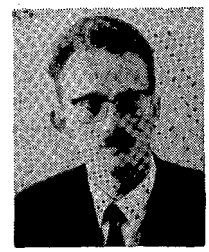
[36] R. J. Sundstrom, "Formal definition of IBM's system network architecture," in *Proc. Nat. Telecommun. Conf.*, Los Angeles, CA, Dec. 1977, p. 3A1.
[37] C. Sunshine, "Formal techniques for protocol specification and verification," *IEEE Comput. Mag.*, 21 pp., Aug. 1979.
[38] C. A. Sunshine and Y. K. Dalal, "Connection management in transport protocols," *Comput. Networks*, vol. 2, pp. 454–473, Dec. 1978.
[39] C. A. Sunshine, "Survey of protocol definition and verification techniques," in *Proc. Comput. Network Protocols Symp.*, Univ. of Liège, Belgium, Feb. 1978, and *Comput. Networks*, vol. 2, pp. 346–350, Oct. 1978.



André A. S. Danthine (M) is a Professor at the University of Liege, Liege, elgium. Since 1972 he has been engaged in computer network research. His research group has done studies for the Belgian Government and several private companies in Belgium and France. He is Editor of *Computer Networks*.

He is a member of the Association of Computing Machinery. Since October 1979, he has been Chairman of the TC6 of the International Federation of Information Processing.

A General Transition Model for Protocols and Communication Services

GREGOR V. BOCHMANN

(Invited Paper)

Abstract—**Different approaches have been used for the formal specification and verification of communication protocols. This paper explains the approach of using a general transition model which combines aspects of finite state transition diagrams and programming languages. Different ways of structuring a protocol into separate modules or functions are also discussed.**

The main part of the paper describes a method for exactly specifying the communication service provided by a protocol. Two aspects of a service specification are distinguished: 1) the local properties which characterize the interface through which the service may be accessed, and 2) the global properties which describe the "end-to-end" communication characteristics of the service. It is shown how the specification method is related to the general transition model for protocol specification. Verification is discussed briefly with emphasis on the use of invariant assertions in the context of finite state as well as programming language protocol descriptions.

The discussed topics are demonstrated with examples based on the HDLC classes of procedures and the X.25 Virtual Circuit data transmission service.

I. INTRODUCTION

DIFFERENT approaches have been used for the formal specification and verification of communication protocols. As explained in another paper of this collection, most of these approaches use finite state transition diagrams or programs written is some high-level programming language or both. The purpose of this paper is threefold.

Manuscript received May 18, 1979; revised December 18, 1979.

The author is with the Department d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montreal, P.Q., Canada, on leave at the Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

First, in Section II, we review some experience with a general transition model, which we called a "unified" approach [1] because it involves state transitions and programming language elements. We believe that such an approach is appropriate for the formal specification of protocols, the specification of the services provided, and the verification of correct operation. In Section IV, we point out certain similarities between finite state transition and programming language approaches to verification. Knowledge of the indicated references may be useful, but are not necessary for the understanding of these sections.

Second, we discuss these issues by considering, as an example, the HDLC classes of procedures.

Third—and this is the main part of the paper in Section III—we describe a method for specifying the communication service provided by a protocol. While certain aspects of this method are related to our "unified" approach, we believe that most elements of the method are of general validity and applicability. In fact, the method is related to software engineering methods [2] for specifying software modules. However, certain elements of our method are specific to protocols due to their distributed nature.

II. A GENERAL TRANSITION MODEL

For a given communication layer of a distributed computer system, we assume that the protocol is specified by separate descriptions for both entities executing the protocol, as shown in Fig. 1. We explain in this section the main features of a general transition model [1] which is based on Keller's transition model [3] for parallel programs. We also discuss the relation of this model to other protocol description methods, and the

0090-6778/80/0400-0643$00.75 © 1980 IEEE

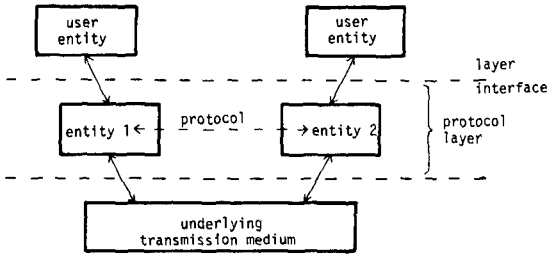


Fig. 1. A protocol layer within a layered system architecture.

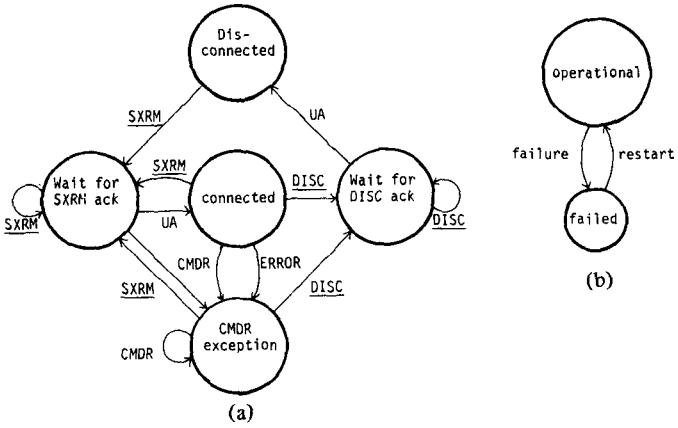


Fig. 2. State transition diagrams for the primary link setup module of an HDLC station. (a) Operational procedure which may be discontinued due to a failure. [The diagram is hierarchically dependent on the *operational* state of diagram (b).] (b) Diagram showing the possible failure and restart transitions.

importance of modularization which may lead to the subdivision of a given communication layer into several sublayers or protocol modules. Without giving the complete definitions which may be found in the literature, these concepts are explained using the HDLC classes of procedures as an example. The complete HDLC specifications, based on this method, may be found in [4]. An experience of using these specifications for the implementation of X.25 link level procedures is described in [8].

A. The Description Method

In our general transition model, an entity is described by the set of possible states in which it may be, and the possible state transitions (which are assumed to exclude one another in time). The possible states are generally described by two components:

1) a finite state transition diagram, and
2) a set of program variables which each may assume certain values.

The state of the entity is characterized by: 1) a token which indicates the active place in the transition diagram, and 2) the values of the program variables. As an example, Fig. 2 shows the transition diagram of an HDLC module which operates the link setup and disconnection procedure. The state space of a complete HDLC station is defined by this and similar diagrams (one for each of the modules shown in Fig. 5) and the program variables shown in Fig. 3.

```
received : frame-type;          } general
status : status-type;

unack : sequence-count;         }
VS : sequence-count;            }
buffer                          } Source
   data (VS:sequence-count):info-block
   to-send (VS:sequence-count): boolean

VR:sequence-count;              } Sink

PF-control.bit : (0.. 1);       } Primary
                                  PF control
```

Fig. 3. Program variables of HDLC station.

The operation of an entity is defined by the possible state transitions. These transitions are indicated in the transition diagram (see, for example, Fig. 2); however, additional information must be provided. For instance, each transition, when executed, may change the values of the program variables and interact with the user entity through the upper layer interface or with the underlying transmission medium through the lower interface (see Fig. 1). A given transition may only be executed when its *enabling predicate,* i.e., a Boolean expression depending on the program variables, is true. This additional information may be given in the form of a table, as shown in Fig. 4. For example, the $\underline{I}$ transition, which sends an information (I) frame to the peer entity, may only be executed when a data block is *to be sent* and not too many I frames are *unack*nowledged. When executed, the action of the transition sends an I frame and updates the value of the send variable VS.

B. Relation to Other Description Methods

It has been pointed out [5] that most protocols contain certain aspects that are naturally described by finite state (FS) transition diagrams and other aspects that are better described by program variables and executable statements written in some programming language. The HDLC procedures provide a typical example. The link setup and disconnection procedure is described relatively completely by the FS transition diagram of Fig. 2, whereas the data transfer, exemplified by the $\underline{I}$ and $I_=$ transitions given in Fig. 4, essentially involves program variables and statements. Different approaches have been taken to cope with this situation (see, for example, [6]).

The approach of attempting to write complete descriptions in the FS model is limited because most protocols are so complex that the resulting FS descriptions becomes too large to be useful. However, partial descriptions in the FS model may be very useful. For example, the FS descriptions of X.21 and X.25 are of this kind. We note that even a relatively complete FS description and analysis [7] of the simple "alternating bit" protocol ignores the contents of the exchanged user messages. The partial description approach corresponds to keeping only the FS transition diagram of our general transition model (in the case of the HDLC procedures, for example, keeping Fig. 2 and ignoring Fig. 3 and 4). But it is clear that such a

Transition	Enabling predicate	Action	Meaning
Primary station:			
SXRM	PF-control.bit = 1	send-unnumbered (SXRM) ;	SXRM is SNRM or SARM depending on the mode to be set
UA	received.kind = UA	init (source) ; init (sink) ; init (transmission) ;	initialize the source and sink components
DISC	PF-control.bit = 1	send-unnumbered (DISC) ;	
CMDR	received.kind = CMDR	init (transmission) ;	
ERROR	Status in [invalid-control-field, invalid-info, invalid-size, invalid-NR]	init (transmission) ;	frame received contained an error to be resolved by a higher level recovery procedure at Primary
OTHER	...	init (transmission) ;	in certain states, the reception of certain kind of frames is simply ignored (not shown in the transition diagrams
I	buffer.to-send (VS) and VS ≠ (unack + window) mod modulus	send-info (VS,VR,buffer.data (VS)); VS := (VS+1) mod modulus ;	when there is an I frame to be sent, which lies within the send window, send it
$I_=$	received.kind = I and received.NS=VR	unack := received.NR; VR := VR+1; init (transmission)	if I-frame is in sequence, pass data to user

Fig. 4. Definition of the transitions shown in Fig. 2(a) and of the *I*-frame receiving $I_=$ and sending *I* transitions.

description, and a protocol analysis based on it, must be complemented with additional information.

On the other hand, the FS aspects may be eliminated from the description of a protocol in the general transition model by replacing each FS transition diagram, which contains one token, by a variable which indicates the place of the token in the diagram, together with appropriate enabling predicates and update actions for the transitions. Such a transformation is straightforward, and is usually performed in order to obtain an implementation of the protocol.

C. Modularization

Most protocols implemented in a given layer of a hierarchical system are so complex that a conceptual subdivision into several sublayers or functions is very useful. In this case, each sublayer or function corresponds to a module within each entity executing the protocol. The different modules of an entity are relatively independent of one another. In the examples which we have considered, i.e., the link level HDLC procedures [4], the X.25 packet level procedures [8], and the ML Protocol [9] providing transport and session layer functions, the following concepts were sufficient to naturally describe the interactions between the modules within one entity, and different entities through a layer interface. We note that all, except the second concept, are also applicable to FS models.

Complete Independence: Each module is described by a separate transition model.

Shared Variables: The modules are independent, except that the transitions of one module may update the program variables of the other module, thus influencing its behavior. Fig. 5 shows a possible modular decomposition of an HDLC primary station. In this example, a shared variable is used to indicate a time-out condition to the *P/F-bit control* module,

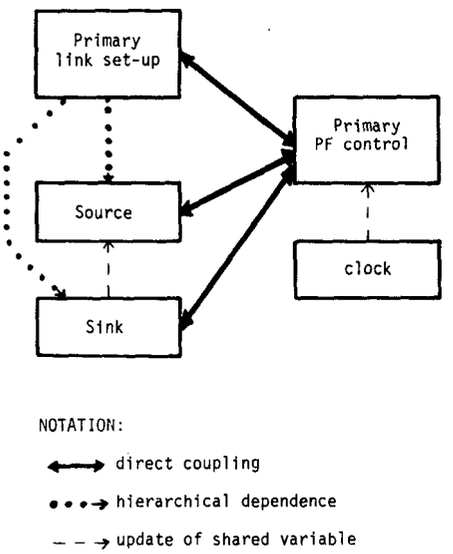


Fig. 5. Modules of an HDLC primary station and their relation.

and the variable *unack* of the *source* module is accessed by the *sink* module when a piggybacked acknowledgment arrives.

Hierarchical Dependence [4]: A module *B* is hierarchically dependent on a module *A* if *B* enters its initial state whenever *A* enters a particular state, which we call the *activating state* for *B*, and the transitions of *B* are only possible while *A* remains in the activating state. In the example of Fig. 5, hierarchical dependence is used to describe the fact that the data transfer executed by the *source* and *sink* modules is only active when the *link setup* module is in the *connected* state (see Fig. 2).

Direct Coupling [4], [7]: This concept introduces a strong synchronization between certain transitions of different mod-

ules. Two transitions of different modules are directly coupled if they can only be executed jointly (only when both respective enabling predicates are true). This mechanism may be used to describe the local interaction of an entity through the upper layer interface with its user or through the lower interface with the underlying transmission medium (see Section III). In a more general form, where a given transition is directly coupled alternatively to several transitions of the other module, this concept was also used for describing the interaction between the *P/F-bit control* module and the other modules of an HDLC station [4], as indicated in Fig. 5. In fact, each sending and receiving transition of the other modules must be coordinated with a transition of the *P/F-bit control* module which checks the validity of the P/F-bit sent or received.

III. SPECIFICATION OF COMMUNICATION SERVICE

The specification of the communication service provided by a given protocol layer (see Fig. 1) defines what the user entities have to know about the protocol layer they use, without being concerned with the details of the protocol. We distinguish the local and the global properties of a communication service. The local properties of the service are those which characterize the local interaction of one user entity with an entity providing the service, ignoring what happens at the other end of the communication link. Given that we consider a *communication* service, the local properties leave an important aspect unspecified, namely, the relation between what happens at the two ends of the communication link. Since the global properties specify this relation, they may be called the "end-to-end" properties of the communication service. We note that the distinction between local and global properties is not "exclusive" since the specification of the global properties of a service usually implies (i.e., includes) its local properties. We include in the following only some simple examples. A complete service specification along the lines discussed here may be found in [14].

A. Local Properties

In this subsection, we concentrate on the local properties of a communication service. These properties clearly determine the local interface through which a user entity accesses the service. The properties may be considered to be the abstract specification for the local interface, which must be satisfied in each local system. At the end of the section, we comment on how this abstract interface may be refined in order to give rise to a particular interface implementation.

1) A Directly Coupled Interface: We assume that both entities that interact through the interface are described by a general transition model, as explained in Section II. We describe the interaction between the two entities by direct coupling. In particular, certain transitions of the service providing entity are directly coupled with certain transitions of the user entity. If we do not want to specify the operation of the user entity (which is usually the case), we may simply give a list of *interface transitions* which may be executed by the user entity subject to some (unspecified) enabling predicates, and which are directly coupled with transitions of the service providing entity. For example, for the entity using the HDLC link layer service, we may define the interface transitions given in Fig. 6. We note that the flow control at the interface is automatically present since a pair of directly coupled transitions may only be executed when the corresponding enabling predicates in both entities are true and no other transition is in progress. Parameterized transitions may be used for passing value parameters between the two entities, such as the *data* parameter in the case of the $\downarrow D$ and $\uparrow D$ transitions.

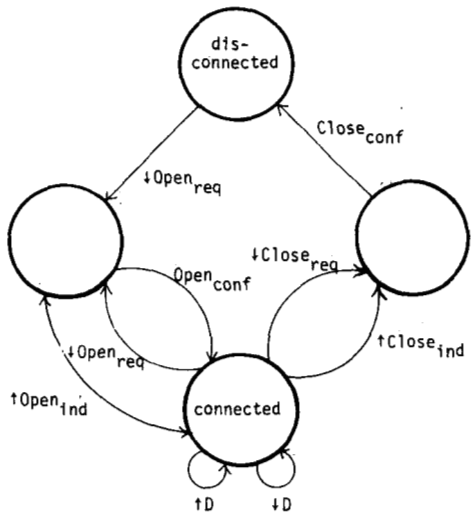
interface transition of user entity	coupled transition of HDLC station (see figures 2 and 4)
$\downarrow Open_{request}$	SXRM starting in *disconnected* or *connected* state
$\uparrow Open_{indication}$	SXRM starting in *CMDR exception* state
$Open_{confirmation}$	UA starting in *Wait for SXRM ack* state
$\downarrow Close_{request}$	DISC starting in *connected* state
$\uparrow Close_{indication}$	DISC starting in *CMDR exception* state
$Close_{confirmation}$	UA starting in *Wait for DISC ack* state
↓D(data:info-block)	a transition appending the *data* parameter into the *buffer* variable of the source module
↑D(data:info-block)	$I_{=}$ where the *data* parameter is equal to *received.data*
Fail	failure

Fig. 6. Interface transitions for the HDLC link layer service and their coupling with the transitions of the service providing HDLC station.

2) Abstraction: While the conceptual operation of the interface may be described by directly coupled transitions, as explained above, we discuss in the following three further abstractions which lead to simpler interface descriptions. The first two abstractions are based on the fact that the user entity does not need to (and should not) know the operation of the protocol which provides the service. The same considerations apply also in the general context of software engineering for the specification of the service provided by a software module. The last abstraction is particular to the context of communications.

Ignoring the Operation of the Protocol: The order in which the interface transitions may be executed by the user entity is clearly determined by the direct coupling and the order in which the transitions of the service providing entity may be executed. Let us consider the example of the layer interface for the HDLC protocol. We may deduce from the information in Figs. 6 and 2 that the interface transitions may be executed in the order shown in Fig. 7. (This diagram is obtained from the diagram of Fig. 2 by merging the *Connected* and *CMDR exception* states, and replacing the transition labels according to the table of Fig. 6. We note that this derivation is generally not so simple because the interaction between the two protocol entities may limit the transition possibilities.)

Combining Interface Transitions into "Service Primitives": Continuing with the example above, we see in Fig. 7 that certain interface transitions are always followed by the same next transition. We may therefore combine these transitions into a single one, thus simplifying the overall transition



(a)

Fig. 7. Local service interface transition diagram based on Figs. 2(a) and 6.

diagram. Adopting the following combinations

↓Open$_{request}$	Open$_{confirmation}$	≡ ↓Open
↑Open$_{indication}$	Open$_{confirmation}$	≡ ↑Open
↓Close$_{request}$	Close$_{confirmation}$	≡ ↓Close
↑Close$_{indication}$	Close$_{confirmation}$	≡ ↑Close

leads to the interface transition diagram of Fig. 8. We call the remaining (partially combined) interface transitions *service primitives.*

Ignoring the Source of Initiation: The symbols "↑" and "↓" in the names of the service primitives have been introduced to explicitly indicate whether the execution of the primitive is initiated by the service providing entity ("↑") or the user entity ("↓"). In the case of the data transmission primitives ↑*D* and ↓*D*, this distinction is clearly important. In the case of the link setup or disconnection primitives, however, this distinction is not always important, in which case one may make abstraction from it. In particular, the diagram of Fig. 8 does not require this distinction; neither does the specification of the global properties of the service discussed in the next section. We therefore drop the symbol "↑" or "↓" whenever this distinction is of no importance.

If we consider the exchange of parameter values between the interacting entities during the execution of a service primitive, the situation may become more complicated. In the case of a primitive for establishing a virtual circuit through a packet-switched data network, for example, a *distant subscriber address* parameter value is provided by the initiating entity, while a *response* parameter value is returned by the other entity. Independently of which entity initiates the primitive, this fact may be described by the following notation:

VC-Establishment ($\rightarrow x$: distant subscriber address, $\leftarrow y$: response code).

3) Discontinuation of Service Primitives: As shown in Fig. 7 and 8, the concept of hierarchical dependence (see Section II-C) may be used to indicate that the normal link layer service is only available as long as the physical circuit is *operational.* Since a *Fail* interface transition may occur any time in the *operational* state, link establishment primitives, for example, will be "interrupted" by a failure which occurs after an Open$_{req}$ transition and before the corresponding Open$_{conf}$ transition (see Fig. 7). We say that the service primitive is *discontinued.* In Fig. 8, this possibility is not shown explicitly, but it must be taken into account. We conclude that whenever the layer interface description involves some hierarchical dependence, the possibility of discontinuation for the dependent service primitives must be considered.

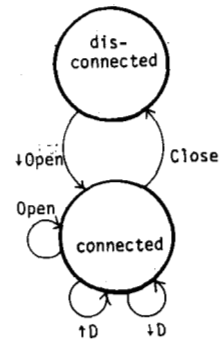


Fig. 8. Simplified local service interface transition diagram.

Another example of discontinuation is given by the virtual circuit data transmission service where, according to X.25, the transfer of a complete user sequence (i.e., variable length data block) between the DTE and the network may be "interrupted" by a reset or circuit clear.

4) Interface Implementation: It is clear that many details must be added to the abstract interface specification suggested in this section in order to obtain an interface implementation. However, these details may be chosen differently for each local implementation, whereas the abstract interface properties discussed in this section must be valid for every actual interface. In particular, the mechanisms for implementing flow control and the distinction between which entity initiates a service primitive may be implemented in quite different ways. For instance, the use of message queues between the service providing entity and its user would be a particular way of implementing the interface.

B. Global Properties

An interface description, as discussed in Section III-A, defines the service primitives and the order in which these primitives may be executed at a local interface between a user and a service providing entity. Here we concentrate on the global properties of a communication service, which are those aspects that make the service useful for *communication.* The local service interface description for the HDLC protocol, for example, states that sending and receiving of user data blocks is possible in the *connected* state (see Fig. 8). Only the global properties state that the first block received at one end is equal to the block first sent at the other end.

The global properties of a communication service usually have two aspects: 1) restrictions on the order in which the service primitives at the two ends of the link may be executed, and

2) restrictions on the possible parameter values exchanged. An example of the second aspect is given above; an example for the first aspect is the fact that (usually) the number of possible receive executions at one end is always smaller or equal to the number of send executions performed at the other end.

Speaking about the execution order of service primitives at different locations brings up the problem of how such an order can actually be observed or enforced. We assume, for the present purposes, that the execution order at different locations can be determined by some hypothetical observer or with sufficiently well synchronized real time clocks.

We use the following notation. Given two service primitives A and B, "$A \Rightarrow B$" means that the beginning of the execution of A is earlier (in real time) than the end of the execution of B (that is, there may be a causal influence of A on B). The notation "$A \Leftrightarrow B$" means that $A \Rightarrow B$ and $A \Leftarrow B$ holds (that is, there is some instant (in real time) when both service primitives are in progress). We say that A and B are *simultaneous.*

For many purposes, instead of considering the execution order to be defined in respect to the real time, it may be adequate to consider that the execution order defined by the global properties of the service determine some partial order of events which represents some "logical time" as discussed by Lamport [10].

A Possible Notation: A possible notation for specifying the global properties of a communication service are production rules of a particular form. We adopt the usual convention of writing the nonterminal symbols in brackets ⟨...⟩, and writing the possible productions after the symbol "::=". Each production is defined in terms of (possibly other) nonterminals and terminals which are written in the form $\{{X \atop Y}\}$. X and Y are sequences of service primitives which describe a possible pair of corresponding execution sequences at the respective ends of the communication link.

As an example, Fig. 9 contains a possible specification of the global properties of an HDLC link layer service. Rule 2, for instance, states that an ⟨Open Sequence⟩ consists of *Open* primitives executed simultaneously at both ends of the communication link followed by a ⟨Data Sequence⟩ with possibly further repetitions. The ⟨Data sequences⟩ are defined by rules 3 and 4, and rule 1 defines the possible global execution sequences which consist of a repetition of an ⟨Open Sequence⟩ followed by a pair of simultaneous *Close* primitives executed at the two ends of the link. (We note that the rules of Fig. 9 imply the "local" transition rules given in Fig. 8 which apply separately at each end of the communication link.)

Restrictions on the possible parameter values may be stated for each of the production rules. In the case of Fig. 9, the only parameters exchanged are the user data sent and received (see rule 4 of Fig. 9). In the case of the establishment of a virtual circuit, using the service primitive given in Section III-A2), the following rule may apply:

$$\langle \text{VC Open} \rangle ::= \left\{ \begin{matrix} \text{VC-Establishment}\ (x, y) \\ \Updownarrow \\ \text{VC-Establishment}\ (x', y') \end{matrix} \right\}$$

where $y = y'$, x = subscriber address of the entity executing the "lower" part, and x' = subscriber address of the entity executing the "upper" part.

(1) <Link Seq> ::= empty

::=<Link Seq> <Open Seq> $\left\{ \begin{matrix} \text{Close} \\ \Updownarrow \\ \text{Close} \end{matrix} \right\}$

(2) <Open Seq> ::= $\left\{ \begin{matrix} \text{Open} \\ \Updownarrow \\ \text{Open} \end{matrix} \right\}$ <Data Seq>

::=<Open Seq> $\left\{ \begin{matrix} \text{Open} \\ \Updownarrow \\ \text{Open} \end{matrix} \right\}$ <Data Seq>

(3) <Data Seq> ::=<Fifo Seq 12> ‖ <Fifo Seq 21> , i.e. arbitrary interleaving of data transfer in both directions

(4) <Fifo Seq 12> ::= $\left\{ \begin{matrix} \downarrow D(x_1) & \downarrow D(x_2) & \ldots & \ldots & \downarrow D(x_n) \\ \Downarrow & \Downarrow & & & \\ \uparrow D(x'_1) & \uparrow D(x'_2) & \ldots & & \uparrow D(x'_m) \end{matrix} \right\}$

where $0 \leq m \leq n$ and

$x_i = x'_i$ for $i = 1,2,\ldots m$

<Fifo Seq 21> ::= ...(similarly)

(5) Discontinuation due to a failure: The execution sequences defined above for entity 1 and entity 2 may be "interrupted" by a local *Fail* transition, such that the last primitive executed by an entity may be discontinued. If A ⇒ B holds between executions of two service primitives the following is true: A is completely suppressed due to the failure implies that B is discontinued or completely suppressed.

Fig. 9. Global properties of the link layer communication service.

C. Elements for a Communication Service Specification

We conclude from the foregoing discussion that the specification of a communication service for a given protocol layer should contain the following elements.

1) An informal explanation of the service provided and the functions included in the layer: this part is given in natural language. It should give an overall understanding of the purpose and operation of the layer.

2) A list of service primitives available at the layer interface: this parts describes precisely each of the service primitives individually.

3) Local properties determining in which order the service primitives may be executed at one end of the communication link without regard to the other end.

4) Global properties relating the execution order and exchanged parameter values at both ends of the communication link: this is the essential part of the service specification.

5) Grade of service considerations: they specify quantitative properties such as throughput, delay, etc., and also indicate in which situations and with which probabilities certain malfunctions, such as undetected errors and failures, may occur. (In contrast to this, points 2)–4) above concentrate on qualitative properties of the service which are always satisfied.)

We believe that any communication service specification that does not contain the equivalent of the elements 2)–5) must be considered incomplete. Elements 2)–4) are discussed in the foregoing sections. We believe that formal methods,

similar to those described here, may be useful for specifying these elements in a more precise manner.

IV. PROTOCOL VERIFICATION

Instead of giving a review of protocol verification (which may be found elsewhere [6]) or describing any particular approach to verification, we give in the following some remarks which show the relation of the previous sections with the problems of protocol verification, and which show also, we hope, that many approaches to verification are basically very simple.

A. What Should be Verified?

The term "protocol verification" usually means to ascertain that the entities executing a given protocol together with the underlying transmission medium (see Fig. 1) actually provide the specified communication service to the user entities in the layer above. It is therefore necessary to determine the service actually provided (based on the specification of the underlying transmission service and the definition of the communicating entities) and compare it with the communication service specified. Let us assume that we want to verify that the service actually provided is equal to the service specified. The proof may be divided into two parts.

1) Partial correctness: to show that every execution sequence of service primitives (at both ends of the communication link, and including specific parameter values) that is actually possible satisfies the constraints imposed by the service specification.

2) Effective progress: to show that every execution sequence of service primitives that satisfies the service specifications is actually possible, and that no situations of deadlock or starvation or infinite loops without progress exist.

B. Various Kinds of Assertions

The use of "assertions" is a well-known technique for the verification of sequential programs and has been extended for use with parallel programs. Similar techniques also apply to the verification of protocols. The basic idea consists of defining an invariant assertion, or briefly "invariant," i.e., a Boolean expression depending on the state of the system which is always true (i.e., as long as no state transition is in progress). Since this technique was developed for verifying programs, it seems natural to use it for verifying protocols that are defined in terms of program variables and executed statements. In this case, the invariants typically involve the program variables of both entities and the state of the underlying transmission medium (i.e., the "messages" in transit) [5], [11].

It is interesting to note that certain approaches to the verification of protocols based of FS description techniques may be shown to be based on a particular form of invariant assertions. For example, the equations given in [7] for the "adjoint states" of a protocol are such that the following assertions are always true when the underlying medium is empty (i.e., no "message" in transit). If $a_i (i = 1, 2, \cdots, n)$ are the possible states for entity 1, and s_1 and s_2 are the actual states of entity 1 and entity 2, respectively, then the assertion

$$s_1 = a_i \text{ implies } s_2 \text{ is an element of Adj } (a_i)$$

holds for every possible state a_i. This is not surprising since the definition of "adjoint state," roughly speaking, is as follows. The adjoint states $Adj(a_i)$ of a given state a_i are those states of entity 2 in which entity 2 may possibly be when entity 1 is in state a_i.

Another example is the detection of incompleteness or overspecifications as described by Zafiropulo *et al.* (see, for example, [12]). Their main idea is as follows. Given an FS protocol definition, an invariant assertion of the following form is derived for each possible state $a_i (i = 1, 2, \cdots, n)$ of entity 1:

$$s_1 = a_i \text{ implies the messages ... may now be received by the entity 1, but no other messages.}$$

Given such assertions, it is easy to check whether the definition of entity 1 includes all necessary receiving transitions and no unnecessary ones. It is sufficient to verify, for any given state a_i, that the definition foresees the handling of exactly those received messages which are mentioned in the corresponding assertion.

In the case of a protocol definition in terms of the general transition model described in Section II where the state of an entity is defined by an FS transition diagram and certain program variables, invariant assertions are in general of the following form:

$$s_1 = a_i \text{ and } s_2 = b_j \text{ implies Assertion}_{ij}$$

where a_i and b_j are possible states of the entities 1 and 2, respectively, and Assertion_{ij} is a Boolean expression depending on the program variables of both entities and possibly also on the state of the underlying transmission medium [13].

As an example, we give the following invariant assertion which may be derived from the definitions of the HDLC procedures given in the Figs. 2–4 and the assumption that each frame received without error notification is an exact copy of a frame sent by the other entity.

$$s_1 = \text{connected and } s_2 = \text{connected implies}$$

$$\left[\begin{array}{l} \text{entity2.received.kind} = I \text{ and} \\ \text{entity2.received.NS} = \text{entity2.VR} \\ \text{implies} \quad \text{entity2.received.data} = \\ \qquad\qquad \text{entity1.buffer.data (entity2.VR)} \end{array}\right]$$

This assertion is important for the verification of correct data transfer of the HDLC procedures. It specifies conditions under which a data block received by entity 2 is equal to the corresponding data block in the buffer of entity 1. Given the definitions of the service primitives $\downarrow D$ and $\uparrow D$ (see Fig. 6) and the transition $I_=$ (see Fig. 4), this invariant assures that the data

blocks received by the user from entity 2 are the same as those submitted by the user to entity 1. This is what rule 4 of the service specification in Fig. 9 postulates.

We conclude that the above invariant assertion proves the partial correctness of the HDLC protocol, as far as rule 4 of the service specification is concerned. However, it does not imply effective progress, which would mean that each data block submitted to entity 1 will eventually be delivered to the user by entity 2. For proving this, we must rely on the underlying transmission service not to make "too many" transmission errors. A more detailed discussion of a simple protocol verification example in the context of the general transition model is given in [1].

V. CONCLUSIONS

In the framework of distributed system architecture involving a hierarchy of different protocol layers, the clear delimitation between the different layers becomes an important issue. The delimitation between a given layer and its user is given by the layer interface which is characterized by the communication service provided through that interface. For the description of the layered architecture of a distributed system, the service specifications for the individual layers seem to be the main tool. For instance, one objective for a layered system architecture is the possibility to change the protocol adopted in a given layer without affecting the other layers of the system. During such a change, the protocol of that layer clearly changes, while the service provided must remain unchanged.

Because the communication service definitions play such an important role in the design of distributed systems, great care should be taken for their exact specification. This paper presents a possible formal approach to the specification of communication services. While a finite state approach seems to be useful for many aspects of communication protocol specification and verification (although not all), we feel that, for the specification of communication *services*, the finite state approach alone is insufficient. It seems that important service characteristics are naturally described by constraints on parameter values which are exchanged over the interface during the execution of the service primitives. The two aspects of "order of execution" and "exchanged parameter values" seem to correspond to the two aspects of our general transition model described in Section II, namely, "state transitions" and "program variables."

ACKNOWLEDGMENT

I would like to thank J. Gecsei for suggesting many improvements on the manuscript.

REFERENCES

[1] G. V. Bochmann and J. Gecsei, "A unified model for the specification and verification of protocols," in *Proc. IFIP Congr. 1977*, pp. 229–234.
[2] D. L. Parnas, "The use of precise specifications in the development of software," in *Proc. IFIP Congr.* 1977, pp. 861–867.
[3] R. M. Keller, "Formal verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 371–384, July 1976.
[4] G. V. Bochmann and R. J. Chung, "A formalized specification of HDLC classes of procedures," in *Proc. Nat. Telecommun. Conf.*, Los Angeles, CA, Dec. 1977, pp. 03A.2-1–2-11; reprinted in *Advances in Computer Communications and Networking*, W. W. Chu., Ed. Dedham, MA: Artech House, 1979.
[5] N. V. Stenning, "A data transfer protocol," *Comput. Network*, vol. 1, pp. 99–110, Sept. 1976.
[6] G. V. Bochmann and C. Sunshine, "Formal methods in communication protocol design," this issue, pp. 624–631.
[7] G. V. Bochmann, "Finite state description of communication protocols," in *Proc. Comput. Network Protocols Symp.*, Univ. Liège, Liège, Belgium, Feb. 1978, pp. F3-1–F3-11; and *Comput. Networks*, vol. 2, pp. 361–372, Oct. 1978.
[8] G. V. Bochmann and T. Joachim, "Development and structure of an X.25 implementation," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 429–439, Sept. 1979.
[9] G. V. Bochmann and F. H. Vogt, "Message link protocol—Functional specifications," *ACM Comput. Commun. Rev.*, vol. 9, pp. 7–39, Apr. 1979.
[10] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558–565, July 1978.
[11] G. V. Bochmann, "Logical verification and implementation of protocols," in *Proc. 4th Data Commun. Sympo.*, ACM/IEEE, 1975, pp. 8-15–8-20.
[12] P. Zafiropulo *et al.*, "Towards analyzing and synthesizing protocols," this issue, pp. 651–661.
[13] G. V. Bochmann, "Combining assertions and states for the validation of process communication," in *Constructing Quality Software*, P. G. Hibbard and S. A. Shuman, Ed. Amsterdam: North-Holland, 1978, pp. 229–232.
[14] ——, "Specification of the services provided by the MLP," Univ. Montreal, Montreal, P.Q., Canada, Tech. Rep., 1979.

★

Gregor V. Bochmann, for a photograph and biography, see this issue, p. 631.