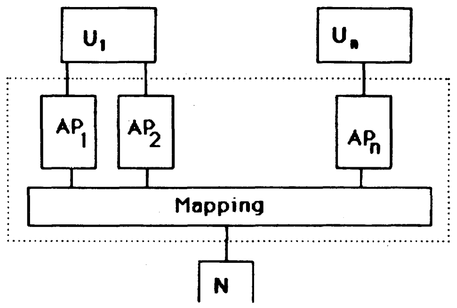


Fig. 2. Structure of the Transport protocol specification.

which handles connection establishment, data transfer and disconnection, and one common **Mapping** module which looks after the multiplexing of several Transport connection over a single Network connection. The figure also shows several instances of user modules U_i and module N providing the Network service; these latter modules represent the environment of the Transport entity.

However, the protocol for a given layer is usually not implemented in isolation, but in combination with the other protocol layers. Given that the software system corresponds to a large number of modules in the specification, it is important to determine how the interactions between these different modules are realized in the implementation. Some of these modules also interact with the environment, e.g., user programs or I/O devices. Often, the specification defines a static structure by which the different modules are interconnected.

Important design decisions relate to the manner in which these different interactions are realized. Another important design decision is the question of how many processes are used to implement the system, and how these processes communicate with one another and the environment using the operating system facilities. The implementation strategy discussed below provides automatically certain alternatives for the implementation of module interactions. It assumes that several specification modules are usually combined into a single process for implementation, although in an extreme case, one process per module could be used.

III. Translating Formal Specifications into a High-Level Programming Language

A. Specifications Using an Extended FSM Model

It is assumed in the following that the protocol specification is given in an extended finite state machine formalism [6], as for instance Estelle or SDL. Using such a formalism, a module of the specification can be described as an extended finite state machine which interacts through input/output interactions with other modules in the system. Alternatively, a module may also be described as consisting of several interconnected more simple modules, as for instance the Transport entity shown in Fig. 2. The behavior of each machine is described as an extended finite state transition machine. It may execute a state transition due to an input interaction received, or spontaneously. During such a transition, it may also generate output interactions. The extensions relate to interaction parameters and additional state variables. The relation of the state transitions with these parameters and state variables is described using a programming language notation (Pascal in the case of Estelle). For instance, an enabling condition may be associated with a transition which may depend on input interaction parameters and state variables, and which must be satisfied if the transition is to be executed.

The output interactions generated by a given module become input for another module of the specified system or for the environment. The destination module is determined by the interconnection structure between the different module instances within the specified system. Usually, it is possible to describe systems with static module interconnection structures, such as shown in Fig. 2, but FDT's and many other languages also allow the specification of dynamically changing interconnection structures.

An important feature of the specification language is the kind of module interactions assumed. Many languages assume that output interactions are placed in an input queue of the destination module, and the destination module will independently consider each interaction at the head of an input queue (Estelle allows for several input queues per module) for execution of a corresponding transition [13], [24]. Another approach is the direct interaction model, similar to rendezvous [15], [18] where an output can only be generated if the destination module accepts the interaction as input for one of its transitions.

B. A Simple Example Specification

In order to demonstrate the principles explained above, and to lead to the discussion of the translation process, this section presents a very simple specification example. It is based on the Transport protocol, but only two (out of many) transitions are considered.

As Fig. 2 shows, the Transport entity is described as consisting of one module of type **AP** per Transport connection and a single **Mapping** module. An (incomplete) formal specification of this Transport entity is given in Fig. 3. Fig. 3(a) defines the interaction structure of the Transport entity in terms of its interaction points with the environment and its refinement in terms of the **Mapping** and **AP** submodules. **TSAP** is an array of interactions points, one for each possible Transport connection, over which the Transport service is provided to the user modules. The different connections are distinguished by an index value of type **TCEP_id_type. NSAP** is the interaction point over which the Network service is used.

Fig. 3(b) shows the definition of the different kinds of interactions that can occur between the Transport entity and its users. The specification of the channel type **TS_primitives** given here defines only two kinds of interactions: the **TCONreq** interaction by which the user

```
module TP_entity;
     (* external port declarations *)
     TSAP : array [TCEP_id_type] of TS_primitives (provider);
     NSAP : NS_primitives (user);
end TP_entity;
          (a)
(* Transport service interactions *)
type
     option_type    = set of (option1, option2);
     data_type      = ...; (* implementation dependent *)
     T_address      = ...; (* implementation dependent *)

channel TS_primitives (user, provider);
by user:
     TCONreq (dest_address : T_address;
            proposed_options : option_type);
     TDATAreq (d : data_type);
     *****
by provider:
     *****
end TS_primitives;
          (b)
module Mapping;
     P : array [TCEP_id_type] of PDU_and_control (mapping);
     NS : NS_primitives (user);
end Mapping;
          (c)
(* PDU definitions *)
type
     sequence_nb = 0 .. 255;
     credit_type = 0 .. 127;

channel PDU_and_control (AP, mapping);
by AP, mapping:
     CR (addresss : T_address;
         options : option_type);
     AK (TR : sequence_nb;
         credit : credit_type);
     *****
end PDU_and_control;
          (d)
module AP;
     TS : TS_primitives (provider);
     MAP : PDU_and_control (AP);
end AP;


process Mapping_process for Mapping;
     *****
end Mapping_process;
```

```
process AP_process for AP;
var
     state : (closed, open, wait_for_CC, ***** );
     options : option_type;
     TRseq : sequence_nb;
     R_credit : credit_type;
     *****

function accepted_options (requested_options : option_type)
                         : option_type;
     begin (* all options supported *)
        accepted_options := requested_options
     end;

trans
     when  TS . TCONreq
     from  closed   to  wait_for_CC
     begin
          TRseq := 0;
          options := accepted_options (proposed_options);
          out  MAP . CR (dest_address, options);
     end;

*****

trans
     provided true (* it is possible to send an AK,
                     implementation dependent choice *)
     from  open   to same
     begin
          out  MAP . AK (TRseq, R_credit);
     end;

*****
end AP_process;
          (e)


(* process instantiations and interconnections *)
A1 : AP with AP_process;
A2 : AP with AP_process;
A3 : AP with AP_process;
A4 : AP with AP_process;

M : Mapping with Mapping_process;

connect
     A1.MAP  to  M.P [1];
     A2.MAP  to  M.P [2];
     A3.MAP  to  M.P [3];
     A4.MAP  to  M.P [4];
          (f)
```

Fig. 3. Example specification.

can request the establishment of a connection, and the **TDATAreq** interaction by which the user may send data over the established connection. The notation · · · · · is used here to indicate that certain parts of the specification are not included. The specification also introduces two roles, **user** and **provider.** As indicated in Fig. 3(a), the module **TP_entity** plays the role **provider** for the interaction points **TSAP.** It is assumed that a similar definition is given for the channel type **NS_primitives.**

Fig. 3(c) shows the definition of the submodule types **Mapping** and **AP.** They are interconnected by channels of type **PDU_and_control,** which are defined in Fig. 3(d). Over this channel coded protocol information is exchanged. The existing module's instances and their interconnection structure are defined in Fig. 3(f).

Fig. 3(e), finally, shows the definition of an extended finite state machine specifying the behavior for a module of type **AP**. Such a behavior is called a "process." Each module instance associated with this process contains the state variables **STATE, options, TRseq,** and **R_credit. STATE** is the so-called major state variable which corresponds to the "finite state" of the machine. Only two transitions are shown. The first is executed when a **TCONreq** is received by the user. It leads to an update of the state variables, including the new major state **wait_for_CC,** and to the generation of output of kind **CR.** The second transition is a spontaneous one, which may be executed whenever the connection is in the **OPEN** (major) state. It has the effect of generating an **AK** output interaction.

C. An Implementation Strategy

The implementation strategy described here applies to the automated implementation of specifications written in an extended finite state machine language in general. It was used by the Estelle compiler [14] which was used for several implementation projects [25], [32]. The objective of this compiler was to translate formal specifications written in an Estelle-like language, as shown in Fig. 3, into Pascal procedures which would be suitable for incorporation, without change, into a Pascal program implementing the protocol. A similar translation approach can be used for other implementation languages, for example C.

The following points present important aspects of the implementation approach, and have a strong impact on the structure of the implementation.

1) The automated translation process gives rise to one procedure written in the implementation language (in our case Pascal) for each process type in the specification. Each of the so obtained procedures can be compiled separately, if the implementation language compiler supports that option.

2) The execution of a transition of a module instance is performed by calling the procedure corresponding to the process type and passing to it as parameters the input interaction (if any) and a record data structure which contains the information about the present state of the module.

3) In a given system state, a number of different transitions belonging to different module instances may be possible. The selection of the next transition to be performed is made by an overall transition scheduler, which is usually part of the Pascal main program. This scheduler determines which module instance and which input interaction (or spontaneous transition) will be processed, and calls the corresponding process procedure. It is important to note that this scheduler is not automatically generated. A simple round-robin scheduler procedure is provided as part of the run-time support for the implementations generated by the Estelle compiler; the implementor is free to write his/her own scheduler to suit any particular implementation objectives.

4) For a given input interaction and a given state of the module instance, there are, in general, several possible transitions. A simple method of selection is to execute the first possible transition in the order in which the transitions are defined in the specification. With such an approach, the scheduler selects the module instance and the input interaction to be processed, or possibly a spontaneous transition is to be executed. The selection of the transition to process a given input is performed by the procedure implementing the module type. This selection may be realized by embedded case statements considering different interaction points, different kinds of interactions, and different major module states. It is also possible to generate embedded IF statements following in lexical order all the transitions in the specification, testing each to see whether it applies. These are two extreme choices for translating a module specification into program code. Combined approaches may often be more suitable. It is also possible to construct a transition table including, for each major state and kind of input, the list of applicable transitions. This form of code is particularly compact. The run-time support will then include a state machine interpreter [5], which may also include facilities for step-wise program execution, traces and other test instrumentation. Which of these approaches is taken for the generation of the program is largely a question of optimization.

5) The record data structure which contains the state information for a module instance also contains pointers to each of the surrounding module instances to which the former is connected. These pointers are used to forward output interactions generated by the module to the appropriate receiving module instances. Two options of module interactions are supported, **queueing** and "**direct call.**" The latter is a simplified version of rendezvous where the outputting module calls the procedure generated for the destination module, passing the generated output as input parameter. It is assumed that an output operation is only performed when the receiving module is in a state where it can accept the given interaction as input. It is the responsibility of the designer of the specification/implementation to make sure that this condition is satisfied.

6) The above considerations apply to all interactions between module instances which are part of the specified system. Interactions with its environment are not easily included in this general implementation scheme. However, most specified systems are "embedded" systems which include interactions with the environment. For instance, the Transport protocol implementation described in Section IV interacts with the user processes and part of the operating system providing the Network communication service. Output to the environment is easily implemented by calling an implementation-dependent procedure for output. Input from the environment can be implemented by including spontaneous transitions in the specification which would be called by the global scheduler and execute the processing related to the input. The scheduler should be aware when the external input is available.

7) The data structures containing the state information of the module instances and the pointers connecting them according to the module interconnection structure defined in the specification are created by support routines, which are called during the execution of the implementation. In the case of static system structure, these routines are called during the initialization phase. The initialization procedures are also generated by the Estelle compiler based on the specification of the modules.

D. Translating a Specification into Implementation Code

In order to make the above discussion more concrete, we present in detail the translation of the example shown in Fig. 3 using the translator mentioned before [14]. It is important to note that the specification of Fig. 3 must be "completed," as explained in Section IV-B, before the translation process is applied. We assume in the following that this has been done.

In summary, the compiler generates three categories of statements. First are the type declarations for the records which will represent the state of the module instances, channels, and interactions as well as the variables and type declarations required by the run-time support. Secondly, there are procedures which implement the processes by code to handle the various defined transitions. Finally, there are procedures for the initialisation of the system and the creation and linking of the various data structures.

Details of the generated code are shown in Figs. 4 and 5. Fig. 4 shows the declarations generated by the compiler. Lines 1-38 show the type declaration SOTYPE for the records which represent the interactions in the speci-

```
 1    S1TYPE = ^SOTYPE;
 2    SOTYPE =
 3      RECORD
 4        NEXT: S1TYPE;
 5        CASE CHANNEL OF
 6          C2TS_primitives:
 7            (CASE T2TS_primitives: S2TS_primitives OF
 8               S2TCONreq:
 9                 (D2TCONreq:
10                    RECORD
11                      dest_address: T_address;
12                      proposed_options : opticn_type
13                    END);
14               S2TDATAreq:
15                 (D2TDATAreq:
16                    RECORD
17                      d: data_type
18                    END); );
19          C1PDU_and_control:
20            (CASE T1PDU_and_control:
21              S1PDU_and_control OF
22               S1CR:
23                 (D1CR:
24                    RECORD
25                      address: T_address;
26                      options: option_type
27                    END);
28               S1AK:
29                 (D1AK:
30                    RECORD
31                      TR: sequence_nb;
32                      credit: credit_type
33                    END); );
34          COAP_process:
35            (CASE TOAP_process: SOAP_process OF
36               R1ANY:
37                 (); );
38      END;
39
40    P1TYPE = ^POTYPE;
41    POTYPE =
42      RECORD
43        IDENT: P2TYPE;
44        CHANLIST: C1TYPE;
45        NEXT, REFINEMENT: P1TYPE;
46        CASE PROCESS OF
47          POMapping_process:
48            ();
49          POAP_process:
50            (DOAP_process:
51               RECORD
52                 STATE: (closed, open,
53                         wait_for_CC);
54                 options: option_type;
55                 TRseq: sequence_nb;
56                 R_credit: credit_type;
57               END);
58      END;
59
60  VAR
61    POVAR: P1TYPE;
62    COVAR: C1TYPE;
63    SOVAR: S1TYPE;
```

Fig. 4. Declaration code produced by FDT compiler.

fication. Each interaction primitive defined in the original specification is implemented as a variant of that declaration. Note the presence of a system attribute, NEXT in line 4, required by the implementation in addition to the variant fields that reproduce exactly the fields present in the original specification (lines 11–12, 17, 25–26, and 31–32). To handle the use of identical field names in different interactions, the user fields are not inserted directly in the variant for each interaction; rather, they are grouped in a dummy record (for example: D2T_CONreq in line 9) which then appears as the only attribute of each variant (S2T_CONreq in this case).

Another point to notice in the SOTYPE declaration is the definition of an empty variant, COAP_process (lines 34–37), which does not correspond to any explicit channel or interaction definition in the original source code. This variant is generated to handle the spontaneous transition in the **AP_process** process. Spontaneous transitions are treated as special interactions on a channel 0 and

```
 1  PROCEDURE AP_process;
 2
 3    LABEL
 4      1;
 5
 6    VAR
 7      C1VAR: C1TYPE;
 8      S1VAR: S1TYPE;
 9
10
11    FUNCTION accepted_options(requested_options: option_type)
12           :option_type;
13      BEGIN
14        WITH POVAR^.DOAP_process DO
15          BEGIN
16          accepted_options := requested_options;
17          END
18      END;
19
20    BEGIN
21      C1VAR := COVAR;
22      S1VAR := SOVAR;
23      WITH POVAR^.DOAP_process DO
24        BEGIN
25        IF COVAR^.IDENT = 1 THEN
26          IF SOVAR^.T2TS_primitives = S2TCONreq THEN
27            WITH SOVAR^.D2TCONreq DO
28              BEGIN
29              IF STATE IN [closed] THEN
30                BEGIN
31                STATE := wait_for_CC;
32                TRseq := 0;
33                options := accepted_options(proposed_options);
34                BEGIN
35                    PUPROCEDURE(2);
36                    NEW(SOVAR, C1PDU_and_control,S1CR);
37                    SOVAR^.T1PDU_and_control := S1CR;
38                    SOVAR^.D1CR.address := dest_address;
39                    SOVAR^.D1CR.options := options;
40                    OUT
41                END;
42                GOTO 1; END;
43              END;
44        IF true THEN
45          BEGIN
46          IF STATE IN [open] THEN
47            BEGIN
48            IF COVAR^.IDENT = 0 THEN
49              IF SOVAR^.TOAP_process = R1ANY THEN
50                BEGIN
51                BEGIN
52                    POPROCEDURE(2);
53                    NEW(SOVAR, C1PDU_and_control,S1AK);
54                    SOVAR^.T1PDU_and_control := S1AK;
55                    SOVAR^.D1AK.TR := TRseq;
56                    SOVAR^.D1AK.credit := R_credit;
57                    OUT
58                END;
59                GOTO 1;
60                END;
61            END;
62          END;
63        END:
64  1:
65    CASE C1VAR^.IDENT OF
66      0:
67        CASE S1VAR^.TUAP_process OF
68          R1ANY:
69            DISPOSE(S1VAR, COAP_process, R1ANY);
70          END;
71      1:
72        CASE S1VAR^.T2TS_primitives OF
73          S2TCONreq:
74            DISPOSE(S1VAR, C2TS_primitives,
75                      S2TCONreq);
76          END;
77      END;
78  END;
```

Fig. 5. Procedure generated by FDT compiler.

dummy variants are generated for each spontaneous transition present in the specification.

The second part of Fig. 4 shows the type declaration POTYPE for the data structures which will represent the module instances during execution. For each module type defined, a variant is added to the POTYPE declaration. The same technique outlined for the interactions is used here and user declared local attributes appear without change within a dummy record in each variant. The attributes for the AP_process module of Fig. 3(e) are shown in lines 49-57. There is also an empty variant (lines 47-48) for the Mapping module present in the full example specification. There are 4 system attributes (lines 43-45) present in all process records; these are used by the run time routines to identify, build and link the data structures modelling the specified system.

Finally, in lines 61-63, are shown three key global variables which determine the context for any transition. During execution, these pointers (POVAR, COVAR and SOVAR) will contain respectively the addresses of the current module instance, the current channel whose interaction is being treated, and the current interaction itself. These are the only global variables shared by all routines in the system.

Fig. 5 shows the procedure generated to handle the transitions for the **AP_process** module. The procedure assumes that its context (POVAR, COVAR, and SOVAR) has been set up correctly before the procedure is called. For each defined transition, a section of code is generated. The example has two transitions: the first, triggered by the receipt of a T_CONreq interaction, leads to the code in lines 25-43 and the second, a spontaneous transition, leads to the code in lines 44-62. All procedures begin with a partial saving of context (lines 21-22) so that the environment is not lost when the output procedure modifies the values of COVAR and SOVAR. There is also a WITH statement on the correct variant of the current module (line 23) so that local module variables may be referred to directly within the procedure body.

Creation and destruction of the interaction records which transfer information between module instances is implemented completely within the generated "module" procedures. At the end of each such procedure (lines 64-77) in this case) Pascal code is generated to dispose of any interactions the process may receive. It is in the nature of the Pascal language that for each variant of a record type, a separate DISPOSE statement is needed where the value of the variant tag is expressed as a constant (it is not possible to use a general-purpose DISPOSE statement where the variant tag would be expressed as a variable). Generation of this specific disposal code is an important task of the compiler.

Each transition follows the same pattern. The code is preceded by tests on the identity of the received interaction and that of the received interaction and that of the channel upon which it came. To this may be added tests corresponding to PROVIDED or FROM clauses (line 29). A WITH statement on the input (line 27) allows the received interaction attributes to be used directly in the transition statements. At the end of the transition a "GOTO 1" statement passes control to the data record disposal code. For a spontaneous transition (lines 44-62), the pattern is the same except that the "spontaneous" channel 0 is used and the interaction is a "pseudo" interaction with no attributes internally generated by the system.

Another aspect shown in Fig. 5 is the implementation of the output statement (lines 34-41 and 51-58). The interaction to be sent is first created (line 53) and its data fields are initialized (line 54-56). The call to the library procedure POPROCEDURE (line 52) had located channel MAP (assigned the number 2 by the compiler) and had placed its address in COVAR. The output routine uses data in the Channel data structure to place the interaction in the right reception queue. If a channel with rendezvous is specified, the procedure representing the recipient module is called directly at this point. If the channel uses queued interactions, control returns immediately to the sender; the recipient being activated later by the scheduler when it comes upon the pending interaction in its scan of channel queues.

E. Translation Issues

The above example gives a simplified introduction to code generated for Estelle specifications. In general, the translation process is more difficult, in particular when the full power of the specification language is taken into account. A number of specification translators have been or are being built for Estelle and similar languages [5], [3], [14], [4].

The Estelle compiler described here [14] is partitioned into the usual phases of lexical, syntactic, and semantic analysis, optimizations, and code generation. The syntax analysis phase creates an internal representation of the specification in the form of a tree structure. The nodes of the tree are characterized by semantic attributes which are evaluated during the subsequent semantic analysis phase. During this phase, certain semantic conditions are also checked. The compiler only verifies those semantic conditions which would not be validated by the subsequent Pascal compilation of the generated Pascal program. It is noted, however, that it would be more convenient to have a complete verification of the static semantics by the specification compiler. Certain optimizations may be performed on the internal representation of the specification. For instance, the different transitions could be sorted in a specific order to make the generated code smaller or more efficient. The code generation phase is relatively straightforward, in particular for those parts of a specification which are already essentially written in the Pascal notation, such as the actions of transitions, or the expressions in PROVIDED clauses. However, there are certain aspects of the specification language which are more difficult to handle, such as the following.

There seems to be no simple translation scheme which correctly handles Estelle's scope rules in relation with the WHEN clause and the associated input parameters. Our compiler simply generates a Pascal WITH statement which opens a scope containing the parameter names. However, this approach does not work when the same names are also used for formal parameters or local variables in a procedure declared within the transition.

Another problem related to scope rules is due to the identifiers of certain run-time utility procedures, variables, and types. These identifiers are referenced in the generated Pascal code. For instance an <output> procedure is used for generating an output interaction. Therefore, the inadvertent use of the same identifiers for other purposes in the specification leads to problems. One (unsatisfactory) solution is to prohibit the use of these predefined identifiers in specifications.

In addition, the compiler generates certain additional identifiers which correspond to parts of the specification. These identifiers should never be in conflict with the other identifiers of the specification which remain present in the translated Pascal code; they should also have a mnemonic form closely resembling the part of the specification which they represent. Our compiler forms such identifiers by preceding the original name by a fixed character and a number which is used to distinguish different cases (see for example Fig. 4). There seems to be no completely satisfactory solution, in particular if the generated code should be compilable by a Pascal compiler that distinguishes identifiers based on only its first (say) 10 characters.

The run-time utility procedures which are used for creating the interconnection structure between modules during the initialization of a system have to work with any type of module and channel. They are a kind of generic procedures which are difficult to write in the Pascal programming language.

IV. Protocol Implementation Methodology

It is important to note that the automated translation of specifications, as described above, covers only part of the implementation effort. As mentioned before, there are aspects of an implementation which are not described by the specification and which must be chosen during the implementation phase. These aspects are discussed in this section. We first present a Transport protocol implementation which was developed in a traditional, ad hoc manner, although a formal specification of the protocol was used as a basis for certain parts of the code. A systematic methodology for implementing protocols in several steps of refinement and based on a formal specification is then described. It was subsequently applied to the same Transport protocol together with the automated specification translation approach described above. A comparison of these two implementations is also given.

A. An ad hoc Implementation Based on a Formal Specification

The structure of this implementation [25] of the OSI Transport protocol classes 0 and 2 is shown in Fig. 6. The

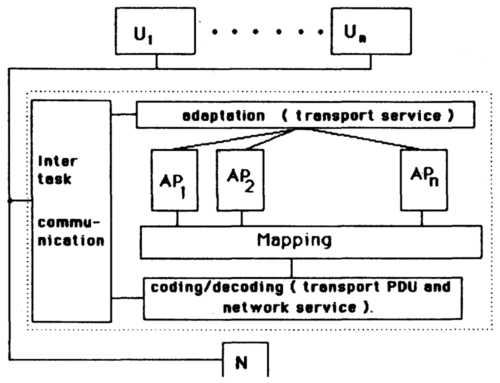

U_1
U_n
adaptation (transport service)
Inter task communication
AP_1
AP_2
AP_n
Mapping
coding/decoding (transport PDU and network service).
N

Fig. 6. Structure of "manual" implementation.

logical behavior of the **AP** and **Mapping** modules is based on the formal Transport protocol specification given in [7]. The Transport entity is a single task in the operating system, communicating through operating system primitives with a task providing the Network service, and several user tasks which may establish one or several Transport connections with remote systems through the Transport entity.

The interactions between the different tasks is based on message exchange provided by the operating system. For the original implementation which was running on a PDP-11 computer, the user data was not directly included in these messages, rather pointers to data buffers were passed between the tasks. The data buffers were allocated in shared memory by the task first generating the data, or receiving it through external I/O primitives, and were shared among all the processes using the data [19].

Overall, the program handles the incoming events in an asynchronous manner. Messages received by the task or internal time-outs give rise to event interrupts which schedule corresponding event handling routines which in turn call the procedures implementing the protocol. This kind of program structure is efficient and relatively transportable. In fact, the program was subsequently transported to run under VMS on a VAX, and it is presently ported to run under Unix. Only the intertask communication package had to be adapted for each of these new environments.

The experience of this implementation showed that the availability of a formal specification simplifies significantly the implementation process; however, only part of the implementation is directly related to the formal specification. Much time was spent in the development of the interfaces with the operating system for interaction with the user processes and the Network communication service, including buffer management. Another important part, not included in the formal specification, is the coding and decoding of PDU's. The size of the Pascal source code for these different program sections is given in Table I.

B. Implementing Protocols by Stepwise Refinement of Specification

Although the number of steps to be used for going from a *protocol specification* to the *implementation code* may

TABLE I
SIZE OF DIFFERENT PARTS OF TWO TRANSPORT PROTOCOL IMPLEMENTATIONS

(A) ad hoc (manual) implementation approach
(B) implementation using the specification compiler

Part of program	Number of source lines (A)	Number of source lines (B)	Program size (in octets) (A)	Program size (in octets) (B)
(a) PDU de- and en-coding	3 000	3 000	11 940	11 940
(b) Code corresponding to the transitions of the formal specification	3 000	5 500**	17 800	29 306
(c) Buffer management and O/S interfaces for intertask communication	3 000	3 000	3 974	3 974
(d) Run-time support routines	1 000	1 400***	2 324	3 452
(e) main program	1 000	400	6 468*	3 282*

Notes:
* including static variables
** this part of the code is automatically obtained by translation of the detailed specification (see Section 4.2)
*** this part of the code is fixed (independent of the specification) and comes with the specification compiler

vary in different cases, we propose a methodology which identifies the following steps:

1) Preparation of a *detailed specification*,
2) Adaptation to the implementation environment,
3) Program code creation, and
4) Testing.

Each of these steps is further discussed below.

1) Detailed Protocol Specification: As further explained in [26], the objective is to obtain a *detailed specification* which would be valid for a large number of implementations of the protocol in question. However, compared to the protocol specification, it is more detailed and specific, i.e., the available choices are reduced. The *protocol specification* should only describe properties that must be satisfied by all conforming protocol implementations, and it should be designed in such a manner that any two communicating implementations that conform to the specification will provide the corresponding communication service (see point 2) in the Introduction).

It seems that the following aspects can normally be described in the detailed specification:

a) More detailed specification of the local service interfaces with the protocol layers below and above;

b) Identification of all error cases (in addition to those already described in the protocol specifications) and definition of the corresponding error handling (in particular, the handling of unexpected interactions from the user module);

c) The handling of the spontaneous transitions.

Point a) includes in particular the definition of certain data types which have been left undefined by the protocol specification, such as **data_type** and **T_address** in Fig. 3. It may also include the definition of certain implementation-dependent procedures and/or functions, or the definition of supplementary service primitive parameters.

A protocol specification usually indicates the required behavior of the protocol entity in the case of certain errors committed by the peer protocol entity. However, often these indications are relatively vague, and more precise choices must be made for any implementation. In addition, erroneous behavior of the user and/or underlying communication service may be foreseen. The detailed protocol specification should define the behavior of the protocol implementation for these situations.

As mentioned in Section III-B, a spontaneous transition may be executed any time its enabling condition is satisfied. For efficiency reasons, however, an implementation should normally select appropriate instants for the execution of the defined spontaneous transitions. The strategy for doing this should be defined in the detailed protocol specification.

For example, Fig. 3 contains one spontaneous transition for sending acknowledgments which is enabled whenever the connection is in the **OPEN** state. The associated **PROVIDED** clause is completely implementation dependent. In fact, the strategy for sending acknowledgments can have an impact on the end-to-end transmission delay, on buffer management at the sending and receiving sites, and on the cost of communication. Therefore, an appropriate acknowledgment scheme must be selected depending on the performance requirements of the protocol implementation. In practice, the spontaneous transition could for instance be executed whenever an additional buffer (and therefore additional credit) is available.

In general, a spontaneous transition has a PROVIDED clause of the form **PROVIDED <boolean expression>**, and whether it is enabled may depend on additional state variables included in the expression. The following systematic approaches to the handling of spontaneous transitions may be considered (it is noted that such choices must be made whether a specification compiler is used, or not):

a) Considered all spontaneous transition for execution at regular time intervals.

b) Maintain two lists of spontaneous transitions, called **active** and **passive**. Initially, the active list contains all spontaneous transitions. Transitions from the active list are considered for execution at regular time intervals. If one of them is not enabled at that time, it is placed on the passive list. A transition from the passive list is put on the active list when the execution of the transition may have led the effect of changing the state variables in such a manner as to make the enabling condition of the spontaneous transition true. An analysis of the data flow between the actions of all transitions (which may update the state variables) and variables used in the enabling conditions of the spontaneous transitions should be made in order to determine whether, after the execution of a given transition, one of the passive transitions should be made active.

c) Maintain a **try** list of spontaneous transitions. A data flow analysis similar as for point 2) above should be used to determine, after the execution of a given transition, which spontaneous transition may be enabled due to the action of the executed transitions. All those spontaneous transitions are placed on the try-list and are considered for execution before the next input transition is executed. It is noted that the data flow analysis may be done in an informal manner by the person writing the detailed protocol specification. Operations of the form "try transition <number>", which have the effect of putting the indicated spontaneous transition into the try-list, may be included in the definition of the transition actions. This latter strategy has been used for the Transport implementation described in Section IV-C.

2) Adaptation to the Implementation Environment: While it can be expected that many design choices made at the level of the detailed protocol specification may be applicable for a large number of different implementation projects in different implementation environments, the next step involves the adaptation to the particular runtime environment. Buffer management issues and operating systems interfaces must be considered in detail, including interprocess (task) communication.

The partition of the communication software into several processes must be decided at this level. Fig. 6 shows how the interprocess communication interfaces were included in the software structure of the Transport protocol implementation developed with the Estelle compiler (see Section IV-C). The formal specification of the Transport protocol was completed by the addition of two interface modules which include environment-specific routines for sending and receiving messages from the other processes in the system.

3) Program Code Creation: Based on the formal specification of the protocol, which has been completed as described above, the implementation code can be produced manually, or through a specification compiler. It is noted that the manual programming task is usually much simpler based on a formal specification than when it is based on an informal protocol specification.

4) Protocol Implementation Testing: Protocol implementation testing usually proceeds in several stages. Debugging testing is often performed using an interactive testing tool by which arbitrary peer and user interactions can be generated and the output of the implementation under test (IUT) be observed. Afterwards, the implementation may be submitted to conformance test [23] which is usually executed in a largely automated manner.

It is important to note that certain tests can be executed based on the formal protocol specification or the detailed protocol specification if a suitable execution support exists for the specification language. Any errors found in these specifications may be corrected before the more detailed steps of the implementation proceeds. If the detailed specification is shared for many implementation projects, they may all benefit from less error-prone specifications.

C. A Semiautomatic Transport Implementation

In order to evaluate the usefulness of the Estelle compiler for the automatic generation of parts of a protocol implementation, the same formal specification that was the basis for the implementation described in Section IV-A was also used for generating an implementation with the Estelle compiler described above.

The same buffer management and intertask communication routines were used in order to make a comparison between the two implementations more meaningful. The resulting program sizes are shown in Table I (column B). It is noted that only row 2 is generated by the specification compiler, and row 4 is a fixed set of run-time support routines. Rows 1 and 3 are the same in the two implementations. As the table shows, the transition code generated by the compiler is larger than the corresponding code of the hand-coded implementation, but it turned out to be of a more regular structure.

As the above table shows, the buffer management and intertask communication routines are relatively complex. However, as explained in Section III-C, the specification compiler allows the integration of several separately specified modules into a single Pascal program. The implemented Transport protocol entity, for instance, consists of one **Mapping** module and several AP modules, as shown in Fig. 6. Also, the specifications of the protocols for several layers could be combined into a single program (task). This would reduce the intertask communication overhead associated with an implementation where each layer protocol would be implemented in a separate program.

It is interesting to note that about half of the overall program code were generated by the specification compiler. It is also interesting to note that most errors found after the initial debugging phase were related to the environment-specific parts of the specification. This indicates that the detailed formal specification contained relatively few errors [26].

V. Discussion and Conclusions

The availability of the formal specification of a protocol can be useful for the validation of the protocol design, as well as for protocol implementation and testing. This paper discusses the semiautomatic implementation of protocols based on their formal specification. It is important to note that a protocol specification usually leaves important design decisions unspecified; these design decisions must be made for each implementation of the protocol depending on the particular implementation requirements. As discussed in Section IV-B, protocol specifications written in a formal description technique (FDT) may be refined in several steps until an implementation-oriented specification is obtained.

It is sometimes argued that specifications in Estelle tend to appear "implementation oriented," in the sense that they imply certain design decisions which are a matter of implementation [17]. Real implementations adopting these decisions can be obtained semiautomatically, as described in the paper. However, it is conceivable that other implementations would use different implementation choices. Their automated generation would require automatic program transformations, which are more difficult to realize. SDL has similar characteristics as Estelle. However, most existing specifications written in SDL describe the "data part" informally, which implies that this important part cannot be processed automatically. Other specification languages, such as Lotos, are primarily intended for more abstract specifications. So-called ''wide band'' languages are designed with the aim to be suitable for writing abstract specifications which can be transformed, within the same language, into implementation-oriented designs and code [31].

The implementation strategy using the stepwise refinements described in Section IV-B can be applied with any of the above languages. An important saving in implementation effort can be obtained if an existing formal protocol specification, or (better) detailed specification, can be used as starting point for the implementation project.

The translation methodology described in Section III is largely adapted to specifications written in an extended finite state machine formalism. As discussed in Section IV in relation with the Transport protocol implementations, a specification compiler can produce readable code which is relatively efficient in space and run-time. However, it is also clear that it would not be used in cases where high-performance implementations are desired.

Further experiences with the semiautomatic implementation approaches are underway. Areas which need particular attention include the following:

1) The automatic inclusion of testing facilities within the generated implementations.

2) Efficiency improvements in the generated code related to interactions between different module instances. An implementation language with generic types or less strong typing rules than Pascal seems to be useful here.

3) Integration, within the specification language and its compiler, of PDU coding and decoding facilities based on the ASN1 notation [2] for OSI Application layer protocols. This would allow for automatic code generation for part (a) in Table I.

References

[1] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications*, vol. 1. New York: Springer-Verlag, 1985.

[2] *Information Processing—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*, ISO DIS 8824.

[3] J. P. Ansart, V. Chari, and D. Simon, ''From formal description to automated implementation using PDIL,'' in *Protocol Specification, Testing and Verification (IFIP/WG6.1)*, H. Rudin and C. H. West, Eds. Amsterdam, The Netherlands: North-Holland, 1983.

[4] J. P. Ansart *et al.*, ''Software tools for Estelle,'' in *Protocol Specification, Testing and Verification VI*, B. Sarikaya and G. Bochmann, eds. Amsterdam, The Netherlands: North-Holland, 1986, pp. 55–62.

[5] T. P. Blumer and R. Tenney, ''A formal specification technique and implementation method for protocols,'' *Comput. Networks* vol. 6, no. 3, pp. 201–217, July 1982.

[6] G. v. Bochmann, ''A general transition model for protocols and communication services,'' *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 643–650, Apr. 1980; reprinted in *Communication Protocol Modeling*, C. Sunshine, Ed. Dedham, MA: Artech House, 1981.

[7] —, ''Example of a Transport protocol specification,'' prepared for CERBO Informatique Inc. under contract for Department of Communications Canada, Oct. 1982.

[8] —, ''Usage of protocol development tools: The results of survey'' (invited paper), presented at the 7th IFIP Symp. Protocol Specification, Testing and Verification, Zurich, Switzerland, May 1987.

[9] G. v. Bochmann, D. Ouimet, and J. Vaucher, ''Simulation for validating performance and correctness of communication protocols,'' Dep. d'IRO, Univ. Montreal, Tech. Rep., 1987.

[10] G. v. Bochmann, R. Dssouli, and J. R. Zhao, ''Trace analysis for conformance and arbitration testing,'' *IEEE Trans. Commun.*, submitted for publication.

[11] G. v. Bochmann and J. P. Verjus, ''Some comments on ''transition-oriented'' vs. ''structured'' specification of distributed algorithms and protocols,'' *IEEE Trans. Software Eng.*, to be published.

[12] G. J. Dickson and P. de Chazal, ''Application of the CCITT SDL to protocol specification,'' *Proc. IEEE*, Dec. 1983.

[13] *Estelle: A Formal Description Technique Based on an Extended State Transition Model*, ISO DIS 9074, 1987.

[14] G. Gerber, ''Une methode d'implantation automatisee de systemes specifies formellement,'' M.Sc. thesis, Univ. Montreal, 1983.

[15] C. A. R. Hoare, ''Communicating sequential processes,'' *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[16] C. Jard and G. v. Bochmann, ''An approach to testing specifications,'' *J. Syst. Software*, vol. 3, no. 4, pp. 315–323, Dec. 1983.

[17] L. Kovacs and A. Ercsenyl, ''Specification versus implementation based on Estelle,'' *ACM Comput. Commun. Rev.*, vol. 16, no. 4, pp. 4–22, July 1986.

[18] *LOTOS: A Formal Description Technique*, ISO DIS 8807, 1987.

[19] M. Maksud, ''Operator's manual for the interaction transport tester,'' prepared for CERBO Informatique Inc. under contract for the Department of Communications of Canada, 1984.

[20] R. Milner, *A Calculus of Communicating Systems* (Lecture Notes in Computer Science, No. 92). New York: Springer-Verlag, 1980.

[21] *Reference Model for OSI*, ISO IS 7498, 1982.

[22] H. Partsch and R. Steinbrueggen, ''Program transformation systems,'' *ACM Comput. Surveys*, vol. 15, no. 3, Sept. 1983.

[23] D. Rayner, ''A system for testing protocol implementations,'' *Comput. Networks*, vol. 6, no. 6, Dec. 1982.

[24] *Recommendation Z.100*, CCITT SG XI, 1987.

[25] J. M. Serre, ''Methodologie d'implantation du protocole Transport classe 0/2,'' M.Sc. thesis, Dep. d'IRO, Universite de Montreal, 1985.

[26] J. M. Seere, E. Cerny, and G. v. Bochmann, ''A methodology for implementing high-level communication protocols,'' in *Proc. 19th Hawaii Int. Conf. Syst. Sci.*, Jan. 1986.

[27] H. Ural and R. L. Probert, ''Automated testing of protocol specifications and their implementations,'' in *Proc. ACM SIGCOMM Symp.*, 1984.

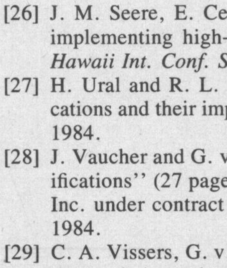
[28] J. Vaucher and G. v. Bochmann, ''A simulation tool for formal specifications'' (27 pages + annexes), prepared for CERBO Informatique Inc. under contract for the Department of Communications Canada, 1984.

[29] C. A. Vissers, G. v. Bochmann, and R. L. Tenney, ''Formal description techniques by ISO/TC97/SC16/WG1 ad hoc group on FDT,'' *Proc. IEEE*, vol. 71, no. 12, pp. 1356–1364, Dec. 1983.

[30] C. Vissers and L. Logrippo, ''The importance of the concept of service in the design of data communications protocols,'' in *Proc. IFIP Workshop Protocol Specification, Verification and Testing*, Toulouse, 1985. Amsterdam, The Netherlands: North-Holland.

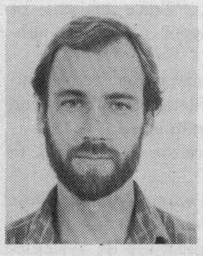
[31] F. L. Bauer *et al.*, *Towards a Wide-Spectrum Language to Support Program Specification and Program Development* (Lecture Notes in Computer Science, No. 69). New York: Springer-Verlag, 1979, pp. 543–552.

[32] G. v. Bochmann, ''Semi-automatic implementation of transport and session protocols,'' *Comput. Standards and Interfaces*, vol. 5, pp. 343–349, 1987.



Gregor v. Bochmann (M'82–SM'84) received the Diploma in physics from the University of Munich, West Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

He has worked in the areas of programming languages and compiler design, communication protocols, and software engineering. He is currently Professor in the Department d'Informatique et de Recherche Operationnelle, Universite de Montreal. His present work is aimed at design, implementation, and testing methods for communication protocols and distributed systems. During 1977–1978, he was a Visiting Professor at the Ecole Polytechnique Federale, Lausanne, Switzerland. During 1979–1980 he was a Visiting Professor in the Computer Systems Laboratory at Stanford University, Stanford, CA.



George Walter Gerber was born in Berne, Switzerland, on August 23, 1954. He received the Diploma in mathematics from the Swiss Federal Institute of Technology (ETH Zurich) in 1978 and the M.Sc. degree in computer science from the University of Montreal, Montreal, P.Q., Canada, in 1983.

From 1978 to 1981 he was with the Development Division for Building Automation at Landis & Gyr, Zug, Switzerland, as a Software Specialist. From 1982 to 1983 he was a Research Assistant at the Department of Computer Science and Operations Research of the University of Montreal, focusing on formal languages, compilation, and communication protocols. Since November 1983 he has been at CEPEL, Brazilian national research centre for electric energy, in Rio de Janeiro, where his work encompasses protocol specification, implementation and testing, and power plant and substation automation. His research interests include communication protocols, distributed systems, and automation.

Jean-Marc Serre finished his Master thesis in 1985 with Professors Eduard Cerny and Gregor v. Bochmann at the University of Montreal, Montreal, P.Q., Canada.

He worked from 1983 to 1986 as a Research Assistant of Professor Bochmann, implementing various transport protocols classes and conformance testers using FDT techniques. Some of those programs were used to perform international OSI tryouts with Sweden, Japan, England, and Australia. From the beginning of 1986 to 1987 he worked as a Programmer Analyst for CWARC, a Canadian government owned R&D center. He is currently working for Bell Northern Research on network management standards.