ARPA ORDER NO.: 3460/3681
OP10 Information Processing Techniques

N-1429-ARPA/NBS

November 1979

FORMAL METHODS FOR COMMUNICATION PROTOCOL SPECIFICATION AND VERIFICATION

Carl A. Sunshine

A Rand Note

prepared for the

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY and the
NATIONAL BUREAU OF STANDARDS

Rand
SANTA MONICA, CA. 90406

The research described in this note was sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA903-78-C-0029; and by Purchase Order No. 815662 from the National Bureau of Standards.

Published by The Rand Corporation

ARPA ORDER NO.: 3460/3681
OP10 Information Processing Techniques

N-1429-ARPA/NBS

November 1979

FORMAL METHODS FOR COMMUNICATION PROTOCOL SPECIFICATION AND VERIFICATION

Carl A. Sunshine

A Rand Note

prepared for the

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY and the
NATIONAL BUREAU OF STANDARDS

Rand
SANTA MONICA, CA. 90406

PREFACE

This is the final report on a Rand study of methods for specifying and verifying computer communication protocols. The study, which was jointly sponsored by the Defense Advanced Research Projects Agency and the National Bureau of Standards, was a small exploratory effort. Its main purpose was to survey the state of the art, identify promising directions for future work, and make some initial progress in some of these directions, rather than to present solutions to major problems. This document should be of interest to technicians and planners in the sponsoring agencies, as well as to others concerned with the design, analysis, procurement, and evaluation of computer networks and communication protocols.

The author wishes to acknowledge the contributions to this report of Ming-Yee Lai, who participated in Rand's summer graduate student program.

SUMMARY

Increasingly numerous and complex communication procotols are being employed in distributed systems and computer networks of various types. The informal techniques used to design these protocols have been largely successful but have also yielded a disturbing number of errors or unexpected and undesirable behavior in most protocols. This Note describes some of the more formal techniques being developed to facilitate design of correct protocols.

A great deal of confusion surrounds the words "specification" and "verification" in the domain of computer communication protocols. Hence our first goal is to define these concepts in the context of a layered model of protocols. Protocol specification requires a clear definition of both the services to be provided by a given protocol layer and the protocol entities within the layer that cooperate to provide the service. Verification, then, consists of two parts: (1) showing that the entities collectively do interact to provide the specified services, and (2) showing that each entity is properly implemented according to its specification. A useful subset of the first part may be described as verification of "general properties" such as deadlock, looping, and completeness. These properties may be checked for most protocols without requiring any particular service specification.

Most work in protocol design and analysis to date has proceeded without a comprehensive specification of the services to be provided by a protocol to its users. Hence a major focus of our work has been to explore techniques for formally specifying protocol services. Two major

approaches from the general software specification domain were identified and applied to a set of example protocols: (1) an "abstract machine" model that defines operations that may be invoked, and (2) an "agent" model where the service is an active process with inputs and outputs.

Abstract machine models successfully handle services that can be defined in terms of individual operations with specified effects on the state of the machine. These models provide a convenient means for handling exception conditions, but they do not easily accommodate several necessary aspects of protocol service specifications. Assertions about sequences of operations are outside the model. Since they have no explicit output, only "polling-type" interfaces with the user can be defined. "Prodding-type" interfaces cannot be defined. Moreover, assumptions must be made about the behavior of the machine's users in order to talk about termination. To overcome some of these problems, it appears necessary to include "users" who are really part of the service, or to include cyclic processes in the specification.

Agent models handle these difficulties more successfully. Transition-type agents retain most of the benefits of machine models in defining the handling of individual inputs. Buffer-history-type agents facilitate assertions about sequences of operations but have difficulty with state-oriented service features and exceptions. It appears that both sorts of specification are useful for different aspects of service--transitions for state-oriented features like connection establishment, and buffer histories for data transfer.

Most of the effort in verification to date may be classified as either state exploration or program-proving. State exploration is based on modeling each entity of a protocol layer as a state machine and then generating all the reachable states of the composite system, starting from some initial state. This type of analysis is relatively straightforward and has been automated to some extent, but it can deal only with the major states or "control" aspects of a protocol. Program-proving is based on specifying each entity as a program and then formulating and proving assertions that represent correct operation of the system (i.e., its service specification). This technique can in principle verify all features of a specification, but a great deal of ingenuity is required to construct proofs of even simple systems.

Several newer techniques promise to reduce some of these difficulties. "Unified" methods use state exploration of a few major states to facilitate program proofs of additional properties involving the other state variables of the protocol. Symbolic execution exploits the ability to group classes of system states in order to minimize the size of the state space that must be explored. Both design rules and transformation methods promise to eliminate the need for post-design verification altogether by constraining the design process to follow correct paths. All of these techniques require further research before their effectiveness can be evaluated.

It is clear from the literature that the use of more formal techniques has already had a positive impact on the protocol design process. State-exploration techniques for verifying general properties are fairly well understood and have the potential for routine

application in the near future. Use and development of more powerful verification techniques require a high level of skill in formal methods and must still be considered research problems. A great deal of work remains to be done in developing techniques that are routinely and widely applicable.

CONTENTS

I. INTRODUCTION

Increasingly numerous and complex communication procotols are being employed in distributed systems and computer networks of various types. The informal techniques used to design these protocols have been largely successful, but they have also yielded a disturbing number of errors or unexpected and undesirable behavior in most protocols. This Note describes some of the more formal techniques being developed to facilitate the design of correct protocols.

As they develop, protocols must be described for many purposes. Early descriptions provide a reference for cooperation among designers of different parts of a protocol system. The design must be checked for logical correctness. Then the protocol must be implemented. If the protocol is in wide use, many different implementations may have to be checked for compliance with a standard. Although narrative descriptions and informal walk-throughs are invaluable elements of this process, painful experience has shown that they are inadequate by themselves.

A great deal of confusion surrounds the words "specification" and "verification" in the domain of computer communication protocols. Hence our first goal is to define these concepts in the context of a layered model of protocols. Section II includes a listing of the necessary elements of a protocol specification. Section III explores methods for formal specification in detail, focusing on the experience gained from an attempt to specify several example protocols. In Section IV we discuss various approaches to protocol verification, identifying the pros and cons of major techniques, as well as some promising new

approaches. Section V cites some applications of formal methods that have been reported in the literature. A comprehensive bibliography indexed by key phrases is also provided.

II. THE MEANING OF PROTOCOL SPECIFICATION AND VERIFICATION

We assume that the communication architecture of a distributed system is structured as a hierarchy of different protocol layers. Each layer provides a particular set of services to the users above. From those users' viewpoint, the layer may be viewed as a "black box" or machine which allows a certain set of interactions with other users (see Figure 1). A user is concerned with the nature of the service provided, but not with how the protocol manages to provide it.

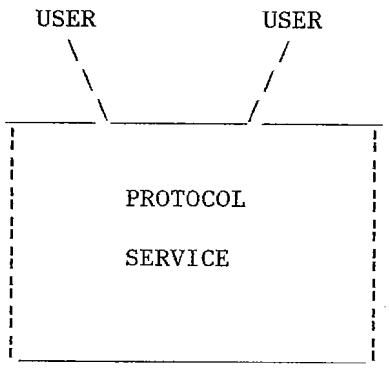


Figure 1--User View of Protocol Layer

This description of the input/output behavior of the protocol layer constitutes a service specification of the protocol. It should be "abstract" in the sense that it describes the types of commands and their effects but leaves open the exact format and means for conveying them (e.g., procedure calls, system calls, interrupts, messages, etc.) A particular instance of the service which does specify exact formats may be called an interface specification.

SERVICE SPECIFICATION

Specifying the service to be provided by a layer of a distributed communication system presents problems similar to those of specifying any software module of a complex computer system. Therefore, methods developed for general software engineering [8,53,75,97,125] should be useful for the definition of communication services. Usually, a service definition is based on a set of service primitives that describe in an abstract manner the operations at the interface through which the service is provided. In the case of a transport service, for example, some basic service primitives are Connect, Disconnect, Send, and Receive. The execution of a service primitive is associated with the exchange of parameter values between the service-providing entity of one layer and the service-using entity of the higher layer. The possible values and the direction of transfer must be defined for each parameter.

Clearly, the service primitives should not be executed in an arbitrary order and with arbitrary parameter values (within the range of possible values). At any given moment, the allowed primitives and parameter values depend on the preceding history of operations. The service specification must reflect these constraints by defining the allowed sequences of operations directly, or by defining a "state" of the service which is used in specifying the results of operations.

In general, the constraints depend on previous operations by the same user ("local" constraints) and by other users ("global" constraints). Considering again the example of transport service, a local constraint is the fact that Send and Receive may be executed only

after a successful Connect. An example of a global constraint is the fact that the "message" parameter value of the first Receive on one side is equal to the message parameter value of the first Send on the other side. To date, little is known about methods for precisely specifying computer communication services [21,53,115,125].

PROTOCOL SPECIFICATION

Although the internal structure of a protocol layer is irrelevant to the user, the protocol designer must be concerned with it. In a network environment with physically separated users, a protocol layer must be implemented in a distributed fashion, with entities (modules or processes) local to each user communicating among themselves via the services of the lower layer (see Figure 2). The interactions among entities in providing the layer's service constitute the actual protocol. Hence a protocol specification must describe the operation of each entity within a layer in response to commands from its users, messages from the other entities (via the lower-layer service), and internally initiated actions (e.g., timeouts).

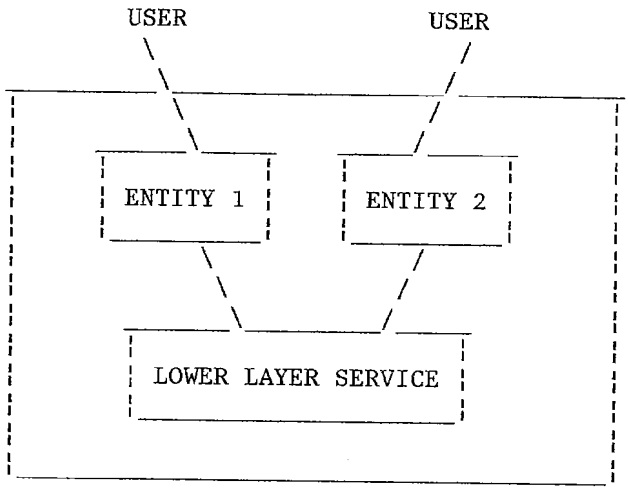


Figure 2--Internal Structure of Protocol Layer

ABSTRACTION AND STEPWISE REFINEMENT

The specifications described above must embody the key concept of abstraction if they are to be successful. To be abstract, a specification must include the essential requirements that an object must satisfy and must omit the unessential. A service specification is abstract primarily in the sense that it does not describe how the service is achieved (i.e., the interactions among its constituent entities) and secondarily in the sense that it defines only the general form of the interaction with its users (not the specific interface).

A protocol specification is a refinement or "implementation" of the service specification because it begins to define how the service is provided by specifying the entities that cooperate to perform it. This "implementation" of the service is what is usually meant by the design of a protocol layer. Each entity remains to be implemented in the more

conventional sense of that term, typically by coding in a particular programming language. There may be several steps in this process until the lowest-level implementation of a given protocol layer is achieved [19,53,55].

WHAT A PROTOCOL DEFINITION SHOULD INCLUDE

Given the above discussion, a complete description of a protocol layer should include the following items:

1. A general description of the purpose of the layer and the services it provides.

2. An exact specification of the service to be provided by the layer.

3. An exact specification of the service assumed for the layer below and required for the correct and efficient operation of the protocol. (This may be redundant with the lower layer's definition, but it makes the protocol definition self-contained.)

4. The internal structure of the layer in terms of entities and their relations.

5. A description of the protocol(s) used between the entities including:

   5.1. An overall, informal description of the operation of the entities.

   5.2. A protocol specification which includes

      (a) a list of the types and formats of messages exchanged between the entities;

      (b) rules governing the reaction of each entity to user commands, messages from other entities, and internal events.

   5.3. Any additional details (not included in point 5.2), such as considerations for improving efficiency, suggestions for implementation choices, or a detailed description which may come close to an implementation.

Reference 15 presents an example of these items for a simple data-transfer protocol. While the descriptive items of this list are important, it is the formal items 2 and 5.2 that are the main focus of this Note. We next consider how this understanding of protocol specification sheds light on the meaning of verification.

THE MEANING OF VERIFICATION

Verification is essentially a demonstration that an object meets its specifications. Recalling from above that services and protocol entities are the two major classes of objects requiring specification for a protocol layer, we see that there are two basic verification problems that must be addressed: (1) The protocol's design must be verified by analyzing the possible interactions of the entities of the layer, each functioning according to its (abstract) protocol specification, to see whether their combined operation satisfies the layer's service specification; and (2) the implementation of each protocol entity must be verified against its abstract specification.

The somewhat ambiguous term "protocol verification" is usually intended to mean this first design verification problem. Because protocols are inherently systems of concurrent independent entities interacting only via exchange of messages, verification of protocol designs takes on a characteristic communication-oriented flavor. Implementation of each entity, on the other hand, is usually done by "ordinary" programming techniques and hence represents a more common (but by no means trivial) program verification problem that has received less attention.

The service specification itself cannot be verified, but rather forms the standard against which the protocol is verified. However, the service specification can be checked for syntax, consistency, or realizability [53]. It must also properly reflect the users' desires and provide an adequate basis for the higher levels that use it. Unfortunately, techniques to achieve these latter goals are still poorly understood.

It is important to note that protocol verification also depends on the properties of the lower-layer protocol. A protocol specification is only a partial specification in that it leaves some actions undefined, namely, those given by the lower layer's service specification. In verifying that a protocol meets its service specification, it will be necessary to assume the properties of the lower layer's service. If a protocol fails to meet its service specification, the problem may rest either in the protocol itself or in the service provided by the lower layer.

In its broadest interpretation, system validation aims to assure that a system satisfies its design specifications and (ideally) operates to the satisfaction of its users. At the present time, the validation of correctness features and that of efficiency features are largely separate disciplines employing different methods. Efficiency considerations are outside the scope of this Note. In the realm of correctness, exploratory techniques such as simulation and testing may be used, but usually only a limited number of situations can be examined. We reserve the term "verification" for methods that allow,

at least in principle, the consideration of all possible situations the system may encounter during operation.

Most of the verification work to date has focused on protocol designs without having any precise and comprehensive service definition. Results have demonstrated what can be verified easily rather than what needs to be verified. Hence a major focus of the work reported here has been on the problems of rigorous protocol service specification, at least in the realm of correctness.

III. FORMAL SPECIFICATION METHODS

As noted above, three things must be specified for a given protocol layer: the service it provides to its users, the entities that make up the layer and comprise the protocol itself, and the implementation of the entities. We shall assume that implementations will use a suitable programming language, and we shall have little to say about them. This section discusses methods for specifying services and protocols.

Although numerous approaches to specifying software systems have been proposed, two major categories may be discerned. We shall call these the "abstract machine" model and the "agent" model.

In the machine model, the service is defined in terms of a set of operations that may be invoked by the machine's users. The machine has a state consisting of the values of variables or value-returning functions that are defined as part of the machine. An operation may return a value (some portion of the state) and/or change the state of the machine, depending, of course, on the current state of the machine.

In the agent model, the service is viewed as an active agent or process with explicit inputs and outputs. The agent functions by consuming its input and producing its output. Two types of agent model are used. State-transition-type models focus on the processing of individual inputs and include an internal state for the agent. For each combination of input and state, a new state and output (possibly null) are specified. The second type of agent model, which we shall call a "buffer-history" model, avoids referring to any internal state of the

agent and attempts to directly specify the relation between the sequence of outputs and the sequence of inputs.

In the following sections, we shall explore the merits of each of these models for specifying the services of a set of representative protocols. These protocols were chosen to include a very simple and widely used example (the alternating bit protocol (ABP)), a more sophisticated general-purpose data-transfer protocol (a transport protocol), and a "function-oriented" or higher-level protocol (file-transfer protocol). We first describe each of these protocols informally to provide a general understanding of their services before attempting more formal specifications.

INFORMAL PROTOCOL SERVICE DESCRIPTIONS

Alternating Bit Protocol

The ABP provides for one-way transfer of messages from a fixed sender to a fixed receiver. Messages must be delivered without error and in the order in which they are sent. The sender must wait until the receiver accepts a message before sending the next message. The protocol gets its name from its implementation, where a single bit sequence number is attached to each message sent over an unreliable transmission medium. The bit is alternated for each new message sent. Further details on the implementation may be found in the literature [13,14,20,82] and, of course, are irrelevant for describing the service provided.

Transport Protocol (TP)

TP can accommodate many users, each identified by a port or address. A pair of users must first request a connection between themselves before exchanging messages. Once connected, users may simultaneously transfer messages in both directions. The receiver (in each direction) controls the flow of messages by giving the sender explicit "credits" or permission to send some number of messages. Connected users may also exchange short "interrupts" which are independent of the data messages exchanged, and which typically travel faster. When users are finished communicating, they ask to be disconnected. Messages and interrupts must be delivered without errors and in the order in which they are sent. Transport protocols are also called host-to-host, virtual circuit, or interprocess communication protocols.

File Transfer Protocol (FTP)

The purpose of FTP is to extend the file-system services available locally into a network environment. The services it provides include copying or appending from one file to another, deleting or renaming files, and listing the contents of file directories. The ability to partially retrieve or replace parts of files may also be provided. Since FTP operations may take significant amounts of time and resources, the ability to run them in "background" mode, as well as to inquire about their status or abort them, is also important.

SERVICE SPECIFICATIONS

Now we will try to formally specify each of the three protocols described above, using the specification approaches mentioned. This will provide further details on each method and will reveal difficulties that arise in performing real specifications.

SRI's High Level Development Methodology (HDM) [97,107] provides a well-developed example of the abstract machine model. The specification language in HDM, which is called SPECIAL, allows a rich variety of data types to be defined. The state of the machine is defined in terms of the values of "Vfunctions" which may be visible to users or hidden. Operations that change the state (set new values for Vfunctions) are called Ofunctions. Each Ofunction may have a number of "exceptions" defined which prevent the normal effects of the operation from taking place, and instead return an error code.

The University of Texas GYPSY system [53,55] provides a well-developed example of the agent model. The basic elements of Gypsy are processes and buffers. Processes may interact with each other only by sending messages through buffers. The effects of processes must be specified using entry and exit assertions, or with "block" assertions that are to hold whenever the process is blocked waiting to read or write a buffer. Typically, these assertions are made in terms of "buffer histories" or the sequences of messages sent to and received from particular buffers by particular processes.

There are numerous examples of transition-type agent models [13,20,98,125]. Rather than using any of these exactly, we have introduced our own form of transition model to demonstrate the similarities to the HDM model. As we shall see below, a machine model can be converted to a transition model by treating the operations as inputs and adding explicit outputs.

Alternating Bit Protocol

A simple ABP service specification in SPECIAL is presented in Fig. A.1 of App. A. (Information on SPECIAL syntax is also given.) The machine includes the definition of an abstract data type "msg," Send and Receive operations, and a buffer for the "state." This specification shows that the ABP service is essentially that of a queue of size one.

It is important to note that SPECIAL specifications are not algorithmic. In particular, Ofunctions do not give a series of assignment statements, but rather present a set of expressions defining the new state after the operation completes.

In addition to defining the individual operations, we would like to specify that the messages delivered by Receive are the same as the messages given to Send. This requires assertions about sequences of operations that are beyond the scope of the SPECIAL language. However, we can make these assertions in either of two ways. First, we could write a "program" for the ABP machine as follows:

```
BEGIN
Send(m1);
m2 := Receive();
END
```

Then we could assert that m2=m1 when this program terminates. Hopefully, we could prove this assertion from the definition of the individual operations of the ABP machine, so we are not adding any new information. However, we may be expressing the service in a more convenient way.

The second approach is to introduce some additional "ghost" variables or state to the ABP machine which allow us to write assertions about the whole sequence of Send and Receive operations. Such an augmented specification is given in Fig. A.2 of App. A, including correctness and liveness assertions which are once again outside the scope of SPECIAL. Note that we have extended the Send and Receive operations to place their message also into the new ghost variables Inseq and Outseq, respectively.

Once again, we should be able to prove these assertions from the definitions of the individual operations. However, an interesting difficulty arises with the progress assertion. To prove this assertion, it is clearly necessary to assume that Receive operations are invoked in a timely fashion. That is, since the machine has no explicit output of its own, we must make assumptions about the behavior of the users of the machine in order to talk about progress.

Unfortunately, neither of the above approaches to handling sequences of operations is directly accommodated in HDM. Assertions

such as those in Fig. A.2 of App. A are simply outside the scope of SPECIAL. To use the "program" approach, we would have to specify a hypothetical higher-level operation called Send_Receive, and then give its implementation as follows:

```
OVFUN Send_Receive(msg m1) -> msg m2;
   EFFECTS
      m2 = m1;

OVFUN PROG Send_Receive(msg m1) -> msg m2;
   BEGIN
   Send(m1);
   m2 := Receive();
   END;
```

We could then use the verification tools of HDM (see below) to prove that the program for Send_Receive correctly implemented its specification, assuming the specification for the lower-level Send and Receive operations. However, the introduction of this higher-level operation is at best artificial.

Buffer-history models are intended to facilitate specification of just such sequences of behavior. Fig. A.3 of App. A shows the ABP service specified in GYPSY. Note that the concept of buffers is part of the specification language, as are functions "allto" and "allfrom" on buffers that return the sequence of all messages sent to or received from a buffer. The Block statement in Fig. A.3 gives the service specification solely in terms of the input and output sequences of the ABP agent, without reference to any internal state. Progress is not included.

To give a transition-type specification, we need to show the processing of each individual input. Fig. A.4 of App. A gives such a

specification in the GYPSY syntax, showing that there is only one type of input "event" for the transition machine: an arriving message. The action of the agent consists solely of copying the input message to the output. No state is necessary, although a temporary variable is used to hold the received message.

Once again, the transition and buffer-history specifications give equivalent information, and it should be possible to prove the latter from the former. However, in this case both are expressable within the scope of the GYPSY system, and it is possible to deal with termination without making any assumptions about other agents that interact with this agent (i.e., its users).

Transport Protocol

A transport protocol includes features such as addressing, connections, flow control, and interrupts that are not found in the simple ABP. These services require more complex specifications. Once again we start with a machine model specification, as shown in Fig. A.5 of App. A.

Send and Receive operations are still present, but they now include exceptions for flow control and connection features. They also make use of a separately specified buffer module to allow multiple messages to be in transit at one time.

Flow control is handled by a state variable that maintains the difference between credits given and messages sent. Violating the flow-control constraints is treated as an error, illustrating the use of

exception conditions and a "non-blocking" interface. However, these limits could also have been specified to block the operation by replacing the exceptions with a Delay statement such as "DELAY UNTIL Credit(me,him) > 0;".

Interrupts provide a parallel channel for short messages with a simple one-at-a-time flow control, and hence are specified almost exactly as message transfer in the ABP. A major difference, however, is the introduction of multiple users and connections. The existence of multiple users requires inclusion of explicit addresses in all operations, to indicate which users are involved.

Connections introduce the first service feature that is not data-transfer oriented, but rather state oriented. Four states are needed to capture the notion of connection between a pair of users A and B: neither connected, A requesting connection to B, B requesting A, and both requesting. These four states are conveniently captured by the two boolean state functions Pconnected A,B and Pconnected B,A (where P means partially). These state functions summarize the results of previous Connect and Disconnect operations and thereby facilitate specification of the effects of new connection requests.

An important feature of connection operations is that user B should be "notified" when user A requests a connection with him. Since the machine model has no explicit outputs, this means that user B must watch for a change in value of the Pconnected(x,B) Vfunctions. This suggests that users must interact with the machine model in a "polling" fashion by constantly invoking Vfunctions to see what has happened, rather than

being "prodded" by the service. Similar problems arise with interrupts and even with data messages.

The same difficulties concerning specification of sequences of data-transfer operations, arise as in the ABP, but the "program" approach is no longer workable. Since the specification of the transport protocol allows multiple Sends to be done before any Receives (unlike the specification for ABP), programs would have to be written for the many possible permutations. Hence, specification in terms of input and output sequences seems the only viable solution.

If we wish to specify a buffer-history-type model for TP, message transfer looks much as it does for the ABP. However, the addition of addresses makes it necessary to extract the messages between a pair of users from all the input and output, as shown in Fig. A.6 of App. A. The function Data_transfer_OK states that the output is an initial subsequence of the input between a given pair of users. A similar statement is also needed for interrupts.

To specify flow control, we must have some way to count the number of messages sent and credits given. Since no state variables may be used, specification must be in terms of the size of the input sequences themselves, as shown in the function Flow_control. This formulation is not totally satisfactory because although it describes correct behavior, it fails to tell what happens in the case of "bad" inputs. In particular, if a sender tries to send too many messages before credit is granted, the specification should state that they are "rejected." Then the output is no longer equal to the input of messages, but only to the

"accepted" messages. Thus buffer-history models appear to handle exception conditions poorly.

Connections pose the most serious difficulties for a buffer-history specification. It appears to be extremely difficult to describe the output sequence of Connect and Disconnect messages as any "closed form" function of the input sequence of those messages. The best that can be done is to define the handling of individual inputs as in a transition model. Thus the function Connection_OK in Fig. A.6 describes the handling of individual Connect and Disconnect inputs. Note that we have had to define input history sequences that are equivalent to the partially and fully connected states of the machine model. Expressing such state-oriented features with buffer histories seems cumbersome at best. It should also be noted that it is necessary to consider the relative timing of previous inputs from both users in order to specify the results of an input from one user.

Figure A.7 in App. A presents a transition model of the TP service that is derived primarily from the machine model of Fig. A.5. Each Vfunction remains as a state function. Each Ofunction is converted to an input-handling routine to process the corresponding input to the agent. These input handlers produce explicit outputs--typically, a copy of their input--eliminating the need for the receiving operations of the machine model. However, since the state of the agent can no longer be observed by its users, we have had to add Give_credit and Int_ack outputs which are given to the sender to inform him of flow-control limits. Exception conditions are carried over unchanged, except they now cause an error message output to the sending user.

Either of the agent-type models allows formulation of assertions about sequences of operations such as, for example, that matching connection requests result in being fully connected, or that the number of Sends accepted must be less than the number of credits given. Both models also allow specification of "prodding" rather than polling-type service.

It is interesting to note that the TP specified in Figs. A.5 through A.7 is quite similar to the CCITT X.25 protocol if Resets are omitted. However, there is an important difference in viewpoint. Our specification might be seen as defining the (end-to-end) service provided by a public network between its subscribers. This is definitiely not the intention of X.25, although it does imply some end-to-end service features. Alternatively, our specification could be used to define the service provided by an X.25 "machine" whose users were the subscriber (DTE) on one side and the public network (DSE) on the other. This is closer to the intention of X.25's designers, but there is still a difference. Our service specification defines the user interactions with the X.25 machine which are missing from the CCITT specification. The X.25 protocol covers only the exchanges between the two entities which make up the machine, and thus it provides only a partial protocol specification and no service specification.

File Transfer Protocol

From an abstract service viewpoint, an FTP is not very different from a local file system. Since the purpose of FTP is to extend the operations of local file systems across a network, FTP is defined in

terms of those local operations. The key operation of copying a file from one system to another requires the definition of equivalence between files on different systems. To do this, we assume the existence of a function Canon(file) that transforms equivalent files on different systems to a canonical form which is identical.

Figure A.8 in App. A presents an abstract machine-model service specification for FTP. The major state functions are Exist(file name), which defines whether a file exists, and Cont(file name), which holds the "contents" of the file. There is also a Status state function which returns the status of previously requested operations (e.g. done, error, in progress, etc.). The status is maintained by a separate table management module. FTP also calls on a TP module to supply data-transfer service.

The operations include functions to start and stop sessions with the FTP service (Ftp and Quit) and to request the usual file-system functions of copying, appending, deleting, and listing names of files. The results of these operations can be defined conveniently in terms of the current state or the new file-system state when the operation completes. A machine model is well-suited to specifying this type of self-contained operation, complete with numerous exception conditions that are important for file operations, as shown in Fig. A.8.

The only major difficulty with FTP concerns specification of a "background" mode of operation. Unlike the "foreground" mode where each function is completed before control is returned to the caller, background-mode operations merely initiate functions before returning.

These functions are then completed by a background demon or server process, and the user must invoke a status operation to determine when the function is completed and what its outcome is. To define this type of service, we need to specify an initiating operation that returns a transaction id, a status operation, and a background process that completes the transaction. Then we can assert that if the status of the initiated transaction is "done," then its end results are true (e.g. for copying, file A equals file B).

This problem can be handled in three ways: First, we can specify an initiating operation which "posts" the work to be done as part of the state, and a separate completing operation that completes the transactions that are posted. However, this begs the question of who is going to invoke the completing operation, and when they will do so, and it leaves us unable to prove termination without assuming the timely invocation of this operation by some unspecified outside user. A variation of this approach is to specify a higher-level operation whose effects are the desired end results and whose implementation is the parallel calling of the initiating and the completing operations; but this, too, is outside the machine specification and is somewhat artificial.

The second approach is to push the definition of the server process into the implementation that does deal with programs. However, this leaves the end results of the background operation completely unspecified at the service level. In effect, we are relying on the implementation to do more than the service specification states, which seems highly unsatisfactory.

The third approach is to extend the machine model to include definition of cyclic "processes" as part of the machine. Such processes would be defined just like other operations, with the understanding that they are constantly invocable (i.e., conceptually, they would be invoked after every normal operation). This approach is illustrated in Fig. A.8 of App. A, where a Background_Server process is specified to complete the background-mode Copy and Append operations that are posted to be done by normal user operations.

As with TP, the machine model of FTP could easily be converted to a transaction-type agent model. Background functions could be accommodated easily in the agent model by forking a process to do the background work and to also produce an immediate output when background commands were received. When the background process was done, a second output could be produced, relieving the user of the need to poll for completion.

As expected for a specialized function protocol, the main portion of the FTP specification concerns the semantics of the specialized functions to be performed (e.g., file-system functions). The details of these functions are outside the scope of the protocol per se and would be defined in a separate file-system module. The main concern of the FTP service itself then becomes flow of control. In most cases this is a simple request/response interaction that is easily modeled with an abstract machine approach. However, the background mode of operation presents greater difficulties, as discussed above.

Summary

In this section we have considered two basic models for describing protocol services: an abstract machine model where operations must be invoked, and an agent model where the service is an active process with inputs and outputs. We have used these models to specify three representative protocols (ABP, TP, and FTP) and have compared them, largely on the basis of their expressive power or their ability to completely and conveniently define the services of interest.

Machine models successfully handle services that can be defined in terms of individual operations with specified effects on the state of the machine. These models provide a convenient means for handling exception conditions, but they do not easily accommodate several necessary aspects of protocol service specifications. First, assertions about sequences of operations are outside the model. Since they have no explicit output, only "polling" interfaces with the user can be defined; prodding-type interfaces cannot be defined. Also because outputs are lacking, it is necessary to make assumptions about the behavior of the machine's users in order to talk about termination. Finally, in order to specify background modes of operation, it is necessary to include "users" who are really part of the service, or to include cyclic processes in the specification.

Agent models handle these difficulties more successfully. Transition-type agents retain most of the benefits of machine models in defining the handling of individual inputs. Buffer-history-type agents facilitate assertions about sequences of operations but have difficulty

with state-oriented service features and exceptions. It appears that both sorts of specification are useful for particular aspects of service--transitions for state-oriented features like connection establishment, and buffer histories for data transfer. Fortunately, these approaches can be mixed within the framework of agent models such as the GYPSY system.

We have focused on comparing the expressive power of different service-specification methods in this section. However, ease of implementation and verification are also important considerations, and we shall return to these subjects later.

PROTOCOL SPECIFICATIONS

The distinction between abstract machine and agent models carries over to protocol specifications. In the SRI HDM machine model, each Vfunction of the service machine must be "mapped" to Vfunctions in lower-level machines, and each Ofunction in the service machine must be implemented as a program which invokes operations of lower-level machines. Hence the protocol specification consists of a combination of these programs, plus some of the lower-level machine definitions. In the agent model, each service agent is typically implemented as a set of cooperating lower-level agents. Hence the protocol specification is just another agent specification.

This section presents protocol specifications in both of these models and compares the results. Since the protocol specification is several times longer than the service specification, we have limited

detailed discussion to the ABP. This is sufficient to reveal the major differences, and examples from the other protocols are included when useful.

Figure B.1 of App. B shows an abstract machine specification that implements the ABP service shown in Fig. A.1. The first part of the example gives the program for each of the Ofunctions in the ABP service. This program calls functions in the lower-level machines which are defined next. These include a Send_Station and a Receive_Station which form part of the protocol. These trivial machines are needed to hold the sequence number and message-text-state information for the protocol.

There are specifications for a medium in both directions (sender to receiver and receiver to sender). These define the service provided by the lower-level protocol--in this case, a simple link which can lose packets (it is assumed that damaged packets are discarded). We have included a limit on the number of messages that may be lost to illustrate one method for constraining loss to facilitate termination proofs. There are also specifications for a timer machine used by the sender, and finally a mapping of the state functions which show that the buffer at the service level actually corresponds to the buffer in the Receive_Station at the protocol level.

The non-determinism of the medium introduces an interesting problem. In the machine model, each operation invoked at the service level may lead to a sequence of operations at lower levels, but all of the low-level operations must complete before the top-level operation returns. Thus the top-level Send operation leads to a series of

Send_Data operations until a proper acknowledgment is received. The receiver needs to execute a corresponding series of Receive_Data operations, but he has no way of knowing when he can stop trying to receive, because although he may receive a proper data message and acknowledge it, his acknowledgment may be lost. Hence he must continue to accept messages and acknowledge them to satisfy the sender who is waiting for an acknowledgment.

Such a receiver can be specified only as a cyclic process that runs across the boundaries of individual Send operations. We have taken the liberty of extending the machine model to include a program for such a process which does not correspond to any operation at the service level, but rather runs continuously once the service machine is initialized.

Figure B.2 of App. B shows an agent-model specification that implements the service shown in Fig. A.3. The ABP agent's "implementation" is just a skeleton that starts five other agents in parallel and sets up the appropriate interconnection of their inputs and outputs. The sender and receiver agents define the protocol and make use of the two medium agents and a timer as before.

It is interesting to note that while the high-level service is specified with a buffer-history-type model, the protocol agents are specified with a transition-type model. This is convenient because the outputs of the protocol agents are much more complex functions of their inputs than are those at the service level.

While this transition specification is convenient for descriptive and perhaps implementation purposes, it is probably insufficient for verification purposes; some additional assertions relating the inputs and outputs of each protocol agent would be needed for verification. We have tried to include a few examples of such buffer-history assertions, but those given are probably inadequate. The construction of such assertions is one of the known difficulties of most program-verification methods.

Most existing protocol-specification efforts fall into the agent category, although the explicit distinction between abstract machine and agent models has not been made at the protocol level. A variety of transition models have been adapted and/or applied to protocol problems [13,85,115]. Most of these define a limited explicit state with additional "context" information or state variables. There have also been a number of protocol specifications using more-or-less abstract programming-language approaches [11,108].

In reality, these approaches are not so different. On the programming-language side, it is possible to define a "state" variable in a program and dispatch to different processing routines, on the basis of state and input type; transition models, however, often depend on program-language descriptions to define their use of context information. Thus a number of hybrid models that use transition models for a limited set of explicit major or control states and programs to define the use of additional state information have been proposed [20,39].

IV. VERIFICATION METHODS

This section presents an extensive, high-level review of approaches to protocol verification. After discussing the general outlines of what can be verified, we review each major approach and discuss various methods for dealing with protocol complexity.

WHAT CAN BE VERIFIED

The overall verification problem may be divided along two axes, each with two categories. On one axis, we distinguish between general and specific properties. On the other, we distinguish between partial correctness and termination or progress.

General properties are those properties common to all protocols that form an implicit part of all service specifications. Foremost among these is the absence of deadlock (the arrival in some system state or set of states from which there is no exit). Completeness, or the provision for all possible inputs, is another general property which requires only the specification of the input set in order to be checked. Progress or termination also require minimal specification of what constitutes "useful" activity or the desired final state.

Specific properties of the protocol, on the other hand, require specification of the particular service to be provided. Examples include reliable data transfer in TP, copying a file in FTP, and clearing a terminal display in a virtual terminal protocol. Definition of these features make up the bulk of service specifications.

On the other axis, partial correctness usually means that if the protocol service performs any action at all, it will be in accord with its specifications. For example, if TP delivers any messages, they will be to the correct destination, in the correct order, and without errors. Termination or progress means that the specified services will actually be completed in finite time. In the case of logical verification, which is the subject of this Note, it is sufficient to ascertain a finite time delay. In cases where the efficiency and responsiveness of the protocol must be verified, it is clearly necessary to determine numerically the expected time delay, throughput, etc.

VERIFICATION METHODS

Verification efforts to date have largely started from protocol specifications, without having any comprehensive service definition; therefore they have generally been shaped by the approach to protocol specification used. As a result, two major verification methods have evolved: reachability analysis for protocols expressed with state-transition models, and program proofs for protocols expressed in programming-language models.

Reachability analysis is based on exhaustively exploring the possible interactions of two (or more) entities within a layer. A composite or global state of the system is defined as a combination of the states of the cooperating protocol entities and the lower-layer service connecting them. From a given initial state, all possible transitions (user commands, timeouts, message arrivals) are generated, leading to a number of new global states. This process is repeated for

each of the newly generated states until no new states are generated (some transitions lead back to already generated states). For a given initial state and set of assumptions about the underlying protocol (the type of service it offers), this type of analysis determines all of the possible outcomes that the protocol may achieve. References 13 and 21 provide a clear exposition of this technique.

Reachability analysis is particularly straightforward to apply to transition models of protocols which have explicit states and/or state variables defined. It is also possible to perform a reachability analysis on program models by establishing a number of "break points" in the program that effectively define control states [65]. Symbolic execution (see below) may also be viewed as a form of reachability analysis.

Reachability analysis is well-suited to checking the general correctness properties described above, because these properties are a direct consequence of the structure of the reachability graph. Global states with no exits are either deadlocks or desired termination states. Similarly, situations where the processing for a receivable message is not defined or where the transmission-medium capacity is exceeded are easily detected. The generation of the global state space is easily automated, and several computer-aided systems for this purpose have been developed [39,65,94,98].

The major difficulty of this technique is "state explosion." The size of the global state space may grow rapidly with the number and complexity of protocol entities involved and the underlying layer's

services. Therefore, to keep the state model manageable (and comprehensible), in most cases only the major "control" variables are explicitly represented as states as described in Section II. Hence only the control portion of the protocol services is verified by reachability analysis. For example, state exploration of the ABP as in Refs. 13 and 20 can show the absence of deadlocks and can show that each message sent results in a message delivered, but the equality of the text of the input and output messages must be verified by other means, since the text of the message is not part of the explicit state. Techniques for dealing with this problem are discussed below.

The program-proving approach involves the usual formulation of assertions that reflect the desired correctness properties. Ideally, these would be supplied by the service specification, but as noted above, services have not been rigorously defined in most protocol work, so the verifier must formulate appropriate assertions on his own. The basic task is then to show (prove) that the protocol programs for each entity satisfy the high-level assertions. This often requires the formulation of additional low-level assertions at appropriate places in the programs [11,108].

A major strength of this approach is its ability to deal with the full range of protocol properties to be verified, rather than with only general properties. Ideally, any property for which an appropriate assertion can be formulated can be verified, but formulation and proof often require a great deal of ingenuity. Only modest progress has been made to date in the automation of this process.

While a large body of work on general program proof exists, several characteristics of protocols pose special difficulties in proofs. These include concurrency of multiple protocol modules and physical separation of modules so that no shared variables may be used. A further complication is that message exchange between modules may be unreliable, requiring methods that can deal with nondeterminism.

A particular form of proof that has been used for protocols with large numbers of interacting entities (e.g., routing protocols) has been called "induction on topology" [85]. The desired properties are first shown to be true for a minimum subset of the entities, and then an induction step is proved showing that if the properties hold for a system of N entities, they also hold for N+1 entities.

As with specification, a hybrid approach promises to combine the advantages of both techniques. By using a state model for the major states of the protocol, the state space is kept small, and the general properties can be checked by an automated analysis. Other properties, for which a state model would be awkward (e.g., sequenced delivery), can be handled by assertion proofs on the variables and procedures that accompany the state model. Such combined techniques are described in Refs. 9 and 12.

When an error is found by some verification technique, the cause must still be determined. Many transitions or program statements may separate the cause from the error which results, for example, when the acceptance of a duplicate packet at the receiver is caused by the too rapid reuse of a sequence number at the sender. In some cases, the

protocol may be modeled incorrectly, or the correctness conditions (i.e., service specification) may be formulated incorrectly. In other cases, undesired behavior may result from transmission-medium properties that were not expected when the protocol was designed (e.g. reordering of messages in transit). Even when an automated verification system is available, considerable human ingenuity is required to understand and repair any errors that are discovered.

Another, more recent approach to achieving correct protocols may be described as development of design rules or sufficiency criteria. In this method, design rules are formulated which are sufficient to guarantee that a set of interacting entities designed according to the rules will obey certain properties. As an example, to guarantee completeness, each Send transition added to one entity should have a corresponding Receive transition added to its partner(s). Further examples are provided in Ref. 136. While the use of such rules avoids the need for any post-design verification of the properties covered, rules for only a limited set of properties have been developed to date.

A related approach that also builds correctness into the design process may be described as a "transformational" method. In this approach, a set of previously defined transformations are applied to the service specification, each one carrying it closer to the desired protocol design. Each transformation has been verified to maintain "equivalence" of the system, so the final result is guaranteed to properly implement the original specifications. An example of such a transformation might be the converting of a copy or assignment operation at the service level into a pair of sending and receiving processes that

use a positive acknowledgment/retransmission protocol between themselves. While this technique has been applied to other software design problems, its use with protocols is still untested.

METHODS FOR REDUCING COMPLEXITY

A major difficulty for protocol verification by any method is the complexity of the global system of interacting protocol entities, also termed "state space explosion." The following methods may be used to reduce this complexity and facilitate verification (several of these points are due to Bochmann [14]).

(1) Partial description and verification: Depending on the description method used, only certain aspects of the protocol are described. This is often the case for transition diagram descriptions which usually capture only the rules concerning transitions between major states, ignoring details of parameter values and other state variables.

(2) Choosing large units of actions: State space explosion may result from the interleaving of the actions executed by the different entities. For example, the preparation and sending of a protocol data unit by an entity may usually be considered an indivisible action which proceeds without interaction with the other entities of the system. The execution of such an action may be considered a single "transition" [75] in the global protocol description.

A particularly powerful application of this idea is that of considering only states where the transmission medium is empty. Such an

"empty medium abstraction" [19] is justified when the number of messages in transit is small. In this case, previously separate sending and receiving or sending and loss transitions of different entities can be combined into single joint transitions of both entities.

(3) Decomposition into sublayers: The decomposition of the protocol of a layer into several sublayer protocols simplifies the description and verification, because the protocol of each sublayer may be verified separately. An example of this idea is the decomposition of HDLC into the sublayers of bit-stuffing, check-summming, and elements of procedure, and the division of the latter into several components as described in Ref. 19.

(4) Classifying states by assertions: Assertions which are predicates on the set of all possible system states may be formed. These define a set of states for which each predicate is true. One may then consider each set of states collectively in reachability analysis instead of considering individual states. By making an appropriate choice of predicates (and therefore classes of states), the number of cases to be considered may be reduced considerably. This method is usually applied for protocol descriptions involving program variables. Typically, the assertions depend on some variables of the entities and the set of messages in transit (through the layer below) [108,11].

Symbolic execution [24] may be viewed as a form of reachability analysis which reduces the number of distinct global states by using symbolic variables to define large classes of states that may be considered together. In symbolically executing the global system, new

states are created at each decision point. Each new state corresponds to a possible outcome of the decision, with predicates associated with it reflecting the conditions necessary for this state to be reached. Whenever possible, the predicates are simplified (symbolically), using axioms that define the types of the variables (e.g., integers, strings, etc.). As an example of the savings which may result, instead of treating all possible values of a sequence countervariable explicitly as different states, it may be possible to consider only the three conditions where the variable is "less than", "equal to", or "greater than" some symbolic value.

(5) Focusing search: Instead of generating all possible states, it is possible to predetermine potential global states with certain properties (e.g., deadlocks) and then check whether they are actually reachable [39].

(6) Automation: Some steps in the analysis process may be performed by automated systems, a few of which have been developed [24,39,53,65,94,98,125]. However, the use of these systems is not trivial, and much work goes into representing the protocol and service in a form suitable for analysis. Human intervention is needed in many cases for distinguishing between useful and undesired loops, or for guiding the proof process.

V. USES OF FORMAL TECHNIQUES

This section presents a (certainly incomplete) list of cases where formal methods were successfully used for designing data communication and computer network protocols. In some cases, the formal description was made after the system design was essentially finished, and served for an additional analysis of correctness and efficiency or as an implementation guide. In other cases, the formal description was used as a reference document during the system design. The references, indexed by keyword, provide further details on these and other examples.

The end-to-end transport protocol of the French computer network Cyclades was first specified in a semiformal manner, using a high-level programming language. This specification was the basis for the different protocol implementations in different host computers. Some of these implementations were obtained through a description in a macro-language, derived from the original protocol specification [138]. The same specification was also the basis for simulation studies which provided valuable results for the protocol validation and performance evaluation [81,X48]. A formalized specification of the protocol has also been given using a hybrid model with state machines augmented by context information and processing routines [40].

The procedures for the internal operation of the Canadian public data network Datapac were described by a semiformal method using state diagrams and a high-level programming language for the specification of the communicating entities [X49]. This description was very useful for doing semiformal verifications of the protocols during the design phase

and served as a reference document during the implementation and testing phases of the system development.

A formal description method was used during the design of several interface standards for the interconnection of minicomputers with measurement and instrumentation components [126,127]. The relatively concise description of the protocols was used as a means for communication between the members of the standards committees and for the verification of the design. It is also part of the final standard documents.

The HDLC link protocol has been specified with a regular grammar model [68] that incorporated an indexing technique to accommodate sequence numbering. The same protocol has also been specified with a hybrid model combining state transitions with context variables and high-level language statements [19]. The latter technique also heavily employed decomposition to partition the protocol into seven separate components and was used in obtaining an implementation of the HDLC link-level procedures of X.25 [18].

IBM's SNA has been specified with a hybrid model using state machines augmented by context information and processing routines [14,73]. Hierarchical decomposition is heavily used to create a large number of more manageable modules.

Call establishment in the CCITT X.21 protocol has been modeled with a state-transition-type model and analyzed with a form of reachability analysis [131]. The analysis checked for general correctness properties of completeness and deadlock, and uncovered a number of completeness

errors (i.e., a protocol module received a message for which no processing was defined).

Virtual circuit establishment in the CCITT X.25 protocol has been modeled with a state transition model and analyzed with a manual reachability analysis [13]. The analysis showed that the CCITT specification was ambiguous and that several cycles with no useful progress could persist if the protocol once entered certain unsynchronized states.

Connection establishment in the transport protocol for the ARPANET (TCP [X29]) has been partially modeled with a hybrid state-transition model and validated with a manual reachability analysis [114]. An automated reachability analysis [65] was also used on a simplified model and revealed an error in sequence-number handling and incorrect modeling of the transmission medium.

A general-purpose data-transfer protocol has been modeled with a high-level programming language and verified using manual program-proving techniques [11,108]. (However, Hajek [65] has identified several flaws in the verification in [11].) The protocol includes window-based flow control and a large but finite sequence number space.

A simple data-transfer protocol has been analyzed with a transition model augmented with time constraints to show that proper data transfer requires certain time constraints to be maintained between retransmission, propagation, and processing times [88].

A simplified version of the ARPANET communications subsystem has been modeled with a high-level programming language and verified using partially automated program-proving techniques [53,55]. Program modules can be both comprehensively verified in advance and checked against their specifications at run time for the particular inputs that occur. A complete software engineering system called GYPSY provides a unified language for expressing both specifications and programs, so that high-level specifications in the design can be progressively refined into detailed programs. A management system maintains the implementation and verification status of all system components under development.

Connection establishment between a requester and a shared server process (the ARPANET Intial Connection Protocol) has been modeled with a state-transition model and analyzed by an automated reachability analysis [94]. The analysis showed that one of a pair of simultaneous requests for service might be rejected. A revised version of the protocol was shown to eliminate this error. The same analysis technique was also used to validate a simple data-transfer protocol.

A basic data-transfer protocol has been specified using a formal language model (BNF) to describe the syntax of both interactions between protocol modules and the detailed structure of individual messages [121]. This specification may be directly used to drive a "recognizer" for the language (protocol). Additional processing routines to perform the "semantics" of the protocol must be added to the grammar-driven recognizer.

A simple link protocol has been specified with a specialized flowchart-type model, along with a means for automatically converting the flowcharts to equivalent data and control networks that are directly realizable in hardware [60].

VI. CONCLUSIONS

Protocol specification requires a clear definition of both the services to be provided by a given protocol layer and the protocol entities within the layer that cooperate to provide the services. Design verification then consists of showing that the interaction of entities is indeed adequate to provide the specified services, while implementation verification consists of showing that the implementations of the entities satisfy their (abstract) protocol specifications. A useful subset of design verification may be described as verification of "general properties" such as deadlock, looping, and completeness. These properties may be checked for most protocols without requiring any particular service specification.

Most protocol design and analysis work to date has proceeded without a comprehensive specification of the services to be provided by a protocol to its users. Hence a major focus of our work has been to explore techniques for formally specifying protocol services. Two major approaches from the general software-specification domain were identified and applied to a set of example protocols: an abstract machine model that defines operations that may be invoked, and an agent model where the service is an active process with inputs and outputs.

Machine models successfully handle services that can be defined in terms of individual operations with specified effects on the state of the machine. These models provide a convenient means for handling exception conditions, but they do not easily accommodate several necessary aspects of protocol service specifications. Assertions about

sequences of operations are outside the model. Since they have no explicit output, only "polling" and not "prodding"-type interfaces with the user can be defined. In addition, it is necessary to make assumptions about the behavior of the machine's users in order to talk about termination. To overcome some of these problems, it appears necessary to include "users" who are really part of the service, or to include cyclic processes in the specification.

Agent models handle these difficulties more successfully. Transition-type agents retain most of the benefits of machine models in defining the handling of individual inputs. Buffer-history-type agents facilitate assertions about sequences of operations but have difficulty with state-oriented service features and exceptions. It appears that the two sorts of specification are useful for different aspects of service--transitions for state-oriented features like connection establishment, and buffer histories for data transfer.

Most verification efforts to date fall into either state-exploration or program-proving categories. The former type is straightforward to accomplish and has been automated to some extent, but it can deal only with the major states or "control" aspects of a protocol. The latter technique can in principle verify all features of a specification, but a great deal of ingenuity is required to construct proofs of even simple systems.

Several newer techniques promise to reduce some of these difficulties. "Unified" methods use state exploration of major states to facilitate program proofs of additional properties involving the

other state variables of the protocol. Symbolic execution exploits the ability to group classes of system states in order to minimize the size of the state space that must be explored. Both design rules and transformation methods promise to eliminate the need for post-design verification altogether by constraining the design process to follow correct paths. All of these techniques require further research before their effectiveness can be evaluated.

It is clear from the references that the use of more formal techniques has already had a positive impact on the protocol design process. State-exploration techniques for verifying general properties are fairly well understood and have the potential for routine application in the near future. Use and development of more powerful verification techniques require a high level of skill in formal methods and must still be considered research problems. A great deal of work remains to be done in developing techniques that are routinely and widely applicable.

BIBLIOGRAPHY

The following references are the results of a comprehensive literature search for items relevant to protocol specification and verification, ending in July 1979. They were compiled with the help of the working group on Formal Protocol Specification and Verification of IFIP Technical Committee 6.1, chaired by John Day. The items are listed alphabetically by author. Each item includes a list of key phrases describing its subject, chosen from the following list. An index of all items relevant to each key phrase follows the reference citations.

KEY PHRASES

State machine and related models
Petri Net and related models
Formal language/grammar models
Programming language models
Graphical techniques

Simple data transfer protocols
Link control protocols
Transport protocols
High-level protocols
N-party protocols
Synchronization protocols

Formal architecture specification
Formal service specification
Formal protocol specification
Formal interface specification
Program proof
State exploration
Implementation
Complete development methodology
Testing
Sufficiency criteria and design rules
Simulation
Symbolic execution
Resiliency
Efficiency
Informal case analysis
Survey
Design methodology
Combined models

1. Azema, P., Ayache, J. M., and Berthomieu, B., Design and Verification of Communication Procedures: A Bottom-Up Approach, Proc. of Third International Conf. on Software Engineering, 1978. [Petri Net and related models, Simple data transfer protocols, Transport protocols, Complete development methodology, Implementation]

2. Azema, P., Valette, R., and Diaz, M., Petri Nets as a Common Tool for Design Verification and Simulation, 13th Design Automation Conf. 1976. [Petri Net and related models, Simulation, Sufficiency criteria and design rules]

3. Bachman, Charles, Data Structure Diagrams, Data Base Quarterly, ACM-SIGBDP Vol. 1, No. 2, 1969. [Formal architecture specification, Formal language/grammar models, Programming language models, Formal protocol specification, Complete development methodology]

4. Bachman, Charles, Architecture Definition Technique: Its Objective, Theory, Process, Facilities and Practice, Tech. Rpt. Honeywell Information Systems, Billerica, MA, 1978. [Formal architecture specification, Formal language/grammar models, Programming language models, Formal protocol specification, Complete development methodology]

5. Bartirssek, W., and Parnas, D. L., Using Traces to Write Abstract Specifications of Software Modules, Report No.TR77-012, Univ. of N. Carolina; Chapel Hill, NC, 1977.

6. Bartlett, K. A., Scantlebury, R. A., and Wilkinson, P. T., A Note on Reliable Full-Duplex Transmission over Half-Duplex Links, CACM Vol. 12, No. 5, May 1969. [State machine and related models, Simple data transfer protocols, Informal case analysis]

7. Belsnes, D., Single Message Communication, IEEE Trans. Comm. Vol. COM-24, No. 2, February 1976. [Simple data transfer protocols, Informal case analysis]

8. Benice, R. J., and Frey, A. H., An Analysis of Retransmission Systems, IEEE Trans. on Comm. Technology, December 1964. [State machine and related models, Simple data transfer protocols, Informal case analysis, Efficiency]

9. Birke, D. M., State-Transition Programming Techniques and Their Use in Producing Teleprocessing Device Control Programs, Proc. 2nd Symp. on Problems in the Optimization of Data Communications, 1971. []

10. Bjorner, Dines, Finite State Automation--Definition of Data Communication Line Control Procedures, Fall Joint Computer Conference, AFIPS 1970. [State machine and related models, Link control protocols, Implementation, Formal protocol specification]

11. Bochmann, G. V., Logical Verification and Implementation of Protocols, Proceedings of the 4th Data Comm. Symp., IEEE, 1975. [State machine and related models, Programming language models, Link control protocols, Program proof, Formal protocol specification, Implementation]

12. Bochmann, G. V., Communication Protocols and Error Recovery Procedures, ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, 1975. [State machine and related models, Simple data transfer protocols, State exploration, Formal protocol specification]

13. Bochmann, G. V., Finite State Description of Communication Protocols, Proc. Symp. on Computer Comm. Protocols, Liege, Belgium, 1978. Also in Computer Networks (North Holland), Vol. 2, No. 4/5, 1978. [State machine and related models, Simple data transfer protocols, Link control protocols, Survey, Transport protocols, State exploration]

14. Bochmann, G. V., Specification and Verification of Computer Communication Protocols, Dept. d' I.R.O., Univ. de Montreal, Rpt. 294. Montreal, Canada, 1978. [Survey]

15. Bochmann, G. V., Synchronization in Distributed System Modules, Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks, August 1979. [Combined models, N-party protocols, Sufficiency criteria and design rules, Synchronization protocols]

16. Bochmann, G. V., Distributed Synchronization and Regularity, Computer Networks, Vol. 3, No. 1, 1979. [Combined models, N-party protocols, Sufficiency criteria and design rules, Synchronization protocols]

17. Bochmann, G. V., Towards an Understanding of Distributed and Parallel Systems, Dept. d'I.R.O., Univ. de Montreal, Publ. No. 317, Montreal, Quebec, Canada, 1978. [Combined models, N-party protocols, Formal protocol specification, Sufficiency criteria and design rules, Program proof]

18. Bochmann, G. V., and Joachim, T., Development and Structure of an X.25 Implementation, IEEE Trans. on Software Engineering, Vol. SE-5, No. 5, September 1979. [Combined models, Link control protocols, Transport protocols, Implementation, Complete development methodology]

19. Bochmann, G. V., Chung, R. J., A Formalized Specification of HDLC Classes of Procedures, Tech. Report No. 265, Dept. d'I. R. O., Univ. Montreal, 1976. Also in Proc. National Telecomm. Conf., 1977. IEEE. Reprinted in Advances in Computer Communication, W. Chu (ed.) 1979. [Combined models, Link control protocols, Formal protocol specification]

20. Bochmann, G. V., and Gecsei, J., A Unified Method for the Specification and Verification of Protocols, Proc. IFIP Congress, 1977. [Combined models, Simple data transfer protocols, Formal protocol specification, State exploration, Program proof]

21. Bochmann, G. V., and Vogt, F. H., Message Link Protocol - Functional Specification, ACM SIGCOMM Computer Communication Review, Vol. 9, No. 2, 1979. [Combined models, Transport protocols, Formal protocol specification, Formal service specification]

22. Bochmann, Gregor V., A General Transition Model for Protocols and Communication Services, Rpt. 322, D. I. R. O., Univ. Montreal, Canada, 1979. Also to appear in IEEE Trans. Comm. [Combined models, Link control protocols, Transport protocols, Formal protocol specification, Formal service specification, Program proof, State exploration]

23. Boebert, W. E., Franta, W. R., and Berg, H., NPN: A Finite State Specification Technique for Distributed Software, Proc. Conf. on Specifications of Reliable Software, IEEE, 1979. [Implementation, Informal case analysis]

24. Brand, D., and Joyner, W. H., Verification of Protocols Using Symbolic Execution, Proc. Symp. on Computer Comm. Protocols, Liege, Belgium, 1978. Also in Computer Networks (North Holland), Vol. 2, No. 4/5, 1978. [Formal language/grammar models, Simple data transfer protocols, Symbolic execution]

25. Brand, Daniel, Algebraic Simulation Between Parallel Programs, IBM Tech. Rpt. RC 7206, Thomas J. Watson Research Center, Yorktown Heights, NY, 1978. [Formal language/grammar models, Simple data transfer protocols, Symbolic execution]

26. Bredt, T. H., Analysis of Operating System Interactions, Proc. AICA Congress on Theoretical Informatics, Univ. Pisa, 1973. [State machine and related models, State exploration]

27. Bremer, J., Representation axiomatique d'un protocole, Description du programme REDUCTION, S.A.R.T. 76/19/10, September 1976, Univ. de Liege, Belgium. [State machine and related models, Programming language models, Transport protocols, State exploration, Program proof]

28. Bremer, J., Representation axiomatique d'un protocole, Notice d'utilisation du programme REDUCTION, S.A.R.T. 76/18/10, October 1976, Univ. de Liege, Belgium. [State machine and related models, Programming language models, Transport protocols, State exploration, Program proof]

29. Bremer, J., Verification de la logique d'un protocole, Description du programme VERIFY, S.A.R.T. 77/03/10 January 1977, Univ. de Liege, Belgium. [State machine and related models, Programming language models, Transport protocols, State exploration, Program proof]

30. Bremer, J., Verification de la logique d'un protocole, Notice d'utilisation du programme VERIFY, SART 76/20/10 December 1976, Univ. de Liege, Belgium. [State machine and related models, Programming language models, Transport protocols, State exploration, Program proof]

31. Campbell, R. H., Habermann, A. N., The Specification of Process Synchronization by Path Expressions, Proc. Int. Symp. Held at Rocquencourt on Operating Systems, 1974. []

32. Chandersekaran, C. S., and Shankar, K. S., Towards Formally Specifying Communications Switches, Proc. NBS Conf. on Computer Networks, 1976. [Formal protocol specification, Implementation]

33. Chen, R. C., Represenation of Process Synchronization, Proc. ACM SIGCOMM/SIGOPS Interprocess Comm. Workshop, 1975. [Petri Net and related models]

34. Chow, T. S., Analysis of Software Design Modeled by Multiple Finite State Machines., Proc. COMPSAC, IEEE, 1978. [State machine and related models, State exploration, Sufficiency criteria and design rules]

35. Chung, Paul, and Gaiman, Barry, Use of State Diagrams to Engineer Communications Software, Proc. Third Int. Conf. on Software Engineering, 1978. [State machine and related models, Link control protocols, Transport protocols, Implementation]

36. Danthine, A., Petri Nets for Protocol Modelling and Verification, Proc. European Symposium on Data Comm., Budapest, 1977. [Petri Net and related models, Simple data transfer protocols, Formal protocol specification, State exploration]

37. Danthine, A. S., and Bremer, J., Communication Protocols in a Network Context, Proc. ACM SIGCOMM/SIGOPS Interprocess Comm. Workshop, 1975. [State machine and related models, Programming language models, Implementation, Formal protocol specification, Complete development methodology, Transport protocols]

38. Danthine, A., and Bremer, J., Definition, Representation, et Simulation de Protocoles dans un Contexte Reseaux, Journ. Intern. Mini-ordinateurs et Transm. de Donnes, January 1975. [State machine and related models, Programming language models, Implementation, Formal protocol specification, Complete development methodology, Transport protocols, Combined models]

39. Danthine, A., and Bremer, J., Modelling and Verification of End-to-End Transport Protocols, Univ. de Liege, Belgium, S. A.R.T. 77/11/13, 1977. Also in Computer Networks (North Holland), Vol. 2, Nos. 45, 1978. [State machine and related models, Programming language models, Implementation, State exploration, Program proof, Transport protocols, Combined models]

40. Danthine, A., and Bremer, J., An Axiomatic Description of the Transport Protocol of CYCLADES, Prof. Conf. on Computer Networks and Teleprocessing, 1976. [State machine and related models, Programming language models, Formal protocol specification, Implementation, Complete development methodology, Transport protocols, Combined models]

41. de Meer, J., and Henken, G., Spezifikation und portable Implementierung des X.25 packet level Protokolls, Diplomarbeit am Fachbereich Informatik, TU Berlin, 1978.

42. Donnelley, James, and Yeh, Jeffrey, Interaction Between Protocol Levels in a Prioritized CSMA Broadcast Network, Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, 1978. Also in Computer Networks, Vol. 3, No. 1, 1979. [Link control protocols, Efficiency]

43. Ellis, D. J., Formal Specification for Packet Communication Systems, Tech. Rpt. MIT, Lab for Computer Science, MIT/LCS/TR-189, 1978.

44. Fayolle, G., Gelenbe, E., and Piyolle, G., An Analytic Evaluation of the "Send and Receive" Protocol, IEEE Trans. on Comm., Vol. COM-26, No. 3, March 1978. [Programming language models, Simple data transfer protocols, Efficiency]

45. Feiertag, Richard J., Shostak, Robert E., and Lamport, Leslie B., Verification of Communications-Oriented Language Problems, SRI International, 1978. [Programming language models, N-party protocols, Formal protocol specification, Program proof]

46. Feldman, J. A., Synchronizing Distant Cooperating Processes, Tech. Report TR26, Computer Science Dept., Univ. of Rochester, October 1977. [State machine and related models, Programming language models, Synchronization protocols, Program proof]

47. Finn, S. G., Resynch Procedures and a Fail-Safe Network Protocols, IEEE Trans. Comm., Vol. COM-27, No. 5, 1979. [Formal language/grammar models, Link control protocols, Transport protocols, Sufficiency criteria and design rules]

48. Fong, N. P., NTS - A Protocol Test and Development System, NBS Computer Networking Symposium, IEEE, December 1978. [State machine and related models, Formal language/grammar models, Link control protocols, State exploration, Implementation, Testing]

49. Furtek, F. C., The Logic of Systems, MIT LCS TR-170, 1976. [Petri Net and related models]

50. Gallager, R., A Minimum Delay Algorithm Using Distributed Computation, IEEE Trans. on Comm., Vol. COM-25 No. 1, January 1977. [Programming language models, N-party protocols, Efficiency, Program proof]

51. Gilbert, P., and Chandler, W. J., Interference Between Communicating Parallel Processes, CACM 15, 6, June 1972. [State machine and related models, State exploration]

52. Goldman, Barry, Deadlock Detection in Computer Networks (Master's Thesis), Computer Science Dept., MIT, Cambridge, MA, 1977. [Formal language/grammar models, Programming language models, High-level protocols, Formal protocol specification, Sufficiency criteria and design rules, Simulation, Symbolic execution]

53. Good, D. I., Constructing Verified and Reliable Communications Processing Systems, ACM SIGSOFT Softw. Eng. Notes Vol. 2, No. 5, October 1977, p. 8-14. [Programming language models, Simple data transfer protocols, Formal protocol specification, Program proof, Implementation, Complete development methodology, Testing]

54. Good, D. I. (ed.), Constructing Verifiably Reliable and Secure Communications Processing Systems, ICSCA-CMP-6, Univ. of Texas, Austin, 1977. [Programming language models, Simple data transfer protocols, Formal protocol specification, Program proof, Implementation, Complete development methodology, Testing, Synchronization protocols]

55. Good, Don I., and Cohen, Richard, Verifiable Communications Processing in GYPSY, ICSCA-CMP-11, Univ. of Texas, Austin, June 1978. Also in Proc. 17th COMPCON, IEEE, 1978. [Programming language models, Simple data transfer protocols, Formal protocol specification, Program proof, Implementation, Complete development methodology, Testing]

56. Good, Don I., Cohen, Richard M., and Keeton-Williams, Jim, Principles of Proving Concurrent Programs in GYPSY, ICSCA-CMP-15, Univ. of Texas, Austin, January 1979. [Programming language models, Simple data transfer protocols, Formal protocol specification, Program proof, Implementation, Complete development methodology, Testing]

57. Gord, E. P., Hopwood, M. D., and Rowe, L.A., Language Constructs for Message Handling in Decentralized Programs, Proc. ACM National Conf., 1974. [Formal language/grammar models]

58. Gouda, Mohamed G., Protocol Machines: Towards a Logical Theory of Communication Protocols (Ph.D. Thesis), Univ. of Waterloo, Waterloo, Ontario, Cananda, 1977. [State machine and related models, Petri Net and related models, Simple data transfer protocols, Link control protocols, N-party protocols, Formal protocol specification, State exploration, Implementation, Complete development methodology, Efficiency]

59. Gouda, Mohamed G., and Manning, Eric G., On the Modelling, Analysis, and Design of Protocols--A Special Class of Software Structures, Proc. 2nd Int. Conf. on Software Engineering, 1976. [State machine and related models, Simple data transfer protocols, Link control protocols, Formal protocol specification, State exploration, Implementation, Complete development methodology]

60. Gouda, Mohamed G., and Manning, Eric G., Protocol Machines: A Concise Formal Model and Its Automatic Implementation, Proc. 3rd Int. Conf. Computer Comm., 1976. [State machine and related models, Simple data transfer protocols, Link control protocols, Formal protocol specification, Implementation]

61. Gouda, Mohamed G., and Manning, Eric G., Toward Modular Hierarchical Structures for Protocols in Computer Networks, Proc. COMPCON, IEEE, 1976. [State machine and related models, Simple data transfer protocols, Link control protocols, Formal protocol specification, Implementation, Complete development methodology]

62. Grotsch, E., and Schlurick, T., Protokollbeschreibung und Protokollmaschine-Entwurf fur Rechnerkopplungen mit Hilfe einer Datenstrukturorientierten Entwurfsmethode, Siemens AG, Erlangen, 1978. [Graphical techniques, Implementation]

63. Gunther, K. D., Prevention of Buffer Deadlocks in Packet Switching Networks, IIASA Workshop on Data Comm., Laxenburg, Austria, 1975.

64. Habermann, A. N., Path Expressions, Tech. Rpt. Computer Science Dept., Carnegie-Mellon University 1975.

65. Hajek, J., Automatically Verified Data Transfer Protocols, Proc. Int. Conf. on Computer Comm., Kyoto, Japan, 1978. [Programming language models, Simple data transfer protocols, Link control protocols, Transport protocols, State exploration, Synchronization protocols]

66. Hajek, Jan, Self-synchronization and Blocking in Data Transfer Protocols, Eindhoven Univ. of Technology, The Netherlands, April 1977. [Programming language models, Simple data transfer protocols, Synchronization protocols, N-party protocols, State exploration]

67. Hajek, Jan, Protocols Verified by APPROVER, SIGCOM Computer Comm. Rev., Vol. 9, No. 1, January 1979. [Programming language models, Simple data transfer protocols, Link control protocols, Transport protocols, State exploration, Synchronization protocols]

68. Harangozo, J., Protocol Definition with Formal Grammars, Proc. Symp. on Computer Comm. Protocols, Liege, Belgium, 1978. [Formal language/grammar models, Link control protocols, Formal protocol specification]

69. Harangozo, J., An Approach to Describing a Data Link Level Protocol with a Formal Language, Proc. 5th Data Comm. Symp., IEEE, 1977. [Formal language/grammar models, Link control protocols, Formal protocol specification, Implementation]

70. Hewitt, C., and Baker, H., Laws for Communicating Parallel Processes, Proc. IFIP Congress, 1977. [Formal language/grammar models, Programming language models]

71. Hoffman, H. J., On Linguistic Aspects of Communication Line Control Procedures, IBM Report RZ 345, 1970. [Formal language/grammar models, Link control protocols]

72. Hopper, K., Kugler, H. J., and Unger, C., Abstract Machines Modelling Network Control Systems, Computer Science Dept., Massey Univ. Palmerston North, New Zealand. 1977. []

73. IBM, Systems Network Architecture Format and Protocol Reference Manual: Architectural Logic, IBM Report SC30-3112-1, June 1978. [State machine and related models, Link control protocols, Formal protocol specification, State exploration]

74. Ideguchi, T., et al., An Abstract Protocol Machine for Communications and Its Application, TGEC IECE, 1977 (in Japanese). [State machine and related models, Programming language models, Simple data transfer protocols, Link control protocols, Transport protocols, Formal protocol specification, Implementation]

75. Keller, R. M., Formal Verification of Parallel Programs, CACM, Vol. 19, No. 7, July 1976. [Petri Net and related models, Formal protocol specification, Program proof]

76. Krogdahl, S., Verification of a Class of Link-Level Protocols, BIT, Vol. 18, 1978. [Programming language models, Link control protocols, Program proof, Sufficiency criteria and design rules]

77. Kuemmerle, K., and Port, E., Towards a Methodology for Representing Computer Network Architectures, Proc. Int. Conf. on Computer Comm., Kyoto, Japan, 1978. [Graphical techniques, State machine and related models, Formal architecture specification]

78. Lamport, Leslie, Time, Clocks, and the Ordering of Events in a Distributed System, CACM, Vol. 21, No. 7, Jul. 1978. [Synchronization protocols, Sufficiency criteria and design rules]

79. Lamport, Leslie, The Implementation of Reliable Distributed Multiprocess Systems, Computer Networks, Vol. 2, No. 2, May 1978. [Programming language models, Resiliency, Sufficiency criteria and design rules, Program proof, N-party protocols]

80. Le Moli, G., A Theory of Colloquies, First European Workshop on Computer Networks, 1973. Also in Alta Frequenza, Vol. 42, No. 10, 1973. [State machine and related models, Formal protocol specification, Simple data transfer protocols]

81. LeLann, Gerard, and LeGoff, Herve, Verification and Evaluation of Communication Protocols, Computer Networks, Vol 2, No. 1, 1978. [Programming language models, Transport protocols, Simulation]

82. Lynch, W. C., Reliable Full-Duplex Transmission Over Half-Duplex Telephone Lines, CACM, Vol. 11, No. 6, June 1968. [State machine and related models, Simple data transfer protocols, Informal case analysis]

83. Merlin, P., A Study of the Recoverability of Computing Systems, Tech Report #58 (Ph.D. Thesis), Computer Science Dept. Univ. of Calif., Irvine, 1974. [Petri Net and related models, Simple data transfer protocols, State exploration, Sufficiency criteria and design rules]

84. Merlin, P. M., A Methodology for the Design and Implementation of Communication Protocols, IEEE Trans. Comm. COM-24, 6 June 1976. [Petri Net and related models, Simple data transfer protocols, State exploration]

85. Merlin, P. M., Specification and Validation of Protocols, accepted for publication in IEEE Trans. Comm. [Survey]

86. Merlin, P. M., and Farber, D. J., Recoverability of Communication Protocols - Implications of a Theoretical Study, IEEE Trans. on Comm., Vol. COM-24, 1976. [Petri Net and related models, Simple data transfer protocols, State exploration]

87. Merlin, P. M., and Segall, A., A Failsafe Distributed Routing Protocol, Report EE 313, Technion, Haifa, Israel, 1978. [Petri Net and related models, N-party protocols, State exploration]

88. Merlin, P., and Farber, D. J., Recoverability of Communications Protocols, IBM Research Rpt. RC 5416, (23664) 1975. [Petri Net and related models, Simple data transfer protocols, State exploration]

89. Mezzalira, L., and Scheiber, F. A., Designing Colloquies, First European Workshop on Computer Networks, 1973. [State machine and related models, Programming language models, Simple data transfer protocols, Transport protocols, Formal protocol specification]

90. Morgan, D. E., and Taylor, D. J., A Survey of Methods of Achieving Reliable Software, Computer, Vol. 10, No.2, 1977. [Program proof, Testing]

91. Morino, K., et al., On the Specification of High Level Data Link Control Procedures, Proc. 17th Annual Convention of IPSJ, 1976 (in Japanese). [State machine and related models, Link control protocols, Implementation, Formal protocol specification]

92. Neumann, Peter G., Boyer, Robert S., and Levitt, Karl N., A Provably Secure Operating System: the System, its Applications, and Proofs, SRI International, February 1977. [Programming language models, Formal protocol specification, Program proof, Implementation, Complete development methodology]

93. Pease, M., Shostak, R., and Lamport, L., Reaching Agreement in the Presence of Faults, to appear in JACM. []

94. Postel, J. B., A Graph Theory Analysis of Computer Communications Protocols, (Ph.D. Thesis) UCLA-ENG7410, 1974. [Graphical techniques, State machine and related models, Transport protocols, State exploration]

95. Postel, Jon B., and Farber, D., Graph Modeling of Computer Communications Protocols, Proc. 5th Texas Conf. on Computing Systems, Austin, Texas, 1976. [Petri Net and related models, Simple data transfer protocols, State exploration]

96. Robinson, L., and Levitt, K. N., Proof Techniques for Hierarchically Structured Programs, CACM, Vol. 20, No. 4, April 1977. [Programming language models, Formal protocol specification, Program proof, Implementation, Complete development methodology]

97. Robinson, Lawrence, The HDM Handbook, Vol. I: The Foundations of HDM, SRI International, June 1979. [Programming language models, Formal protocol specification, Program proof, Implementation, Complete development methodology]

98. Rudin, H., West, C. H., and Zafiropulo, P., Automated Protocol Validation: One Chain of Development, Proc. Symp. on Computer Comm. Protocols, Liege, Belgium, 1978. Also in Computer Networks (North Holland), Vol. 2, Nos. 4/5, 1978. [State machine and related models, Link control protocols, State exploration]

99. Rusbridge, R. E., and Langsford, A., Formal Representation of Protocols for Computer Networks, Report AFRE-R-7826, UKAEA, Harwell, England, 1974.

100. Schindler, S., de Meer, J., and Henken, G., Formal Specification and Portable Implementation of the X.25 Packet Level Protocol, to be published. [State machine and related models, Link control protocols, Transport protocols, Formal protocol specification, Implementation]

101. Schindler, S., Didier, J., and Steinacker, M., Design and Formal Specification of an X.25 Packet Level Protocol Implementation, Technische Univeritat Berlin, Proc. COMPSAC, IEEE, 1978. [State machine and related models, Link control protocols, Transport protocols, Formal protocol specification, Implementation]

102. Schindler, S., de Meer, J., and Henken, G., Flow Control in the X.25 Packet Level Protocol - A Formal Description, Workshop uber Rechnernetze und Datenfernverarbeitung, Berlin, 1978. [State machine and related models, Link control protocols, Transport protocols, Formal protocol specification, Implementation]

103. Schindler, S., Didier, J., and Steinacker, M., Design and Formal Specification of an X.25 Packet Level Protocol Implementation, Proc. COMPSAC Conference, 1979. []

104. Schreiber, Fabio A., Some Linguistical Problems about Colloquies, ACM SIGCOMM Computer Comm. Review, 1975. [State machine and related models, Programming language models, Formal protocol specification, Transport protocols]

105. Shankar, C. S., and Chandersekaran, C. S., Data Flow, Abstraction Levels and Specifications for Communications Switiching Systems, 2nd Int. Conf. on Software Engineering, 1976. [Programming language models, Link control protocols, Transport protocols, Informal case analysis]

106. Siedler, J., A Method of Analysis of Discrete Communication Feedback Systems, Information and Control, Vol. 29, No. 2, October 1975. [State machine and related models, Graphical techniques, Simple data transfer protocols, Link control protocols, Transport protocols]

107. Silverberg, Brad A., Robinson, Lawrence, and Levitt, Karl N., The HDM Handbook, Vol. II: The Languages and Tools of HDM, SRI International, June 1979. [Programming language models, Formal protocol specification, Program proof, Implementation, Complete development methodology]

108. Stenning, N. V., A Data Transfer Protocol, Computer Networks, Vol. 1, No. 2, September 1976. [Programming language models, Simple data transfer protocols, Program proof]

109. Stenning, N. V., Definition and Verification of Computer Network Protocols (Ph.D. Thesis), Univ. of Sussex, England, 1978. [Programming language models, Transport protocols, Program proof]

110. Sundstrom, R. J., Formal Definition of IBM's System Network Architecture, Proc. Nat. Telecomm. Conf., December 1977. [State machine and related models, Formal protocol specification, Implementation]

111. Sunshine, C. A., Interprocess Communication Protocols for Computer Networks, Tech. Rpt. #105 (Ph.D. Thesis), Digital Systems Lab, Stanford Univ., 1975. [State machine and related models, Transport protocols, State exploration, Program proof, Sufficiency criteria and design rules, Combined models]

112. Sunshine, C. A., Survey of Communication Protocol Verification Techniques, The Rand Corporation, P-5725, 1976. Also in Proc. NBS Symp. on Computer Networks, IEEE, 1976. [Survey]

113. Sunshine, C. A., Survey of Protocol Definition and Verification Techniques, Proc. Symp. on Computer Network Protocols, Liege, Belgium, 1978. Also in Computer Networks, Vol. 2 No. 4/5, 1978. [Survey]

114. Sunshine, C. A., and Dalal, Y. K., Connection Management in Transport Protocols, Computer Networks, Vol. 2, No. 6, December 1978. [State machine and related models, Transport protocols, Program proof, State exploration, Sufficiency criteria and design rules, Combined models]

115. Sunshine, C. A., Formal Techniques for Protocol Specification and Verification, IEEE Computer Magazine, Vol. 12, No. 9, 1979. [Survey]

116. Symons, F. J. W., A Generalized Model of Processing Systems Using NPN's, A Generalisation of Petri Nets, Telecom Group Rpt. No. 141, Elec. Eng. Dept., Univ. of Essex, England, 1976. [Petri Net and related models, State machine and related models, Transport protocols, Formal protocol specification, State exploration, Implementation]

117. Symons, F. J. W., Modelling and Analysis of Communication Protocols Using Petri Nets, Telecomm. Group Report No. 140, Elec. Eng. Dept., Univ. of Essex England, 1976. [Petri Net and related models, State machine and related models, Transport protocols, Formal protocol specification, State exploration, Implementation]

118. Symons, F. J. W., The Application of Numerical Petri Nets to the Analysis of Communication Protocols and Signalling Systems, Tech. Rpt. 144, Elec. Eng. Dept., Univ. of Essex, England, May 1977. [Petri Net and related models, State machine and related models, Transport protocols, Formal protocol specification, State exploration, Implementation]

119. Symons, F. J. W., Modelling and Analysis of Communications Protocols Using Numerical Petri Nets, Telecommunications Systems Group Rpt. 152, Univ. of Essex, England, May 1978. [Petri Net and related models, State machine and related models, Transport protocols, Formal protocol specification, State exploration, Implementation]

120. Tajibnapis, William, A Correctness Proof of a Topology Information Maintenance Protocol for a Distributed Computer Network, CACM, Vol. 20, No. 7, July 1977. [State machine and related models, Programming language models, Transport protocols, N-party protocols, State exploration, Program proof, Sufficiency criteria and design rules]

121. Teng, Albert Y., and Liu, Ming T., A Formal Model for Automatic Implementation and Logical Validation of Network Communication Protocol, NBS Computer Networking Symposium, IEEE, 1978. [Formal language/grammar models, Link control protocols, Formal protocol specification, Implementation]

122. Teng, Albert Y., and Liu, Ming T., A Formal Approach to the Design and Implementation of Network Communication Protocol, Proc. COMPSAC, IEEE, 1978. [Formal language/grammar models, Link control protocols, Formal protocol specification]

123. Tomonaga, M., et al., A Study of Protocol Description Technique, WG C NET IPSJ, 1977 (in Japanese). [Petri Net and related models, Simple data transfer protocols, Link control protocols, Transport protocols, Implementation]

124. Valette, R., Sur la description, l'analyse en la validation des systemes de commande paralleles, These d'Etat, Universite Paul Sabatier, Toulouse, France, 1976.

125. van-Mierop, D., Design and Verification of Distributed Interacting Processes, UCLA-ENG-7920 (Ph.D. Thesis), Computer Science Dept., UCLA, 1979. [Petri Net and related models, Programming language models, Simple data transfer protocols, Formal service specification, Formal protocol specification, State exploration, Implementation, Complete development methodology, Symbolic execution]

126. Vissers, C. A., Interface, a Dispersed Architecture, Proc. 3rd Annual Symp. on Computer Architecture, 1976. [Formal interface specification, State machine and related models, Implementation, Formal protocol specification]

127. Vissers, Chris, Formal Description with Asynchronous State Machines, IEC TC65A WG6, Twente Univ. of Technology, November 1978. [State machine and related models, Simple data transfer protocols, Implementation, Formal protocol specification]

128. Wells, R. E., Specification and Implementation of a Verifiable Communications System, (Master's Thesis) ICSCA-CMP-4, Univ. of Texas, Austin, 1976. [Programming language models, Simple data

transfer protocols, Formal protocol specification, Program proof, Implementation, Complete development methodology, Testing, Synchronization protocols]

129. West, C. H., An Automated Technique of Communications Protocol Validation, IEEE Trans. on Comm. Vol. COM-26, No. 8, August 1978. [State machine and related models, Simple data transfer protocols, State exploration]

130. West, C. H., General Technique for Communications Protocol Validation, IBM Journal of Research and Development, Vol. 22, No. 4, July 1978. [State machine and related models, Simple data transfer protocols, State exploration]

131. West, C. H., and Zafiropulo, P., Automated Validation of a Communications Protocol: the CCITT X.21 Recommendation., IBM Journal of Research and Development, Vol. 22, No. 1, January 1978. [State machine and related models, Link control protocols, State exploration]

132. Whitby-Strevens, C., Towards the Performance of Distributed Computing Systems., Proc. COMPSAC, IEEE, 1978. [State machine and related models, Petri Net and related models, Formal language/grammar models, Efficiency, State exploration]

133. Wolfinger, B., and Drobnik, O., Simulation of Protocol Layers of Communication in Computer Networks, in Computer Networks and Simulation, S. S. Shoemaker (ed.), North-Holland Publishing Co., 1978. [State machine and related models, Simulation, State exploration]

134. Yaegashi, S. et al., A Consideration about Description and Implementation of Communication Control Protocols, Proc. 19th Annual Convention of IPSJ, 1978 (in Japanese). [State machine and related models, Implementation, Simple data transfer protocols, Link control protocols, Transport protocols]

135. Zafiropulo, P., Protocol Validation by Duologue-Matrix Analysis, IEEE Trans. on Communications Vol. COM-26, No. 8, August 1978. [State machine and related models, Link control protocols, Simple data transfer protocols, State exploration]

136. Zafiropulo, P., Design Rules for Producing Logically Complete Two-Process Interactions and Communications Protocols, Proc. COMPSAC, IEEE, 1978. [State machine and related models, Simple data transfer protocols, Sufficiency criteria and design rules]

137. Zafiropulo, Pitro, A New Approach to Protocol Validation, Proc. Int. Comm. Conf., 1977. [State machine and related models, Link control protocols, State exploration]

138. Zimmermann, H., "The Cyclades End-to-End Protocol," Proc. 4th Data Comm. Symp., IEEE, 1975. [Transport protocols, Implementation, Informal case analysis]

Key Phrase Index

Appendix A

FORMAL SERVICE SPECIFICATIONS

```
MODULE AB_Protocol   $ "Alternating bit" is an implementation detail
                     not visible in this specification.
TYPES
   msg: VECTOR OF CHAR ~= EMPTY;

PARAMETERS
   msg EMPTY;  $ Reserved value of message buffer

FUNCTIONS

VFUN Buf() -> msg m;
   HIDDEN;
   INITIALLY m = EMPTY;

OFUN Send(msg m);
   DELAY UNTIL Buf() = EMPTY;
   EFFECTS
        'Buf() = m;

OVFUN Receive() -> msg m;
   DELAY UNTIL Buf() ~= EMPTY;
   EFFECTS
        m = Buf();
        'Buf() = EMPTY;

END MODULE AB_Protocol;
```

Notes:

All text following $ is comment. Keywords are written in upper case.

The Types paragraph defines abstract data types from known types.

The Parameters paragraph defines constants or functions set outside this module.

The Functions section defines value-returning functions (VFUNs) which represent the state of the machine, and value-setting functions (OFUNS). Functions which both set and return values (OVFUNs) are also allowed.

VFUNs may be visible (the default) or hidden from the users of the machine. Each VFUN must have its initial value defined.

OFUNS may have an Exceptions section defining named exception conditions which cause an error return, a Delay condition which causes the operation to wait until the condition is satisfied, and an Effects section which defines the effects of the operation in terms of the new values given to VFUNs. New values for VFUNs (after the operation is complete) are denoted with a single quotation mark (').

Fig. A.1--Alternating bit protocol service specification in SPECIAL

```
MODULE AB_Protocol

TYPES
        msg: VECTOR OF CHAR;

DEFINITIONS
        Append(msg m, VECTOR OF msg v) IS
            VECTOR( FOR i FROM 1 TO LENGTH(v) + 1:
                IF i <= LENGTH(v) THEN v[i] ELSE m);

FUNCTIONS

VFUN Buf() -> msg m;
   HIDDEN;
   INITIALLY m = EMPTY;

OFUN Send(msg m);
   DELAY UNTIL Buf() = EMPTY;
   EFFECTS
        'Buf() = m;
        'Inseq() = Append(m,Inseq());    $ Ghost variable

OVFUN Receive() -> msg m;
   DELAY UNTIL Buf() ~= EMPTY;
   EFFECTS
        m = Buf();
        'Outseq() = Append(Buf(),Outseq());      $ Ghost variable
        'Buf() = EMPTY;

$ Ghost variables for use in assertions.
VFUN Inseq() -> VECTOR OF msg v;
        HIDDEN;
        INITIALLY v = VECTOR ();

VFUN Outseq() -> VECTOR OF msg v;
        HIDDEN;
        INITIALLY v = VECTOR ();

END MODULE AB_Protocol;
```

Fig. A.2--Alternating bit protocol service specification in SPECIAL, with added assertions

```
Assertions
        $ Correctness: Output is initial subsequence of input
        FOR i FROM 1 TO LENGTH(Outseq()): Outseq[i] = Inseq[i];

        $ Progress: All input reaches output in finite time
        IF (SOME INTEGER i | i = LENGTH(Inseq())) > LENGTH(Outseq())
                THEN "after finite time"  LENGTH(Outseq()) = i;
```

Notes:

The Definitions paragraph defines "macro"-type substitutions for use in the body of the module.

The built-in Vector data type is indexed from 1.

Fig. A.2 (cont'd)

```
SCOPE AB_Protocol =

BEGIN

  TYPE msg_unit = SEQUENCE OF CHARACTER;  *Basic unit for transmission*
  TYPE info_buf = BUFFER OF msg_unit;

  PROCEDURE AB_Protocol (info_sent:       info_buf <INPUT>
                         info_delivered: info_buf <OUTPUT>) =
  BEGIN
    BLOCK ALLFROM(info_sent) = ALLTO(info_delivered);
  END;

END;
```

Notes:

Keywords of GYPSY are written in upper case.

The parameters of a process in GYPSY define the buffers that interconnect it with other processes. In this case, process AB_Protocol has one buffer of msg_units which it uses only for input, and one buffer of msg_units which it uses only for output.

The term "block" is shorthand for the condition "when blocked waiting to receive from an input buffer."

The functions Allfrom and Allto return, respectively, the sequences of all messages received from and sent to the buffer given as argument. (The "activation ID" parameter which selects for a specific process has been omitted throughout.)

Fig. A.3--Alternating bit protocol service specification in GYPSY--buffer-history-type model

```
SCOPE AB_Protocol =

BEGIN

  TYPE msg_unit = SEQUENCE OF CHARACTER;  *Basic unit for transmission*
  TYPE info_buf = BUFFER OF msg_unit;


  PROCEDURE AB_Protocol (info_sent: info_buf <INPUT>
                         info_delivered: info_buf <OUTPUT>) =
  BEGIN
    VAR msg: msg_unit;
    LOOP
      AWAIT
      ON RECEIVE msg FROM info_sent THEN (
           SEND msg TO info_delivered; )
    END;

  END;

END;
```

Notes:

The "var" statement declares a local variable msg of type msg_unit.

The "await" statement waits until one of its receive clauses is satisfied (i.e., until input is available from the specified buffer) and then performs the "then" portion of the clause.

Send and Receive are defined as blocking operations on the buffers available to a process (i.e. Receive blocks until its buffer is not empty, and Send blocks until its buffer is not full).

Fig. A.4--Alternating bit protocol service specification in GYPSY--transition-type model

```
MODULE Transport_Station

TYPES
   interrupt: INTEGER 0..255;
   port: INTEGER;

PARAMETERS
   buffer B(port i,j);  $ Mapping from ports to buffer id;
   INTEGER empty;

EXTERNALREFS
   from Buffer:
buffer: DESIGNATOR;
msg: VECTOR OF CHAR;
OFUN Put(msg m; buffer b);
OVFUN Get(buffer b) -> msg m;
BOOLEAN Full(buffer b);
BOOLEAN Empty(buffer b);

FUNCTIONS

VFUN Pconnected(port i,j) -> BOOLEAN n;
   INITIALLY  n = FALSE;

VFUN Connected(port i,j) -> BOOLEAN n;
   DERIVED;   n = Pconnected(i,j) AND Pconnected(j,i);

VFUN Credit(port i,j) -> INTEGER n;
   EXCEPTIONS
      no_connection:  ~Connected(i,j);
   INITIALLY  n = 0;

VFUN Int(port i,j) -> interrupt x;
   HIDDEN;  INITIALLY x = empty;

OFUN Connect(port me,him);
   EXCEPTIONS
      already: Pconnected(me,him);
   EFFECTS
      Pconnected(me,him) = TRUE;

OFUN Disconnect(port me,him);
   EXCEPTIONS
      no_connection: ~Pconnected(me,him) AND ~Pconnected(him,me);
   EFFECTS
      'Pconnected(me,him) = FALSE;   $ one side disconnects both;
      'Pconnected(him,me) = FALSE;
      'Int(me,him) = empty;
      'Int(him,me) = empty;
      EFFECTS OF Clear_buf(me,him);
      EFFECTS OF Clear_buf(him,me);
```

Fig. A.5--Transport protocol service specification in SPECIAL

```
OFUN Send(port me,him; data_msg m);
   EXCEPTIONS
      no_connection:  ~Connected(me,him);
      flow_limit:     Credit(me,him) = 0;
   EFFECTS
      EFFECTS OF Put(B(me,him,),m);
      'Credit(me,him) = Credit(me,him) - 1;

OVFUN Receive(port me,him) -> msg m;
   EXCEPTIONS
      no_connection: ~Connected(me,him);
      empty:  Empty(B(him,me));                    $ no blocking
   EFFECTS
      m = Get(B(him,me));

OFUN Give_Credit(port me,him);
   EXCEPTIONS
      no_connection: ~Connected(me,him);
   EFFECTS
      'Credit(him,me) = Credit(him,me) + 1;

OFUN Send_Int(port me,him; interrupt i);
   EXCEPTIONS
      no_connection:  ~Connected(me,him);
      flow_limit: Int(me,him) ~= empty;
   EFFECTS
      'Int(me,him) = i;

OVFUN Receive_Int(port me,him) -> interrupt i;
   EXCEPTIONS
      no_connection: ~Connected(me,him);
      empty:  Int(him,me) = empty;          $ no blocking
   EFFECTS
      i = Int(him,me);
      'Int(him,me) = empty;

END MODULE Transport_Station;
```

Notes:

See notes for Fig. A.1.

The Externalrefs paragraph lists operations and data types defined in other modules--in this case, a separate buffer module which follows.

The Designator type used for buffers allows creation of buffer objects with the New operation, which returns a pointer to the object.

Fig. A.5 (cont'd)

```
MODULE Buffer

TYPES
    buffer: DESIGNATOR;
    msg:    VECTOR OF CHAR;

DEFINITIONS

    BOOLEAN Full(buffer b) IS LENGTH(b) >= Max_size(b);

    BOOLEAN Empty(buffer b) IS LENGTH(b) = 0;

    VECTOR OF msg Append(msg m, VECTOR OF msg v) IS
            VECTOR( FOR i FROM 1 TO LENGTH(v) + 1:
                IF i <= LENGTH(v) THEN v[i] ELSE m);
FUNCTIONS

VFUN Buf(buffer b) -> VECTOR OF msg buf;
        HIDDEN;   INITIALLY buf = VECTOR();   $ empty vector

VFUN Max_size(buffer b) -> INTEGER i;
        HIDDEN;   INITIALLY i = 0;

OVFUN Create_buf(INTEGER sz) -> buffer b;
    EXCEPTIONS
        resource error: RESOURCE ERROR;
    EFFECTS
        b = NEW(buffer);
        'Max_size(b) = sz;

OFUN Clear_buf(buffer b);
    EXCEPTIONS
        nonexistent: Max_size(b) = 0;
    EFFECTS
        'Buf(b) = VECTOR();

OFUN Put(buffer b; msg m);
    EXCEPTIONS
        nonexistent: Max_size(b) = 0;
    DELAY UNTIL ~Full(b);
    EFFECTS
        'Buf(b) = Append(m, Buf(b));

OFUN Get(buffer b) -> msg m;
    EXCEPTIONS
        nonexistent: Max_size(b) = 0;
    DELAY UNTIL ~Empty(b);
    EFFECTS
        m = Buf(b)[1];
        'Buf(b) = VECTOR( FOR i FROM 2 TO LENGTH(Buf(b)): Buf(b)[i]);

END MODULE Buffer;
```

Fig. A.5 (cont'd)

```
Scope Transport_Protocol =
Begin


type msg:record(from: portid;
                to: portid;
                text: string; );

type int:record(from: portid;
                to: portid;
                interrupt:integer; );

type control:record(
                op: one of (credit, ack, connect, disc, error);
                from: portid;
                to: portid; );

type msgbuf: buffer of msg;
type intbuf: buffer of interrupt;
type cntrlbuf; buffer of control;


process Transport_Protocol(
        msg_in: msgbuf <input>;
        msg_out: msgbuf <output>;
        int_in: intbuf <inbuf>;
        int_out: intbuf <output>;
        cntrl_in: cntrlbuf <input>;
        cntrl_out: cntrlbuf <output>; ) =

Begin
   Block
        Data_transfer_OK();
        Flow_control_OK();
        Connection_OK();
End;


function Data_transfer_OK() =
        * Messages delivered from S to D are initial subsequence of
            messages sent from S to D *
    forall adr:source
        forall adr:dest
            extract_msg_out(source, dest) is an initial sequence of
                extract_msg_in(source,dest);

function extract_msg_out(adr: from, to) =
   subsequence of msg m in allto(msg_out) |
        m.from = source and  m.to = dest;
```

Fig. A.6--Buffer-history-type transport protocol service specification

```
function extract_msg_in(adr: from, to) =
   subsequence of msg m in allfrom(msg_in) |
        m.from = source and  m.to = dest;


function Flow_control_OK() =
        * Number of messages sent from S to D <= number of credits
            sent from D to S *
    forall adr:source
        forall adr:dest
            size(extract_msg_out(source, dest)) <=
                size(extract_credit_in(dest,source);

function extract_credit_in(adr: from, to) =
   subsequence of control c in allfrom(cntrl_in) |
        m.op = credit and m.from = source and  m.to = dest;


function Connection_OK() =
        * results of connect *
    if Last(extract_con_in(i,j)) = connect(i,j) and
        Rest(extract_con_in(i,j)) = connect(j,i)* connect(i,j)  !!
        then Last(extract_err_out(i)) = "already connected"
        else Last(extract_con_out(i,j)) = connect(i,j);

        * results of disconnect *
    if Last(extract_con_in(i,j)) = disc(i,j) and
        Rest(extract_con_in(i,j)) = disc(i,j) or disc(j,i)
        then Last(extract_err_out(i)) = "not connected"
        else Last(extract_con_out(i,j)) = disc(i,j);

function extract_con_in(adr: i,j) =
    subsequence of control c in allfrom(cntrl_in) |
        (c.op=connect or c.op=disc) and
            ((c.from=i and c.to=j) or (c.from=j and c.to=i));

function extract_con_out(adr: i,j) =
    subsequence of control c in allto(cntrl_out) |
        (c.op=connect or c.op=disc) and c.from=i and c.to=j;

function extract_err_out(adr: i) =
    subsequence of control c in allto(cntrl_out) |
        c.op=error and c.to=i;

End;
```

Fig. A.6 (cont'd)

Notes:

Text between * is comment.

The character ¦ means "such that."

Functions Last and Rest on sequences return the last element and all but the last element, respectively.

The right side of the line marked with !! defines a regular expression where * means zero or more occurrences of an item.

The terms "sequence," "initial sequence," and "subsequence" are assumed to be part of the specification language.

Fig. A.6 (cont'd)

```
AGENT Transport_Station

TYPES
   interrupt: INTEGER
   port: INTEGER
   command: DYNAMIC STRUCT {opcode op; port me,him; $... other parameters}
   opcode: ONE OF {data, int, int_ack, give_credit, connect, disconnect}
   msg: VECTOR OF CHAR;

STATE

Pconnected(port i,j) -> BOOLEAN n
   INITIALLY  n = FALSE

Connected(port i,j) -> BOOLEAN n
   DERIVED
   n = Pconnected(i,j) AND Pconnected(j,i)

Credit(port i,j) -> INTEGER n
   INITIALLY  n = 0

Int_Pending(port i,j) -> BOOLEAN n
   INITIALLY n = FALSE


INPUT HANDLERS (DISPATCH ON op OF command c)

connect (port me,him)
   EXCEPTIONS
      already: Pconnected(me,him)
   EFFECTS
      'Pconnected(me,him) = TRUE
      OUTPUT c

disconnect (port me,him)
   EXCEPTIONS
      no_connection: ~Pconnected(me,him) AND ~Pconnected(him,me)
   EFFECTS:
      'Pconnected(me,him) = FALSE  /* one side disconnects both
      'Pconnected(him,me) = FALSE
      'Int(me,him) = FALSE
      'Int(him,me) = FALSE
      OUTPUT c

give_credit (port me,him)
   EXCEPTIONS
      no_connection: ~Connected(me,him)
   EFFECTS
      'Credit(him,me) = Credit(him,me) + 1
      OUTPUT c
```

Fig. A.7--Agent-type transport protocol service specification

```
data (port me,him; msg m)
   EXCEPTIONS
      no_connection:  ~Connected(me,him)
      flow_limit:     Credit(me,him) = 0
   EFFECTS
      OUTPUT c
      'Credit(me,him) = Credit(me,him) - 1

int (port me,him; interrupt i)
   EXCEPTIONS
      no_connection:  ~Connected(me,him)
      pending: Int_pending(me,him)
   EFFECTS
      OUTPUT c
      'Int_pending(me,him) = TRUE

int_ack (port me,him)
   EXCEPTIONS
      no_connection:  ~Connected(me,him)
      no_int:    ~Int_pending(him,me)
   EFFECTS
      OUTPUT c
      'Int_pending(him,me) = FALSE

default
   EXCEPTIONS
      bad_command: TRUE

END AGENT Transport_Protocol
```

Notes:

A dynamic structure is a record containing some number of <name,type,value> triplets. Input and output are in this format for convenience.

The operation of the agent is to continuously read commands from its input and to invoke the handler corresponding to the op field of the input. The parameters of each handler are the command fields expected. Their absence causes an error exception.

Exceptions are handled by outputting an error message addressed to the sender of the offending message and containing the name of the exception.

Output c means send the dynamic structure c to the output buffer.

Fig. A.7 (cont'd)

```
MODULE FTP      $ See Notes at end

TYPES

  record: VECTOR_OF BIT;
  file: VECTOR_OF record;
  user_id: INTEGER;
  trans_id: INTEGER;
  name_spec: VECTOR_OF CHAR;
  host_name: VECTOR_OF CHAR;
  file_name: VECTOR_OF CHAR;
  path_name: STRUCT_OF (host_name hn; file_name fn);
  password: VECTOR_OF CHAR;
  account: INTEGER;
  access_mode: ONE_OF {"read", "write", "append", "delete"};
  dest_stat: ONE_OF {"must", "can't", "don't_care"};
  acc_crl: STRUCT_OF (user_id u; password pw; account ac);
  action: ONE_OF {"ftp", "quit", "abort", "status", "listname",
                  "copy", "append", "delete", "replace"}
  data_type: CHAR;
  trans_mode: CHAR;
  file_struct: CHAR;
  data_spec: STRUCT_OF (file_data_type ty; trans_mode md; file_struct st);
  partial_spec: STRUCT_OF (INTEGER upper; INTEGER lower);
  new_data: record;
  message: record;
  parm: STRUCT_OF (BOOLEAN f/b; dest_stat ds; acc_crl a1,a2;
                   data_spec d; path_name p1,p2; trans_id id;
                   partial_spec ps);

DECLARATIONS

  INTEGER j;
  BOOLEAN b, f/b;
  file f;
  user_id uid;
  trans_id id;
  message msg, msg1, msg2;
  name_spec ns;
  path_name p, p1, p2;
  access_mode c, c1, c2;
  dest_stat ds;
  acc_crl a, a1, a2;
  data_spec d;
  partial_spec ps;
  new_data n;
  stat_despt s;
  VECTOR_OF path_name vp;
  stat st;
```

Fig. A.8--File transfer protocol service specification in SPECIAL

```
DEFINITIONS

  VECTOR_OF path_name Match(a; ns) IS  ...
        $"return all pathnames matching ns for which user has access".

  BOOLEAN Access_check(a; p; c) IS ...
        $"check the access control information a for permission to
          access file p in mode c".

  BOOLEAN Data_spec_check(d; p1, p2; c1, c2) IS ...
        $"check if the data specification is accepted".

  file Canon(f) IS ....
        $"return the canonical form of file f".

  INTEGER Length(p) IS .....
        $"return the length of the file specified by p".

EXTERNALREFS

  FROM Transport_Protocol:
       connect_id: k, k1,k2;
       VFUN On_net(p.hn) -> b;

  FROM Table_Management:
       Type stat: ONE_OF {"done", "error", "in_progress", "aborted",
                          "suspended"};
       Type stat_descpt: STRUCT_OF (action act; stat st; parm pa;
                                    connect_id k; errorcode errcode);
       Type errorcode: ONE_OF {"bad_message", ... );
       VFUN Stat_table(id) -> s;
       OFUN Stor_table(s);
       VFUN Unique_id() -> id;
          $"return a unique id for a transaction".

  FROM System:
       VFUN Get_uid() -> uid;


FUNCTIONS

  VFUN Error(id) -> b;  $"true if error of any kind during operation".
       EXCEPTIONS
          no_such_trans: Stat_table(id) = ?;
       HIDDEN;
       INITIALLY b = FALSE;

  VFUN In_session(uid) -> b;    $"is user uid in an FTP session?".
       HIDDEN;
       INITIALLY b = FALSE;
```

Fig. A.8 (cont'd)

```
VFUN Exist(p) -> b;                    $"does file named p exist?".
     INITIALLY
        b = FALSE;

VFUN Cont(p) -> f;                     $"content of file named p".
     EXCEPTIONS
        nonexist_file: ~Exist(p);
     INITIALLY
        f = ?;

OFUN Ftp();                            $"start FTP session".
     EXCEPTIONS
        already_in_session: In_session(Get_uid());
     EFFECTS
        'In_session(Get_uid()) = TRUE;

OFUN Quit();                           $"quit FTP session".
     EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
     EFFECTS
        'In_session(Get_uid()) = FALSE;

OFUN Abort(id);                        $"abort transaction id".
     EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
        unknown: Stat_table(id) = ?;
        bad_command: Stat_table(id).op ~= "copy" or "append";
        finished: Stat_table(id).st = "done";
     EFFECTS
        Stat_table(id).st = "aborted";

VFUN Status(id) -> s;                  $"status of transaction id".
     EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
        unknown: Stat_table(id) = ?;
     EFFECTS
        s = Stat_table(id);

VFUN Listname(a; ns) -> vp;    $"list file names matching name spec ns".
     EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
        nomatch: Match(a, ns) = VECTOR();
        error: Error(Unique_id());
     EFFECTS
        vp = Match(a, ns);
```

Fig. A.8 (cont'd)

```
OVFUN Copy(f/b; ds; a1, a2; d; p1, p2) -> id;
                                  $"copy file p1 to p2".
     EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
        no_access: ~Access_check(a1, p1, "read") or
                   ~Access_check(a2, p2, "write");
        source_file_nonexist: ~Exist(p1);
        dest_file_nonexist: ~Exist(p2) and ds = "must";
        dest_file_exist: Exist(p2) and ds = "can't";
        unconnected_host: ~On_net(p1.hn) or ~On_net(p2.hn);
        bad_data_spec: ~Data_spec_check(d, p1, p2, "read", "write");
        error: Error(id);
     EFFECTS
        id = Unique_id();
        s.id = id;
        s.act = "copy";
        s.parm = {f/b, ds, a1, a2, d, p1, p2, id, -};
        IF (f/b = foreground)  THEN(                    $"foreground".
           IF (~Exist(p2))  THEN 'Exist(p2) = TRUE;
           Canon('Cont(p2)) = Canon(Cont(p1));
           s.st = "done";
        );
        ELSE(                                           $"background".
           s.st = "in_progress";
        );
        EFFECTS_OF Stor_table(s);

OVFUN Append(f/b; a1, a2; d; p1, p2) -> id;
                                  $"append file p1 to p2".
     EXCEPTIONS
        not_in_session: ~In_session(Get_uid);
        no_access: ~Access_check(a1, p1, "read") or
                   ~Access_check(a2, p2, "append");
        source_file_nonexist: ~Exist(p1);
        unconnected_host: ~On_net(p1.hn) or ~On_net(p2.hn);
        bad_data_spec: ~Data_spec_check(d, p1, p2, "read", "append");
        error: Error(id);
     EFFECTS
        id = Unique_id();
        s.id = id;
        s.act = "append";
        s.parm = {f/b, ds, a1, a2, d, p1, p2, id, -};
        IF (f/b = foreground)  THEN(                   $"foreground".
          IF (~Exist(p2)) THEN (
              'Exist(p2) = TRUE;
              Canon('Cont(p2)) = Canon(Cont(p1));
          );
```

Fig. A.8 (cont'd)

```
          ELSE
          (Canon('Cont(p2)) = Canon(Cont(p2)) & Canon(Cont(p1));
          $ "& indicates the concatenation".
          s.st = "done";
        );
        ELSE(                                          $"background".
          s.st = "in_progress";
        );
        EFFECTS_OF Stor_table(s);

  OVFUN Delete(a; p) -> id;          $"delete file p".
      EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
        no_access: ~Access_check(a, p, "delete");
        no_such_file: ~Exist(p);
        error: Error(id);
      EFFECTS
        'Exist(p) = FALSE;
        id = Unique_id();
        'Cont(p) = ?;
        s.st = "done";
        s.id = id;
        EFFECTS_OF Stor_table(s);

  OVFUN Replace(a; p; ps; n) -> id; $"replace portion ps of file p by
                                      new data n".
      EXCEPTIONS
        not_in_session: ~In_session(Get_uid());
        no_access: ~Access_check(a, p, "write");
        no_such_file: ~Exist(p);
        bad_spec: ps.lower < 0 or ps.upper > Length(p);
        error: Error(id);
      EFFECTS
        Canon('Cont(p)) = VECTOR(FOR j FROM 1 TO Length(p) :
                          IF ps.lower =< j =< ps.upper
                          THEN Canon(n[j -ps.lower +1]);
                          ELSE Canon(Cont(p)[j]));
        id = Unique_id();


PROC Background_Server();
  EFFECTS
    FORALL s=Stat_table(id) | s.st = "in_progress"
      IF ~Error(id) THEN(
        IF Stat_table(id).act = "copy" THEN(
            IF (~Exist(p2))  THEN 'Exist(p2) = TRUE;
            Canon('Cont(p2)) = Canon(Cont(p1));
        );
```

Fig. A.8 (cont'd)

```
        ELSE IF Stat_table(id).act = "append" THEN(
            IF (~Exist(p2)) THEN (
                'Exist(p2) = TRUE;
                Canon('Cont(p2)) = Canon(Cont(p1));
            );
            ELSE
            (Canon('Cont(p2)) = Canon(Cont(p2)) & Canon(Cont(p1));
        );
        s.st = "done";
        EFFECTS_OF Stor_table(s);
    );
```

Notes:

All text following $ is comment.

The Declarations paragraph declares the types of parameters and variables used in the rest of the module.

The value ? means "undefined."

The Proc paragraph defines a "process"-type cyclic operation, as discussed in the text.

The & operator (for the Append function) means concatenation.

The following table shows the FTP commands and the parameters that are needed for each command.

Command	Parameters							
	Fore/ Back-ground	Dest. Status	Authen. Control	Data Xfer Spec.	Source Path name	Dest. Path name	Trans-action ID	Part. Spec.
Listname			*		*			
Copy	*	*	*	*	*	*		
Append	*		*	*	*	*		
Delete			*		*			
Replace			*		*	*		*
Abort							*	
Status							*	
Ftp								
Quit								

Authentication control contains the user's login name, password, and account number.
Data Transfer Spec. specifies the data type, transmit mode, and file structure.

Fig. A.8 (cont'd)

Appendix B

FORMAL PROTOCOL SPECIFICATIONS

```
PROGRAM MODULE AB_Protocol
$ This is an "implementation" of each OFUN in the service spec.

TYPES           $ These types used in all modules
        msg: VECTOR OF CHAR;
        seqnum: INTEGER 0..1 OR empty;
        data_pkt: STRUCT OF (msg text; seqnum sq);

DEFINITIONS
        INTEGER empty is -1;

EXTERNALREFS
    $ These 5 modules are used by the AB_Protocol implementation.

    FROM Send_station:
VFUN Sendseq() -> seqnum s;
OFUN Inc_sendseq();

    FROM Receive_station:
VFUN Rcvseq() -> seqnum s;
OFUN Inc_rcvseq();
VFUN Rbuf() -> msg m;
OFUN Store(msg m);

    FROM Medium StoR:
OFUN Send_data(data_pkt d);
OVFUN Wait_data() -> data_pkt d;

    FROM Medium RtoS:
OVFUN Wait_ack() -> seqnum a;
OFUN Send_ack(seqnum a);

    FROM Timer
OFUN Start_timer();


IMPLEMENTATIONS

OVFUN PROG User_Receive() -> msg m;
    BEGIN
        WAIT UNTIL Rbuf() ~= empty;
        m <- Rbuf();
        Store(empty);
    END;
```

Fig. B.1--Alternating bit protocol specification in SPECIAL

```
OFUN PROG Send(msg m);

    DECLARATIONS
        data_pkt p;
        seqnum a;

    BEGIN
        p.text <- m;
        p.sq <- Sendseq();
        UNTIL done DO
            Send_data(p);
            Start_timer();
            a <- Wait_Ack();          $ Wait for ack or timeout
            IF a = Sendseq() THEN
                Inc_sensdseq();
                SIGNAL done;
            ELSE FI;
        OD;
    END;


PROC Receiver();

    DECLARATIONS
        data_pkt p;

    BEGIN
        WHILE TRUE DO
            p <- Receive_data();
            Send_ack(p.sq);
            IF p.sq = Rcvseq() AND Rbuf() = empty THEN
                    Store(p.text);
                    Inc_rcvseq();
            ELSE FI;
        OD;
    END;

END PROGRAM MODULE AB_Protocol;



MODULE Send_station

VFUN Sendseq() -> seqnum s;
    INITIALLY s = 0;

OFUN Inc_sendseq();
    EFFECTS
        'Sendseq() = (Sendseq() + 1) MOD 2;

END MODULE Send_station;
```

Fig. B.1 (cont'd)

```
MODULE Receive_station

VFUN Rcvseq() -> seqnum s;
    INITIALLY s = 0;

OFUN Inc_rcvseq();
    EFFECTS
        'Rcvseq() = (Rcvseq() + 1) MOD 2;

VFUN Rbuf() -> msg m;
    INITIALLY m = empty;

OFUN Store(msg m);
    EFFECTS
        'Rbuf() = m;

END MODULE Receive_station;



MODULE Medium_StoR
$ Used to send data packets from sender to receiver

PARAMETERS
        INTEGER maxerrs;

EXTERNALREFS
    from Misc:
VFUN Random() -> BOOLEAN t;

FUNCTIONS

VFUN StoRbuf() -> data_pkt d;
    HIDDEN;
    INITIALLY d.sq = empty;

VFUN Error_count -> INTEGER e;
    HIDDEN;
    INITIALLY e = 0;

OFUN Send_data(data_pkt d);
    EFFECTS
        IF Error_count() < maxerrs AND Random() THEN  $ lost
                'StoRbuf() = StoRbuf;    $ no change
                'Error_count() = Error_count() + 1;
        ELSE 'StoRbuf() = d;
        FI;
```

Fig. B.1 (cont'd)

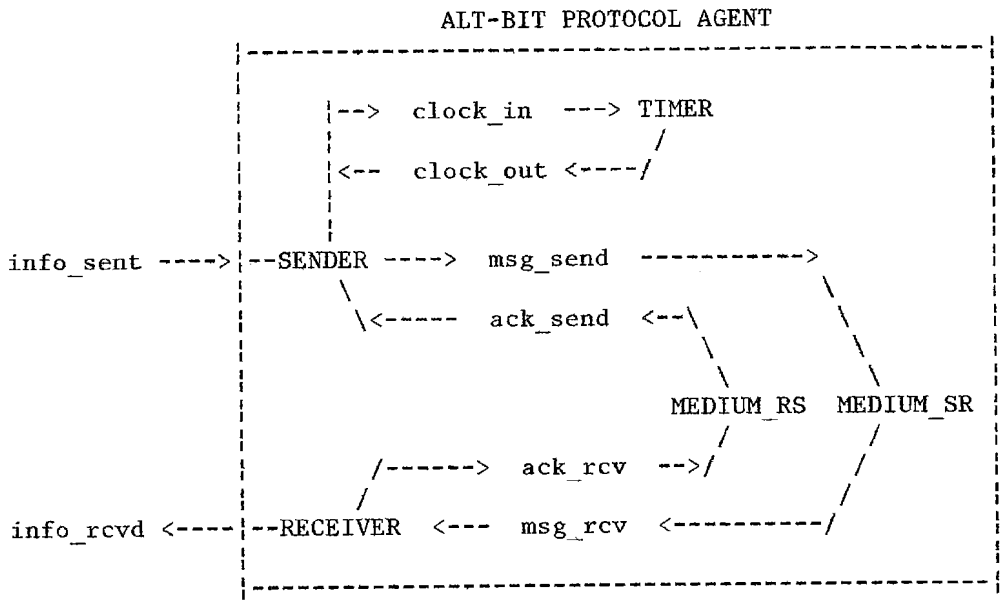
```
OVFUN Wait_data() -> data_pkt d;
    DELAY UNTIL StoRbuf().sq ~= empty;
    EFFECTS
        d = StoRbuf();
        'StoRbuf().sq = empty;

END MODULE Medium StoR;



MODULE Medium_RtoS
$ Used to send acknowledgements from receiver to sender

PARAMETERS
        INTEGER maxerrs;

EXTERNALREFS
    from Misc:
VFUN Random() -> BOOLEAN t;

    from Timer:          $ Don't like this here
VFUN Time() -> INTEGER t;

FUNCTIONS

VFUN RtoSbuf() -> seqnum a;
    HIDDEN;
    INITIALLY a = empty;

VFUN Error_count -> INTEGER e;
    HIDDEN;
    INITIALLY e = 0;

OFUN Send_ack(seqnum a);
    EFFECTS
        IF Error_count() < maxerrs AND Random() THEN  $ lost
                'RtoSbuf() = RtoSbuf;    $ no change
                'Error_count() = Error_count() + 1;
        ELSE 'RtoSbuf() = a;
        FI;

OVFUN Wait_ack() -> seqnum a;
$ Includes timeout which should really be in send_station, not here
    DELAY UNTIL RtoSbuf() ~= empty OR Time() = 0;
    EFFECTS
        a = RtoSbuf();
        'RtoSbuf() = empty;

End MODULE Medium_RtoS;
```

Fig. B.1 (cont'd)

```
MODULE Timer

PARAMETERS
        INTEGER timeout;

VFUN Time() -> INTEGER t;
    INITIALLY t = 0;

OFUN Start_timer();
    EFFECTS
        'Time() = timeout;

OFUN Tick();
    EFFECTS
        IF Time() > 0 THEN  'Time() = Time() - 1;
        FI;


PROC Clock();       $ Hardware process
    BEGIN
        WHILE TRUE DO
                $Wait for clock period
                Tick();
        OD;
    END;

END MODULE Timer;



MAP AB_Protocol TO Receive_station;

EXTERNALREFS
    from AB_Protocol:
        VFUN Buf() -> msg m;
    from Receive_station:
        VFUN Rbuf() -> msg m;

MAPPINGS
    Buf():  Rbuf();

END_MAP;
```

Fig. B.1 (cont'd)

```
* Alternating_Bit_Protocol in Gypsy *
```




```
Scope Alt_Bit_Protocol =
begin

 Type msg_unt = sequence of character; *Basic unit for transmission*
 Type msg_pkt = record (message : msg_unt; seqnum : integer);
 Type msg_buf = buffer of msg_pkt;
 Type clk_buf = buffer of integer;
 Type inf_buf = buffer of msg_unt;


 Procedure AB_Protocol (info_sent: inf_buf<input>,
                        info_rcvd: inf_buf<output>) =
 begin
   block allfrom(info_sent) = allto(info_rcvd) *Service Specification*

   var msg_send, msg_rcv, ack_send, ack_rcv: msg_buf;
   var clock_in, clock_out: clk_buf;

   cobegin
     Sender (info_sent, msg_send, ack_send, clock_in, clock_out);
     Medium_sr (msg_send, msg_rcv);
     Medium_rs (ack_rcv, ack_send);
     Receiver (info_rcvd, msg_rcv, ack_rcv);
     Timer (clock_in, clock_out);
   end;
 end;
```

Fig. B.2--Alternating bit protocol specification in GYPSY

```
Procedure Sender (var info_sent: inf_buf<input>;
                  var ack_send: msg_buf<input>;
                  var msg_send: msg_buf<output>;
                  var clock_in: clk_buf<output>;
                  var clock_out: clk_buf<input> ) =

begin
  block <input> all_sent(info_sent, msg_send);   * service spec *
       *Sender only sends next message if last was acknowledged,
        and seqnum of next message is incremented *

       *Code*
  var i: integer := 1;
  var msg, ack: msg_pkt;
  var tick, start: integer;
  var m: msg_unt;

  loop
    receive m from info_sent;
    msg.message := m;
    msg.seqnum := i mod 2;
    send msg to msg_send;
    if not empty(clock_out) then receive tick from clock_out;
    send start to clock_in;

    assert pending;

    loop
      await
        on receive ack from ack_send then
            if ack.seqnum = i mod 2 then
               begin
               i := i + 1 mod 2;
               quit;
               end;
        on receive tick from clock_out then  *timeout and retransmit*
               begin
               send msg to msg_send;
               send start to clock_in;
               end;

      assert seq_num_advanced(i,ack_send);
    end;

    assert unique_seq_num(msg_send);
  end;
end;
```

Fig. B.2 (cont'd)

```
Procedure Medium_sr (var msg_send: msg_buf<input>;
                     var msg_rcv: msg_buf<output> ) =
begin
  block subseq(msg_send, msg_rcv);
end;


Procedure Medium_rs (var ack_rcv: msg_buf<input>;
                    var ack_send: msg_buf<output> ) =
begin
  block subseq(ack_rcv, ack_send);
end;


Procedure Receiver(var info_rcvd:inf_buf<output>;
               var msg_rcv: msg_buf<input>;
               var ack_rcv: msg_buf<output> ) =
    * Receiver only accepts message if seqnum is the successor of
      last accepted, but always acknowledges *
begin
  block pending;

  var i: integer := 1;
  var msg: msg_pkt;
  var exp: integer;

  loop
    receive msg from msg_rcv;
    send msg to ack_rcv;
    if exp = msg.seqnum then (
       send msg.message to info_rcvd;
       exp := exp + 1 mod 2
       );
    assert pending;
  end;
end;


Procedure Timer(var clock_in: clk_buf<input>;
               var clock_out: clk_buf<output>  ) =
begin
  block size(allto(clock_out)) = size(allfrom(clock_in))

  var tick, start: integer;

  loop
    receive start from clock_in;
    pause(timeout);
    send tick to clock_out;
  end;
end;
```

Fig. B.2 (cont'd)

```
Function lost: boolean = pending;
       * returns random T/F value *


Function pause(set_time: integer): boolean = pending;
       * pauses for given time *


Function all_sent(info_sent:inf_buf; msg_send:msg_buf) : boolean =
begin
  exit assume all j: integer, j <= size(allfrom(info_sent)
               -> some k: integer,
               allfrom(info_sent)[j] = allto(msg_send)[k].message;
               * all input were sent *

       assume all j: integer, j <= size(allto(msg_send)
               -> some k: integer,
               allto(msg_send)[j].message = allfrom(info_sent)[k];
               * all sent were input *
end;


Function unique_seq_num(buf: msg_buf) : boolean =
begin
  exit assume all k: integer, k <= size(allto(buf)) and
               allto(buf)[k-1].seqnum =  allto(buf)[k].seqnum
               -> allto(buf)[k-1].message = allto(buf)[k].message
       *succeeding messages with same seqnums have same texts *
end;


Function seq_num_advanced(i:integer; ackbuf:msg_buf) : boolean =
  exit assume last(allfrom(ackbuf)).seqnum ~= i
       * send seqnum is advanced after getting a current ack *
end;


Function subseq(buf1, buf2: msg_buf) : boolean =
begin
    exit assume all j,k < size(allfrom(buf1)),
               some j2,k2: integer | j2 < j, k2 < k, and
               allfrom(buf1)[j] = allto(buf2)[j2]  and
               allfrom(buf1)[k] = allto(buf2)[k2]  and
               j > k
       <==>    j2 > k2;
       * allto(buf2) is subsequence of allfrom(buf1) *
end;

end Alt_Bit_Protocol;
```

Fig. B.2 (cont'd)