Formal Methods in Communication Protocol Design

Article *in* IEEE Transactions on Communications · May 1980

DOI: 10.1109/TCOM.1980.1094685 · Source: IEEE Xplore

Gregor V. Bochmann
University of Ottawa

serves as Vice-Chairman (United States) of IFIP Working Group 6.4 on Local Computer Networks.

★

Edward A. Taft received the B.A. degree in applied mathematics from Harvard University, Cambridge, MA, in 1973.

Since then he has been a member of the Computer Science Laboratory of the Xerox Palo Alto Research Center, Palo Alto, CA, working in the areas of internetwork protocols, distributed systems, and personal computing.

Robert M. Metcalfe received the S.B. degree in electrical engineering and the S.B. degree in management from the Massachusetts Institute of Technology, Cambridge, in 1969, and the M.S. degree in applied mathematics and the Ph.D. degree in computer science from Harvard University, Cambridge, MA, in 1970 and 1973, respectively.

His Ph.D. dissertation is titled "Packet Communication." He is presently President of 3Com Corporation and Consulting Associate Professor of Electrical Engineering at Stanford University, Stanford, CA, where he has been lecturing on distributed computing since 1975. He was with Xerox Corporation, Palo Alto, CA, between 1972 and 1979, where he worked on ARPANET, Ethernet, Fibernet, Pup, and Laurel. In June 1979 he formed 3Com Corporation to promote, develop, and exploit communication compatibility among computers in the office and home.

Formal Methods in Communication Protocol Design

GREGOR V. BOCHMANN AND CARL A. SUNSHINE

(Invited Paper)

Abstract—**While early protocol design efforts had to rely largely on seat-of-the-pants methods, a variety of more rigorous techniques have been developed recently. This paper surveys the formal methods being applied to the problems of protocol specification, verification, and implementation.**

In the specification area, both the service that a protocol layer provides to its users and the internal operations of the entities that compose the layer must be defined. Verification then consists of a demonstration that the layer will meet its service specification and that each of the components is correctly implemented. Formal methods for accomplishing these tasks are discussed, including state transition models, program verification, symbolic execution, and design rules.

I. INTRODUCTION

As evidenced by the earlier papers of this issue, increasingly numerous and complex communication protocols 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 paper describes some of the more formal techniques which are being developed to facilitate 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, and 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 by themselves they are inadequate.

In the following sections, we shall discuss the use of formal techniques in each of the major design steps of specification, verification, and implementation. Section II clarifies the meaning of specification in the context of a layered protocol architecture, identifies what a protocol specification should include, and describes the major approaches to protocol specification. Section III defines the meaning of verification, discusses what can be verified, and describes the main verification methods. Section IV provides pointers to some important case histories of the use of these techniques. For detailed examples, we refer to the subsequent papers of this issue which generally provide additional support for the points which we have had to treat briefly in this survey. A complete bibliography may be found in [18], and complementary surveys in [44], [8], [33], [43].

II. PROTOCOL SPECIFICATION

As noted above, protocol descriptions play a key role in all stages of protocol design. This section clarifies the meaning of specification in the domain of communication protocols,

Manuscript received August 8, 1979; revised January 8, 1980. This work was supported in part by the National Sciences and Engineering Council of Canada and the United States Advanced Research Projects Agency.

G. V. Bochmann is with the University of Montreal, Montreal, P.Q., Canada.

C. A. Sunshine is with the Information Sciences Institute, University of Southern California, Marina del Rey, CA 90221.

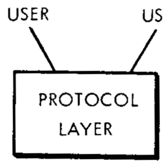

Fig. 1.

identifies the major elements that comprise a specification, and presents the major methods for protocol specification.

A. The Meaning of Specification

We assume that the communication architecture of a distributed system is structured as a hierarchy of different protocol layers, as described in earlier papers of this issue. Each layer provides a particular set of *Services* to its users above. From their viewpoint, the layer may be seen as a "black box" or machine which allows a certain set of interactions with other users (see Fig. 1). A user is concerned with the nature of the service provided, but not with how the protocol manages to provide it.

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 mechanisms for conveying them (e.g., procedure calls, system calls, interrupts, etc.). These formats and mechanisms may be different for users in different parts of the system, and are defined by an *Interface Specification*.

Service Specifications

Specifying the service to be provided by a layer of a distributed communication system presents problems similar to specifying any software module of a complex computer system. Therefore methods developed for software engineering [36], [31] are useful for the definition of communication services. Usually, a service specification is based on a set of *Service Primitives* which, in an abstract manner, describe 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 entities involved, i.e., the service providing and using entities of two adjacent layers. The possible parameter 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 making use of a "state" of the service which may be changed as a result of some 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 a transport service, a local constraint is the fact that *Send* and *Receive* may only be executed 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.

Protocol Specifications

Although irrelevant to the user, the protocol designer must be concerned with the internal structure of a protocol layer. In a network environment with physically separated users, a protocol layer must be implemented in a distributed fashion, with *Entities* (processes or modules) local to each user communicating among one another via the services of the lower layer (see Fig. 2). The interaction among entities in providing the layer's service constitutes 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).

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 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 distributed "implementation" of its service specification in the sense that it partly defines how the service is provided (i.e., by a set of cooperating entities). This "implementation" of the service is what is usually meant by the design of a protocol layer. The protocol specification should define each entity to the degree necessary to ensure compatability with the other entities of the layer, but no further. 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 [20], [11].

B. What a Protocol Definition Should Include

A protocol cannot be defined without describing its context. This context is given by the architectural layer of the distributed system in which the protocol is used. A description of a layer should include the following items [28].

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 provided by the layer below, and required for the correct and efficient operation of the protocol. (This of course is redundant with the lower layer's definition, but makes the protocol definition self-contained.)

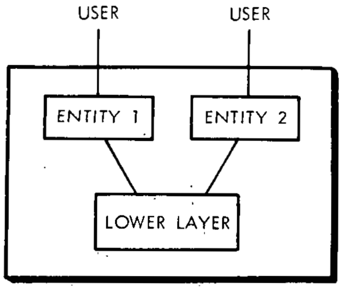

Fig. 2.

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:

a) An overall, informal description of the operation of the entities.

b) A protocol specification which includes:

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

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

c) Any additional details [not included in point b)], such as considerations for improving the efficiency, suggestions for implementation choices, or a detailed description which may come close to an implementation.

Reference [8] presents an example of these items for a simple data transfer protocol.

C. Specification Methods

Descriptions of communication services and protocols must be both easy to understand and precise—goals which often conflict. The use of natural language gives the illusion of being easily understood, but leads to lengthy and informal specifications which often contain ambiguities and are difficult to check for completeness and correctness. The arguments for the use of formal specification methods in the general context of software engineering [36] apply also in our context.

Protocol Specifications

Most of the work on formal specification of protocols has focused on the protocol itself and not on the service it provides. A variety of general formalisms such as state diagrams, Petri nets, grammars, and programming languages have been applied to this problem and in many cases adaptations or extensions to facilitate protocol modeling have been made [14]. These techniques may be classified into three main categories: transition models, programming languages, and combinations of the first two.

Transition models are motivated by the observation that protocols consist largely of relatively simple processing in response to numerous "events" such as commands (from the user), message arrival (from the lower layer), and internal timeouts. Hence state machine models of one sort or another with such events forming their inputs are a natural model. However, for protocols of any complexity, the number of events and states required in a straightforward transition formalism becomes unworkably large. For example, to model a protocol using sequence numbers, there must be different states and events to handle each possible sequence number [37]. Models falling into this category include state transition idagrams [4], [50], grammars [25], [47], Petri nets and their derivatives [35], [46], *L*-systems [19], UCLA graphs [37], and colloquies [30].

Programming language models [24], [41], [6], [21] are motivated by the observation that protocols are simply one type of algorithm, and that high-level programming languages provide a clear and relatively concise means of describing algorithms. Depending on how high level and abstract a language is used, this approach to specification may be quite near to an implementation of the protocol. As noted above, this proximity is a mixed blessing, since unessential features in the program are often combined with the essential properties of the algorithm. A major advantage of this approach is the ease in handling variables and parameters which may take on a large number of values (e.g., sequence numbers, timers), as opposed to pure state machine models.

Hybrid models [45], [16], [10], [40], [3] attempt to combine the advantages of state models and programs. These typically employ a small-state model to capture only the main features of the protocol (e.g., connection establishment, resets, interrupts). This state model is then augmented with additional "context" variables and processing routines for each state. In such hybrid models, the actions to be taken are determined by using parameters from the inputs and values of the context variables according to the processing routine for each major state. For example, the sequence number of an arriving message may be compared with a variable giving the expected next sequence number to determine whether to accept the message, what the next state should be, and how to update the expected sequence number. Bochmann and Gecsei [10] have demonstrated the potential for trading off the complexity of the state model with the amount of context information for a given protocol. (Other techniques for managing the complexity of protocols are discussed in Section III-C.)

As noted above, one major goal of protocol specification is to provide a basis for implementation. Some specification methods facilitate this goal more than others. Programming language specifications may be quite close to implementations (but often lack the desired degree of abstraction). Direct implementation of transition or hybrid model specifications by some form of interpreter or "compiler" is also relatively straightforward [11]. In many cases, these implementation methods have been at least partially automated [40], [47], [22], [2].

Service Specifications

It is only recently that the need for comprehensive protocol service specifications has been realized [3], [43]. Initial efforts at formal service specifications have been directed towards applying general software engineering methodology. As noted in Section II-A, definition of the primitive operations supported by the layer (e.g., *Send, Connect*) is a basic feature of any specification. In abstract machine type specifications, internal "states" of the layer are also defined. These states are used in defining the effects of each operation, and may be changed as a result of the operation [38].

Other specification methods that do not require definition of explicit states may also be used. I/O history type methods define the allowed input and output sequences of the layer and their relation to each other (e.g., that the sequence of messages delivered is identical to the sequence of messages sent) [20], [3]. Algebraic specifications [23] provide another way of defining the allowed sequence of operations. Sunshine [42] provides a comparison of several of these methods, but work in this area is just beginning.

III. PROTOCOL VERIFICATION

In its broadest interpretation, system validation aims to assure that a system satisfies its design specifications and (hopefully) operates to the satisfaction of its users. Validation activity is important during all design phases, and may include testing of the final system implementation, simulation studies, analytical performance predictions, and verification. Verification is based on the system specification, and involves logical reasoning. Therefore it may be used during the design phase before any system implementation exists, in order to avoid possible design errors. While testing and simulation only validate the system for certain test situations, verification allows, in principle, the consideration of all possible situations the system may encounter during actual operation.

A. The Meaning of Verification

Verification is essentially a demonstration that an object meets its specifications. Recalling from Section II that *Services* and *Protocol Entities* are the two major classes of objects requiring specification for a protocol layer, we see 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 and communicating through the underlying layer's service, to see whether this combined operation satisfies the layer's service specification; and 2) the *Implementation* of each protocol entity must be verified against its abstract protocol 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 via (possibly unreliable) 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 from protocol verifiers.

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 consistency [20]. It must also properly reflect the users' desires, and provide an adequate basis for the higher levels which 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. 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.

Most of the verification work to date has been on design rather than implementation, and we shall focus on design verification in the remainder of this section. While a protocol design need only be verified once, each different implementation must be verified against the design.

B. 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 may be considered to 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. Progress or termination may also be considered in this category since they 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 a transport protocol, copying a file in a file transfer protocol, or 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 has the usual meaning that if the protocol performs any action at all, it will be in accord with its service specification. For example, if a transport protocol 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 report, it is sufficient to ascertain a finite time delay. In the case that the efficiency and responsiveness of the protocol is to be verified, it is clearly necessary to determine numerically the expected time delay, throughput, etc.

C. Verification Methods

Approaches to protocol verification have followed two main paths: reachability analysis and program proofs. Within the scope of this paper, we can only outline these two approaches. The references cited in Section IV provide more details on particular techniques.

Reachability analysis is based on exhaustively exploring all 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 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 layer (the type of service it offers), this type of analysis determines all of the possible outcomes that the protocol may achieve. Reference [50] provides 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 [24]. Symbolic execution (see the following) 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 for transition models is easily automated, and several computer aided systems for this purpose have been developed [16], [50], [37]. The major difficulty of this technique is "state space explosion" because 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. Techniques for dealing with this problem are discussed below.

The program proving approach involves the usual formulation of assertions which 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 of his own. The basic task is then to show (prove) that the protocol programs for each entity satisfy the high-level assertions (which usually involve both entities). This often requires formulation of additional assertions at appropriate places in the programs [41], [6].

A major strength of this approach is its ability to deal with the full range of protocol properties to be verified, rather than 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.

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 which accompany the state model. Such combined techniques are described in [16] and [10].

While a large body of work on general program verification 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 few early applications of general program verification methods to protocols are cited in Section IV.

A particular form of proof that has been useful for protocols with large numbers of interacting entities (e.g., routing protocols) may be called "induction on topology" [33]. The desired properties are first shown to be true for a minimum subset of the entities, and then an induction rule is proved showing that if the properties hold for a system of N entities, they also hold for $N + 1$ entities.

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 place where the error occurs, as 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 may be formulated incorrectly. In other cases, undesired behavior may be due to 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 approach to achieving correct protocols that has been proposed recently is based on constructive design rules that automatically result in correct protocols. In one case [50], design rules are formulated which guarantee that the specifications obtained for a set of interacting entities will be complete. For each send transition specified by the designer, the rules determine the corresponding receive transition to be added to the partner entity. In another case [34], the specification of a second entity is determined by a design rule such that it will operate with a specified first entity to provide a given overall service.

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 keep this complexity within manageable limits.

1) *Partial Specification and Verification*: Depending on the specification method used, only certain aspects of the protocol are described. This is often the case for transition diagram

specifications 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 is due to 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" in the global protocol description.

A particular application of this idea is to consider only states where the transmission medium is empty. Such an "empty medium abstraction" [4] 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 and/or Phases*: The decomposition of the protocol of a layer into several sublayers and/or phases of operation simplifies the description and verification, because the protocol of each part may be verified separately. An example of this idea is the decomposition of HDLC into the sublayers of bit stuffing, checksumming, and elements of procedure, and the division of the latter into several components as described in [9].

4) *Classifying States by Assertions*: Assertions which are predicates on the set of all possible system states may be formed. Each predicate defines a set (or class) of states which consists of those states for which the predicate is true. One may then consider classes 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 proving partial correctness of protocol specifications given in some programming language [41], [6], and also in the case of symbolic execution [13]. Typically, the assertions depend on some variables of the entities and the set of messages in transit (through the layer below).

To illustrate the possible savings in the number of cases to be analyzed, consider the state of an entity receiving numbered information frames. Instead of treating all possible values of a sequence counter variable explicitly as different states, it may be possible to consider only the three cases where the variable is "less than," "equal to," or "greater than" the number in the information frame received.

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 [16].

6) *Automation*: Some steps in the analysis process may be performed by automated systems [13], [16], [20], [24], [50], [37], [13]. 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.

IV. USES OF FORMAL TECHNIQUES

We give in the following 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 specification 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 specification was used as a reference document during the system design.

Standards

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 [49]. The analysis checked for general correctness properties of completeness and deadlock, and uncovered a number of completeness errors (i.e., a protocol module could receive 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 by a manual reachability analysis [3], [1], [7]. It was found that several cycles with no useful progress could persist after the protocol once entered certain unsynchronized states.

A formal specification method was used during the design of several interface standards for the interconnection of minicomputers with measurement and instrumentation components [26], [48]. The relatively concise description of the protocols was used as means for communication between the members of the standard 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 [25] 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 [9]. The latter specification 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 [11].

Arpanet

Connection establishment in a transport protocol (TCP) for the ARPANET has been partially modeled with a hybrid state transition model and validated with a manual reachability analysis [45]. An automated reachability analysis [24] was also used on a simplified model and revealed an error in sequence number handling, and incorrect modeling of the transmission medium.

A simplified version of the ARPANET IMP-IMP link 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 [35].

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 [20], [21]. A software engineering system (called *Gypsy*) was used which 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. Program modules can be both comprehensively verified in advance, or checked against their specifications at run time for the particular inputs which occur.

Connection establishment between a requester and a shared server process (the ARPANET Initial Connection Protocol) has been modeled with a state transition model and analyzed by an automated reachability analysis [37]. 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.

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 macrolanguage, derived from the original protocol specification [51]. The same specification was also the basis for simulation studies which provided valuable results for the protocol validation and performance evaluation [29], [17]. A formalized specification of the protocol has also been given using a hybrid model with state machines augmented by context information and processing routines [15].

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 [32]. 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.

IBM's SNA has been specified with a hybrid model using state machines augmented by context information and processing routines [40]. Hierarchical decomposition is heavily used to create a large number of more manageable modules. The model provides a basis for both automated verification of general properties, and for compilation of executable code.

The Message Link Protocol [12] for process-to-process communication has been formally specified in a hybrid model. A formal service specification was also given, and the design has been partially verified by a manual reachability analysis using symbolic execution [5]. The verification uncovered a synchronization problem that has been corrected in a more recent version of the protocol.

V. CONCLUSIONS

The specification of a protocol layer must include definitions of both the services to be provided by the layer, and the protocol executed by the entities within the layer to provide this service. "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 the more abstract protocol specification. 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 in many cases without requiring any particular service specification.

Although protocol specifications must serve many purposes, verification and implementation are two critical tasks which require rigorous or formal specification techniques in order to be fully successful. Formal protocol specifications are more precise than descriptions in natural language, and should contain the necessary details for obtaining compatible protocol implementations on different system components. The cases mentioned in Section IV demonstrate that formal methods may be used profitably for the specification, verification, and implementation of communication protocols. However, a great deal of work remains to be done in improving verification techniques and high-level system implementation languages, in integrating performance (efficiency) analysis with analysis for logical correctness, and in automating these analysis techniques.

Most published papers on protocol verification present some particular verification technique, and demonstrate this technique by discussing its application to a simple protocol of more or less academic nature. This is not surprising, considering the short history of this specialized discipline. Some *a posteriori* verifications of protocol standards of general concern have been presented pointing out certain difficulties with the adopted procedures [1], [7], [49]. These verification efforts were based on a state reachability analysis, and in one case [49] an automated system was used. The results will influence the implementation of these protocols, and may have an impact on future revisions of the standards.

We believe that more effort should be spent on the logical verification of protocols during the design phase. Based on a formalized description method, this effort may in the future be simplified by the use of interactive automated systems for protocol verification. The same protocol specification used for the verification should also serve as an official definition of the protocol, and could be transformed, possibly through a semiautomated process into a usable protocol implementation [40], [22], [11]. It is clear that such an approach would increase the reliability of the protocols, decrease compatibility problems, and lower the cost of the protocol implementations.

REFERENCES

[1] D. Belsnes and E. Lynning, "Some problems with the X.25 packet level protocol," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 7, pp. 41–51, Oct. 1977.

[2] D. Bjorner, Finite state automation—definition of data communication line control procedures," in *Fall Joint Comput. Conf., AFIPS Conf. Proc.*, 1970.

[3] G. V. Bochmann, "A general transition model for protocols and communication services," this issue, pp. 643–650.

[4] ——, "Finite state description of communication protocols," *Comput. Networks*, vol. 2, pp. 361–372, Oct. 1978.

[5] ——, "Formalized specification of the MLP," "Specification of the services provided by the MLP," and "An analysis of the MLP," Univ. Montreal, Dep. d'I.R.O., June 1979.

[6] ——, Logical verification and implementation of protocols," in *Proc. 4th Data Commun. Symp.*, Quebec, Canada, 1975, pp. 8-5–8-20.

[7] ——, "Notes on the X.25 procedures for virtual call establishment and clearing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 7, pp. 53–59, Oct. 1977; see also [4].

[8] ——, "Specification and verification of computer communication protocols," in *Advances in Distributed Processing Management.* Philadelphia, PA: Heyden & Son, 1980, to be published.

[9] G. V. Bochmann and R. J. Chung, in "A formalized specification of HDLC classes of procedures," in *Proc. Nat. Telecommun. Conf.*, Los Angeles, CA, Dec. 1977, Paper 3A.2.

[10] G. V. Bochmann and J. Gecsei, "A unified model for the specification and verification of protocols," in *Proc. IFIP Congress*, 1977, pp. 229–234.

[11] 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.

[12] G. V. Bochmann and F. H. Vogt, "Message link protocol—Functional specifications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 9, pp. 7–39, Apr. 1979.

[13] D. Brand and W. H. Joyner, Jr., "Verification of protocols using symbolic execution," *Comput. Networks*, vol. 2, pp. 351–360, Oct. 1978.

[14] A. Danthine, "Protocol representation with finite state models," this issue, pp. 632–643.

[15] A. S. Danthine and J. Bremer, "An axiomatic description of the transport protocol of Cyclades," presented at Prof. Conf. Comput. Networks and Teleprocessing, Aachen, Germany, Mar. 1976.

[16] ——, "Modeling and verification of end-to-end transport protocols," *Comput. Networks*, vol. 2, pp. 381–395, Oct. 1978.

[17] A. Danthine and E. Eschenhauer, "Influence on the node behavior of a node-to-node protocol," in *Proc. 4th Data Commun. Symp.*, Oct. 1975, pp. 7-1–7-8.

[18] J. Day and C. Sunshine, Eds. "A bibliography on the formal specification and verification of computer network protocols," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 9, Oct. 1979.

[19] C. A. Ellis, "Consistency and correctness of duplicate database systems," in *Proc. 6th Symp. Op. Syst. Principles*, Purdue Univ., West Lafayette, IN, Nov. 1977; *ACM Op. Syst. Rev.*, vol. 11, pp. 67–84, 1977.

[20] D. I. Good, "Constructing verified and reliable communications processing systems," *ACM Software Eng. Notes*, vol. 2, pp. 8–13, Oct. 1977; also Rep. ICSCA-CPM-6, Univ. Texas at Austin.

[21] D. I. Good and R. Cohen, "Verifiable communications processing in GYPSY," Univ. Texas at Austin, Rep. ICSCA-CPM-11, June 1978.

[22] M. G. Gouda and E. G. Manning. "Protocol machines: A concise formal model and its automatic implementation," in *Proc. 3rd Int. Conf. Comput. Commun.*, Toronto, Canada, 1976, pp. 346–350.

[23] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Commun. Ass. Comput. Mach.*, vol. 21, Dec. 1978.

[24] J. Hajek, "Automatically verified data transfer protocols," in *Proc. 4th Int. Comput. Commun. Conf.*, Kyoto, Japan, Sept. 1978, pp. 749–756; also see progress Rep. in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 8, Jan. 1979.

[25] J. Harangozo, "An approach to describing a link level protocol with a formal language," in *Proc. 5th Data Commun Symp.*, Utah, 1977, pp. 4-37–4-49.

[26] IEEE Standard 488-1975; see also D. E. Knoblock, D. C. Loughry, and C. A. Vissers, "Insight into interfacing," *IEEE Spectrum*, May 1975.

[27] IFIP WG 6.1, "Proposal for an internetwork end-to-end transport protocol," INWG Gen. Note 96.1; also in *Proc. Comput. Network Protocols Symp.*, Univ. Liege, Belgium, Feb. 1978, p. H-5.

[28] International Organization for Standardization, "Reference model of open systems architecture," TC97/SC16/N227, June 1979.

[29] G. LeLann and H. LeGoff, "Verification and evaluation of communication protocols," *Comput. Networks*, vol. 2, pp. 50–69, Feb. 1978.

[30] G. LeMoli, "A theory of colloquies," *Alta Frequenza*, vol. 42, pp. 493-223E–500-230E, 1973; also in *Proc. First European Workshop on Comput. Networks*, Arles, France, Apr. 1973, pp. 153–173.

[31] B. Liskov and S. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7–18, Mar. 1975.

[32] F. Mellor, W. J. Olden, and C. J. Bedard, "A message-switched operating system for a multiprocessor," in *Proc. COMPSAC '77*, IEEE, Chicago, IL, 1977, pp. 772–777.

[33] P. M. Merlin, "Specification and validation of protocols," *IEEE Trans. Commun.*, vol. COM-27, pp. 1671–1680, Nov. 1979.

[34] P. Merlin and G. V. Bochmann, "On the construction of communication protocols and module specifications," Stanford Univ., Tech. Rep. SEL 182, Dec. 1979.

[35] P. M. Merlin and D. J. Farber, "Recoverability of communication protocols—Implications of a theoretical study," *IEEE Trans. Commun.*, vol. COM-24, pp. 1036–1043, Sept. 1976.

[36] D. L. Parnas, "The use of precise specifications in the development of software," in *Proc. IFIP Congress 1977*, pp. 861–867.

[37] J. B. Postel. "A graph model analysis of computer communications protocols," Ph.D. thesis, Comput. Sci. Dep., Univ. California, Los Angeles, UCLA ENG-7410, 1974.

[38] L. Robinson, K. N. Levitt, and B. A. Silverberg, *The HDM Handbook*, vol. I-III, SRI Int., 1979.

[39] A. M. Rybczynski and D. F. Weir, "Datapac X.25 service characteristics." in *Proc. Fifth Data Commun. Symp.*, 1977, pp. 4-50–4-57.

[40] G. D. Schultz *et al.*, "Executable description and validation of SNA," this issue, pp. 661–677.

[41] N. V. Stenning, "A data transfer protocol," *Comput. Networks*, vol. 1, pp. 99–110, Sept. 1976.

[42] C. A. Sunshine, "Formal methods for communication protocol specification and verification," The Rand Corp., N-1429, Nov. 1979.

[43] ——, "Formal techniques for protocol specification and verification," *Comput. Mag.*, vol. 12, pp. 20–27, Sept. 1979.

[44] ——, "Survey of protocol definition and verification techniques," *Comput. Networks*, vol. 2, pp. 346–350, Oct. 1978.

[45] C. A. Sunshine and Y K. Dalal, "Connection management in transport protocols," *Comput. Networks*, vol. 2, pp. 454–473, Dec. 1978.

[46] F. J. W. Symons, "Modelling and analysis of communications protocols using numerical Petri nets," Dep. Elec. Eng., Univ. Essex, England, Tech. Rep. 152, May 1978.

[47] A. Y. Teng and M. T. Liu, "A formal model for automatic implementation and logical validation of network communication protocols," in *Proc. Comput. Networking Symp.*, Nat. Bureau Standards, Dec. 1978, pp. 114–123.

[48] C. A. Vissers and B. V. D. Dolder, "Generative description of DIN 66 202 (E)" German, English, Twente Univ., Rep. 1261, 1881, Mar. 1977.

[49] C. H. West and P. Zafiropulo, "Automated validation of a communications protocol: The CCITT X.21 recommendations," *IBM J. Res. Develop.*, vol. 22, pp. 60–71, Jan. 1978.

[50] P. Zafiropulo *et al.*, "Towards analyzing and synthesizing protocols," this issue, pp. 651–661.

[51] H. Zimmermann, "The Cyclades end-to-end protocol," in *Proc. Fourth Data Commun. Symp.*, 1975, pp. 7-21–7-26.

Gregor V. Bochmann received the Diploma in physics from the University of Munich, Munich, Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

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

Carl A. Sunshine, for a photograph and biography, see this issue, p. 412.