CUCS-140-84

DEVELOPMENT TOOLS FOR COMMUNICATION PROTOCOLS: AN OVERVIEW

Nihal Nounou and Yechiam Yemini
Department of Computer Science
Columbia University
New York, New York 10027

March 1984

ABSTRACT

This paper presents an overview of commonly used protocol development tools falling under two categories: construction tools and validation tools. Construction tools are used to develop protocols from specifications to working systems. They include tools for specification, synthesis, and implementation. Validation tools are used to analyze protocols behavior. They include tools for formal verification, performance evaluation and testing. For each tool, we examine the key underlying issues, outline the main approaches, and illustrate its application to a send-and-wait protocol.

Table of Contents

List of Figures

1 Introduction

A **communication protocol** consists of a set of rules and a set of message formats used for transferring information among distributed processes in a computer network. Communication protocols (briefly protocols) involve processes that are distributed, concurrent, time-dependent, asynchronous, and that communicate through an unreliable transmission medium that could lose, duplicate, reorder, and/or corrupt messages. These render the protocol design problem complex.

The International Standards Organization (ISO) proposed an Open System Interconnection (OSI) standard protocol architecture [Zimm 81], in which each protocol layer (as depicted in Fig. 1) provides *services* to the layers above, concealing the details of the lower layers. A description of the exchange of primitive events between the protocol layer and the layers above and below constitutes the **service specification.** The rules that describe how the protocol layer entities (also called modules, processes, or parties) provide these services are referred to as the **protocol specification.** Specifically, a protocol specification describes how the entities respond to commands from lower and upper layers, messages from other peer entities, and internally initiated actions (e.g., timeouts). The code implementing the procedures constituting the protocol entities is the **protocol implementation.**

Protocol development typically follows three phases corresponding to the model above: *service statement*, *protocol design*, and *implementation*. This phased development is supported by *construction tools* and *validation tools*. Construction tools are used for successive refinement of protocol description. They include specification, synthesis and implementation methods. Validation tools could be used at each phase to assess how the protocol description at that phase meets its design objectives (e.g. freedom of deadlock behavior). They include formal verification, performance evaluation and testing methods.

Recently, there has been a growing interest (see for example [Ansa 82a, Boch 82, Yemi 83]) in building development environments that integrate the various tools required throughout the protocol development process. An ideal development environment should encompass a comprehensive set of development tools, and a methodology for the application of these tools. Therefore, a prime objective of this overview is to examine the **complete** set of commonly used protocol development tools.

The organization of this paper is as follows. In sections 2 through 4 we overview construction tools. Sections 5 through 7 are overviews of validation tools. Finally, in section 8 we conclude with few comments on protocol development environments.

2 Specification Methods

Experience has shown that protocols specified in an informal manner are error-prone (e.g., the specification of the X.21 interface [X.21 76]) due to the ambiguity and incompleteness of the specifications [West 78a]. Formal specifications are concise, complete, unambiguous, and can serve as the basis for other protocol development tool. Indeed, protocol development tools are highly dependent on the specification method used. For example, a different

verification method would be required if the specification method used in the protocol environment is changed.

Key requirements of specification methods for protocols include:

1. Supporting modeling of concurrency.
2. Supporting modeling of nondeterminism, which is required to represent either nondeterministic behavior (e.g., a communication medium could either lose or transmit messages), or nondeterministic mapping between events in the specifications at different development phases (e.g., a user request in the service specification might be realized by more than one behavior in the protocol specification).
3. Allowing for the description of both local views (i.e., describe each entity behavior separately) and global views (i.e., the whole protocol layer behavior) of the protocol, and the composition of these local view into a global view.
4. Supporting the specification of the timing requirements of protocols together with their functional requirements. The value of the time-out period, for example, might greatly affect the behavior of retransmission protocols [Yemi 82].
5. Separating the specification of the protocol function from the specification of its topology (how the distributed entities are connected together). This feature is especially valuable in specifying high-level protocols where the topology might vary without influencing the function of the protocol.
6. Supporting the description of both the syntax (the allowed events and their order) and semantics (the actions taken as a consequence of an event) of the protocol.

Consider, for example the following simple send-and-wait protocol. In this protocol, there are three distributed processes: a sender S, a receiver R, and a transmission medium M. The operation of the protocol is as follows: the sender receives a new message m from a source C, sends it to the receiver through the medium which either delivers it to R or lose it. The sender waits for a time period T (time-out) for an acknowledgment a to arrive, upon which it returns to its initial state waiting for another new message. The receiver process waits for the new message m to arrive from the medium, after which it delivers it to a destination D and then sends an acknowledgment a to the sender through the medium. It is assumed, for simplicity, that the medium does not lose acknowledgments, and that the timeout is ideally set such that a timeout occurs only after a message is lost. Note that if the send-and-wait protocol is at one protocol layer, then the source and destination would be at the next higher layer and the medium represents the next lower layer.

Let us illustrate how the send-and-wait protocol can be specified using a simple specification method: a pure *finite state machine* (FSM). A FSM consists of the following components: 1) finite set of states, 2) finite set of input commands, 3) finite set of outputs 4) transition functions from (command $\times$ state) $\rightarrow$ state, and 5) an initial state. FSMs are a natural choice for specifying protocol since protocols consist primarily of simple processing (reactions) in response to different kinds of events (actions). Actions (reaction) could be commands from (to) the upper protocol layer, messages from (to) the lower layer or internal events from (to) peer entities in the same layer. A FSM responds to an input command according to the input itself and the history of past inputs which is represented explicitly in terms of states.

A FSM specification of the send-and-wait protocol is shown in Fig. 2, where specifications of the three local processes are given. In that figure, events with an overbar are send events (output commands), events with an underbar are receive events (input commands) and subscripts are used such that for event $e_{i,j}$, flow of data is from i to j. Non-deterministic behavior, such as the choice at state 3 of the sender between receiving a timeout or an acknowledgment, is modeled by multiple output arcs from that state. Since this specification describes the operation of the different entities involved in the protocol, then it is a protocol specification. A service specification for the same protocol is shown in Fig. 3 in which the service primitive events GET and DELIVER between the protocol system and its users (source and destination processes) and their order are described.

For this simple protocol, FSM specifications prove to be adequate. Unfortunately, this is not the case for more complex protocols. For example, FSM specifications of protocols that involve message sequence numbers (the send-and-wait protocol has a sequence number window of size one) suffer from an explosion in the number of states. These problems are typically addressed by schemes allowing more complex state and transitions descriptions. Therefore, extensions of FSMs , which we call *state machines*, allowing state variables and constructs for expressing predicates and actions to be associated with transitions have been introduced e.g. [Boch 77, Schw 81, Divi 82] .

Other specification methods include: *petri-nets-based methods* which allow modeling of concurrency and interactions between the distributed processes [Dant 78, Symo 80, Ayac 81], *formal languages* with their similarities to FSMs [Teng 78], *sequence expressions* which specify only the valid protocol events and their order without any explicit mention of states [Schi 81], *algebraic specifications* in which the protocol system is specified in terms of objects and operations on these objects [Muss 80, Miln 80] *temporal logic* with its reasoning about temporal properties required of a protocol execution as well as its static properties [Schw 82], and *procedural languages* where the unit of specification is a procedure containing detailed data declarations and computational statements [Good 82].

3 Protocol Synthesis

The job of composing a specification for an entire protocol system is rather complex in nature. Also, formal verification of some important desirable properties of such specifications have been shown to be generally undecidable (see [Bran 83]). Consequently, some research has been directed towards synthesizing complete specifications (i.e., including specifications of all the local entities involved) of protocols from incomplete ones and in some efforts the produced specifications would be also guaranteed to be free from design errors. Key variations among synthesis methods include the input specification(s) they accept and whether they produce error-free output specification(s). However, they all take advantage of the duality inherent in the interactions among the protocol entities where a message sent by one local entity should be received at another local entity.

Let us briefly describe some of the protocol synthesis algorithms as they are applied to the send-and-wait protocol:

- Given the send events of the sender, receiver and medium entities, a synthesis

algorithm finds the receive events in all three entities [Zafi 80] using a set of production rules. This is done incrementally i.e., each time the designer enters a send event to the synthesis program. The produced specifications are guaranteed to be free from certain design errors (such as deadlock).

- Given the service specifications of a protocol system and the specifications of the sender and medium processes (for example) a synthesis algorithm finds the specification of the remaining receiver process such that together with the sender and medium they would produce the given service specification [Boch 83]. However, the produced specification of the receiver might include redundant transitions and may reach deadlock when interacting with the other two processes.
- Given a global system specification (see Fig. 4 to be described in section 5.) and a partition of the set of global events between the three local processes, a synthesis algorithm finds the specification of each of the local processes such that their interactions would produce the given global specification [Prin 82].

Gouda et al [Goud 84] have also proposed a synthesis algorithm that accepts a FSM specification of one process and produces specifications of this process (with some modifications) and another interacting process that are free from some design errors. However, the algorithm is limited to two interacting processes and thus does not apply to our model of the send-and-wait protocol.

4 Protocol Implementation Methods

A protocol implementation method is a construction tool (a compiler in effect) that transforms a protocol specification into an implementation. While first layer protocols and possibly second layer protocols in the ISO hierarchy are implemented in firmware, protocols of layers 3 through 7 are implemented in software. For an example of the former, the reader is referred to [Goud 76]. In this section we will limit our discussion to software implementations.

The typical approach for implementing a FSM specification, as described in [Boch 82], is to implement it as a looping program, with each cycle of the loop executing a transition. The loop would consist of a single large CASE (or a set of conditional statements) statement with one case for each kind of input interaction, and for each of these cases another CASE statement would test the major state of the module and compute the next state accordingly.

Ideally, we would like to automate the implementation process so that both the effort involved and the possibility of errors are minimized. This possibly depends not only on the protocol specification method used, but also on the programming language used for implementation, and the complexity of the protocol. State-transition-based specifications (such as state machines and petri-net-based-models) lend themselves more easily than event-based specification (such as sequence expressions) to translations into implementation. This is because specifications for the former describe the flow of execution of a protocol step by step, while specifications for the latter are concerned with the valid outcomes of the protocol operation and not with how the outcomes are produced.

Unfortunately, there are some issues involved in the implementation process that preclude completely automated implementations. Human intervention in protocol implementations is

required for two purposes. First, to add the implementation dependent parts (e.g. buffer management functions), and message coding which is necessarily omitted from the specifications during the earlier phases because it is highly dependent on the implementation language used and not of importance to the protocol developer in the service statement or the protocol design phases. Second, the implementor often has to make certain choices based on the specific protocol being implemented. For example, whether to implement the protocol modules as part of the operating system or as cooperating user processes, and how will the different modules interact : using shared memory, or using some kind of interrupt mechanism.

5 Verification Methods

Protocol verification consists of logical proofs of first the *correctness* of each of the service, protocol, and implementation specification independently, and second, of the *mapping* between the service and the protocol specifications and between the protocol and implementation specifications. Proof of correctness of a specification constitutes proving the validity of certain desirable properties (typically stated in predicate calculus) that would assure its correct operation under all conditions, and proof of mapping constitutes of proving that a specification at one level of abstraction correctly implements the specification at the higher level.

To prove that a specification is correct, one has to prove that it satisfies protocol *safety* and *liveness* properties. Safety properties state the design objectives that a specification must meet if the protocol ever achieves its goals and liveness properties state that the specification is guaranteed to achieve these goals. For example, an informal description of a safety property S and a liveness property L for the send-and-wait protocol specification could be

S : the order of messages sent is the same the order of the messages received.

L : having received a new m, then retransmission must continue until an acknowledgment is received at the sender.

Safety and liveness properties are highly dependent on the protocol under consideration. However, there are some general desirable properties that are common to any protocol. These include:

- *Deadlock-freedom* (a liveness property) i.e. the protocol does not enter a state from which there is no exit.
- *Completeness* of the protocol in handling all situations that may arise during execution.
- *Progress* (a liveness property) or absence of tempo-blocking where the protocol enters an infinite cycle accomplishing no useful work.
- *Stability* i.e. the property that the protocol will return to a normal mode of operation after an exceptional condition occurs.

The *perturbation* approach [West 78b] to verification takes FSM specifications of the local protocol entities and derives a reachability graph. In this graph, each node represents the states of all the local entities, and each arc represents an entity event. Starting from the initial state of the graph, interactions of the entities are examined by following all possible ways in which the initial states and all subsequent states can be perturbed. Each node that the system can reach is checked for deadlock and whether all entities are able to receive in their current state all events that can be sent to them. The whole graph can be then checked for progress and stability.

The reachability graph for the send-and-wait protocol is shown in Fig. 4, and the fact that it was possible to produce the complete perturbation indicates deadlock-freedom and completeness. Also, the protocol is stable because it returns to its initial state after a finite number of events, and there is no tempo-blocking (assuming a less than one loss probability in the medium).

To see how a deadlock behavior would be detected by this approach, consider removing the timeout transition from the Sender process. Then, the system would deadlock at state 5 in Fig. 4 if the medium loses a message. Also, if the timeout value was too small, then we would have a timeout transition from each of states 7 through 12 back to state 2 in Fig. 4, and there would be a possibility of tempo-blocking due to any of these timeout loops. This shows how protocols are time-dependent systems, and therefore verification of timing requirements should be integrated with functional verification.

A critical disadvantage of the perturbation technique is the state explosion problem related to FSM models. *Symbolic execution*, another approach to verification, avoids this problem by considering classes of states instead of single states [Bran 78]. Also, although the perturbation approach is useful in verifying general properties, other approaches are required to prove specific protocol properties such as properties S and L given above for the send-and wait protocol. These approaches include *assertion-based methods* to express and prove Hoare/Floyd [Hoar 69, Floy 67] style safety assertions and temporal logic which has been used successfully in proving both safety and liveness properties of protocols [Schw 82].

In assertion-based verification, global system are proved using already proven local assertions. An assertion is a logical statement attached to a control point in the specification describing the requirements of the system state at this point. Verifying an assertion means demonstrating that it will always be true whenever the control point it is attached to is reached, regardless of the execution path taken to reach that point. For local assertions, this is typically done by using *inference rules* which define the effect of each specification construct on the assertions preceding it, and other proven assertions. The reader is referred to [Sten 76] for an assertion-based verification of a generalized data transfer protocol (it allows a window size of more than one) with a safety assertion similar to S given for the send-and-wait protocol.

Formulating assertions and proving them require a great deal of user ingenuity. This difficulty could be partially alleviated by automating proof checking and employing induction on the structure of the specification in building proofs [Muss 80, Divi 81].

Consider a service, protocol, and implementation specifications all proven to be correct. Still, it has to be shown that they all represent the same protocol. This is addressed by proof of mapping which in the case of FSM specifications constitutes a *static* mapping between each state at the higher level and the state(s) implementing it at the lower level, and a *dynamic* mapping between each transition at the higher level and the sequence (possibly nondeterministic) of transitions at the lower level. This could be done for the send-and-wait protocol as follows: in Fig. 3 states 1 and 2 are implemented by states 1 and 8 in Fig. 4 respectively (static mapping), and events GET and DELIVER in Fig. 3 correspond to $\underline{m}_{C,S}$ and $\overline{m}_{R,D}$ in Fig. 4 (dynamic mapping) respectively.

6 Performance evaluation

Performance evaluation is concerned with the evaluation of performance measures such as *delay*, and *throughput* to indicate how well a protocol meets its functional requirements. Delay in a data transfer protocol is defined as the time from the start of a message transmission until the successful arrival of its acknowledgment. Throughput is the transmission rate of useful data between protocol entities (i.e., excluding any control information or retransmission required by the protocol).

In order to analyze the performance of a protocol, a representation of the protocol and a representation of the communication medium are required. What parameters are required to specify the medium? This depends primarily on the protocol layer under consideration since the medium at a certain layer includes all the protocols of the lower layers. For example, in the case of data link protocols (at layer 2), the following parameters of the medium are required [Reis 82]: data rate, propagation delay, transmission-error process, topology, channel operation (half or full duplex), arrival process, frame-length process, nodal processing power and buffer space. These parameters are typically stochastic in nature.

The representation of the protocol depends on the performance evaluation approach adopted. Two approaches are used: *analysis*, and *simulation*. Analysis proceeds to formulate and solve a performance model of the protocol. There are two modeling approaches: formulating a mathematical model of the protocol from first principles [Bux 80a], or extracting a performance model from a formal specification of the protocol [Moll 81, Yemi 83]. Complex protocols are usually difficult to model and analyze and therefore approximate models are often used [Reis 82]. Another solution to complexity is to consider the simulation approach. Protocol simulation is typically based upon either the protocol specification [Baue 81], or some implementation of the protocol [Bux 80b].

Let us see how the delay of the send-and-wait protocol can be computed using basic probability laws and the protocol FSM specification as a description of its operation. Assume that the time involved in each transition of the reachability graph in Fig. 3 is an exponentially distributed random variable, and a negligible delay at both the sender and receiver ends of the medium. Based on these assumptions and considering a single cycle operation of the protocol, a modified reachability graph is shown in Fig. 5. The problem can be stated as follows: given a medium bandwidth of 9600 bits/sec (for terrestrial links), mean message and acknowledgment lengths l of 1024 bits (therefore the mean transmission

time t_s is 0.017sec/message), bit error rate p_b of 10^{-5}, mean propagation delay t_d of 0.013 sec/message, and mean timeout t_T of 1 sec/message, evaluate the mean value of delay d between state 2 to 8 in Fig. 5.

Recall from section 1. our assumption that timeout only occurs after the medium has lost a message. this indicates that the probability of timeout is the same as the probability of an erroneous (taken as lost) message. Therefore, the probability of the timeout loop denoted by p is

$$p = 1 - (1 - p_b)^l \qquad 6.1$$

which is approximately $1 - e^{-lp_b}$ if $lp_b << 1$

The mean delay is given by

$$E[d] = p/(1-p)\ (t_T + t_s) + 3t_s + 2t_d \qquad 6.2$$

$$= 0.357 \text{ sec/message}$$

and the second moment of d is

$$E[d^2] = p/(1-p)\ (2t_T^2 + 2t_s^2) + 2p^2/(1-p)^2\ (t_T + t_s)^2 + 6t_s^2 + 4t_d^2 \qquad 6.3$$

$$= 0.09$$

Derivations of equations 6.2 and 6.3 are given in appendix I. Assume that message arrive at state 2 in Fig. 5 with rate λ, then the protocol's *mean transfer time* T which is the sum of delay and a waiting time (a message has to wait in a queue if it arrives and the protocol system is still waiting for an acknowledgment for a previously sent message) is given by the Pollaczek-Khinchine formula [Klei 75]:

$$T = E[d] + (\lambda\ E[d^2])/(2[1-\lambda\ E[d]]) \qquad 6.4$$

In Fig. 6, we plot T versus λ for various message lengths. As expected, T increases as λ increases and the system becomes saturated when λ approaches $1/E[d]$. Also, as l increases T increases due to the increases in transmission times and p.

7 Testing

Testing is a process of examining whether a protocol satisfies its functional requirements, and measuring its performance. Consider a protocol implementation under test (IUT), the most common [Ansa 82b, Rayn 82a] testing approach consists of applying a set of test sequences to the IUT placed in its normal network environment. The official center that is conducting the testing would then provide some authorized certificate (with some level of confidence) when the testing process is successful indicating that the IUT could internetwork with other standard protocol implementations. Next we will examine two key issues involved in testing: the logical architecture used, and the test sequences that drive the IUT.

A Logical Architecture For Testing

Within the framework of the ISO model, a common logical testing architecture is given in Fig. 7.

In this architecture the peer protocol implementation (PPI) of the IUT is a combination of a reference implementation and a protocol-data-units generator (see Fig. 7). The PPI at layer N together with reference implementations for layers 4,5,...N-1 are located at the test center, while the IUT is at the implementor's site. Both ends are connected to an X.25 network which provides the first three network layers[1] . This testing configuration is referred to as *remote* testing since the test center controls the tests remotely.

The protocol-data-units generator is responsible for generating correct N level service requests, requests for the generation of N-th level protocol errors, indications of undetected N-th level protocol errors, and acts as an encoder and decoder of both valid and invalid (N-1) service. The PPI and the protocol-data-units generator are driven by a test driver (TD) at the testing center. The test responder (TR) is the software module which acts as the user of the N service, and whose operation is totally predictable so that the results of the tests depend only on the behavior of the IUT. The TD and TR communicate through a non-standard protocol.

To be able to assess the IUT, it is necessary to test its response to erroneous and correct requests across both the N and N-1 interfaces. However, if the N-1 service of the protocol being tested is not end-to-end (as in the case of the packet-level of the X.25), then it is not possible to control it remotely. Therefore, a portable box is introduced between the communication medium and the implementor's system (see Fig. 8) to detect any errors introduced by the sub-network and introduce errors in it upon request from the testing center.

Ideally, the testing process should be independent of the protocol being tested as much as possible so that only minimum variations need to be made when a new protocol is tested. This can be achieved by minimizing the protocol dependent parts of the architecture, and automating the process of test sequences selection. The only part of the testing architecture that needs be protocol dependent is the protocol-data-units generator, especially the part for testing normal and faulty N service. This dependency could be minimized by automating that part of the generator such that it is derived from some specification of the protocol.

Remote testing could be used only to test complete protocol implementations. Alternatively, in *direct* testing, a protocol representation could be a service, protocol, or implementation specification. In this approach, the specification is tested in a simulated environment where correct and faulty requests and responses across the N and N-1 interfaces are simulated and the results compared with those of a standard reference specification.

Test Sequence Selection

A test sequence is an input request to the IUT generated by the TD or TR. Test sequences

[1]Only end-to-end protocols above X.25 are tested in such architectures

could be specified either as state tables, or using a test specification language that might be then translated into state tables [Rayn 82b]. Testing is said to be *complete* if all the possible requests that could be applied to the IUT are covered by the test sequences. Unfortunately, theoretical results [Piat 80] show that with no knowledge of the protocol internal state the size (measured as the number of distinct sequential inputs applied to the IUT) of a complete test sequence has an upper bound of $O(n^n)$ where n is the size of the state set of the protocol reference model. Otherwise, with an access to the protocol internal state this figure comes down to $O(n^2)$. These bounds could be very large for complex protocols such as those involving sequence numbers.

However, there are other methods for near complete tests sequence selection. As an example, we will use the *transition tours* [Sari 82] method to calculate a test sequence for the send-and-wait protocol. This method is used to derive test sequences from a protocol specified formally as a state machine but using only its FSM part. A transition tour sequence is an input sequence starting at the initial state and covering all the transitions at least once. The length of the sequence for our protocol example (see Fig. 5) is 8 and the sequence is given by

$$\underline{m}_{C,S}\ \overline{m}_{S,M}\ T\ \overline{m}_{S,M}\ \underline{m}_{M,R}\ \overline{m}_{R,D}\ \overline{a}_{R,M}\ \underline{a}_{M,S}$$

In general, the upper bound on the sequence length is $q + (q-1)(n-1)$, where q is the number of possible transitions. This is the worst case where a traversal of all (n-1) states is required to include each transition in the test sequence. This method detects all operation errors (errors in the output function of the state machine), but it does not detect all transfer errors (errors of the next state function).

8 Conclusions

We have demonstrated in the previous sections how the various protocol development tools are dependent on the specification method used. Hence, in addition to the set of requirements given in section 2, a specification method must exhibit other features to facilitate the application of other development tools. These include:

1. Providing constructs for expressing desired properties of the protocol and thus facilitating their automated formal verification.
2. Executability of the specification to facilitate the simulation of the specification, and enhance the automation of the implementation process.
3. Providing constructs for expressing performance aspects to facilitate automated performance evaluation.
4. Supporting the clear definition of the interfaces between the protocol layer concerned and the layers above and below to allow for separate testing of the implementation of each protocol layer.

Future research is required for developing individual tools and automating them, building integrated protocol environments with advanced human interfaces, and more experience in applying these tools and environments to more protocol standards as they are developed --both low level and high level protocols.

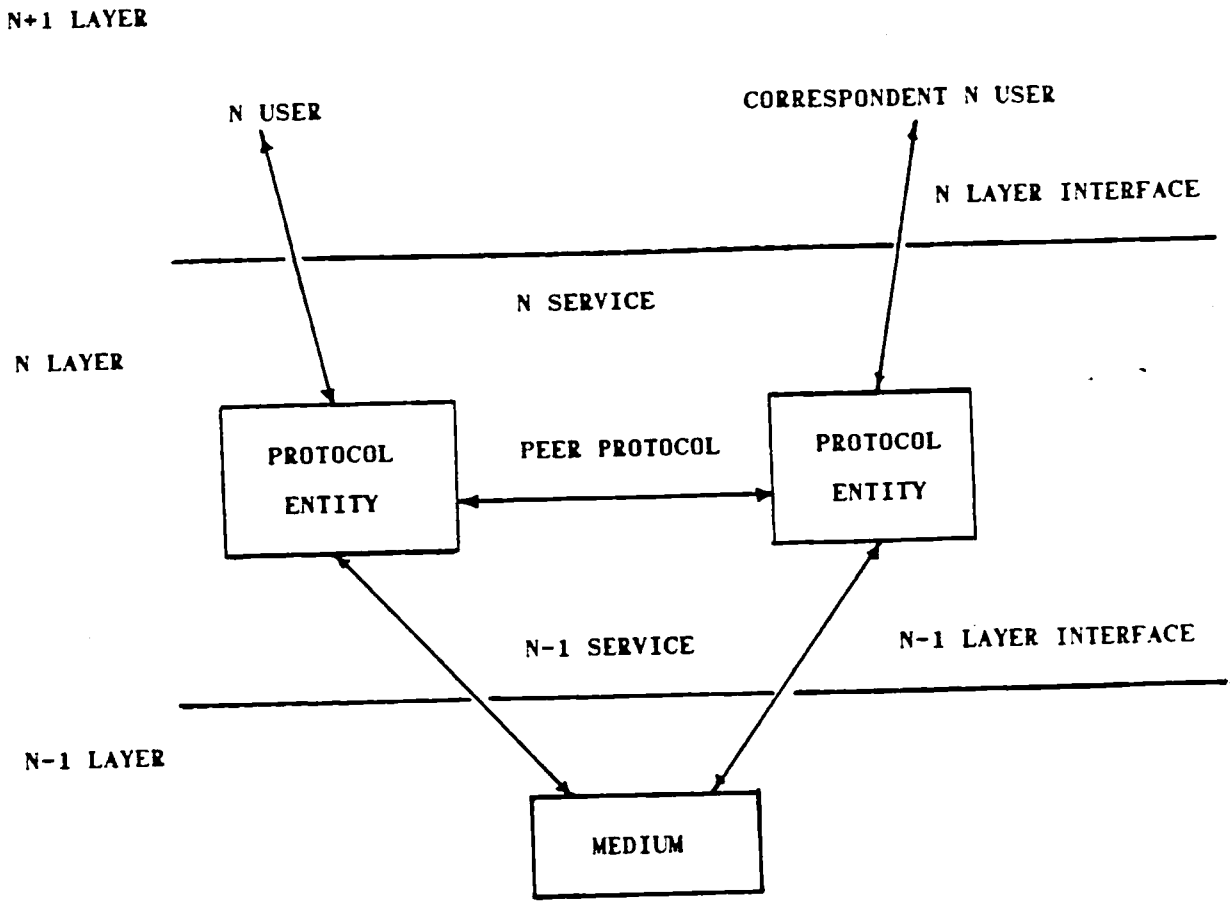
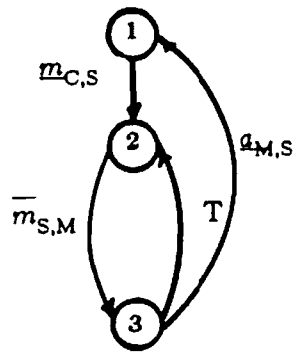


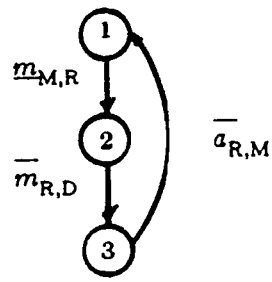
Figure 1: A local view of a protocol layer

Figure 2: A protocol specification for the send-and-wait protocol
(a) Sender (b) Receiver (c) Medium

(a)



(b)



(c)

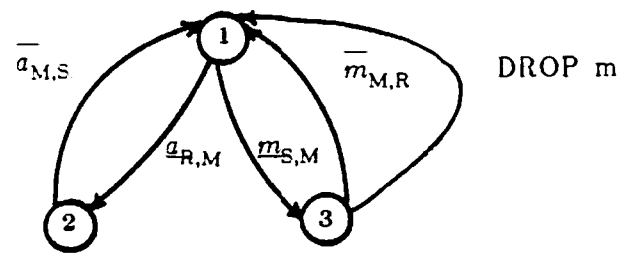
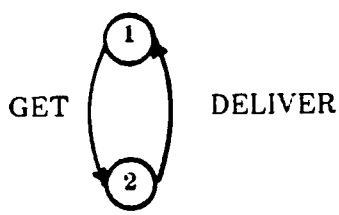


Figure 3: A service specification for the send-and-wait protocol



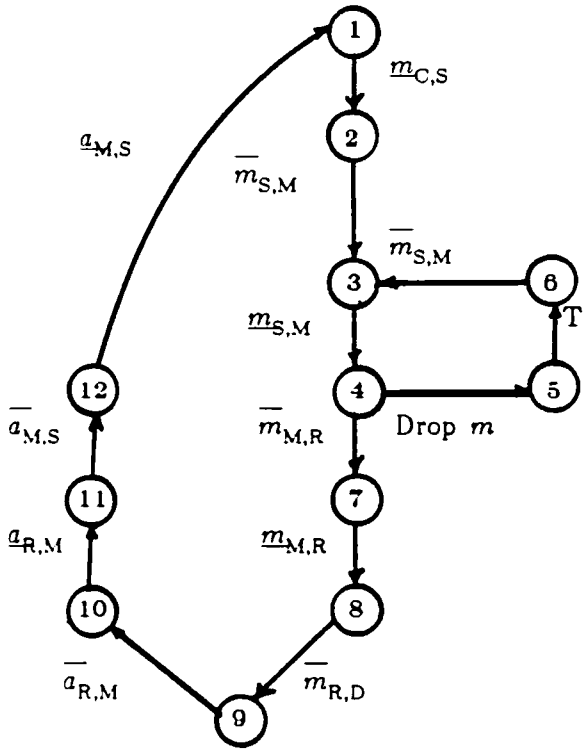


Figure 4: A reachability graph for the send-and-wait protocol

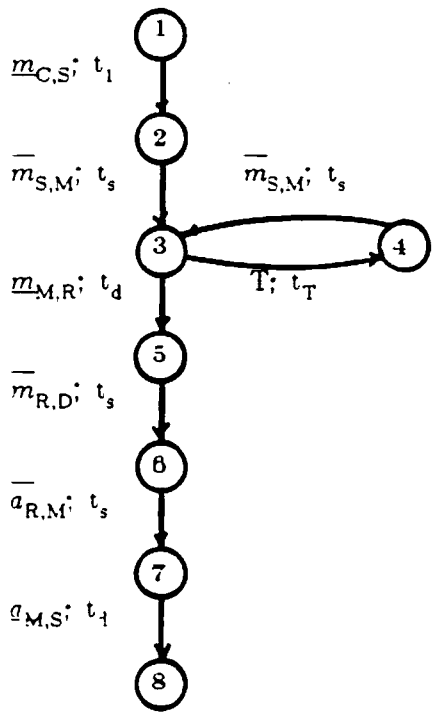


Figure 5: A modified reachability graph for the send-and-wait protocol

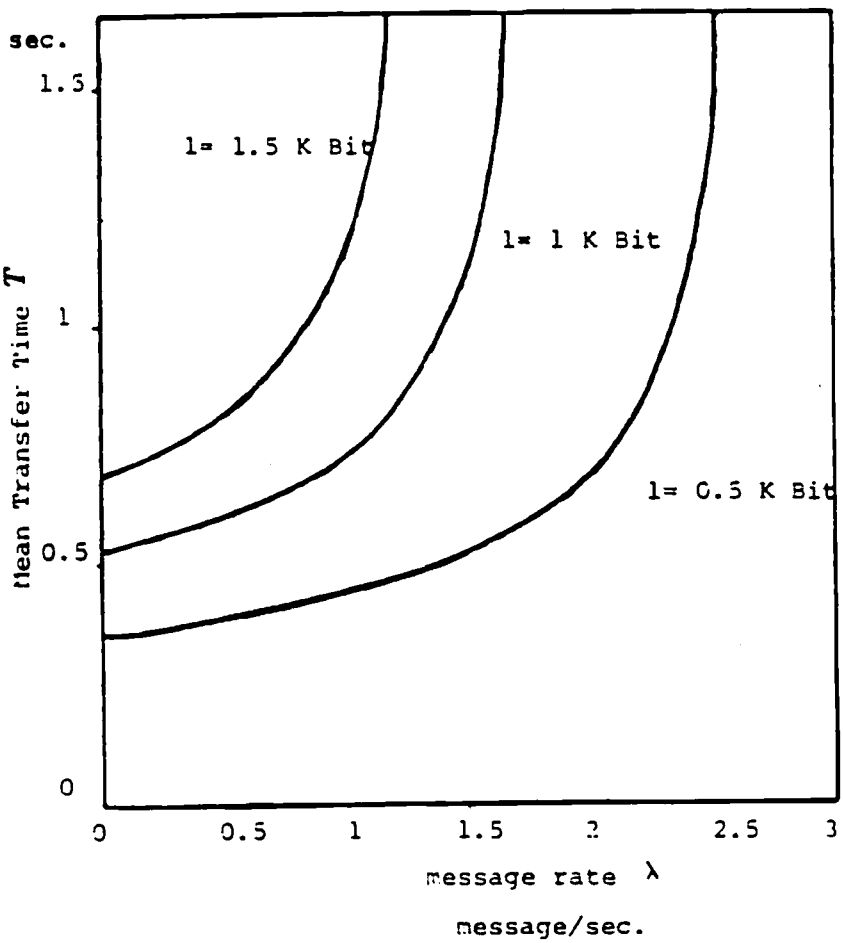


Figure 6: Transfer time vs. arrival rate of the send-and-wait protocol

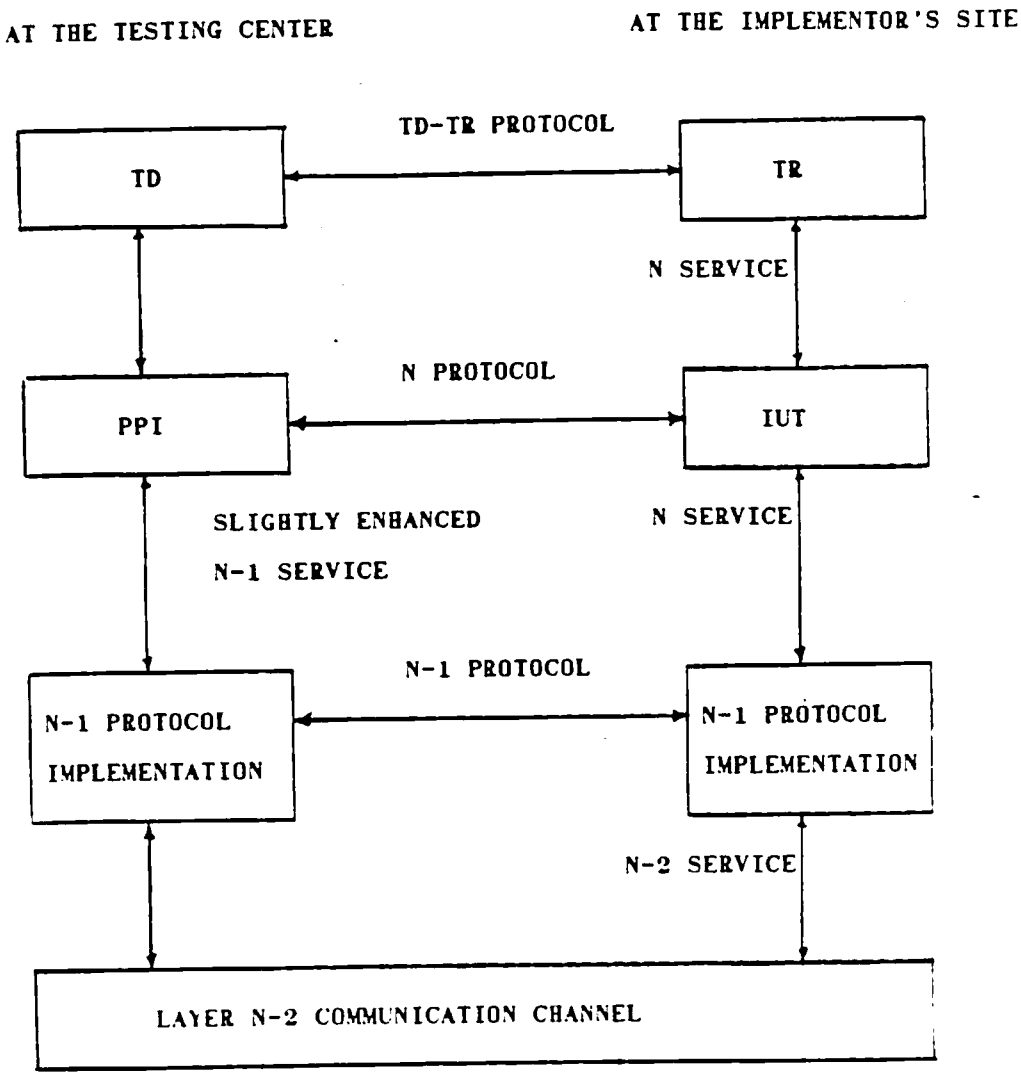


Figure 7: Logical architecture for testing

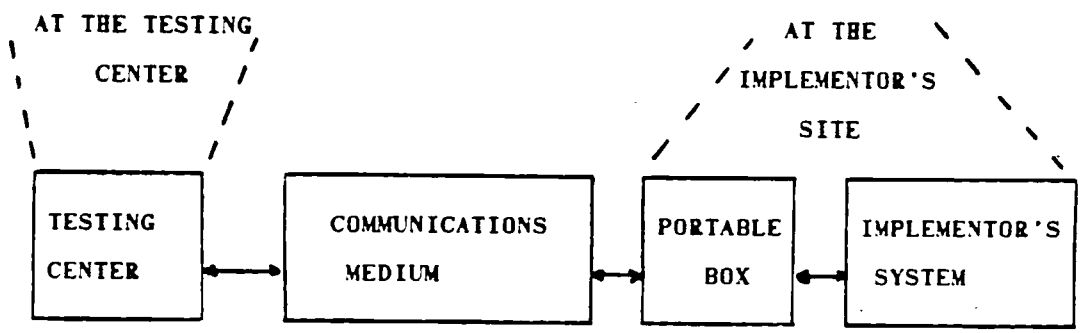


Figure 8: Physical architecture including the portable unit

I. Proof of equations 6.2 and 6.3

Since we are considering mean values, then we can add up sequentially composed event times, and E[d] can be calculated by adding up the time for the timeout loop with the rest of the event times. Let the total time spent in the loop be L and the time of a single loop be t. Then,

For equation 6.2:

$$E[L|\ i=n] = n\ e[t]$$

and by using the theorem of total expectation, we get

$$E[L] = E[n]\ E[t]$$

since i has a modified geometric distribution, then

$$E[L] = p/1\text{-}p\ E[t]$$

but $E[t] = t_T + t_s$, therefore

$$E[L] = p/1\text{-}p\ (t_T + t_s)$$

and the mean delay is

$$\begin{aligned} E[d] &= E[L] + 3t_s + 2t_d \\ &= p/1\text{-}p\ (t_T + t_s) + 3t_s + 2t_d \end{aligned}$$

For equation 6.3: The second moment for D could be calculated by adding the second moments of the sequentially composed event times (which is $2t_i^2$ for event i because they are exponentially distributed). The second moment of L is

$$\begin{aligned} E[L^2|\ i=n] &= Var\ [L|\ i=n] + (E[L|\ i=n])^2 \\ &= n\ Var[t] + n^2\ (E[t])^2 \end{aligned}$$

Using the theorem of total moment, we get

$$\begin{aligned} E[L^2] &= E[n]\ Var[t] + E[n^2]\ (E[t])^2 \\ &= p/1\text{-}p\ [E[t^2]\text{-}(E[t])^2] + [(p+p^2)/(1\text{-}p)^2]\ (E[t])^2 \\ &= p/1\text{-}p\ E[t^2] + 2p^2/(1\text{-}p)^2\ (E[t])^2 \end{aligned}$$

and the second moment of the delay is then given by

$$\begin{aligned} E[d^2] = \ & p/(1\text{-}p)\ (2t_T^2 + 2t_s^2) \\ & + 2p^2/(1\text{-}p)^2\ (t_T + t_s)^2 + 6t_d^2 + 4t_s^2 \end{aligned}$$

[Ansa 82a] J.Ansart, O.Rafiq, and V.Chari.
PDIL - Protocol Description and Implementation Language.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Ansa 82b] J.Ansart.
GENEPI/A -A Protocol Independent System for Testing Protocol Implementation.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Ayac 81] J.Ayache, P.Azema, J.Courtiat, M.Diaz and G.Juanole.
On the Applicability of Petri Net-Based Models in Protocol Design and Verification.
Proceedings of the First International INWG/NPL Workshop : Protocol Testing - Towards Proof? Vol.1.:349-370, 1981.

[Baue 81] W.Bauerfeld.
Description, Verification and Performance Prediction of Computer Network Protocols.
Proceedings of the First International INWG/NPL Workshop : Protocol Testing - Towards Proof? Vol.1.:253-270, 1981.

[Boch 77] G.Bochmann and J.Gecsei.
A Unified Method for the Specification and Verification of Protocols.
In *Proceedings of IFIP Congress*, pages 229-234. August 8-12, 1977.

[Boch 82] G.Bochmann et al.
Some Experience with the Use of Formal Specifications.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Boch 83] G.Bochmann and P.Merlin.
On the Construction of Communication Protocols.
ACM Transactions on Programming Languages and Systems Vol.5,No.1:1-25, January, 1983.

[Bran 78] D.Brand and W.Joyner,Jr.
Verification of Protocols Using Symbolic Execution.
Computer Networks Vol.2:351-360, October, 1978.

[Bran 83] D.Brand and P.Zafiropulo.
On Communicating Finite-State Machines.
Journal of the ACM 30:433-445, April, 1983.

[Bux 80a] W.Bux,K.Kummerle, and H.Truong.
Balanced HDLC Procedures: A Performance Analysis.
IEEE Transactions on Communications Vol.COM-28,No.(11):1889-1898, November, 1980.

[Bux 80b] W.Bux and K.Kummerle.
HDLC Performance: Comparison of Normal Response Mode and Asynchronous Balanced Mode of Operation.
In *IEEE Proceedings of the NTC*, pages 15.3.1-15.3.6. 1980.

[Dant 78] A.Danthine and J.Bremer.
Modelling and Verification of End-to-End Transport Protcols.
Computer Networks Vol.1:381-395, October, 1978.

[Divi 81] B.Divito.
A Mechanical Verification of the Alternating Bit Protocol.
Univ. of Texas at Austin ICSA-CMP-21, June, 1981.

[Divi 82] B.Divito.
Verification of Communications Protocols and Abstract Process Models.
PhD thesis, Univ. of Texas at Austin, August, 1982.

[Floy 67] R.Floyd.
Assigning Meanings to Programs.
Mathematical Aspects of Computer Science 19:19-32, 1967.

[Good 82] D.Good.
The Proof of a Distributed System in Gypsy.
Technical Report 30, The Univ. of Texas at Austin, September, 1982.

[Goud 76] M.Gouda and E.Manning.
Protocol Machine: A Concise Formal Model and its Automatic Implementation.
In *Proceedings of the Third ICCC*, pages 346-350. 1976.

[Goud 84] M.Gouda and Y. Yu.
Synthesis of Communicating Finite-State MAchines with guaranteed Progress.
IEEE Transactions on Comunications COM-32(7):779-788, july, 1984.

[Hoar 69] C.Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM Vol.12,No.(10):576-583, October, 1969.

[Klei 75] L.Kleinrock.
Queueing Systems.
Wiley Interscience, 1975.

[Miln 80] R. Milner.
A Calculus of Communicating Systems.
Springer Verlag, 1980.

[Moll 81] M.Molloy.
On the Integration of Delay and Throughput Measures in Distributed Processing Models.
PhD thesis, Univ. of California Los Angeles, 1981.

[Muss 80] D.Musser.
Abstract data Type Specifications in the AFFIRM System.
ieeetose SE-6(1), January, 1980.

[Piat 80] T.Piatkowski.
Remarks on ADCCP Validation and Testing Techniques.
NBS Trends and Applications Symposium , May 29, 1980.

[Prin 82] R.Prinoth.
An Algorithm to Construct Distributed Systems From State-Machines.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification 261-282, May, 1982.

[Rayn 82a] D.Rayner.
A System for Testing Protocol Implementations.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Rayn 82b] D.Rayner ed.
A System for Testing Protocol Implementations.
NPL Report DITC 9/82 , August, 1982.

[Reis 82] M.Reiser.
Perfromance Evaluation of Data Communication Systems.
Proceedings of the IEEE Vol.70,No.2.:171-196, February, 1982.

[Sari 82] B.Sarikaya and G.Bochmann.
Some Experience with Test Sequence Generation for Protocols.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Schi 81] S.Schindler.
The OSA Project: Basic Concepts of Formal Specification Techniques and of RSPL.
Proceedings of the First International INWG/NPL Workshop : Protocol Testing - Towards Proof? Vol.1.:143-176, 1981.

[Schw 81] D.Schwabe.
Fromal Techniques for the Specification and Verification of Protocols.
PhD thesis, Univ. of California Los Angeles, April, 1981.

[Schw 82] R.Schwartz and P.Melliar-Smith.
From State Machines to Temporal Logic: Specification Methods for Protocol Standards.
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Sten 76] N.Stenning.
A Data Transfer Protocol.
Computer Networks (1):99-110, 1976.

[Symo 80] F.Symons.
Introduction to Numerical Petri Nets, a General Graphical Model of Concurrent Processing Systems.
Australian Telecommunication Research 14(1):28-32, 1980.

[Teng 78] A.Teng and M.Liu.
A Formal Approach to the Design and Implementation of Network Communication Protocol.
In *Proceedings, COMPSAC.* 1978.

[West 78a] C.West and P.Zafiropluo.
Automated Validation of a Communications Protocol: the CCITT X.21 Recommendation.
IBM J. Res. Dev. Vol.22,No.(1):60-71, January, 1978.

[West 78b] C.West.
An Automated Technique of Communications Protocol Validation.
IEEE Transactions on Communications Vol.COM-26,No.(8):1271- 1275, August, 1978.

[X.21 76]
Recommendation X.21 (Revised).
CCITT (International Telegraph and Telephone Consultative Committee), Geneva, Switzerland , March, 1976.

[Yemi 82] Y.Yemini and J.Kurose.
Towards the Unification of the Functional and Performance Analysis of Protocols, or is the Alternating-Bit Protocol Really Correct?
Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification , 1982.

[Yemi 83] Y. Yemini and N. Nounou.
CUPID: A Protocol Develeopment Environment.
Proceedings of the Third IFIP International Workshop on Protocol Specification, Testing and Verification , May, 1983.

[Zafi 80] P.Zafiropulo, C.West, H.Rudin, D.Cowan, and D.Brand.
Towards Analyzing and Synthesizing Protocols.
IEEE Transactions on Communications Vol.COM- 28,No.(4):651-661, April, 1980.

[Zimm 81] H.Zimmermann.
Progression of the OSI Reference Model and it's Applications.
In *Proceedings of The NTC*, pages F8.1.-F8.1.5. 1981.