PROTOCOL CONSTRUCTIONS
FOR
COMMUNICATION NETWORKS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Yanpyng Albert Teng, B.Sc., M.Sc.

* * * * *

The Ohio State University

1980

Approved by

Ming T. Liu

Adviser
Department of Computer
and Information Science

This dissertation is dedicated to my wife.

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my advisor, Dr. Ming T. Liu. He pointed out the importance of investigating a formal model for protocol design about three years ago and thus gave me the initial motivation of this research. His criticism and suggestions have also contributed in a significant way to the development of this dissertation. In particular I thank his encouragement and assistance for the publication of several parts of this research.

I wish to thank the other members of my reading committee. Dr. Sandra A. Mamrak has offered many insightful suggestions that have proven to be very valuable with regard to the organization of this dissertation. Dr. Clinton R. Foulk has patiently read the dissertation drafts, corrected many grammatical mistakes, and given valuable comments. It was very easy to approach both of them with problems.

I thank my colleagues at the IRCC/CIS computing laboratory, Herb Johnson, James Davenport and Lora Williams, who have proofread part of the dissertation draft. I would also like to thank Instruction and Research Computer Center for the financial support as a graduate research associate during my entire stay at OSU. In addition, the support from the National Science Foundation (Grant MCS-7723496) is acknowledged.

Finally, I would like to thank my loving wife, Suh-Fen, for her encouragement, patience, understanding and assistance during all phases of this work.

VITA

Feburary 23, 1951. . .	Born - Taipei, Taiwan
1973	B.Sc., Electrical Engineering, Cheng-Kung University, Tainan, Taiwan, R.O.C.
1976	M.Sc., Computer and Information Science, The Ohio State University
1975-1979	Graduate Research Associate, Instruction and Research Computer Center, The Onio State University

PUBLICATIONS

"A Formal Approacn to the Design and Implementation of Network Communication Protocols," Proc. COMPSAC 78, Nov., 1978, pp. 722-727. (M. T. Liu, co-author).

"A Formal Model for Automatic Implementations and Logical Validation of Network Communication Protocols," Proc. Computer Networking Symposium, Dec. 1978, pp. 114-123. (M. T. Liu, co-author).

"The Transmission Grammar Model for Protocol Construction," Trends and Applications 1980, Computer Network Protocols, May 1980. (M. T. Liu, co-author).

FIELDS OF STUDY

Major Field: Computer and Information Science

Computer Organization and Architecture.
Dr. Ming T. Liu

Computer and Systems Programming.
Dr. Sandra A. Mamrak

Programming Languages.
Dr. Clinton R. Foulk

TABLE OF CONTENTS

LIST OF TABLES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Communication Protocols

Computer components for data processing are becoming much more powerful while simultaneously becoming less expensive. To take advantage of this advance in hardware technology, today's data processing is often distributed across a network of mini and micro computers. A well-defined and agreed-upon set of rules must be established to regulate the interactions between the attached entities in a network and to ensure that these interactions proceed in an orderly fashion. This set of rules is called the communication protocol and is often a major factor in providing correct and reliable operation for the network. The design and implementation of protocols is really a key to the effective exploitation of computer networks on a large scale. In this dissertation, we present a formal model for constructing correct protocols for computer networks.

1.1.1 Functions of Protocols

In a computer network, a message is transmitted over a "raw" communication medium that may be noisy and unreliable. During its passage, the message may be damaged, lost, duplicated, or arrive at the destination out of order. Carefully designed protocols can overcome those problems and create a "virtual" communication channel that is more reliable than the raw communication medium. In order to provide reliable communication between communicating entities over an unreliable medium, protocols in general have the following functions:

(1) Message transport control -- The detection of and recovery from errors caused by damage, loss, duplication, and an out-of-order sequence.
(2) Synchronization -- The control of the interactions between communication entities to avoid deadlock, incompatibility, infinite idle looping, improper termination, and unbounded demand for resources.
(3) Flow control -- The management of message flow from the source to the destination.

(4) Failure recovery -- The recovery from network component failures.

Besides these functions, some protocols may have additional features such as addressing, connection management, sequence control, message fragmentation and priority. Construction of a protocol for a computer network is therefore a very complicated problem and a layered approach is usually adopted for protocol design.

1.1.2 Layered Approach

The advantage of having multiple layers of protocols is that layering provides a separation of functions which is always useful in designing any complex system. It allows implementation and verification of protocols to be carried out separately for each layer. Also, failure recovery and error detection can operate independently to handle errors at each layer.

For any given layer of the protocol, there are three desirable features [Davies73]: (1) it should be transparent to layers above it, so that nothing sent or modified by a higher layer can be affected or modified by its lower layers; (2) it should be self-limiting so that it is impossible to require the provision of potentially infinite storage, or to spend infinite time in a program loop or waiting state; (3) it should be possible to change one layer of the protocol without requiring extensive changes at other layers. These features provide support for evolutionary changes of the protocol since each layer of the protocol acts as an independent module.

A typical distributed computer network has several layers of protocol [Ray76]: hardware level protocols by which hardware devices exchange signals; communication sub-network protocols by which communication processors exchange messages; and network protocols by which host computers exchange messages.

For example, a layered approach has been adopted in ARPANET for the specification of network protocols. The inner layers include the IMP to IMP, HOST to IMP, and HOST to HOST protocols, and several high-level function-oriented protocols have also been defined. Fig. 1 shows what the network protocols in ARPANET look like: (1) the IMP to IMP protocol which describes the communication in the communication subnetwork; (2) the HOST to HOST protocol (NCP, TCP) [McKenzie72, TCP79] which describes the communication between hosts connected to the communication subsystem; and (3) function-oriented protocols (such as ICP, TELNET, FTP, RJE, DCP, GP etc.) [Feinler78] which describe the communication between processes of different host computers and represent high-level virtual connections of the network.

```
        ICP,TELNET,FTP,RJE,DCP,GP
  ------------------------------------
  \                                  /
   \          HOST/HOST             / PROCESSES'
    \------------------------------/ PRIMITIVES
     \                            /
      \        IMP/IMP           / HOST/IMP
       \------------------------/
        \                      /
         \   COMMUNICATION    /
          \     SYSTEM       /
           \                /
            \--------------/
```

Fig. 1 ARPANET protocol hierarchy

For the case of interconnecting different networks, we need to add internetwork protocol [TCP79] to the hierarchy.

1.1.3 Protocol Verification

Sunshine [79] and ISO [78] have used the term "service specification" of the protocol to clarify the concepts of specification and verification in the context of a layered protocol model.

Service specification is similar to the concept of a set of desirable properties of protocols. A "structured verification" concept [Teng78a] is useful in dealing with network protocol verification. Verification of the properties of one layer should be done by utilizing the proved properties or provided service of its lower layers.

Those desirable properties have two aspects: (1) the absence of protocol syntax errors, and (2) the absence of protocol semantics errors. Protocol syntax errors include deadlock, incompatibility, infinite idle looping and improper termination. Protocol semantics errors include those violations of the assertions defined in the service specification. The checking of protocol syntax errors is more amenable to automatic techniques since it only depends on the protocol's syntax specification (we assume that those problems which may cause syntax errors can be detected by examination of the syntax specification). The checking of protocol semantics errors is more difficult since it depends on the protocol's semantics specification to form assertions.

Protocol verification is a demonstration of the freedom of a protocol specification from syntax and semantics errors. Sunshine [79] and West [78a] have called the checking of protocol syntax errors "protocol validation". A major research objective of this dissertation is to provide a systematic scheme for specifying, verifying and implementing protocols which are syntax-error free.

1.2 Objectives of This Research

Although informal descriptions and testing provide helpful tools in protocol design and validation, they are inadequate by themselves to handle the increasing variety and complexity of communication protocols. Our present ability to handle protocols is similar to the ability we had with programming languages in the 1950's, when no convenient formalism, like the Backus-Naur form [Naur63] used now, was available to describe syntactic constructs of languages [Fraser76]. Without a formal model as a convenient means of representation, there are few clues that can lead us to a systematic scheme for automatic software/hardware implementation of communication protocols and little can be done for the automatic verification of these protocols.

In this dissertation, we will investigate formal techniques and methodologies for constructing correct protocols in a computer network environment. We address many protocol construction problems. Five major research objectives addressed in this dissertation are given below:

(1) Develop software engineering techniques for specifying correct protocol programs which are modifiable and maintainable, and develop automatic (or semiautomatic) tools for verifying and implementing protocols in a straightforward manner.

(2) Provide a systematic scheme to investigate the fundamental correctness problems of computer network protocols, and develop a protocol problem checklist (for a given protocol layer and network topology) as a guideline for correctness checking during the protocol construction process. The checklist serves as the basis for the design of verifiable protocols.

(3) Develop a formal specification model which is theoretically applicable to a broad range of protocols and is capable of providing a clear and concise description of protocol syntax structures. The specification should permit a complex protocol to be described with a well-structured notation that is suitable for human readability and should be useful for both protocol verification and implementation. The specification should also provide clear and concise documentation for both network users and protocol designers.

(4) Provide protocol designers an effective solution (with acceptable complexity) to existing protocol correctness problems by using automatic protocol <u>verification</u> techniques. The verification techniques should be flexible and powerful enough to analyze protocols more complicated than a finite state language, and should be convenient for checking design and syntax errors.

(5) Provide protocol designers a direct and straight-forward <u>implementation</u> scheme from a validated protocol specification, thus eliminating the burden of program coding with its attendant errors.

The above five research objectives represent the basis for a fundamental study into the design and analysis of computer network protocols in this dissertation. To meet these objectives, a comprehensive technique for correct protocol construction has been developed by applying software engineering and formal language techniques. A formal model, called the Transmission Grammar (TG) has been developed for defining, verifying, implementing and documenting network protocols, in addition to a conceptual analysis of protocol problems and their solutions.

1.3 Related Work

Considerable progress has been made in using formal techniques to meet our objectives of protocol specification and verification during the past three years. However, many new techniques must be applied and some totally new concepts developed in order to accomplish all of the above research objectives.

A number of important formal protocol models have been proposed, including finite state automata, Petri nets, programming languages and formal grammars. Sunshine[78a] and Merlin [79] have provided surveys of various models. We will also review and compare various models in Chapter 3. The following discussion will summarize some of the important features of these models.

Finite state automata are convenient for representing complex control structures of protocols. The schemes using finite state automata have been very successful for automatic analysis of syntax errors by exhaustively generating and checking the global states. However, the theoretical application of finite state automata is limited to verifying protocols with a bounded number of states (i.e., a finite state language). These schemes generally have the difficulty of "state explosion" and are not directly applicable to automatic implementation. They are also hard to be applied to produce "well-structured" specification and documentation.

Petri nets are also convenient for representing complex control structures of protocols, and allow for automatic reachability analysis. Petri nets are theoretically applicable to a broader range of protocols than finite state automata [Merlin79] but cannot analyze even a simple reader-writer problem [Chen75]. The protocol specification of Petri nets is usually a description of the global interactions of communicating entities for the purpose of verification and must be decomposed for implementation.

Programming languages are convenient for representing variables and counters, but not complex control structures. The schemes using programming languages provide a relatively concise means of describing protocols, and proof of protocol program correctness provides a rigorous, formalized verification process to handle a wide range of assertions of the desired properties. Program proving is an extremely important <u>long-term</u> research project to handle semantics errors but is considered to be at least 10 years away from being useful to programs of any significance [Glass79]. Using program proving schemes, a protocol designer often requires several times the amount of work to prove a protocol program than was required to write the program, since programming languages often deal with many implementation details besides design issues.

Harangozo[78] has used a regular grammar model to specify the HDLC link protocol by incorporating an indexing technique to accommodate sequence numbering. However, his model seems to be intended only for the purpose of protocol specification and cannot define complex protocol features which are context-free or context sensitive.

Most of the previous research efforts that we have mentioned above have concentrated on protocol specification and/or verification. A general and comprehensive methodology is just being applied to the design of complicated protocols [Teng78b, Sunshine79]. In order to achieve the objectives of this research, fundamental protocol correctness problems should be investigated and considered during protocol design, specification and implementation. There are also several neglected problems in the protocol research field, such as structured and modular design, entity characteristic analysis, automatic (or semiautomatic) implementation and documentation maintenance.

1.4 Our Approach and Contributions

This research is also concerned with correct protocol construction techniques. We have developed a comprehensive protocol construction methodology by applying software engineering and formal language techniques to achieve the design objectives discussed in Section 1.2. Our construction techniques are based on a nongraphic formal model using a context-free grammar, called the Transmission Grammar (TG), for defining, verifying and implementing network protocols.

The transmission grammar model is similar to the Backus-Naur form that has been used to define the syntax of programming languages, but it is used to define the control and message structures of communication protocols. The step-wise TG specification allows the protocol designer not only to specify protocol program modules in a well-structured manner, but to keep the specified structure (context-free grammar) so simple that automatic validation and implementation can be easily carried out. The TG protocol model is more concise and powerful than the finite state automata (FSA) model, and is also more convenient to represent and analyze complex control structures than the programming language model can. The TG model is capable of specifying and verifying protocols (languages) which are more complicated than the FSA model can [Teng78a]. A TG protocol specification can contain necessary redundancies for better human readability, as compared with a finite automaton model. Since there does not exist any graphic diagram interpreter for the direct processing of graphic models (such as FSA, Petri nets and UCLA graphs), the linear (nongraphic) property of the transmission grammars is also crucial to _direct_ analysis and implementation of protocols.

Based on the TG model and software engineering techniques, we develop protocol construction techniques for systematic and structured design of correct protocols. The following discussion summarizes some of the more important capabilities and features of our construction techniques.

(1) Validation Checklist -- We investigate the fundamental protocol correctness problems that may cause protocol syntax errors. We summarize and explain the nature of three important types of correctness problems: message transport problems, transmit interference problems and reliability problems. While protocol correctness problems are a major source of protocol syntax errors, only a _subset_ of these problems actually exist in a given protocol topology and layer. Even an expert may make validation mistakes if he does not have a clear understanding of the potential correctness problems in his protocol design.

A deficiency of existing validation techniques [Hajek78, West78b] is the lack of a systematic scheme for examining protocol correctness problems. We introduce a validation checklist, which contains the "minimum" and "sufficient" set of existing correctness problems for a given topology and layer, to be used during our validation and construction process. The checklist can prevent us from wasting our effort in examining non-existent problems and can lead us to a systematic study of protocol correctness problems. We will present an example of this validation checklist in Chapter 2 and use the checklist to simplify global validation process in Chapter 5.

(2) Specification and Design -- To the best of our knowledge, we are the first to present: (1) the context-free and context-sensitive characteristics of various protocols, (2) hierarchically structured specification and documentation for protocol design, and (3) the TG substitution and shuffle operations for constructing inherently correct protocols from verified protocol components and/or validation independent parts [Teng78a].

Using the TG model, a protocol program is hierarchically structured by step-wise refinement. The TG specification of protocols can preserve the history of the step-wise refinement for hierarchically structured documentation. We apply the TG model to specify the protocol first at an abstract or design level which is easy to understand and use during the design stage. The abstract TG specification is related to the concept of a Program Design Language (PDL) [Caine75] in software engineering and design. The abstract TG specification contains abstract communication actions as its terminal symbols. Essentially any set of English statements could be used to describe the abstract actions. This abstract specification provides an easy way to read a protocol program in the design stage, and also provides documentation which can be refined for future validation and implementation. We also develop theories and algorithms to show the importance of TG decomposition and integration in step-wise design for both validation and implementation. This "divide and conquer" scheme allows the construction of inherently correct protocols from "decomposed" and verified protocol modules to ease the task of specification and verification.

(3) Validation -- We present an automated approach to verify protocols which are more complicated than finite state languages. Our correctness checking includes TG structure errors and protocol syntax errors. We first check the program structure errors since the check of TG structure errors can reveal those simple errors before checking more complicated protocol syntax errors.

A local validation system is developed to automatically check TG structure errors from the entity's TG specification. The local validation system is based on the analysis of <u>transitive closure</u> of grammar relations and is very efficient in locating structure errors.

A global validation system based on the validation automaton (VA) model is also developed to check the <u>timing</u> and interactions between communicating entities and to reveal protocol syntax errors. Using this VA model, the designer can conveniently examine such protocol correctness problems as message transport, transmit interference, and reliability problems, by using several basic complexity reduction rules which will be explained in Chapter 5. The VA model can comprehensively represent the inter-dependency between communicating entities and the change of complicated channel status by a matrix of message and acknowledgment queues. We can easily derive a VA specification of protocols from a TG specification. The global validation system can directly read the VA specification and automatically generate the global transitions (in a tractable number of states) for checking protocol syntax errors.

(4) Implementation -- The TG model can be generalized to have two grammar specifications: (1) action grammar and (2) message grammar. The action grammar is used to specify the action sequences of the protocol. The message grammar allows us not only to represent the hierarchical structure of message formats, but to implement automatic syntax parsing and error handling of the message structure. From a validated TG, we can use refinement techniques to specify a detailed protocol description and to preserve the validated protocol properties. A protocol of the communication processor layer in the ARPANET is specified by the generalized TG model to describe both the interactions between communicating entities and the structure of individual messages. This specification can contain proper semantics which is suitable for automatic software/hardware implementations, and can be directly used to drive a recognizer for the protocol. We outline the structure of such a recognizer, called the General Protocol System (GPS), for automatic software implementations. Other syntax-directed implementation techniques are also studied.

The above features and capabilities make our construction techniques for correct protocols very powerful, flexible and efficient. Our contribution is both methodological in developing a framework for systematic protocol construction; and substantial in presenting solutions to the problems of protocol specification, design, verification and documentation. In addition, the overall research effort has provided not only some insights into protocol correctness problems but also practical strategies for handling these problems during all phases of protocol construction.

1.6 Organization of Dissertation

This dissertation is arranged so that each chapter addresses a distinct topic of the protocol research area. This first chapter serves as a foundation from which the rest of the research may be presented. It contains an overview of the functions of computer network protocol and outlines protocol construction techniques. The major design objectives and the important features of this research are also presented and discussed.

The organization of the remaining chapters of this dissertation can be viewed in the context of a software engineering life-cycle for protocol construction as shown in Fig. 2.

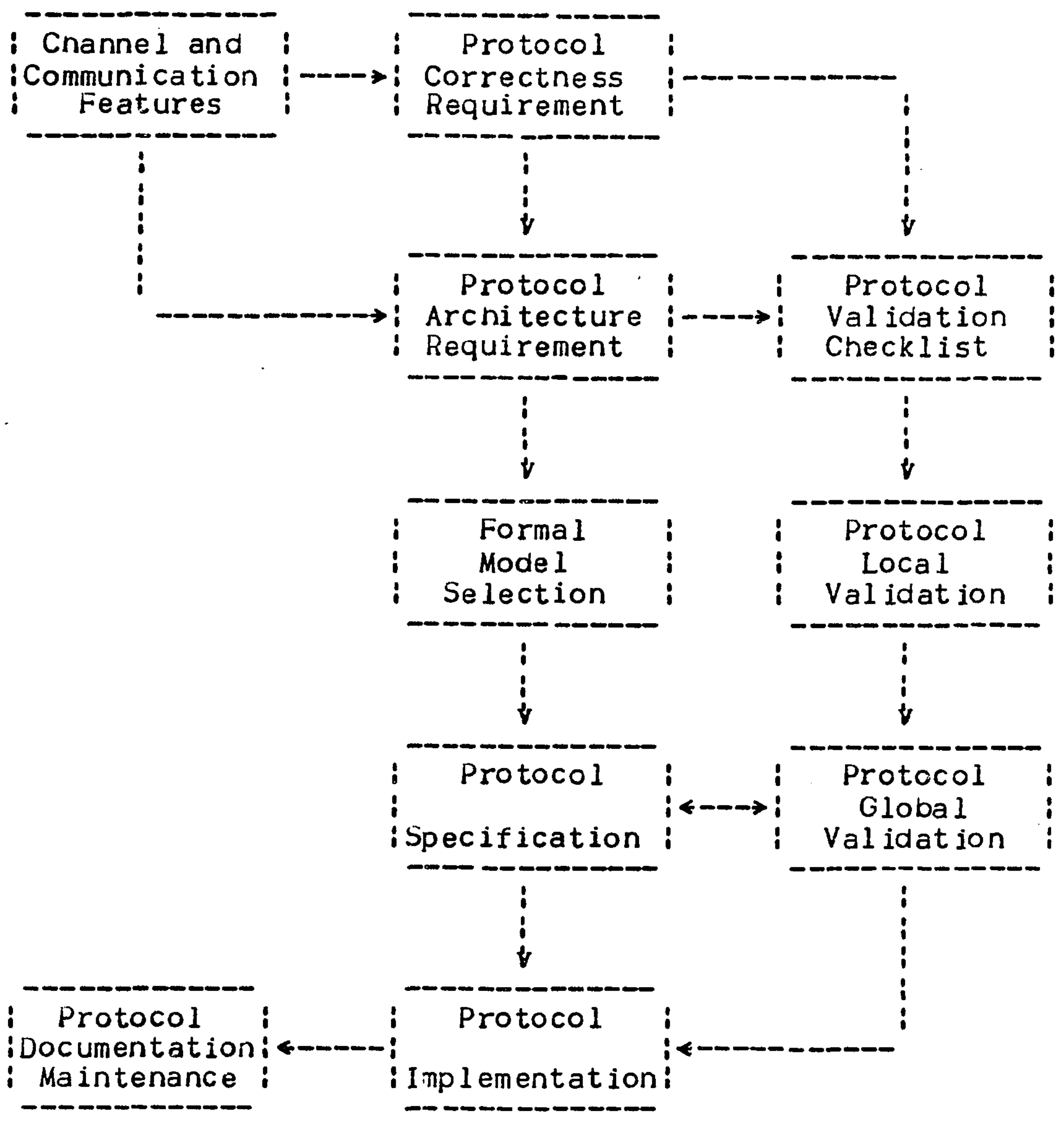

Channel and Communication Features
Protocol Correctness Requirement
Protocol Architecture Requirement
Protocol Validation Checklist
Formal Model Selection
Protocol Local Validation
Protocol Specification
Protocol Global Validation
Protocol Documentation Maintenance
Protocol Implementation

Fig. 2 A software engineering life-cycle for protocol constructions

In Chapter 2 we present the protocol correctness and the protocol architecture requirements for given communication channel features. We investigate the basic correctness properties for protocol design and introduce a systematic scheme to derive a minimum yet "sufficient" validation checklist with respect to the design requirement for any given protocol layer and topology. The investigation of this checklist provides the groundwork for future validation of communication protocols in Chapter 5.

In Chapter 3 we motivate the need for a formal model to design communication protocols and introduce the transmission grammar model to specify protocols. This model can represent and interrelate the hierarchical relationships of protocols. Using this model, We can first decompose a protocol into several components for validating the desired properties and then integrate the validated components for implementation. The arbitrary shuffle and substitution operations of formal languages are introduced to construct <u>inherently</u> correct protocols from those validated components. This model provides for step-wise refinement of the design while preserving the history of the refinement. Also, it allows the application of many existing techniques of formal languages and compilers to protocol validation and implementation. Shuffle and substitution operations are applied to manipulate the TGs in a step-wise design. The concise TG description of a protocol can provide documentation of the protocol design. The model has the flexibility of describing a complicated protocol with different degrees of details for validation and implementation.

Protocol validation of the TG model is based on the reachability analysis of protocol behavior. It includes both local validation and global validation of the protocol. The local validation checks whether a TG has the desired properties and eliminates program structure errors. The global validation can analyze the interactions between communicating entities and can reveal many complex syntax errors. We introduce a validation automaton model and automatically generate the global states represented by the VA model. We provide ten basic reduction rules which can be used to reduce the number of states and transitions in the global validation analysis. A validation system is also implemented to automate the reachability analysis. These important topics are discussed in Chapters 4 and 5.

After the specified TG has been validated, we can use refinement techniques to define a more detailed protocol description with proper semantics for automatic software/hardware implementation. This detailed TG should still preserve the validated properties. In Chapter 6 we present several automatic techniques for protocol implementation. The TG model can be used to describe both the interactions between communication entities and the detailed structure of individual messages. This specification may be directly used to drive a "recognizer" for the language (protocol). We call this recognizer a General Protocol System (GPS). The GPS is designed as an interpreter although it may also be implemented as a "protocol compiler". A data transfer protocol for the communication processor of ARPANET is specified using the TG model and GPS operations to demonstrate the overall design methodology.

Finally, Chapter 7 is the summary for this dissertation. In addition, further research topics in this area are also suggested.

In the Appendix, a PASCAL program of the validation system is listed. This system can automatically perform the local validation for protocol specified by the TG model.

CHAPTER 2

PROTOCOL CORRECTNESS

2.1 Introduction

In this chapter we study the protocol correctness problem. As the number of computer and communication networks increases, the organizations they serve are becoming increasingly concerned about protocol correctness. Compared with centralized systems, the remote communication nature of computer networks introduces protocol syntax errors. The problems of detecting, diagnosing, and recovering from these errors are more complex to solve because of the complex interdependencies between communicating entities and their channels. A global software engineering technique is needed to design and verify protocols in the networks. Until recently, the problems of protocol validation and verification were still not seriously considered unless the protocol had been found not to work.

The importance of protocol verification is only now becoming generally recognized. Protocol validation can be performed at (1) the design phase, (2) the development and test phase, or (3) the operation phase. The relative cost of correcting protocol errors during the operation phase is a lot higher than during the design phase. Clearly, it pays off to invest effort in finding design errors early and correcting them rather than waiting to find errors during operations and having to spend in the order of 10 times more correcting them. We therefore propose to verify protocol correctness during protocol design and specification in this dissertation.

The protocol correctness problem is very tricky. For example, a data transport protocol is correct under normal operation if it will completely transport messages (data) between entities without error, loss or improper order. A correct transport protocol will define what actions are to be taken to detect and correct error, loss and disorder. The complete transfer is an essential part of the correctness argument. Even if all the delivered messages are error-free, we still might suffer from deadlock, degradation, interferences, incompatibility, infinite idle looping, constant transmission noise and other catastrophic (failure) conditions which prevent complete delivery of the messages and proper termination. To verify that all the messages will eventually be delivered is an important verification problem.

In this chapter we first study the basic property of protocol correctness and then investigate in detail communication channel characteristics which have an important impact on protocol architectures and correctness. It is derived from both the lessons learned from practical experience and the result of theoretical reasoning. We then provide a validation checklist of protocol correctness problems. The checklist contains the "minimum" and "sufficient" set of existing correctness problems for a given topology and layer. The checklist can prevent the designer from wasting effort in examining non-existing problems and lead us to a systematic study of protocol correctness problems.

2.2 Protocol Correctness

Because protocols are a particular type of computer program, we first study computer program correctness before we investigate protocol correctness. Conway[78] lists eight different levels for a correct program:

1. A program contains no syntactic errors.
2. A program contains no compilation errors or failures during program execution.
3. There exist test data for which the program gives correct answer.
4. For typical sets of test data, the program gives correct answers.
5. For difficult sets of test data, the program gives correct answers.
6. For all possible sets of data which are valid with respect to the problem specification, the program gives correct answers.
7. For all possible sets of valid test data and all likely conditions of erroneous input, the program gives correct answers.
8. For all possible input, the program gives correct answers.

The appropriate level of correctness needed for any application should be determined at the requirement stage. Verification of the behavior of a protocol presupposes a definition of what the protocol is designed to do and the properties to be verified. Because of the complexity of protocols, level 6 correctness is considered to be sufficient in this dissertation. That is, a protocol is verified to be correct, if for all the global interactions which are realizable according to the problem specification, the protocol gives correct answers. The problem specification should specify the "minimum" correctness requirement as we will discuss later.

Current technology is inadequate for general and automatic verification of a computer program to level 6 correctness. For the verification of a complex protocol program we have an even harder problem of global interactions between communicating entities. As mentioned in Chapter 1, we will restrict ourselves to the investigation of an important subset of protocol correctness (protocol validation) in this dissertation.

Protocol validation is usually based on a straightforward reachability exploration of the protocol's behavior to reveal protocol syntax errors. Protocol validation techniques based on the TG model will be discussed in Chapters 4 and 5 to assure the absence of protocol syntax errors.

Since different protocols have different correctness problems, a "perfect" validation technique for one protocol may work poorly for others. To clarify some confusion over the validation problem, we study the fundamental properties of message transport problems, transmit interference problems and protocol syntax errors in the next three sections. The detailed discussion of protocol reliability problems is beyond the scope of this dissertation. However, our validation models can also handle the reliability problems.

2.3 Message Transport Problems

The message transport problems include damage, delay, loss and improper sequence.
They are applicable only to protocols below application layers and can often be handled by properly selected architectures. For example, a positive acknowledgment and retransmission (PAR) protocol can usually take care of the problems of damage, loss, duplicate packets, and improper sequence by assigning sequence numbers (or ID's) and checksums to each packet transmitted, and retransmitting any packet not positively acknowledged by the other side.

Correctness Hierarchy

In the protocol hierarchy, we know that protocol requirements for a protocol layer do not include any message transport problem that does not exist or has been fully corrected by its lower layers. This property can alleviate the complexity of validation. That is, no correctness checking for a message transport problem is needed in any layer if the problem has been fully corrected by its lower layers.

Transport Protocol

Transport protocols should be available in a computer network to guarantee that the messages are transported to the application layers in sequence and without damage, loss or duplication (i.e., no message transport errors). Transport protocols can provide fully reliable message communication between higher layer entities which must communicate over a less reliable medium as shown in Fig. 3. In this way we can prevent redundant checking in application layer protocols and provide a clean and correct input to them to simplify their design and verification.

Application layer protocols can therefore be assumed to have a perfectly reliable virtual channel provided by the transport protocols. The relationship is shown in Fig. 4. The design of the application layer protocol therefore need not consider message transport problems.

Distributed Checking

The checking of message transport problems can be put off as long as the problems can be fully solved by some combinations of protocol layers which are lower than the application layers.

We can use the disorder checking of messages in ARPANET as an example. The interface message processor (IMP) is responsible for checking packet sequences within a message (or segment) while the transmission control program (TCP) at a host is responsible for checking message sequences within the current session and between sessions. These checks are fully implemented by the combination of the two protocol layers, which are lower than the application protocol layers.

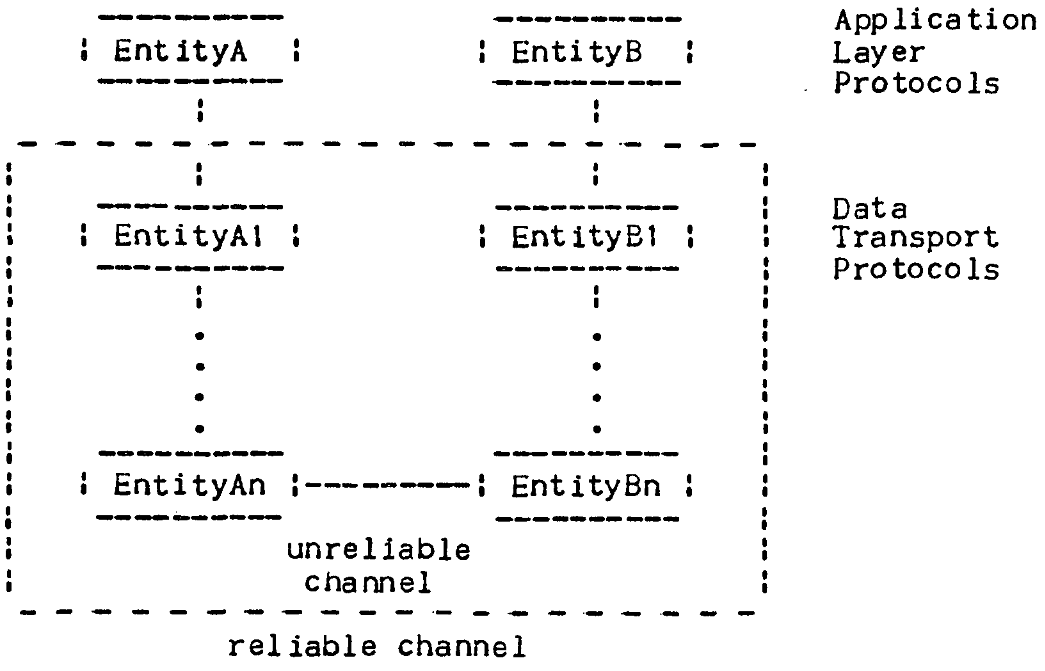


Fig. 3 Transport Protocols

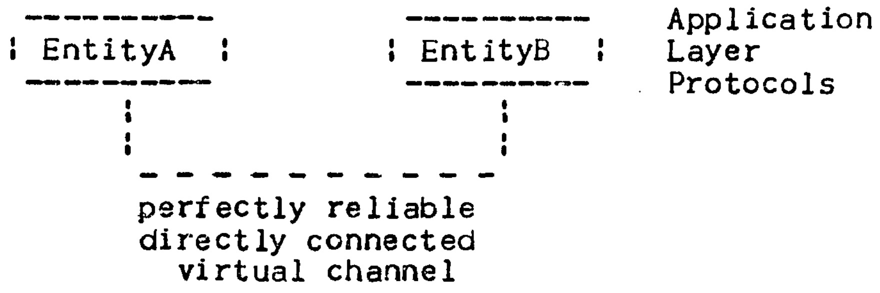

EntityA
EntityB
Application Layer Protocols
perfectly reliable directly connected virtual channel

Fig. 4 Virtual channel for application protocols

2.4 Transmit Interference

A _transmit interference_ occurs when the timing of the delivery of two (or more) transmitted messages is such that their relative timing results in interference, thereby causing the receiver(s) of the messages to react incorrectly and to obtain erroneous results.

The transmit interference problem as mentioned above is very difficult to resolve because the error is not reproducible. If the specific timing of two transmit actions which results in interference is very rare in practice, we would not be able to easily locate the error, because it is not likely to appear again within a short period of time. It is therefore important to prevent such errors from happening at the design stage by some formal verification and designing techniques.

To simplify the discussion we will restrict our consideration of interference to that between two transmit messages. We will examine four types of transmit interference: delay, reply, collision, and race.

(1) Delay Interference

Delay interference is the interference between two transmit messages which have the same syntax format and the same source and destination. As shown in Fig. 5, this type of transmit interference is caused by the critical timing of deliveries between a message m1 and a succeeding message m2, both of which were sent from send module A of entity A to receive module B of entity B. Duplicate interference occurs when m1 and m2 are the same message and one of them is duplicated. Disorder interference occurs when m1 and m2 are different messages and the arrival sequence of the two at entity B is different from the departure sequence at entity A.

```
 ------------                          ------------
|  -------   |      send m1       |     -------    |
| | send   |--+------------------+-->|receive|    |
| |moduleA|--+------------------+-->|moduleB|    |
|  -------   |      send m2       |     -------    |
|            |                    |                |
|            |                    |                |
|  -------   |                    |     -------    |
| |receive|  |                    |    | send  |   |
| |moduleA|  |                    |    |moduleB|   |
|  -------   |                    |     -------    |
 ------------                          ------------
   Entity A                              Entity B
```

Fig. 5. Transmits that cause delay interference

A single message submitted for transmission may be duplicated by the timeout retransmission mechanism because of either: (1) a premature retransmission by a sender while positive acknowledgment from its receiver is underway, or (2) loss or damage of positive acknowledgment.

To attempt resolution of these duplicate messages, the receiver can maintain a list of sequence number(s) of transmission(s) which it is currently expecting. The duplicate message will not be one of the expected messages and can be discarded. Sometimes message delay may also cause a duplicate from a previous closed connection to arrive during the current open connection. The delay could cause the following misinterpretations: (1) a "cross line" problem if incorrect acceptance of the "old line" duplicate by the "current line", (2) an incorrect opening or closing problem if the duplicate is an old open or close command, or (3) a replay problem [Fletcher78] if a series of old duplicates causes the opening of a connection and subsequent acceptance of "duplicate" messages. Unless we can afford to check exhaustively all possible ambiguities to verify that the misinterpretations will not lead to incorrect outcomes, a more elaborate sequence control mechanism is required [TCP79] to allow the receiver to distinguish all the possible duplicates from new messages.

In general, disorder interference can also be prevented if we have a sequence control mechanism. The scheme is well covered in the TCP protocol of ARPANET and will not be discussed here.

(2) Reply Interference

The second type of transmit interference is reply interference. Reply interference can either be piggyback interference or multiple reply interference.

(A) Piggyback interference -- This interference appears in a full-duplex channel and is the interference between a reply message and a send message at a communication entity. As we can see from Fig. 6, the send module A transmits a message m1, and then waits for a reply of m1 from receive module B. If at this time send module B wants to send message m2 which has piggyback control information and has the same syntax construct as the reply of m1, then a piggyback interference might occur.
When the send message m2 of entity B arrives at entity A before the reply message m1, entity A will "believe" that entity B is sending a reply m1. Unless careful verification techniques are used to verify that there is no ambiguity or confusion between m2 and the correct reply of m1, piggyback interference is likely to happen (especially when combined with the possibility of the damage and loss of m1).

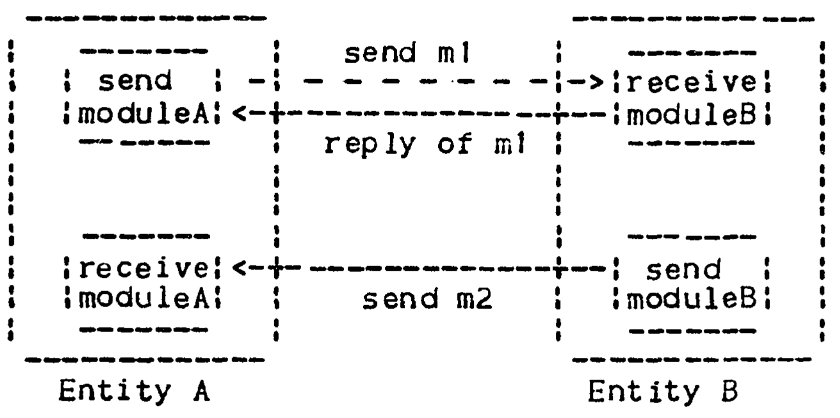


Fig. 6. Transmits that cause piggyback interference

When m2 contains piggybacked acknowledgment information, the resulting interference can lead to two kinds of misleading results:

(a1) Loss unawareness -- This happens when entity B unintentionally contains a positive ACK in m2 for the outstanding message m1 while m1 was actually lost because of a noise burst. So, before entity A's timeout has expired, it will receive the incorrect "acknowledgment" for its lost message. This sequence of actions of loss unawareness thus causes the irrecoverable loss of message m1.

(a2) Receive unawareness -- This happens when entity B unintentionally contains a negative ACK in m2 for the outstanding message m1 while m1 was actually correctly received. Constant receive unawareness can cause constant errors (called tempo-blocking [Hajek78]). For example, if entity B always sends a negative acknowledgment with a retransmit m2 when a timeout has expired, then a situation may arise such that entity A cannot decide whether the transmit of m2 is due to a negative reply of m1 or just a premature retransmission of entity B.

(B) Multiple reply interference -- This interference may occur between two replies as shown in Fig. 7. If two outstanding control messages m1 and m2 have been sent by entity A and the reply of m1 and the reply of m2 have the same syntax format, entity A would "believe" reply m1 to be reply m2 and could thus be led to an erroneous result.

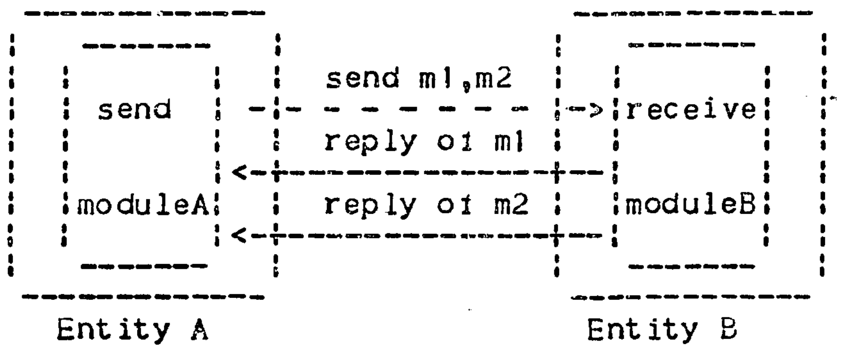


Fig. 7 Transmits that cause multiple reply interference

To avoid multiple reply interference, we can assign sequence numbers to replies and restrict the receiver to accept the replies in order. We can also check to formally verify the outcome is correct even with the ambiguity. For example, in ARPANET's host/host protocol, the CLS command is used either to initiate closing, aborting and refusing a connection, or to acknowledge closing. Suppose that host A sends host B a request for connection, and then A sends a CLS to host B because it is tired of waiting for a reply. However, just when A sends its CLS to B, B sends a CLS to A to refuse the connection. A will "believe" that B is acknowledging the abort, and B will "believe" that A is acknowledging its refusal [Feinler78]. Therefore, even though a communication entity can take a wrong message as its reply because of ambiguous interpretation, the outcome could still be correct.

(3) Collision Interference

The third type of transmit interference is collision interference. This interference occurs when two communicating entities A and B try to access a channel and transmit messages m1 and m2 respectively at the same time as shown in Fig. 8. This could cause call collision if the channel between the entities are half duplex and they call one another at the same time and, due to transmission delays, each finds the other one busy. If the channel is a common broadcast type in the areas of satellite broadcasting or radio switching, mutual interference will produce incorrect reception for both transmissions.

```
 -------------                -------------
|  -------    |              |  -------    |
| | send  |   |   send m1    | |receive|   |
| |moduleA|---+--------------+->|moduleB|  |
|  -------    |              |  -------    |
|             |              |             |
|             |              |             |
|  -------    |              |  -------    |
| |receive|   |   send m2    | | send  |   |
| |moduleA|<--+--------------+--|moduleB|  |
|  -------    |              |  -------    |
 -------------                -------------
   Entity A                     Entity B
```

Fig. 8 Transmits that cause collision interference

Constant collision is a deadlock and should be recovered or prevented. To recover from collision usually requires random delays and retransmits. To prevent it two types of control schemes can be used [Kleinrock78]. (1) Fixed Allocation: This is a static reservation access method which preassigns channel capacity to users, thereby creating "dedicated" channels. (2) Dynamic Reservation: This assigns channel capacity to an entity when the entity has data to send. Such schemes as Polling (where one waits to be asked if he has data to send), Active Reservation (where one asks for capacity when he needs it), and Mini-Slotted Alternating Priority (where a token is passed among numbered users in a prearranged sequence, giving each permission to transmit as he receives a token) all fall in this category. Another scheme using delay buffers [Liu78] can also be used to prevent collision in a loop network.

(4). Race Interference

This type of transmission interference happens when two communication entities A and C try to transmit messages m1 and m2 respectively to entity B and to process common data bases in entity B concurrently and asynchronously as shown in Fig. 9. It is similar to the consistency problem in a centralized system where many users update a common data base. The situation is more complicated in a computer network when the consistency of distributed data bases has to be controlled. The interference could cause a race or inconsistency condition. In a centralized system, we can use synchronization mechanisms to synchronize the access of these common data bases, but the approach is more difficult in a distributed system [Ellis77a].

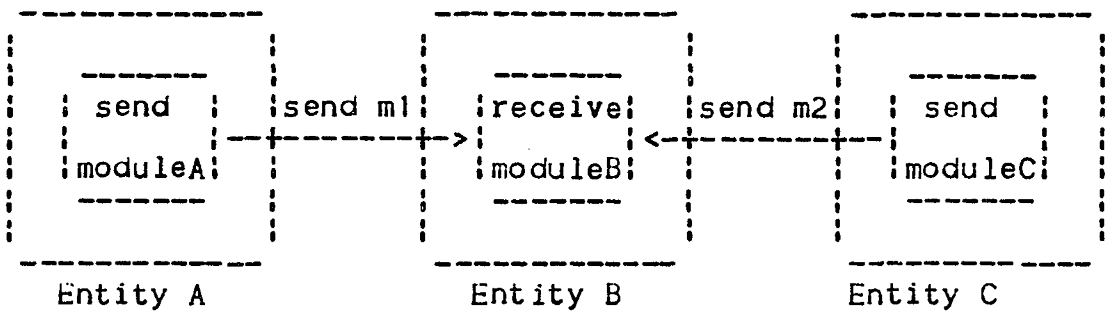


Fig. 9. Transmits that cause race interference

2.5 Protocol Syntax Errors

Message transport problems, transmit interference problems and reliability problems may cause protocol syntax errors. Protocol syntax errors occur when communication entities lose synchronization, and they consist of incompatibility, infinite idle looping and deadlock and improper termination. These errors are usually applicable to all protocols. They have to be fully checked within each individual layer by validation techniques.

(1) Incompatibility

Two communicating modules (send and receive) are compatible or dually-complete if they contain neither unspecified message receptions (receive incompleteness) nor unspecified message transmissions (transmit incompleteness).

The receive completeness of a protocol means that it specifies all the possible message receptions under normal operating conditions (same as the logical completeness in [Zafiropulo78]). One of two things can happen if, in the design of a protocol, a message reception is inadvertently omitted, resulting in loss of the message. The protocol can either initiate an error recovery procedure or it may lose synchronization and become unpredictable with respect to the design. Receive completeness is therefore important in a protocol design [West78a].

Another design requirement, called transmit completeness, requires that the design contain no unspecified message transmission. A natural way of thinking about unspecified transmission is to consider a message which can never be received due to the fact that the corresponding message has never been conceived by the sender. Transmit incompleteness can also cause adverse behavior and should be avoided.

In general, receive incompleteness is more likely to be overlooked by designers than transmit incompleteness during the design phase. For a hardwired protocol like X.21 there is no input queue between the entities of both sides of a channel, and a message reception has to be processed immediately; otherwise, it is likely to be lost. To assure the receive completeness for a receiving entity with no input queue, the entity usually has to remain idle when waiting for a receive message to come, because it does not "know" when the message will arrive. The time period of idle waiting is a waste of resources for a processor entity. In the case that a protocol has an input queue and input interrupt handling, it can avoid waiting idly by queueing input messages before processing them. This is called queued input completeness.

(2) Infinite Idle Looping (Oscillation)

Looping is repeated execution of a cycle of states and can cause by definition unboundness and infinite idle looping. An unboundness error occurs when an entity requests an unlimited or unbounded amount of storage or CPU time. Idle looping is repeated execution of a cycle of states (actions) by one of the two communicating entities and no effective progress can be made by the execution of any actions in the cycle. Effective progress is made during communication whenever a message is fetched or accepted by one of the communicating entities. Infinite idle looping is of course an error situation. An example of infinite idle looping is the continuous retransmission of an outstanding message over a failing channel.

For example, a correct data transfer protocol has to correctly and completely transfer all the messages. Infinite idle looping is one of the common situations that a complete transfer might suffer. When we verify a data transfer protocol to be correct, we have to prove that the protocol is free of infinite idle looping and it will terminate properly. We will discuss techniques to identify loopings in Chapters 4 and 5, and show that the potential unboundness can be handled by inductive reasoning on the loopings.

(3) Deadlock

Deadlock is one of the most serious system malfunctions possible, and one must take great care to avoid it or to recover from it. The concept of deadlock is not as well understood for computer networks as for computer systems. Five types of deadlock can be classified in a communication system.

(A) Resource deadlock -- This type of deadlock may occur when a user's processes request resources at distant hosts. A resource (usually buffer space) deadlock in a communication system is the situation in which two or more competing demands are unknowingly waiting for unavailable resources held by each other.

Flow control procedures coordinate the sending entity at one end of a channel with the receiving entity at the other end to prevent overrun of the input buffer space at the receiving entity. It presents constraints on the flow of data and if the situation ever arises whereby the constraint cannot be met, then the flow will stop, resulting in a buffer space deadlock.

For example, in ARPANET a direct store-and-forward deadlock happens when neighboring switches (IMPs) cannot successfully transmit packets to each other because all the store-and-forward buffers are filled and cannot be released. An indirect store-and-forward deadlock occurs when all store-and-forward buffers in a loop of IMPs become filled with messages, all of which travel in the same direction and none of which are within one hop of their destination. Other types of resource deadlock will be discussed later in this chapter.

(B) Synchronization deadlock -- This type of deadlock occurs if the protocol is improperly designed or a component failure occurs in the communication system. The sender and receiver may lose their synchronization and thus no effective progress can ever be made.

(C) Receive deadlock -- A receive deadlock occurs when the sender or receiver is waiting idly for the other to send a message while there is no such message coming through the channel between them. The deadlock may be caused by an incompatibility or a message lost problem. The incompatibility problem can be avoided if we have a carefully validated design. A timeout and retransmission mechanism can be used to recover from a message lost.

(D) Static deadlock -- A static deadlock is a state that cannot reach a proper terminate state as we will formally define in Chapter 4.

(E) Dynamic deadlock -- Infinite idle looping is an extreme form of looping and is also a dynamic deadlock. A timeout mechanism can usually eliminate the static deadlock by timing out to a recovery state when it happens. However we may still have the unpleasant dynamic deadlock situation in which a cycle of actions is executed but no effective progress can _ever_ be made.

We can see some combinations of the five types of deadlock. A store-and-forward deadlock is a dynamic resource deadlock. A constant collision in a half-duplex channel is a dynamic synchronization deadlock. For a hardwired connection protocol like X.21, we may experience a static synchronization deadlock.

The classification is useful when we discuss protocol validations using a formal model. To check the resource (buffer) deadlock, we should include channel and buffer storage in the model. To check static deadlock, we only need to identify a global state with no exit, but to check dynamic deadlock, we must also check looping.

2.6 Channel, Protocol and Correctness

Many communication and channel characteristics have a profound impact on protocol architecture and correctness. In this section we present a list of channel and communication characteristics and discuss their respective impact on protocol architecture and correctness. The study is aimed at investigating the correctness problem checklist in a systematic way for protocols of various layers and topologies. Appropriate protocol architectures should be used in the protocol design to solve these problems. Validation models should also be developed to verify that these architectures do indeed solve these problem.

In the following, we explain each channel characteristic and then discuss its respective impact on protocol architecture and correctness.

(1) Physical Proximity

Computers and other digital systems, whenever they are in close physical proximity, usually operate on parallel data, so data is transferred in parallel by a computer bus with separate control, data and address lines. However, as the distance between these devices increases, not only do the multiple wires become more costly, but the complexity of the line drivers and receivers becomes greater owing to the increased difficulty of properly driving and receiving signals on a long wire. In most data communication and networks, serial transmission over a single set of lines is therefore preferable to parallel transmission to reduce communication cost.

<u>architecture impact</u>

For protocols of remote communication, the bandwidth becomes more limited, noise on the channel increases and the transmission error and delay probability increases. We need techniques to control the data exchange and correct the errors. For a fair allocation of the limited bandwidth, we often need <u>pipelining</u> to break a long message into short ones. We also need <u>data transparency</u> and <u>formatting</u> techniques to tell where a data block begins and still be able to send any data pattern. Data transparency is usually done by byte stuffing and bit stuffing (like SDLC protocol) where the sender adds special bytes or bits which the receiver removes, thereby preserving the original pattern. Clark [78] indicates that communication lines in a computer network differ from a "big bus" in both <u>defensiveness</u> and <u>generality</u>. We tend to feel a need for more fault-tolerant and generally applicable links procedures and protocols to control data communication links in a network environment.

<u>correctness impact</u>

Communication between two remote entities will introduce transmission delay and propagation delay. Delay can be defined as the time between transmission and delivery of the first bit of the message. There are four components of delay for remote communication: (1) transmission delay, which is proportional to the size of the message and inversely proportional to the transmission bandwidth; (2) propagation delay, which is a function of the distance of the transmission circuit (satellite links have a large propagation delay of about 250 msec); (3) processing delay incurred at any switching node or store-and-forward facility; and (4) queueing delay, which is a function of system load. Delay can cause problems for many conventional approaches. Certain types of applications may not become economically feasible because this delay causes unsatisfactory response time. Variable delay for different messages in a packet switching network can also create problems such as out of order, cross talk and replay.

(2) Asynchronous/Synchronous

There are two primary modes of data communication: asynchronous and synchronous. In asynchronous mode the time intervals between transmitted data units (which are characters) may be of unequal length. This mode is easily generated by electromechanical equipment (e.g., Teletype keyboard). Synchronous transmission, on the other hand, transmits data units (which can be characters or bits) at a fixed rate with equal length of time intervals.

architecture impact

Asynchronous transmission is controlled by start and stop bits at the beginning and end of each character unit. A start bit is used to indicate the beginning of a character and the receive side will be able to correctly receive the character when protocol rules are followed by both sides. In the synchronous mode the entire block of data string can be sent without start/stop bits and achieve higher efficiency. However, a synchronizing signal must be provided to transmit and to synchronously receive.

correctness impact

The asynchronous mode is distortion-sensitive because the receiver depends upon the incoming signal to become synchronized. Any distortion will affect the correctness with which the character is assembled. This impact restricts its speed because a reasonable amount of margin must be build in to accommodate distortion. In synchronous mode, characters or messages must be sent synchronously which means buffering is required. Also, even a slight timing error from the data bit string can cause the entire message to be faulty.

(3) Input Queue (Buffering)

For the type of protocols with input queues, an interrupt handler can be used to handle the queueing of input interrupts. This will alleviate the entity from having to wait indefinitely for an input and then waste resources. The handler will handle input processing when the entity is doing something else. The queued input also relaxes the requirement of input completeness to a loosened requirement of queued input completeness, and we only need to make sure that an input is processed sometime before the buffer overflows.

architecture impact

Buffer space usually is limited and a flow control procedure usually is needed to prevent buffer overflow and to match the sender's rate with the receiver's rate. Several schemes for flow control will be discussed in later sections.

correctness impact

(A) Incompatibility — For buffered type protocol, queued compatibility will need to be verified.

(B) Resource (Buffer) deadlock -- This synchronization error has already been discussed in Section 2.5.

(4) Transmission Errors and Loss

Noise and distortion are some undesirable characteristics and disturbances in most long distance channels. They can generate errors in transmission. On a hardware line, unless the line is completely open, it is likely that a transmitted message will arrive damaged but it is not likely to be lost. In networks with multiple lines and nodes in the transmission path, the message is more likely to be lost. Except for voice and speech protocols, a damaged message usually has to be discarded if it cannot be corrected. In the case of networking, if the addressing field is damaged then the source (sender) can no longer be identified and the message has to be treated as lost. We will therefore discuss these two characteristics together.

architecture impact

To perform error recovery (error control) of transmission errors, error detection mechanisms using parity and cyclic redundancy checksums are often used to identify the error. In a data communication system data integrity is essential. Refinements are usually added to correct the detected errors, either by operations on the received data or by activating retransmission from the source. Better detection/correction requires longer redundant codes, increases overhead and reduces efficiency.

For lost message recovery, the desirable approach is to allow for a timeout at the source to either (1) timeout and retransmit outstanding messages, such as Positive Acknowledgment Retransmission Protocols (PAR) or (2) timeout and send out enquiry such as Binary Synchronization Communication (BSC) protocols. Generally, a positive acknowledgment is sent to acknowledge correctly received message(s).

For the retransmitting type of protocol (PAR), the destination has the possibility of receiving a duplicate when a premature timeout or lost acknowledgment happens. Therefore the source has to identify its transmitting message (usually using a sequence number) for the destination to be able to detect duplication. For the enquiry type of protocol, the source never retransmits a correctly received message and thus has no duplication problem. However, a message delivery of the enquiry type protocol requires more turnaround overhead.

correctness impact

The following are some protocol error conditions that can result from error/loss characteristics.

(A) Undetected error -- Detecting a damaged message is a well developed technique in coding theory. However, it is not possible in reality to have a scheme detecting all of the transmission errors, and occasional acceptance of a faulty control/data message could be resulted. An extreme example which shows why an undetected error is difficult to eliminate is this: In theory, given any elaborate error detecting scheme, there always exists a set of bits in a message and a method of changing these bits during transmission to result in an undetected error. However rare this coincidence might seem to be, it is still possible for it to happen in a real-world communication channel.

An undetected error could result in improper acceptance of a damaged message and could cause protocol malfunctions. One of the most serious undetected errors, called ACK error, occurs when a negative acknowledgment signal changes to a positive acknowledgment signal due to damage and thus results in unrecoverable loss of that message. In the case of an undetected addressing error, a message could circulate forever in a network or could be accepted by a wrong destination.

From coding theory, we know that a protocol designer has various techniques available to detect errors in messages to any desired degree of reliability with some redundancy and checking overhead. Some protocols (as in ARPANET) even have layers of error detection to reduce the possibility of an undetected error. In general, it is reasonable to assume perfect error detection (i.e. no undetected errors) when dealing with reliability.

(B) Idle Looping -- Retransmitting an outstanding message for the loss recovery protocol could cause infinite idle looping.

(C) Dynamic Deadlock -- This protocol syntax error has already been discussed in Section 2.5.

(D) Duplicate Interference -- In a transmission channel where error and loss are presented, a duplicate detection mechanism is necessary at the receiving site.

(E) Error Delay -- Besides transmission and propagation delays, transmission errors and loss can also introduce error delay when a transmitted message is damaged or lost.

(5) Half-Duplex

A half-duplex link is a physical link connecting two communication entities and allowing data to be transmitted by either entity. However, data may not be transmitted by both nodes at the same time.

architecture impact

Line control schemes are needed to determine which entity is going to transmit and which entity is going to receive in the half-duplex line.

correctness impact

A collision interference could happen on a half duplex channel when two or more entities try to transmit at the same time.

(6) Full-Duplex

A full duplex link is a link connecting two communication entities and allowing both entities to transmit data simultaneously.

architecture impact

To maximize the effective use of bandwidth and to take advantage of a full-duplex link, control information such as acknowledgments is usually piggybacked in data messages by the acknowledging entity. To avoid piggyback interference, an echo mechanism can be used to always retransmit acknowledgments representing the current status of correctly received messages.

correctness impact

(A) Piggyback interference -- Receive unawareness and loss unawareness can happen when piggyback acknowledgment is used in a full-duplex line. Receive unawareness can result in a tempo-blocking problem. Tempo-blocking is somewhat similar to the collision problem which happened in a half-duplex line. It is a speed dependent lockup during which no effective progress can be made by either of the communicating entities but might disappear after a suitable change of relative speed ratio of both entities. Tempo-blocking is different from idle looping and is actually a global idle looping (i.e., looping occurring at both communicating entities), when the useful flow of information in both directions of the channel stops because of two-sided receive unawarenesses.

(B) Collision interference — This interference problem has already been discussed in Section 2.4.

(C) Race condition — When the send module and receive module of an entity can access channel and buffer concurrently in a full-duplex line, we should avoid race conditions between the two modules.

(7) Multiple (outstanding) sends

Single outstanding sending requires a sender to wait for the acknowledgment of the previous outstanding sending before it can send the next message. The turnaround time between sending a message and receiving its acknowledgment is wasted as far as the channel bandwidth is concerned. This waste could be substantial in the case of satellite links with very large propagation delays. To increase channel utilization and transmission efficiency, some of the powerful protocols (like SDLC, TCP and NCP) have a feature to allow several sending messages to be outstanding.

architecture impact

To keep the multiple sending messages in order at the receiving entity, a message ID will not be enough for sequence control. We need to assign a sequence number (usually in an ascending order) for each message to detect missing and duplicate messages and to maintain state information at the sender and receiver. This will allow a sender to identify the messages which need to be retransmitted and a receiver to detect duplicate and out-of-order messages.

Different acknowledgment schemes can be introduced to handle multiple outstanding sends. For protocols with "expect receive number" schemes such as SDLC, we can have a multiple acknowledgment (sequential) scheme which can acknowledge more than one outstanding message at a time. The acknowledgment can be piggybacked if necessary. For protocols with "reassembly buffer" schemes such as the interface message processor of ARPANET, we can have queued acknowledgment and acknowledge any acceptable packet upon reception without restricting packets to arrive in order.

To send a high priority message, we need to have the capability to interrupt the sequence control mechanism to process the priority message before those regular messages which have arrived earlier.

correctness impact

(A). Disorder interference -- If control mechanisms are not well defined in the protocol, the messages could be received out of order or be duplicated and cause interference.

(B) Sequence number overflow -- It is essential to remember that the actual sequence number space is finite and is limited by the number of bits assigned for the sequence number field in a message. Sequence number overflow can be handled by recycling the number using the MOD function.

(C) Gaps degradation [Kleinrock78] -- A degradation is defined to be a reduction in the network's level of performance. For multiple outstanding sends, we usually put a limit (n) on the number of messages that are allowed to be outstanding at a time. If n messages are in flight, then the next one may not proceed until an acknowledgment is returned to the source for some of the n outstanding messages. Gaps degradation comes when the round trip delay is greater than the time an entity takes to feed the n messages into the network. This will result in the source being blocked awaiting ACK's to release further messages. This will clearly introduce unnecessary gaps in the message flow resulting in reduced throughput and should be avoided when we design the multiple outstanding send protocol.

(D) Multiple reply interference -- This interference problem has already been discussed in Section 2.4.

(8) Multiple Switches

A switch entity is an entity whose primary function is switching data in a network. For computer networks with multiple switches for end-to-end communication, three switching technologies can be used — circuit switching, message switching, and packet switching. Comparisons of the switching techniques are well covered in the literature and will not be repeated here.

architecture impact

To allow switches in a computer network to route or fetch an incoming message, a destination ID is needed in the multiple switch end-to-end communications. This ID is defined to be the addressable entity and is the end point(s) of logical connection(s). To acknowledge a message, we also should consider end-to-end or logical layer acknowledgment. We therefore introduce a protocol hierarchy consisting of two protocols which are a switch-to-switch protocol for the basic transmission function (destination ID checking and routing) and an end-to-end protocol to deal with overall transmission integrity.

In a packet switching network, store-and-forward deadlock can occur if proper precautions are not taken [Kleinrock76]. A deadlock prevention protocol therefore must be considered. For example, a "buffer class" scheme as in [Raubold76] to partition the buffers in a switch into classes can prevent the deadlock.

correctness impact

(A) Processing delay -- In a computer network with many switches, the messages experience combinations of transmission, propagation, processing and queueing delays at each switch along a communication path. The increased delays need to be properly handled when dealing with protocol correctness.

(B) Switch addressing error -- If a switch in the network does not detect addressing damage of a message, it could either incorrectly accept the message which is not destined for it (acceptance error), or it could incorrectly reject the message which is destined for it (rejection error). An acceptance error could damage the integrity of data transfer or control synchronization. A rejection error could result in the message circulating around in the network and cause degradation.

(C) Store-and-forward deadlock -- This protocol syntax error has already been discussed in Section 2.5.

(9) Multiple Paths

In a multiple switched network, we sometimes have multiple paths between end communication entities. This is usually true for a general distributed topology and is especially true for the internetwork system. The multiple paths may provide the opportunity to make the link fault-tolerant. However, it may also introduce some complex problems such as routing and variable delays to the protocol design.

architecture impact

If multiple paths exist in a network, the switch then has to choose a proper route (path) for its outgoing message. This is the so called the routing problem and a routing table/algorithm is required to calculate a suitable route for a complex network.

Alternate paths between two entities also introduce variable delays when different paths are taken by the packets. Duplicate messages from a previously closed connection could arrive and be accepted by the current connection when the delays vary in a large range. This is the "cross line" problem as we discussed in Section 2.4. Cross line detection schemes [Fletcher78] such as sequence number selection, and three-way handshake of open or close are needed to solve the cross line problem.

correctness impact

(A) Path-fault tolerant -- An important opportunity for increasing the reliability of a network of computers results from the multiple paths between entities. When a switch or line failure does occur in a single path type of network (such as a single loop network or a point to point connection), it means a termination of communication for those entities which need to use the node or line. Alternate paths, on the other hand, can be more fault tolerant by allowing another path for message delivery.

(B) Variable delay -- This can be introduced when multiple paths exist.

(C) Disorder interference -- Variable delay can cause problems such as out of sequence, cross line and replay.

(D) Looping degradation -- Looping degradation occurs due to independent routing decisions made by separate switches which cause traffic to return to a previously visited switch or cause traffic to make unnecessarily long excursions on the way to its destination. The occurrence of loops causes occasional large delays in message delivery. Some loop-free routing algorithms have been published [Nayer75, Segall78].

(E) Direct store-and-forward deadlock -- This protocol syntax error has already been discussed in Section 2.5.

(10) Advance Receive

In some sequential acknowledgment protocols like SDLC, the message segments have to arrive in sequence; otherwise they will be discarded. For a packet switching network when variable delays have a relatively large range of values, this will result in an unacceptably large amount of retransmission. The advance receive feature in a protocol will solve this problem by accepting disordered segments and reassembling them in order.

architecture impact

For advance receive of segments, a window mechanism or allocation mechanism can be used for flow control. A window mechanism has been used in the design of TCP of ARPANET. Within each message, called segment, an indication of the amount of data that the sender of this segment is willing to receive is presented. The segment does not have to arrive in sequence to be accepted. It will be acceptable as long as it is within the acceptable window (a range of buffer space). An allocation mechanism on the other hand allocates storage and changes the storage via special allocation control messages. Both schemes require features in the protocol to do the segmentation/reassembly and to maintain an expected receive list at the receiver and a retransmission list at the sender.

correctness impact

The advance receive scheme can cause more complex correctness problems of dynamic resource deadlock such as reassembly lockup, piggyback lockup and christmas lockup [Kleinrock78]. All these lockups have been experienced as bugs in previous ARPANET designs. This scheme can also cause severe performance degradation. Single packet turbulence and phasing are two phenomena that were experienced in previous ARPANET designs. We should make an effort to prevent them when we design this type of protocol.

(11) Internetworking

The internetwork environment consists of hosts connected to networks which are in turn interconnected via gateways.

architecture impact

Besides gateways, a internet protocol needs to have internet header [TCP79] to fragment and reassemble internet packets for the transmission between "large packet" and "small packet" networks. This type of protocol also need addressing schemes to transmit the internet packets toward their destinations.

correctness impact

Besides the problem of message incompatibility between internet packets, internetwork communication has a longer delay than single network communication and the delay is usually significant enough to make cross line detection schemes a necessity.

The result of these communication characteristics and their corresponding impact on protocol architectures and correctness problems is summarized in Table 1.

Table 1 Channel characteristics and their impact on protocol architecture and correctness.

CHANNEL/COMMUNICATION CHARACTERISTICS	Architecture impact	Correctness impact
1. long distance channel	defensiveness, generality, pipelining, formatting, data transparency.	incompatibility, channel capacity limitation, transmission/propagation delay.
2. asynchronous	start/stop signal (code).	distortion sensitive.
synchronous	timing signal.	timing error.
3. input queues (buffering)	interrupt handler, flow/congestion control.	incompatibility, resource deadlock.
4. error(damage) loss	Parity/CRC error detection (ACK,Timeout, Retransmit, duplicate detection, ID); (ACK/NACK, Timeout, enquire for single outstanding).	Undetected error, idle looping, (no termination), ACK error; dynamic deadlock, duplicate interference, error delay.
5. HDX	line control schemes.	collision interference.
6. FDX	piggybacked ACK, Echo mechanism.	receive unawareness, loss unawareness, collision interference, race condition (channel/buffer)

Table 1 (continue) Channel characteristics and their impact on protocol architecture and correctness.

CHANNEL/COMMUNICATION CHARACTERISTICS	Architecture impact	Correctness impact
7. multiple outstanding sends	seq control, interrupt, multiple ACK,echo mechanism duplicate/missing detection retransmission list.	disorder interference, sequence no. overflow, gaps degradation, multiple reply interference.
8. multiple switches (store/forward)	dest ID checking, end-to-end ACK, addressing, buffer class allocation.	acceptance error, rejection error, processing delay, S/F deadlocks.
9. multiple paths	Routing table, switching, cross line detection scheme (connection SN selection scheme, 3-way handshake open/close).	variable delay, can be link-fault tolerant, delay interference, loopings degradation, direct S/F deadlock.
10. advance receive	window/allocation mechanism segmentation/reassembly, expected receive list.	boundness, piggyback lockup, reassembly lockup, christmas lockup, phasing degradation, single packet turbulance.
1^1. Internetworking	fragmentation, gateway.	incompatibility, delay interference.

2.7 Validation Checklist

Protocol correctness problems are the major source of protocol syntax errors. As we can see in the discussion of the previous section, different protocols have different correctness problems. Only a subset of correctness problems actually exist in a given protocol topology and layer. Even an expert may make validation mistakes if he does not have a clear understanding of potential correctness problems in his protocol design.

A deficiency of existing validation techniques is the lack of a systematic scheme for examining protocol correctness problems. We introduce the use of a validation checklist, which contains a set of existing correctness problems (hopefully is the "minimum" and "sufficient" set) for a given topology and layer, during our validation and construction process. The checklist can prevent the designer from wasting effort in examining non-existent problems and can lead us to a systematic study of protocol correctness problems.

The protocol validation checklist can be easily obtained in the following way. We first establish a list of channel and communication characteristics for the topology and layer we are working on.
Table 2 shows some protocol layers/topologies and their possible channel characteristics.

Table 2 Designer's checklist of protocol characteristics

topology and LAYERS

CHANNEL/COMMUNICATION	point to point			star	loop	arpanet			internet
CHARACTERISTICS	X.21	BSC	HDLC	IMP	LIU	IMP	NCP	PROCESS	TCP
input queues	N	-	Y	Y	Y	Y	Y	Y	Y
error,loss	Y	Y	Y	Y	Y	Y	Y	N	Y
HDX	-	Y	-	-	N	-	-	-	-
FDX	-	N	Y	-	N	Y	Y	Y	Y
multiple switches	N	N	N	-	Y	Y	N	N	N
multiple sends	N	N	Y	-	N	Y	N	-	Y
multiple paths	N	N	N	N	N	Y	N	N	N
advance receive	N	N	N	-	N	Y	N	-	Y
concurrent update	N	N	N	N	N	N	N	Y	N

Y: Yes, N:No, -: Depend on the design choice

From these characteristics and the result in Table 1 we can obtain the validation problem checklist. In Chapter 5 we will show how this checklist can help us construct a validation model to reveal protocol syntax errors in a systematic way.

CHAPTER 3

FORMAL APPROACHES TO PROTOCOL DESIGN

3.1 Introduction

Because protocols for communicating entities require complex global software developmemt, it is generally difficult to verify the correctness of a protocol directly from its informal description. Formal models are usually required to allow some of the significant features of the operation of the protocol to be formally specified and then verified. A formal model will also be useful in protocol implementation and documentation.

In this chapter we review formal approaches and introduce a formal model, called the transmission grammar (TG), using a context-free grammar (CFG) for the design and specification of communication protocols. It is similar to the Backus-Naur form that has been used to define the syntax of an algorithmic language like ALGOL60 [Naur63].

To set up the background of a formal approach to protocol design, we first introduce design problems of communication protocols. We then summarize and compare some other formal approaches to specification and verification of communication protocols. The transmission grammar (TG) is used to define the protocol for the communication entities of a computer network. For the layered protocol design, the communication entity of each layer is decomposed into more detailed inner-layered components and/or validation independent parts. The local approach is first used to define the TG for each of the decomposed components and logical parts. The shuffle and substitution operations are then applied to integrate the TGs of the logical parts and the TGs of the components, respectively. Examples are given to illustrate the grammatical properties of protocols and some specification techniques of TGs.

3.2 Communication Entity

A communication entity in a network system is a protocol module that can communicate with other entities in a distributed fashion. It could be a logical (software) entity or a physical (hardware) entity. The network system is usually logically structured into layers in a layered approach. The principle of layering can be found in the Reference Model of Open System Architecture by ISO [ISO78]. Each layer provides communication entities in the next higher layer with sets of services. Entities within a layer can communicate directly only by using services of the next lower layer.

Each entity in a layer may have its own inner-layered components and/or several validation independent parts (VIPs) to simplify its design. (For example, an IMP entity may have sender and receiver parts.) The decomposition and integration techniques of communication entities will be discussed in Chapter 4.

For the TG model, we use a _local_ approach in the following sense: instead of having a single set of rules governing the interactions between several communicating entities, we define a set of grammar rules within _each_ communication entity to regulate the interactions between them. Each entity, therefore, corresponds to a transmission grammar (TG) which defines its protocol program. The entity's TG may be further decomposed into more detailed inner-layered component TGs or VIP TGs.

The entities of each layer are designed to be as independent as possible so that they can be changed without requiring extensive changes at other layers. This requires that the protocol designer must have a clear understanding of structured design. This structured design of communication protocols will also :(1) reduce the complexity of verification by way of structured verification (That is, prove the properties of one layer by utilizing the known (proved) properties of its lower layers); (2) reduce the software implementation complexity and cost; and (3) increase the reliability of the system by redundant error checking and recovery at different layers.

3.3 Formalism of Protocols

For a network to provide the reliable and flexible service required by its users, it must be based on a sound, extendable and well-defined design. A well-defined protocol design should provide a precise and unambiguous description of protocol rules to enable its verification, implementation and future extension.

Motivation of Formalism

Although the protocol serves an essential role in communication, protocol design for computer networks remains a relatively difficult task and an "arcane art". Our present ability to handle protocols is just like the ability we had with programming languages in the 195Ø's, when no convenient formalism (like the Backus-Naur form used now) was available to describe syntactic constructs of languages [Fraser76]. Without such a means of expression, it was very difficult to make proofs about the languages we invented or to have a systematic way of implementing a compiler for the language. Without such a means of expression, it was impossible to lead us to the concept of a compiler compiler, a compiler generator or a general grammar parser (like the LR(K) grammar parsers).

The breakthrough came in around 1956 when Noam Chomsky gave a mathmatical model of a grammar in connection with his study of natural languages. In 1958, the grammar concept was found to be of great importance with the official syntax description of ALGOL6Ø using the Backus-Naur Form (BNF).

For the design of a communication protocol, we would also like to have a clear, concise and precise model to describe various protocol features and to achieve the following goals.

Goals of Formalism

A formal model cannot be used blindly without considering the nature of protocols in a complex computer network. For a network as complex such as ARPANET, the protocol may involve considerations such as connection setup, sequence numbering, buffer allocation, flow control, error recovery and message format conversion. The protocol design is often so complicated that the hierarchical approach (layered approach) becomes a necessity. In order to provide a tool for the structured design of the communication protocol, any formal specification first has to provide a convenient way to represent and inter-relate various layers of protocols in the hierarchical structure.

Secondly, it should permit a rich (complex) protocol to be described with notations that are tractable, and should also provide a relatively straightforward analysis for correctness. The correctness proof for a complicated protocol might be intractably difficult, but the formal model should enable the designer to validate some "important" properties of the protocol (such as deadlock freeness, proper termination etc.). Thirdly, it should provide a systematic way of implementing protocols and should ease the debugging, modification and future extension of protocols.

The last, and perhaps most important, goal for the formal model is to provide clear and concise documentation of the protocol for both the network users and the protocol designers. Considering the amount of effort that has gone into protocol design, there has been relatively little documentation published. Good modeling documentation can provide a concise and precise way of conveying the designer's ideas and the features of protocols to other people.

3.4 Choice of Models

During the past several years, formal models such as UCLA graphs [Postel74], finite state automata (FSA) [Bochmann77b, 78, Danthine78, Gouda76a, 76b, Sundstrom77, SNA78], type-3 grammars [Harangozo77,78], Petri nets [Merlin76,79] and high level programming languages [Bochmann75, Hajek78, Stenning76] have been used for the specification of communication protocols. However, a general and strong theoretical basis for protocol design in the context of software engineering has not yet been established.

3.4.1 Types of Models

There are three basic types of models that can be used for the description of protocols: the local approach, the global approach and the combined approach.

In the local approach, each communication entity is described by one automaton or one program. In this approach, we do not worry about global synchronization and interactions between communicating entities. Since the local verification does not take the global interactions of the communicating entities into consideration (as we can see from the detailed discussion of the local and global validations in Chapter 4 and 5), it is unable to check timing problems during communication. However, the model is easier to construct and directly related to implementation, since it describes the action of the entity under consideration.

The global approach uses only one automaton to describe the global interactions of two communicating entities and their channel. It is designed to show the precise global interactions between the entities and is thus more complicated than the local approach. We can check the interactions for possible existence of deadlock, infinite idle looping, improper termination and other error conditions. However, global modeling is harder to construct and is less related to the actual implementation than local modeling.

The model of the combined approach uses both the local and global approaches. The local approach is first used to describe each communication entity and the channel between them. Global validation techniques are then applied to combine the individual automaton of two communicating entities and the channel to form a global automaton. The composite automaton of both entities and the channel together is an automaton whose transitions can be analyzed to check for global error conditions.

3.4.2 Features of Different Models

The formal techniques may be classified as either state transition techniques or programming language techniques.

State transition techniques are natural to model the action sequences during communication and allow for more straightforward reachability analysis to explore protocol syntax errors. Programming language techniques make it easier to model variables and allow for more straightforward implementation of the models.

State transition techniques show promising results when applied to protocol verification. Many errors in published protocols have been uncovered by this analysis. The difficulties are state explosion for complex protocols and a more restrictive view of verification (i.e., they only perform reachability analysis). Programming language techniques provide the ability to deal with the full range of correctness problems. The difficulty is that program proving is very hard to apply. General program proof techniques do not include the consideration of message transport problems, concurrency and synchronization.

Hybrid techniques have been attempted to combine the advantages of both techniques [Bochmann77a, 77b, Danthine78]. Merlin [79] and Sunshine[79] have outlined the formal techniques and results to date; Sunshine has also provided a wider survey of various models in [Sunshine78a].

As we mentioned in Chapter 1, the TG model use the local validation technique to reveal program structure errors by analyzing the TG grammar rules, and use the global validation techniques to reveal protocol syntax errors by state transition techniques.

3.4.3 Features of Transmission Grammar Model

This dissertation uses the combined approach to protocol modeling and uses state transition techniques for protocol validation. The transmission grammar model may be used for botn local implementation and global validation of protocols. The simple grammar structure allows state transition techniques to be used for reachability analysis.

Since there is no graphical high-level language for the graphical models (FSA, Petri nets, and UCLA graphs), the linear property of the transmission grammar model is crucial to facilitate the software auto-implementation directly from a validation TG. From formal language theory, we understand that a type-3 grammar (FSA type) is a special case of a type-2 grammar (CFG). The reason for choosing a context-free grammar (BNF-type) in the TG model to describe action sequences is that it is much more concise and powerful than a type-3 grammar model. The property of conciseness will result in a tremendous reduction of dimensionality as compared to the FSA-type model. The power of the CFG model will enable the Reader-Writer type problems [Chen75] (whicn are hard to describe by a type-3 grammar model or the Petri net model) to be easily represented by applying the recursive (push-down store) method. We can integrate different layers of protocols together by mapping the terminals of a higher-level TG to non-terminals of its lower-level TG (by the substitution operation). This integration would give the designer the flexibility of using different degrees of complexity in his design. For several concurrently operated TGs in an entity, the arbitrary shuffle operation can not only be used to ease the TG construction complexity, but also get an exact and adequate description of all the possible, valid action sequences. If we write the model in a validation form, the TG becomes the validation automaton. The validation automaton can automate the global validation process and will be covered in Chapter 5.

3.5 The Transmission Grammar Model

This dissertation presents a complete protocol construction technique based on a grammar model. We use the combined approach which allows for complex reachability analysis. In this section, we motivate the need for the transmission grammar model and present several protocol specification examples.

3.5.1 Motivation

Toward a Syntactic Description of Protocols

In this section, we will illustrate the correspondence between programming languages and protocols. From each communication entity's point of view, the protocol simply consists of a set of rules that can be used to define the action sequences during the phase of communication. We will investigate how to represent the actions of each communication entity when we outline the generalized TG model. Some examples of actions in an action sequence are: (1) put an ASCII character (or packet, message, file, etc., depending on the layers of protocol description) into an output buffer; (2) get an EBCDIC character from an input buffer; (3) set the time; (4) time out; (5) erase the buffer; or (6) send the contents of an output buffer to a lower layer.

There are surprisingly many similarities between programming language compilers [Gries71] and communication protocol software (see Table 3). Here, the communication system is rather like computer hardware. On top of the computer hardware, we can design the machine language, internal forms and high level languages. In a similar way, we design hardwired layer protocols from the primitive actions of communication equipment. On top of the communication system, we can design other layers of protocols which are more convenient and already have safeguards against various kinds of failure by their lower layer protocols.

Table 3 Comparisons between programming language compilers and communication network protocols

PROGRAMMING LANGUAGE COMPILERS	COMMUNICATION NETWORK PROTOCOLS
BNF defines languages	Transmission Grammar defines protocols
each syntactically correct program	each valid and recoverable communication activity sequence
the end of a program	the end of a transmission sequence (as defined by the transmission grammar)
each token (word) input to the compiler	each action (send,receive, timeout etc.) that occurred during communication
program syntax error	unrecoverable error for message transfer
programs in the form of: 1) high level languages 2) internal forms 3) machine language	action sequences in the form of: 1) higher layer TG terminals 2) lower layer TG terminals 3) hardwired layer TG terminals

If we take all these defined actions as terminals (or words in a sentence of a language), then each protocol simply defines a language which has all the valid action sequences as its sentences. Just as the Backus-Naur Form (a context-free grammar) is used to describe the syntax of a programming language, a transmission grammar can be used to describe the communication protocol between communicating entities. Each communication entity is regulated by a set of local protocol rules and the rules are represented by a transmission grammar.

We can then see a protocol as a set of grammar rules, each of which enables a communication entity to produce or generate communication action sequences (sentences defined by the grammar) to communicate with other entities. A communication entity may know how to do the communication actions well but still may not be able to communicate with other entities. It must also know how to assemble actions (words) into valid action sequences (sentences) by the protocol grammar rules in order to communicate. The rules also enable the communication entity to understand the receiving action sequences of other entities regulated by compatible protocols. We say the protocols for a set of communication entities are correct if they follow the designer's original specification and if they will result in a proper dialogue among the communicating entities. One purpose of grammatical description of this set of rules is to present in a precise and explicit way those facets of the protocol that each communicating entity must follow.

Let us now take into consideration the hierarchical property of protocols and regard the TG for each entity in a layer as a black box. (We will investigate how to interrelate (integrate) the TGs of the different layers in Chapter 4.) If a particular action sequence S can achieve a correct communication activity in one layer, then the syntactic description of protocol L for that layer must be such that it can generate S as a sentence. Similarily, if S is not considered as a proper action sequence for communication, then the description must fail to generate S and must also inform the higher layer protocol entity as to the reason for the failure. To summarize these requirements, it is convenient to think of the syntactic description of L as a black box that can receive any action sequence S as input and output its validity, as shown in Fig. 1Ø.

```
                           S grammatical
                             |-------->
                             | S is a syntactically
                             | valid action sequence
          |--------------|   |
    S     | syntactic    |   |
    ---->| description  |--o
          | of protocol L |   |
          |--------------|   |
           Trans. Grammar    | report ungrammaticality to
                             | higher layer, or go to
                             | error handling routines
                             |--------->
                           S ungrammatical
```

Fig. 10 Transmission Grammar as a black box

One thing worth noting is that just as the same meaning can be conveyed by a number of different sentences when we speak English, or the same algorithm can be coded in a number of different ways when we write a program, correct communication can be achieved by a number of different communication action sequences when two entities communicate. This is because we can go through various channels and various error recovery procedures at various layers for the same communication purpose. We say a grammatical description of a particular protocol is adequate if the language it defines includes all the sentences of action sequences that achieve correct communication.

TG Model Description Notations and Conventions

The context-free grammar (BNF type) is chosen as the TG modeling tool to describe an entity's action sequences. The modeling of grammars, FSA and push-down automata in this dissertation will mostly follow the notational conventions in [Hopcroft69].

A transmission grammar G is denoted by G = (Vn,Vt,P,<S>), where

(a) Vn is a set of non-terminal action symbols always enclosed by < >;
(b) Vt is a set of terminal action symbols;
(c) Every element in P, the set of production rules, is of the form <A> ::= x, where <A> e Vn and x e (Vn U Vt)*, and it represents a syntactical rule for G; and
(d) <S> (e Vn) is the starting symbol.

In the grammar which specifies the syntax, the following notational conventions are used. Nonterminals (names of syntactic classes) are written in lowercase letters and enclosed by the brackets "< >". The brackets "[ ]" are used to enclose a sequence of one or more items, all of which must occur exactly once. The brackets "( )" are used to enclose a sequence of one or more optional items, that is, they all occur exactly once or not at all. The brackets "//" are often used to enclose a terminal action written in uppercase letters. The symbol "*" is used to indicate that the preceding item or bracketed sequence of items may be repeated an indefinite number of times in succession.

The production rules can also be written in the form:

```
  nonterminal ::= alternate-1 | alternate-2 | .... |
                  alternate-n     (n>0)
```

An alternate may be any sequence of terminals, nonterminals and bracketed sets of alternates. A context-free grammar can represent all the distinct but equally valid action sequences by the property that every left part non-terminal of a production rule can have several alternative right part strings. For each production rule of the action grammar, we may arrange its alternates in the desired order of processing. This arrangement enables us to represent action processing in order of its priority.

3.5.2 Grammatical Properties of Protocol

For the TG model, we use the <u>local</u> approach in the following sense: instead of having a single set of rules (like Petri net models) governing the interactions between several communicating entities, we define a set of TG rules within _each_ entity to regulate the interactions between these entities. Each entity, therefore, corresponds to a transmission grammar which defines its protocol program. The entity's TG may be decomposed into inner-layered component TGs or VIP TGs to simplify the design. We can use the transmission grammar to define each of the components and VIPs of the communication entity in a layer in a hierarchical way, and then _integrate_ the TGs of the components and VIPs for the entity.

Fig. 11 shows typical communication entities in an ARPANET host system [Feinler78]. Each entity in the black box corresponds to a TG and interacts with neighboring entities through its external channels.

```
    USER1  USER2        USER3 USER4        USER
     : :    : :          : :   : :         LEVEL
---:-:---:-:-----------:-:---:-:-------
:   -----  -----       -----  -----      :
:   : P1:  : P2:  ...  : P3:  : P4:   ...: PROCESS
:   -----  -----       -----  -----      : LEVEL
:   \ \      / /       \ \      / /      :
:    --------            --------        :
:    : NCP1 :            : NCP2 :   ...  : HOST
:    --------            --------        : LEVEL
:          \ \          / /              :
:           -----------                  :
:           :   IMP   :                  : IMP
:           -----------                  : LEVEL
:            / /      \ \                :
:      -----------  -----------          :
:      : MODEM 1 :  : MODEM 2 :     ...  : HARDWIRED
:      -----------  -----------          : LEVEL
--------/-/----------------\-\---------
      / /                   \ \
 TO NEIGHBORING HOST      TO NEIGHBORING HOST
```

Fig. 11 Decomposed entities of a host system in ARPANET

With the local approach, it is easier to specify protocol TG(s) for each communication entity, but the local protocol specifications for all the entities in a network system is not complete without considering the entity's interactions with other entities in the system through the _global_ approach.

TG Design Methodologies Overview

In the following example [Teng78a], we illustrate the grammatical properties of protocols by the step-wise design of some typical TGs of the communication processor level in a packet switching network. I/O interrupt handling, priority of action, buffer requests/allocations, refinement (internal detail) of each action, and message format structure are excluded from consideration for the purpose of simplicity (although these considerations can be easily added to the TG model).

In general each entity at the communication processor level (for example the interface message processor (IMP) in the ARPANET [Heart7Ø]) mainly consists of two validation independent parts (VIPs), i.e. sender rules and receiver rules, which are processed concurrently. We first define a set of actions in both parts, and then design the TG for the sender and receiver VIPs. The TG integration techniques are discussed later in Chapter 4.

Actions of the protocol TG:

Messages and Buffers:

ACK:	acknowledgment packet
Ui:	packet unit i
Pb:	Packet buffer
ACKb:	ACK buffer
Rb:	Reassembly buffer

Sender:

Gs:	"Generate Ui and put in Pb"
Sn:	"Send Ui Never sent before"
As:	"find ACK for an outstanding Ui in ACKb"
Fs:	"Free Pb, ACKb space for Ui"

Receiver:

Rl:	"Receive Ui in Pb destined here"
Cr:	"Consume Ui in Pb and put it in Rb"
Fu:	"Free Pb space unit for Ui"
Ar:	"transmit ACK out to lower level channel"
Sn:	"Send all Ui's in Rb to host"
Fr:	"Free Receive buffer Rb"

Sender TG design:

Let's first consider the following language L, where

L = { w ! w e {Gs,Sn,As,Fs}* and (# of Gs's in w = # of Sn's in w = # of As's in w = # of Fs's in w) and (for all x,y such that w=xy then # of Gs's in x $\geq$ # of Sn's in x $\geq$ # of As's in x $\geq$ # of Fs's in x) }

We can see that L consists of all the sentences that result in correct sending activities and each correct sending action sequence has to be a sentence defined by L. In order to investigate the linguistic nature of protocol transmission grammar, we construct several different types of grammars, generating sentences that belong to L.

Case 1: single outstanding send

Single outstanding send requires each sender to wait until the previous send is completed. This is the simplest type of protocol and we can define a FSA or a type-3 grammar G1 for it. We write G1 in BNF format as:

G1 = ({<S>},{Gs,Sn,As,Fs},P,<S>), where P consists of:
<S> ::= GsSnAsFs<S> ¦ ϵ

Obviously, L(G1) is a proper subset of L, so TG G1 defines a set of correct send action sequences as its sentences.

<u>Case 2: multiple outstanding sends</u>

The TG G1 defined in case 1 is not efficient because packets cannot be sent out until the previous send is acknowledged. Some powerful protocols have a feature which allows several sending packets to be outstanding. This type of protocol cannot be defined by a type-3 grammar but by a type-2 grammar (context-free grammar, or CFG) G2:

G2 = ({<S>},{Gs,Sn,As,Fs},P,<S>), where P consists of:
<S> ::= GsSn<S>AsFs<S> ¦ ϵ

By induction on the length of any sentence w in L(G2), we can prove that: (the # of Gs's (or Sn's) in w = the # of As's (or Fs's) in w) and ($\forall$ x,y such that w=xy the # of Gs's (or Sn's) in x $\geq$ # of As's (or Fs's) in x). Therefore, L(G2) defines a set of correct sending sequences. Notice that L(G1) is a proper subset of L(G2) while L(G2) is a proper subset of L.

Case 3: TG for L

G1 and G2 only define two proper subsets of L. In order to define a TG for L to represent the most generalized (fully multiplexing) protocol,we need to use a type-1 grammar (context-sensitive grammar, or CSG) G3:

```
  G3  =  (  {<S>,<Gs>,<Sn>,<As>,<Fs>},{Gs,Sn,As,Fs},P,<S>),
where P consists of:

<S> ::= <Gs><Sn><As><Fs><S> | ∈
<X><Gs> ::= <Gs><X>  ∀ <X> ∈ {<Sn>,<As>,<Fs>}
<Y><Sn> ::= <Sn><Y>  ∀ <Y> ∈ {<As>,<Fs>}
<Z><As> ::= <As><Z>  ∀ <Z> ∈ {<Fs>}
<Gs> ::= Gs
<Sn> ::= Sn
<As> ::= As
<Fs> ::= Fs
```

G3 first generates a sentence of L(G1), and then applies the production rules to do certain types of interchanges for the sentence of L(G1) to get a valid permutation that satisfies the desired property of L; i.e., ∀ w ∈ L(G3), and ∀ x,y such that w = xy, (# of Gs's in x ≥ # of Sn's in x ≥ # of As's in x ≥ # of Fs's in x). Therefore, the type-1 grammar G3 defined above satisfies the condition that L(G3) = L.

Although the CSG is perhaps the best and simplest way of representing this generalized sender protocol, the complexity of relating a CSG to protocol software implementation is increased as compared with CFG. This is why we have decided to choose context-free grammar as the model and leave some context sensitive features in the semantics part.

Case 4: Erroneous channel and routing

Grammars G1,G2 and G3 mentioned above work satisfactorily for a perfectly error-free channel. Extension to the unreliable (erroneous) channel and addition of flow control actions are also straightforward. We first introduce the following actions:

To: "Time Out for Ui"
Ss: "Send Ui out a lower level channel"
Rt: "Reset the time for Ui"
St: "Set the time for Ui"

We can modify G1 to get

G11 = ({<S>},{Gs,Sn,As,Fs,To,Ss,Rt,St},P,<S>), where P consists of:

<S> ::= GsSnSsSt(ToSsRtSt)*AsRtFs<S> | ϵ

We can also modify G2 to get

GI2 = ((<S>,<Ret>),(Gs,Sn,As,Fs,To,Ss,Rt,St),P,<S>),
where P consists of:

```
<S> ::= GsSnSsSt<S>[<Ret><S>AsRtFs<Ret>]<S>  |  ε
<Ret> ::= (ToSsRtSt)*
```

The modification of G3 is also straightforward and not shown here.

Receiver TG design:
==================

The protocol for the receiver part is in general less complicated than the sender part and we will only list a type-3 grammar Gr (in BNF format) for the case of single outstanding send:

Gr = ((<R>),(Rl,Ar,Cr,Fu,Sh,Fr),P,<R>), where P consists of:

```
<R> ::= [RlArCrFu]*ShFr<R> | ε
```

3.5.3 TG Specification of TCP

We have demonstrated the grammatical properties of simple protocols in the previous section. In this section, we demonstrate a TG specification of a more complicated protocol used in the connection management part of the ARPANET Transmission Control Program (TCP).

Fig. 12 is a TG specification of the connection management protocols of TCP. The specification has been derived by following the step-wise method presented in the previous section. The detailed description of the protocol program can be found in [TCP79]. RSND represents a send action with retransmission. RCV represents a receive action. The specification shows protocol structures and state transitions in a concise and clear form.

```
<closed> ::= <connecting> <exchange> <closed>
          | "RCV.<nonrst-segment>" "RSND.<rst>" <closed>
          | "RCV.<rst>" <closed>
  <connecting> ::= <active.open> <wait.synack>
                 | <passive.open> <listen>
    <active.open> ::= "RCV.<a.open>" "create tcb"
                      "RSND.<syn>"
    <wait.synack> ::= "RCV.<synack>" "RSND.<ack>"
                    | "RCV.<syn>" "clear" "RSND.<rst>"
                    | <break.connect> | <refused> <connecting>
      <break.connect> ::= "RCV.<close>" "delete tcb"
                          <connecting>
      <refused> ::= "RCV.<rst>"
    <passive.open> ::= "RCV.<p.open>" "creat tcb"
    <listen> ::= <break.connect> | "RCV.<syn>"
                 "RSND.<synack>" <listen 3-way>
               | "RCV.<send>" "RSND.<syn>" <wait.synack>
      <listen.3-way> ::= "RCV.<ack>" | <refused> <listen>
                       | "RCV..<syn>" "RSND.<rst>"
                         "clear" <listen>
  <exchange> ::= "send.receive" <exchange> | <disconnect>
    <disconnect> ::= <end send> | <end receive>
      <end.send> ::= "RCV.<close>" [<send.segment>]*
                     "RSND.<fin>" <wait.finack>
      <end.receive> ::= "RCV.<fin>" "RSND.<ack>"
                <wait.close> | "RCV.<finack>" "RSND.<ack>"
        <wait.finack> ::= <receive> <wait.finack>
                        | "RCV.<fin>" "RSND.<ack>" <fin.3-way>
        <wait.close> ::= <exchange> <wait.close>
                       | "RCV.<close>" "RSND.<fin>" <fin.3-way>
          <fin.3-way> ::= "RCV.<ack>" "delete.tcb"
                        | "timeout" "abort"
```

Fig. 12 Connection management protocols of TCP

3.6 The Generalized TG Model

The action of a transmission grammar often has transmission message units as its operands, and each transmission message unit often consists of several variables or constants. For example, the action of "sending a packet out through the send port channel 3" has the packet as its operand, and the packet has the packet number, source, destination, etc., as its variables or constants. For many higher-level protocols, the message syntax could be so complicated as to make the BNF-type syntax description a necessity (see the BNF message syntax specifications for the File Transfer Protocol and Mail Protocol in [Feinler78]).

The TG model, outlined in [Teng78a], only specifies the action sequences of protocols. To represent the interrelationship of hierarchical message structures in the model, the _generalized TG model_ should have two grammar specifications: 1) message grammar, and 2) action grammar. The addition of a message grammar to the TG model is very natural because both action and message grammars have the same BNF format.

The addition of the message grammar to the TG model enables us to: (1) represent the hierarchical structure of message format; (2) represent actions in the action grammar with message grammar non-terminals as their operands, thereby eliminating the overhead of representing actions in the action grammar at the bit or character level; and (3) implement the automatic syntax parsing and error handling. An example of a message grammar is shown in Fig. 13. It is the message format rules for the network control program (NCP) in ARPANET.

```
<trans>  ::=   [<message>]*
  <message>  ::=  <regular msg> ¦ <control msg> ¦ RFNM
    <regular msg>  ::=  <source message> ¦ <dest message>
      <source message>  ::=  <send header> <text>
      <dest message>  ::=  <receive header> <text>
        <send header>  ::=  <send leader> <byte size>
                            <byte count>
        <receive header>  ::=  <receive leader> <byte size>
                               <byte count>
          <send leader>  ::=  "dest" "msg type" <link no>
          <receive leader>  ::=  "source" "msg type"
                                 <link no>
    <control msg>  ::=  <leader> "byte size" "byte count"
                        [<control command>]*
      <leader>  ::=  <send leader> ¦ <receiver leader>
      <control command>  ::=  <rts> ¦ <str> ¦ <cls>
                              ¦ <all> ¦<gvb> ¦<ret>
        <rts>  ::=  RTS <socket> <socket> "link no"
        <str>  ::=  STR <socket> <socket> "link no"
        <cls>  ::=  CLS <socket> <socket>
        <all>  ::=  ALL <link no> "msg space" "bit space"
        <gvb>  ::=  GVB <link no> "msg fraction"
                    "bit fraction"
        <ret>  ::=  RET <link no> "msg space" "bit space"
          <socket>  ::=  <user no> <host no>
            <user no> ::=  "24-BIT  UNIQUELY IDENTIFY EACH
                            USER"
            <host no> ::=  "8-BIT OF  HOST IDENTIFY NO"
```

Fig. 13 Message grammar of NCP in ARPANET

3.7 Properties of TG Model

The TG model has the following properties:

(1) It provides for a precise and concise description and documentation for both message and action rules of protocols. It is the basic reference and guide for protocol program designers and communication protocol users.

(2) It permits complex protocols to be described with descriptive notations which are tractable. Its linear representation (grammar rules) has the flexibility to describe some context-free and context-sensitive properties of protocols which are not possible in models using finite state automata.

The model uses terminals and non-terminals in the grammar, and the terminals at a higher level can be easily mapped to the corresponding non-terminal at a lower level. The model has the flexibility of describing a complicated protocol with different degrees of details for validation and implementation.

(3) It allows the protocol designer not only to specify protocol program modules in a well-structured manner, but to keep the specified design structure (context free grammar) so simple that automatic validation and implementation can be easily carried out (which will be covered in Chapters 4, 5 and 6). Since there does not exist any graphic diagram interpreter for the direct processing of graphic models (such as FSA, Petri nets and UCLA graphs), the linear and nongraphic property of the transmission grammar model is also crucial to direct analysis and automatic implementation of protocols (which also will be covered in Chapters 4, 5 and 6).

CHAPTER 4

PROTOCOL DECOMPOSITION AND VERIFICATION TECHNIQUES

4.1 Introduction

Because of the complexity of protocol verification, we shall cover the topic in two chapters. This chapter defines general validation terms, outlines the overall approach and discusses local validation techniques. The global validation techniques will be covered in the next chapter.

A difficult step in protocol design is to write down the TG specification for a given communication entity of a protocol layer. The hierarchical decomposition of the entity into several components is one way to reduce the complexity. The lateral decomposition of the entity into independent parts is also feasible if the parts are mostly validation independent. These decompositions allow us to reduce the complexity of protocol validation and design. The decomposed components and parts can be integrated for either implementation or final modifications of the validation modeling. Three integration operations for TGs are introduced and investigated in this chapter.

TG local validation techniques are also introduced to detect the syntactic error of protocols. A validation system has been constructed to automatically format a TG and validate the "syntax" of the input TG. It demonstrates the feasibility of building a more powerful and general validation system.

4.2 Definition of Terms

In this section, we formally define terms and notations for future discussion of protocols. The notation basically follows the convention used for programming languages in [Gries71].

Def.1 A symbol can be either a message symbol or an action symbol. A transmit symbol is an action symbol, which consists of actions of sending and receiving messages, and may have message symbols as its operands.

Def.2 A message string is a finite sequence of message symbols. An action string is a finite sequence of action symbols. An idle string is a finite sequence of action symbols such that the string contains no effective communication symbol.

Def.3 An action rule is an ordered pair (<u>,x) where <u> is an action symbol and x is an action string. It is written as <u> ::= x .
A message format rule is an ordered pair (<m>,n) where <m> is a message symbol and n is a message string.

We shall concern ourselves principally with action rules and will focus on discussing them throughout this chapter.

Def.4 A transmission grammar (TG) is a finite non-empty set of action and message rules which is used to define a protocol. A protocol is a set of agreements by which entities exchange information.

A step-wise refinement approach is necessary for the design of a sophisticated network protocol. We introduce the idea of non-terminal and terminal symbols to formalize this approach.

Def.5 The non-terminal action symbols of a TG are the set of symbols occurring as left parts of the TG rules. Each non-terminal is enclosed with the notation < >.

Def.6 The terminal action symbols are the set of symbols in a TG which are not non-terminals.

In the step-wise refinement definition of a TG, we can substitute a non-terminal for terminals and make detailed specifications for the new non-terminal.

Def.7 The starting action symbol of a TG is one of the non-terminals and the starting symbol is usually obvious from the TG.

Def.8 Let G be a grammar; we say that the string w is a direct derivation of the string v, written v => w, if we can write

v = x<u>y, w = xuy

for some strings x and y, where <u> ::= u is a rule of G. We say that w is a derivation of the string v, written v =>+ w, if there exists a sequence of direct derivations

v => u1 => u2 =>..... => un = w (n>0). Finally we write v =>* w if v =>+ w or v = w.

Def.9 Let G be a transmission grammar for an entity. A string x is called a sentential form if x is derivable from the starting symbol <st> (i.e., if <st> =>* x). We use the substring y of x to represent the state of the entity, where <st> => x = wy, w consists only of terminals and the head symbol of y is a non-terminal. If y is ϵ then we say the state is a terminating state of the entity and we say the tail symbol of w is a terminating symbol.

From the definition of the state of an entity we know that the substring y will determine all the possible derivations from the sentential form x. Therefore, all future state transitions are independent of the past state transitions.

Def.1Ø A sentence is a string x derivable from the starting symbol consisting only of terminal symbols. A proper sentence should always have a proper terminating symbol as tail symbol.

This string may consist of a sequence of actions of one transmission transaction (send or receive), or a sequence of actions of one connection opening transaction.

Def.11 A TG is well structured if and only if it is reduced and generates only proper sentences. A TG is reduced if and only if every non-terminal <u> of the TG is such that (1) <u> is reachable from the starting symbol and (2) we can derive a terminal string from <u>.

A well structured TG preserves several nice properties, such as freedom from deadlock, as we will discuss in Section 4.5.2.

We know a (binary) relation on a set is any property that either holds or does not hold for any two ordered symbols of the set. We usually represent relations over the set in a computer by a Boolean matrix. The following relations and their transitive closures are useful when we discuss the algorithms for protocol validation.

Def.12 For a relation R, the transitive closure R+ of R can be defined as a R+ b, if and only if a R!n b for some n>Ø. (We use R!n to represent the repetition of relation R n times .)

For a non-terminal <u> and a symbol S of a TG, we can define the relation FIRST, LAST and WITHIN over the finite vocabulary Vn U Vt of the transmission grammar.

Def.13 <u> FIRST S if and only if there is a rule <u> ::= S...

Note that the three dots "..." represent a (perhaps empty) string which at this point does not interest us. We then have by the definition of transitive closure:

<u>Def.14</u> <u> FIRST+ S if and only if <u> =>+ S...

<u>Def.15</u> <u> LAST S if and only if <u> ::= ...S

<u>Def.16</u> <u> LAST+ S if and only if <u> =>+ ...S

<u>Def.17</u> <u> WITHIN S if and only if <u> ::= ...S...

<u>Def.18</u> <u> WITHIN+ S if and only if <u> =>+ ...S...

These relations will be used for protocol validation when we discuss TG validation techniques in Section 4.5.

<u>EXAMPLE 4.1</u>. To illustrate the above relations, we list some rules.

```
<TG> ::= <receive> <TG>  | <send> <TG>

  <receive> ::= "receive packet" | "receive message"
  <send> ::= <receive> <send> | "send packet" <wait ACK>
           | "send message to host" | ε

    <wait ACK> ::= <receive> <wait ACK> | "acknowledgment"
             | "retransmit" <wait ACK> | <send> <wait ACK>
```

We then have <TG> FIRST <receive>, <TG> LAST <TG>, <TG> FIRST+ "send packet", <send> WITHIN+ "retransmit" and <send> LAST+ "acknowledgment".

4.3 TG Decomposition Techniques

A difficult task for the design and verification of a complex protocol is to write down the formal specification. A general strategy to reduce design complexity is to decompose the protocol into smaller layered components and/or validation independent parts. This strategy has been discussed for the decomposition of communication entities in Section 3.2.

The TG decompositions can be either hierarchical or lateral. The layered approach to protocol design is an example of hierarchical decomposition to decompose a host computer system into several hierarchically related sub-systems. Further decompositions within a sub-system (layer) - either hierarchical or lateral - may also be possible. The hierarchical decomposition of a protocol decomposes its TG into hierarchically related components. It corresponds closely to the step-wise refinement concept in structured programming. An example of hierarchical decomposition is a protocol component for character handling (such as byte stuffing in SDLC) within another component for message handling. The lateral decomposition of protocols decomposes a TG into validation independent parts (VIP). (These are also called logically independent parts in [Teng78a,78b].) Each VIP of a sub-system can be validated independently of other VIPs of the sub-system even though the VIPs may be dependent on each other in the actual implementation. An example of lateral decomposition is the sending and receiving part of a protocol. Lateral decomposition may also correspond to alternate parts of a TG rule.

After designing the TGs components and VIPs, we can integrate them together by substitution and shuffle operations. The shuffle operation is used for the integration of VIPs. The substitution operation is used for the integration of hierarchically related components; the relationships of TG decomposition and integration are shown in Fig. 14. Notice that local VIPs within an entity can be viewed as concurrent processes in the field of concurrent programming. VIP processes often share common variables such as I/O channels and buffer areas, and need some synchronization mechanisms to prevent race conditions. However, VIPs in TG notations often describe a design level action as an indivisible unit, and race conditions can generally be prevented by viewing design action as a <u>critical section</u>.

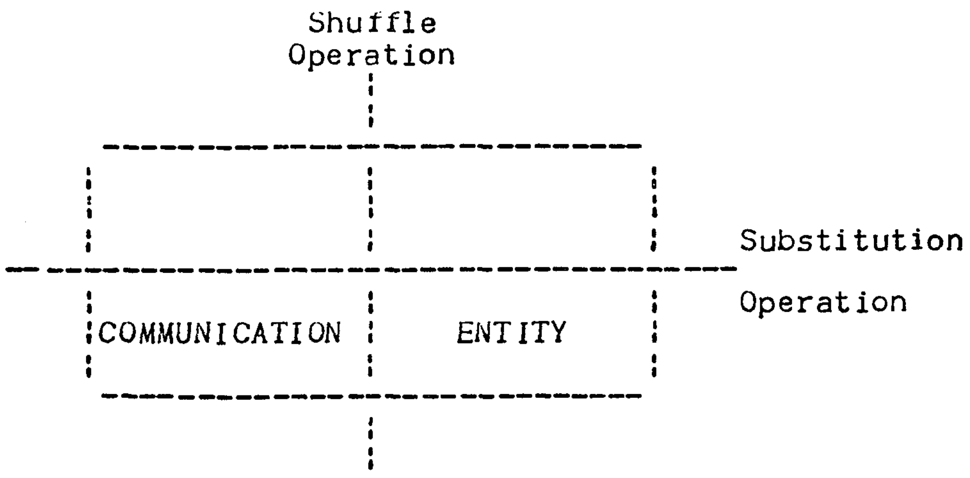


Fig. 14 Decomposition and integration relationships

4.4 TG Integration Techniques

Three operations are available to integrate TGs: the arbitrary shuffle, the restricted shuffle and the substitution operation.

4.4.1 Arbitrary Shuffle (//) Operation

The arbitrary shuffle operation (similar to the shuffle production of [Eilenberg74]) can be used for the integration of several validation independent TGs. In this section, it is used to integrate local VIPs within an entity. It will be used to integrate global validation automata in the next chapter. The integration allows us to check race conditions between VIPs and to ensure the integrity of the channels and buffers shared by the VIPs for actual implementations.

Def.19 Let V1 and V2 be two sets of terminals. Given A, a subset of $V1^*$; B, a subset of $V2^*$; and $V1 \cap V2 = \phi$; then the arbitrary shuffle product of A and B (A // B) is a subset of $(V1 \cup V2)^*$ and consists of all sentences $w \in (V1 \cup V2)^*$ such that $w1 \in A$, $w2 \in B$, and w1 is derived by erasing all the symbols of V2 in w, w2 is derived by erasing all the symbols of V1 in w. The // operation is called the arbitrary shuffle operation or local shuffle operation.

The // operation of two languages can be imagined as the shuffling of their respective sentences. The shuffling of two sentences is similar to an arbitrary shuffling of two decks of cards (each deck of cards corresponds to a sentence of a distinct language). The original order of the individual decks is maintained in the combined deck. That is, if the original sequence of the first deck is 1,2, and that of the second deck is A,B, then the sequence of the combined deck could be: 1,2,A,B; 1,A,B,2; 1,A,2,B; A,B,1,2; A,1,B,2; or A,1,2,B. This arbitrary shuffle operation is very useful when we design the TGs of several concurrently operating parts and wish to integrate the TGs.

For example, (L(GI2) // L(Gr)) contains the combined action sequences of the sender and receiver parts, and still keeps the original sequence of the individual parts. We can thus use the integrated grammar to represent the communication-processor level protocol.

The following algorithms can be used to derive the arbitrary shuffle product of various types of languages.

Theorem 1 If V1 and V2 are two sets of terminals such that $V1 \cap V2 = \Phi$, then any regular set R1 of V1* is closed under the arbitrary shuffle operation with another regular set R2 of V2*.

The steps required to construct the shuffle product of R1 and R2 are given in Algorithm 1.

Algorithm 1

Let M1 = (K1,V1,f1,pØ,F1), and
M2 = (K2,V2,f2,qØ,F2)

be the finite state automata that recognize R1 and R2 respectively, and let $V1 \cap V2 = \Phi$. Then the finite state automaton

M3 = (K1 x K2,V1 U V2,f3,[pØ,qØ],F1 x F2) recognizes

R1//R2, where f3 is defined as follows:

$\forall$qi e K1, f3([p,qi],a) = [p′,qi]

whenever f1(p,a) = p′ in M1 and

$\forall$pi e K2, f3([pi,q],b) = [pi,q′]

whenever f2(q,b) = q′ in M2.

It is clear from the construction that M3 recognizes R1//R2.

Theorem 2 If V1 and V2 are two sets of terminals such that $V1 \cap V2 = \Phi$, then the class of CFLs of V1* is closed under the arbitrary shuffle operation with a regular set of V2*.

The steps required to construct the shuffle product of a CFL and a regular set are given in Algorithm 2.

Algorithm 2
Let L be a CFL and R a regular set. Let

P1 = (Kp,V1,T,f1,pØ,ZØ,Fp)

be a non-deterministic push-down automaton that recognizes L; and

A = (Ka,V2,f2,qØ,Fa)

be a deterministic finite state automaton that recognizes R. Then the non-deterministic pda

P2 = (Kp x Ka,V1 U V2,T,f3,[pØ,qØ],ZØ,Fp x Fa)
recognizes L // R, where f3 is defined as follows:

$\forall$ qi e Ka, f3([p,qi],a,Z) = ([p′,qi],x)
whenever f1(p,a,Z) = (p′,x) in P1; and

$\forall$ pi e Kp, Z e T, f3([pi,q],b,Z) = ([pi,q′],Z)
whenever f2(q,b) = q′ in A.

Corollary The arbitrary shuffle operation is reflexive and associative.

The corollary is clear from Algorithm 2.

EXAMPLE 4.2 To show an example of the shuffle, we construct a push-down automaton P2 to recognize L(GI2) // L(Gr) by the following three steps:

(1) First, we construct a push-down automaton P1 that recognizes L(GI2)

P1 = (Kp,V1,T,f1,PØ,ZØ,Fp)

where Kp = { pØ,p1,p2,p3,p4,p5 }
V1 = { Gs,Sn,Ts,Rs,As,Fs,Rt,St }
T = { ZØ,X }
Fp = { p9 }

f1 is defined as follows:

```
f1  ( pØ,ϵ,ZØ )   = ( p9,ϵ )
f1  ( p9,ϵ,ϵ )    = ( pØ,ZØ )
f1  ( p5,ϵ,ZØ )   = ( pØ,ZØ )
f1  ( pØ,Gs,ZØ )  = ( p1,XZØ )
f1  ( pØ,Gs,X )   = ( p1,XX )
f1  ( p1,Sn,X )   = ( p2,X )
f1  ( p2,Ss,X )   = ( p3,X )
f1  ( p3,St,X )   = ( pØ,X3)
f1  ( pØ,As,X )   = ( p4,ϵ )
f1  ( p4,Rt,X )   = ( p5,X )
f1  ( p4,Rt,ZØ )  = ( p5,ZØ )
f1  ( p5,Fs,X )   = ( pØ,X )
f1  ( p5,Fs,ZØ )  = ( pØ,ZØ )
f1  ( pØ,IØ,X )   = ( p6,X )
f1  ( p6,Rt,X )   = ( p7,X )
f1  ( p7,Ss,X )   = ( p8,X )
f1  ( p8,St,X )   = ( pØ,X )
```

The automaton is also shown in Fig. 15.

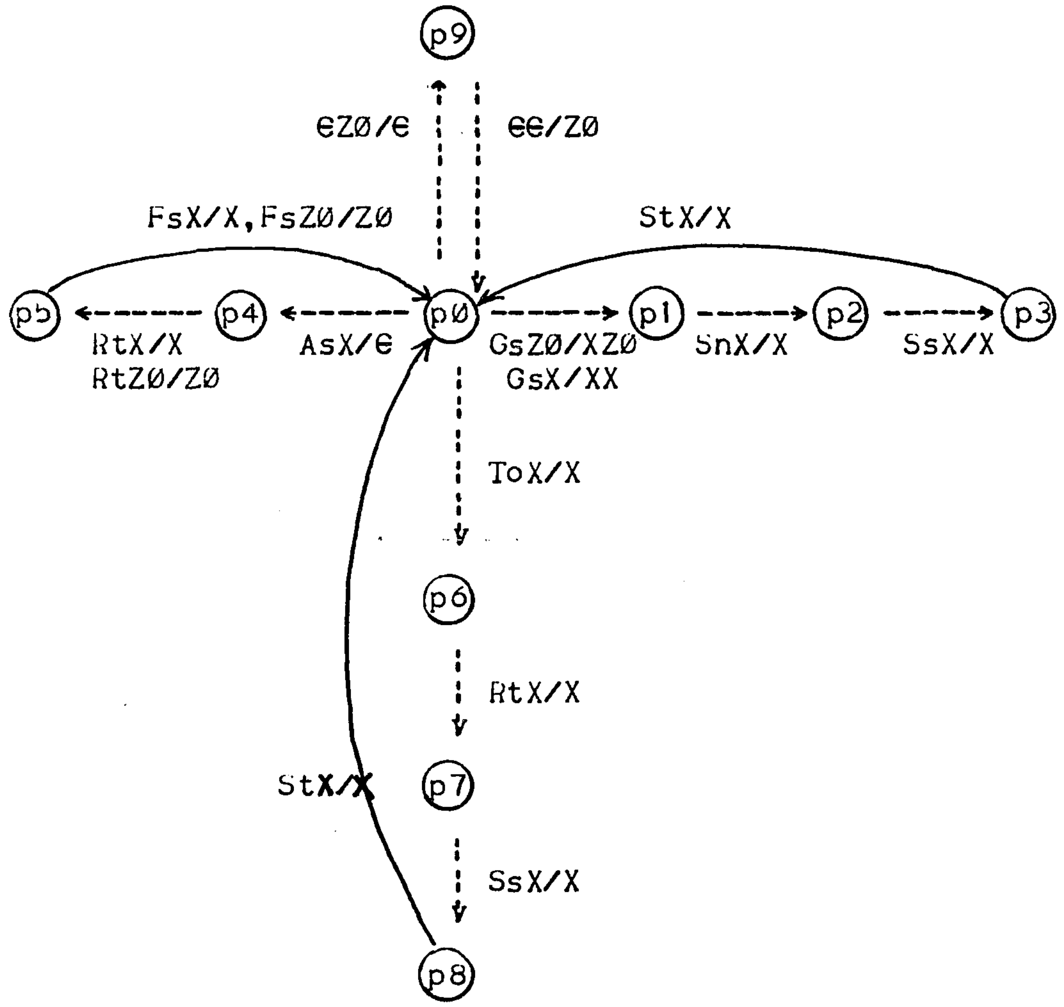


Fig. 15. Push-down automaton that recognizes L(GI12)

(2) Secondly, we can construct a deterministic state automaton A to recognize L(Gr)

```
    A = ( Ka,V2,f2,qØ,Fa )

where     Ka = { qØ,q1,q2,q3,q4,q5 }
          V2 = { R1,Ar,Cr,Fu,Sh,Fr }
          Fa = { qØ }
```

and f2 is defined as follows:

```
          f2 ( qØ,R1 ) = q1
          f2 ( q1,Ar ) = q2
          f2 ( q2,Cr ) = q3
          f2 ( q3,Fu ) = q4
          f2 ( q4,R1 ) = q1
          f2 ( q4,Sh ) = q5
          f2 ( q5,Fr ) = qØ
```

The automaton is also shown in Fig. 16.

$$\begin{tikzcd}
\boxed{q\emptyset} \arrow[r, dashed, "R1"] & \boxed{q1} \arrow[r, dashed, "Ar"] & \boxed{q2} \arrow[r, dashed, "Cr"] & \boxed{q3} \arrow[r, dashed, "Fu"] & \boxed{q4} \arrow[r, dashed, "Sh"] \arrow[lll, bend right, "R1"'] & \boxed{q5} \arrow[lllll, bend right, "Fr"']
\end{tikzcd}$$

Fig. 16 Finite state automaton that recognizes L(Gr)

(3) We can then construct a push-down automaton P2 that recognizes L(GI) // L(Gr) by following Algorithm 1.

P2 = (Kp x Ka,V1 U V2,T,f3,[p9,qØ],ZØ,Fp x Fa)

```
where   Kp x Ka { [pØ,qØ],[pØ,q1],........,[pØ,q5],
                  [p1,qØ],[p1,q1],........,[p1,q5],
                     .                        .
                     .                        .
                     .                        .
                  [p9,qØ],[p9,q1],........,[p9,q5] }
        V1 U V2 =
             { Gs,Sn,Ts,Rs,As,Fs,Rt,St,R1,Ar,Cr,Fu,Sh,Fr }
        Fp x Fa = { [p9,qØ] }
```

f3 is defined as follows:

```
        f3 ( [pØ,qi],Є,ZØ )  = ( [p9,qi],Є )
        f3 ( [p9,qi],Є,Є )   = ( [pØ,qi],ZØ )
        f3 ( [p5,qi],Є,ZØ )  = ( [pØ,qi],ZØ )
        f3 ( [pØ,qi],Gs,ZØ ) = ( [p1,qi],XZØ )
        f3 ( [pØ,qi],Gs,X )  = ( [p1,qi],XX )
        f3 ( [p1,qi],Sn,X )  = ( [p2,qi],X )
        f3 ( [p2,qi],Ss,X )  = ( [p3,qi],X )
        f3 ( [p3,qi],St,X )  = ( [pØ,qi],X3)
        f3 ( [pØ,qi],As,X )  = ( [p4,qi],Є )
        f3 ( [p4,qi],Rt,X )  = ( [p5,qi],X )
        f3 ( [p4,qi],Rt,ZØ ) = ( [p5,qi],ZØ )
        f3 ( [p5,qi],Fs,X )  = ( [pØ,qi],X )
        f3 ( [p5,qi],Fs,ZØ ) = ( [pØ,qi],ZØ )
        f3 ( [pØ,qi],TØ,X )  = ( [p6,qi],X )
        f3 ( [p6,qi],Rt,X )  = ( [p7,qi],X )
        f3 ( [p7,qi],Ss,X )  = ( [p8,qi],X )
        f3 ( [p8,qi],St,X )  = ( [pØ,qi],X )   ∀ qi e Ka

and     f3 ( [pi,qØ],R1,y )  = ( [pi,q1],y )
        f3 ( [pi,q1],Ar,y )  = ( [pi,q2],y )
        f3 ( [pi,q2],Cr,y )  = ( [pi,q3],y )
        f3 ( [pi,q3],Fr,y )  = ( [pi,q4],y )
        f3 ( [pi,q4],R1,y )  = ( [pi,q1],y )
        f3 ( [pi,q4],Sh,y )  = ( [pi,q5],y )
        f3 ( [pi,q5],Fr,y )  = ( [pi,qØ],y )   ∀ pi e Kp
                                         and ∀ y e {X,ZØ}
```

Lemma Let C be a class of languages closed under intersection, Є-free substitution, and union and intersection with regular sets. Then C is closed under the arbitrary shuffle operation.

Proof:

Let L1 and L2 be two languages in C which are subsets of V1* and V2* respectively. We assume without loss of generality that the sets V1 and V2 are disjoint.

Define two substitution functions f1 and f2 by

f1(a) = V2*a for each a in V1, and
f2(b) = V1*b for each b in V2.

The closure under union and intersection with regular sets guarantees that all regular sets are in C and hence V2*a and V1*b are in C.

Let L1′ = f1(L1) U V2* and L2′ = f2(L2) U V1* if ϵ is in L1 and L2, and L1′ = f1(L1) and L2′ = f2(L2) otherwise. Then L1′ is the set of <u>all</u> strings of the form y1b1y2b2....ynbn, n≥1, where the b′s are in V1, b1b2....bn is in L1 and the y′s are in V2*, plus V2* if ϵ is in L1.

L2′ is the set of <u>all</u> strings of the form x1a1x2a2....xmam, m≥1, where the a′s are in V2, a1a2....am is in L12 and the x′s are in V1*, plus V1* if ϵ is in L2.

The ϵ-free substitution guarantees that L1′ and L2′ are in C, and we can easily see that the intersection of L1′ and L2′ is the same as L1 // L2. Since L1′ and L2′ are closed under intersection, we know C is closed under the shuffle operation.

From the closure properties of regular, deterministic context-free, context-sensitive, and type Ø languages, we know they are all closed under intersection, ϵ-free substitution, and union and intersection with regular sets. We therefore have the following result from the above lemma.

<u>Theorem</u> <u>3</u> The classes of regular, deterministic context-free, context-sensitive and type Ø languages are closed under the arbitrary shuffle operation.

4.4.2 Restricted Shuffle (\\) Operation

An important protocol verification step is to validate that the global interactions of the TGs we design are free of protocol syntax errors such as communication incompatibility, communication looping and communication deadlocks. Global validation techniques will be the topic of Chapter 5. Basically, the restricted (global) shuffle operation is used to integrate TGs of two communicating entities. It can validate the global interactions between the TGs and take into consideration the communication channel states. This shuffling usually integrates a send module and a remote receive module as shown in Fig. 17.

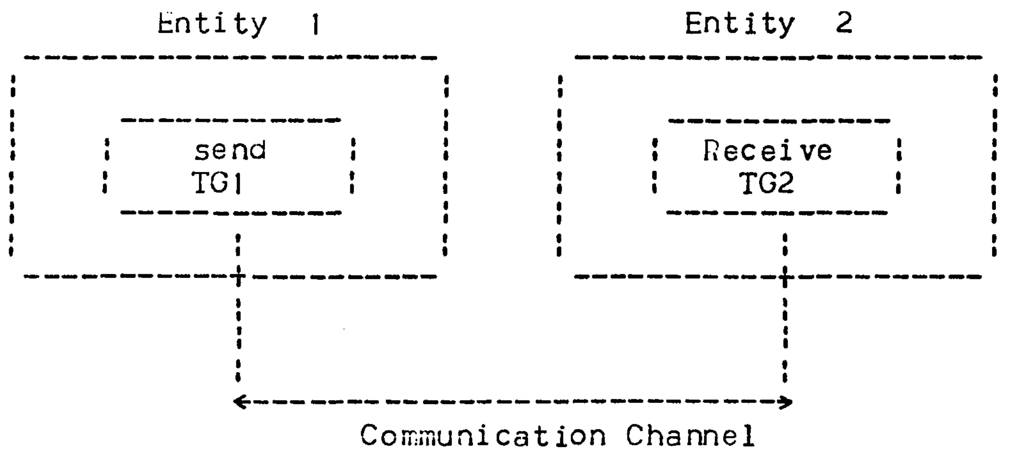


Fig. 17 Global (Restricted) shuffle operation

Def.20 Let V1 and V2 be two sets of terminals. Given A, a subset of $V1^*$; B, a subset of $V2^*$; and $V1 \cap V2 = \Phi$; then the restricted shuffle product of A and B (A \\ B), is a subset of (A // B) and consists of all the sentences of (A // B) which are realizable action sequences. A realizable action sequence cannot contain any string which has a receive action of V2 (or V1) before its corresponding send action of V1.

The algorithm to construct the restricted shuffle product is similar to Algorithm 1 and Algorithm 2 of the arbitrary shuffle product. However, for restricted shuffling, we have to remove all the impossible state transitions and unrealizable action strings.

TG global validation will be discussed further in Section 4.7.

4.4.3 Substitution Operation

To integrate TGs with a hierarchical relationship, we use the substitution operation.

The substitution operation [Hopcroft69], which corresponds closely to the step-wise refinement concept [Wirth71] in structured programming, can perform: (1) the mapping of a terminal of an entity's TG to an action sequence for more detailed protocol description; and (2) the integration of an entity's TG with its upper and/or lower layer TGs by properly substituting low-level non-terminals for high-level terminals to form a new integrated TG. This operation can be used for top-down design of protocols and to inter-relate upper and lower layers of protocols in a protocol hierarchy.

The theorems and algorithms of the substitution operation are discussed in Theorem 9.7 of [Hopcroft69]. It is known that the classes of regular sets and CFLs are closed under the substitution operation. The class of CFLs is closed under ϵ-free substitution. The following is a substitution example of three CFLs:

EXAMPLE 4.3

Let L be generated by the grammar

```
    ( {<S>},( "send", "ack" ), P,<S> ), where P consists of:
        <S> ::= "send" "ack" <S> | ϵ
```

We can substitute the terminals "send" and "ack" to get f(L) in the following:

```
Let f("send") = { Gs Sn Ss St(Rs Rt Ts St)* } and
    f("ack")  = { As Rt Fs };

let  f("send") be generated by
     ( {<S1>,<retransmit>},{Gs,Sn,Ss,St,Rs,Rt,Ts},P1,<S1> ),
where P1 contains:
     <S1> ::= Gs Sn Ss St <retransmit>,
     <retransmit> ::= Rs Rt Ts St<retransmit> | ∈.

and let f("ack") be generated by
     ( {<S2>},{As,Rt,Fs},{<S2>::=As Rt Fs},<S2> ).


Then f(L) is generated by

( {<S>,<S1>,<S2>},{Gs,Sn,Ss,St,Rs,Rt,Ts,As,Fs},P',<S> ),
where P' contains:
     <S> ::= <S1><S2><S> | ∈
     <S1> ::= Gs Sn Ss St<retransmit>,
     <retransmit> ::= Rs Rt Ts St<retransmit> | ∈,
     <S2> ::= As Rt Fs.
```

The first production comes from

<S> ::= "send""ack" <S> | ∈

with <S1> substituted for "send" and <S2> for "ack". Note that f(L) is a well-formed TG of the language L(GI1) which is defined in Chapter 3.

For the new grammar, we may again use the substitution operation to map terminal Ts ("transmit a packet out to a lower level") to a sequence of actions which are described at the character (ASCII) level. More complex substitution of protocol components (such as SDLC bit stuffing) is also possible.

4.5 TG Local Validation Techniques

We concentrate on local TG validation in this section and leave global validation as the topic of Chapter 5. TG validation techniques for verifying large-scale software systems can improve their reliability. The techniques are mechanical in nature and have been implemented. The cost of correct protocol development can be greatly reduced using these validation techniques.

Automatic techniques for TG syntax validation can be applied without regard to their semantic meanings and actual execution. They can be used to locate possible structure flaws to achieve a correct structure for the TG. Protocol validation can be considered as the analysis of the following: (1) no-backup parsing, (2) well structured TG, (3) receive deadlock, (4) incompatibility and (5) loops, each of which is discussed below.

4.5.1 No-backup Parsing

Automatic implementation techniques discussed in Chapter 6 require that the TG of a protocol is able to parse without backup. The TG grammar rules that enable the parsing without backup have to follow two restrictions: (a) deterministic TG, and (b) no left-recursion. Note that these restrictions are not needed for validation and will not affect the power of modeling.

a) Deterministic TG

Every pair of initial symbols x and y of alternates in the grammar production rule of a deterministic TG must satisfy the restriction that

If $x \Rightarrow^* A\ u$ and $y \Rightarrow^* B\ v$, then $A \neq B$,
$\forall\ A, B \in Vt$ and $u, v \in (Vn \cup Vt)^*$.

b) No left-recursion

This means that there does not exist in the grammar a derivation:

$\langle u \rangle \Rightarrow^+ \langle u \rangle\ y$,
$\forall \langle u \rangle \in Vn$ and $\forall\ y \in (Vn \cup Vt)^*$.

We can design algorithms to list all the non-terminals and production rules that violate these two restrictions. Before introducing these algorithms, we first introduce a technique for constructing the transitive closure of a relation.

For some relation R on a set (alphabet, set of symbols) A and some element U we often wish to construct the set

K = {S ¦ U R+ S, S e A}

for validation checking. From the well-known theory of the transitive closure of relations [Gries71], we know that if the set A is finite, then the length of the relation chain is bounded by the number of elements in A.

An important and efficient algorithm using Boolean matrix aritnmetic can also be used to construct the transitive closure. For a relation R defined on a set S of n symbols S1, ...,Sn we can construct an n by n Boolean matrix B to represent this relation by putting 1 in B[i,j] if and only if Si R Sj. The matrix B+ defined by

B+ = B + BB + BBB + ... + B!n

can also be proved to represent the transitive closure R+ of R. The following algorithm [Warshall62] is often used for deriving the matrix B+ from B

```
1.  Set a new matrix B+ = B.
2.  Set i := 1.
3.  For all j, if B+[j,i] = 1 then for k = 1,...,n, set
    B+[j,k] := B+[j,k] + B+[i,k].
4.  Add 1 to i.
5.  If i ≤ n then go to step 3; otherwise stop.
```

Returning to the previous restrictions that enable the TG parsing without backup, it is obvious that a non-terminal

<u> of a TG satisfies the left-recursion condition if and only if <u> FIRST+ <u> where the relation FIRST was defined in Definition 12. The following steps can be used to test left recursion. We can use an algorithm to construct B+ from a matrix B representing the relation FIRST on Vn of the TG. We then have B+[i,i] = 1 if and only if the non-terminal Si e Vn is left recursive. The deterministic restriction can also be validated from the relation FIRST+ on Vn U Vt of a TG.

4.5.2 Well Structured TG

The transitive-closure construction technique can also be used to check a transmission grammar for well structured form. If the TG is not well structured, it could either cause errors or increase complexities.

A well-structured TG requires that there is no undefined non-terminal or superfluous rules, and that each sentence (action sequence) of the TG terminates properly. We say a TG is well structured if it follows the following two restrictions: (A) the TG is reduced, and (B) the TG can generate only proper sentences.

The reduced TG defined in Definition 11 prevents superfluous rules and undefined non-terminals. A TG is called reduced if and only if every non-terminal <u> e Vn satisfies the following two conditions:

(1) <u> must be reachable from the starting symbol, i.e. <u> must appear in the following sentential form:

<st> =>* x <u> y, for some x, y e (Vn U Vt)*, and <st> is the starting symbol of the grammar.

A non-terminal is reachable from the starting state if it has this property, in which case we know that an unreachable non-terminal is either caused by design errors or represents superfluous redundancy. A non-terminal <u> satisfies the restriction if and only if <st> WITHIN+ <u>. The algorithm for checking reachability is therefore straightforward. We first construct an n by n Boolean matrix B representing the relation WITHIN over the set of non-terminals Vn. The matrix B+ is then derived by Warshall's algorithm. It is clear that all the non-terminals are reachable from <st> if and only if B+[1,j] = 1 for j = 2,...,n.

(2) It must be possible to derive a string t of terminal symbols from <u>:

<u> =>+ t, for some t e Vt+

Figure 2.11 of [Gries71] contains a flow chart of an algorithm which recursively checks each non-terminal of a grammar to see if it can generate a terminal string.

The proper sentence restriction requires the proper termination of all action sequences of a TG. It is clear that x is a terminating symbol if and only if <st> =>+ ...x and x e Vt. Note that this relation is the same as <st> LAST+ x. A symbol x is a terminating symbol if and only if <st> LAST x or there exists a non-terminal <u> such that <st> LAST+ <u> and <u> LAST x. A simple recursive routine can be used to list all the terminating symbols for a given TG.

For a well-structured TG, it is obvious that all the action sequences will end at a proper exit. We can be sure that from any state of the TG it will be possible to reach an appropriate terminating state. This is an important property and is also called liveness of all the states of a protocol.

A static deadlock is a state that cannot reach an appropriate terminating state. A well-structured TG has the nice property that it will never reach a static deadlock.

EXAMPLE 4.4 To show the checking of the above syntax errors, let us consider the following grammar:

```
<TG> ::= <a> <b> <c> # ¦  b a b
<a>  ::= <d> a
<b>  ::= b a ¦ b b
<d>  ::= <a>
<e>  ::= <f>
<f>  ::= <e> <e>
```

In this grammar, we can see that <c> is an undefined non-terminal, <a> =>+ <a> a is left recursive, and both <b> =>* b a and <b> =>* b b have the same initial symbol. Since <TG> is the starting symbol, both <e> and <f> do not satisfy the two condition that there should not be any superfluous rules. This example will be also shown as the test example 2 in Section 4.6 when we discuss automatic TG validations.

4.5.3 Receive Deadlock

A receive deadlock happens when an entity is idle waiting for another entity to send a message but there is no such message coming through the channel between them. Even in a well-structured TG, we may still have the problem of receive deadlock. It can be caused by either message loss or incompatibility. The general technique of preventing a receive deadlock caused by message loss is to have a timeout scheme to recover from it. Therefore, for each action of receive waiting, we should have a timeout and retransmit action as an alternate part of the transition. The technique for overcoming the problem of incompatibility is the topic of the next section.

4.5.4 Incompatibility

A simple but useful technique is to list all the sending and receiving messages (in non-terminal form) for two communicating entities. The two sets should have a one-to-one correspondence if they are compatible. However, this technique is inadequate to fully check for incompatibility. A global reachability analysis is necessary for complete checking. Global validation techniques will be covered in Chapter 5.

4.5.5 Loops (Oscillations)

Action loops in the action sequences of a protocol TG should also be identified by the validation system. There are two types of loops: (1) loops with no exit and (2) loops with a proper exit. We say a loop is looping with no exit if there exists a non-terminal in the loop which is not one of the liveness states. In a well-structured TG, it is clear that all the loops have proper exits. For a loop with a proper exit, we are interested in self-limiting and dynamic deadlock checking.

The technique we use to detect loops is to identify the non-terminals that could generate the following sentential form:

```
<u> =>+ x <u> y for some x,y e (Vn U Vt)*.
```

Note that a non-terminal <u> can derive loops if and only if <u> WITHIN+ <u>. Thus we can construct an m by m Boolean matrix A representing the relation WITHIN over Vn of m non-terminals: S1,S2,...Sm. We can then derive A+ representing the WITHIN+ relation. It is clear that a non-terminal Si e Vn can derive a loop leading to itself if and only if A+[i,i] = 1.

In order to prevent the possible infinite looping of action sequences, one of the action terminals in each loop has to limit the number of executions of the loop. For example, a loop in the TG of Fig. 20 (shown in the next section) is:

```
<wait> ::= <retransmit> <wait.ack>
```

Since the number of retransmissions has to be limited, we should include the limitation in the TG. By a similar argument we should limit the number of outstanding packets in the TG. Some loops are inherently limited by a higher-layer protocol and do not need to be limited in the TG.

Dynamic deadlock is a special type of looping, called infinite idle looping. Idle looping happens during a time period T3-T1 when time T3>T2>T1 exists such that

State (T1) = State (T3) $\neq$ State (T2),

and no effective communication action has been made during the time period. Idle looping could be a send action which repeats a number of times without a positive ACK or a receive action which waits for a message, but no such message is underway. In designing a TG, we should detect all such loops and have a scheme to recover from them.

The technique for detecting idle looping is to identify the non-terminals that could generate the following sentential form:

<u> =>+ x <u> y for some x,y $\in$ (Vn U Vt)* such that x can generate only idle strings.

As defined in Definition 2, an idle string contains no effective communication symbol and therefore no effective communication progress can be made during idle looping. We can check and identify potential infinite idle looping when we generate loops using the above techniques.

4.6 The Validation System

A validation system has been constructed to check for the properties discussed in the previous section. The complete program is listed in Appendix A. The reachability analysis of a TG should be performed for both local and global transitions.

The system reads a TG that needs to be validated and stores it in a carefully designed data structure so that efficient testing and checking can be made. After informing the system of the starting symbol, it can parse any test action sequence and tell whether it is a valid sentence of the TG. It can check for the terminating symbols, left recursion, unreachable nonterminals and proper terminations. It will also indent and list all the loops (cycles) for boundedness checking.

Three test examples are shown in Fig. 18, 19 and 20. Fig. 18 is a test of a valid TG with four testing action sequences. Two of them are incorrect. Fig. 19 is a test of an invalid TG which is the same TG as Example 4.4. The system prints out undefined symbols, terminating symbols, left recursion nonterminals and unreachable nonterminals more completely than our hand checking result of the previous section. Fig. 20 is another test of a valid TG which specifies the sending part of a communication processor protocol.

```
<A>   ::= X , ( <B> ) .
<B>   ::= <A> <C> .
<C>   ::= + <A> .
----------------------------------------
<A>
( X + ( ( X + X ) + X ) ) .
( X + X ) .
( A + X).

1       <a>  ::= x ; ( <b> ) .
2        <b>  ::= <a> <c> .
3        <c>  ::= + <a> .
4        ----------------------------------------
          PROPER TERMINATION ANALYSIS
          ***************************
          ** WARNING ** MORE THAN ONE EXITS
          THE TERMINATE SYMBOL IS: X          )

          NO LEFT RECURSION
          REACHABILITY ANALYSIS
          *********************
          NO UNREACHABLE NONTERMINAL
          ALL NONTERMINALS CAN TERMINATE PROPERLY.
          ACTION SEQUENCE TESTING
          **********************
5        <a>
6        ( x + ( ( x + x ) + x ) ) .
          THE SYNTAX OF THE ABOVE ACTION
                           SEQUENCE IS: CORRECT
7        ( x + x ) .
          THE SYNTAX OF THE ABOVE ACTION
                           SEQUENCE IS: CORRECT
          THE SYNTAX OF THE ABOVE ACTION
                           SEQUENCE IS:  INCORRECT
8        ( a + x).
          THE SYNTAX OF THE ABOVE ACTION
                           SEQUENCE IS: INCORRECT
          INDENTED CYCLES LISTING
          **********************
9
10       <a> ::= ( <b> )
11          <b> ::= <a> <c>
12
13          <b> ::= <a> <c>
14             <c> ::= + <a>
```

Fig. 18 Test 1 of a valid TG

```
<TG> ::= <A> <B> <C> # , B A B  .
 <A> ::= <D> A .
 <B> ::= B A , B B .
 <D> ::= <A> .
 <E> ::= <F> .
 <F> ::= <E> <E>  .
-----------------------------------
<TG>
A B A # .
```

```
1       <tg> ::= <a> <b> <c> # ¦ b a b  .
2         <a> ::= <d> a .
3         <b> ::= b a ¦ b b .
4         <d> ::= <a> .
5         <e> ::= <f> .
6         <f> ::= <e> <e>  .
7        ------------------------------------
          UNDEFINED SYMBOL --> <C>
          PROPER TERMINATION ANALYSIS
          ***************************
          ** WARNING ** MORE THAN ONE EXITS
          THE TERMINATE SYMBOL IS: #   B
          LEFT RECURSION --> <A>
          LEFT RECURSION --> <D>
          LEFT RECURSION --> <E>
          LEFT RECURSION --> <F>
          REACHABILITY ANALYSIS
          *********************
          UNREACHABLE NONTERMINAL ---> <E>
          UNREACHABLE NONTERMINAL ---> <F>
```

Fig. 19 Test 2 of an invalid TG

```
<S.VIP> ::= <S> <S.VIP> , # .
<S> ::= <S.PACKET> <WAIT.ACK> .
<S.PACKET> ::=    <S.ROUTING> , <S.HOST> .
<S.ROUTING> ::= SS RO SN ST SS .
<S.HOST> ::= SS GS SN ST SS .
<WAIT.ACK> ::= <ACK> , <RETRANSMIT> <WAIT.ACK> ,
               <S> <WAIT.ACK> .
<ACK> ::= AS RT FS .
<RETRANSMIT> ::= TO RT ST SS .
--------------------------------------------
<S.VIP>
SS RO SN ST SS AS RT FS #  .

1      <s.vip> ::= <s> <s.vip> ¦ # .
2       <s> ::= <s.packet> <wait.ack> .
3       <s.packet> ::=    <s.routing> ¦ <s.host> .
4       <s.routing> ::= ss ro sn st ss .
5       <s.host> ::= ss gs sn st ss .
6       <wait.ack> ::= <ack> ¦ <retransmit>
                       <wait.ack> ¦ <s> <wait.ack> .
7       <ack> ::= as rt fs .
8       <retransmit> ::= to rt st ss .
9       --------------------------------------------
         PROPER TERMINATION ANALYSIS
         ***************************
         THE TERMINATE SYMBOL IS: #
         NO LEFT RECURSION
         REACHABILITY ANALYSIS
         *********************
         NO UNREACHABLE NONTERMINAL
         ALL NONTERMINALS CAN TERMINATE PROPERLY.
         ACTION SEQUENCE TESTING
         ***********************
10       <s.vip>
11       ss ro sn st ss as rt fs #  .
         THE SYNTAX OF THE ABOVE ACTION
                           SEQUENCE IS: CORRECT
         INDENTED CYCLES LISTING
         ***********************
12
13      <s.vip> ::= <s> <s.vip>
14
15      <s> ::= <s.packet> <wait.ack>
16         <wait.ack> ::= <s> <wait.ack>
17
18      <wait.ack> ::= <retransmit> <wait.ack>
19
20      <wait.ack> ::= <s> <wait.ack>
```

Fig. 20 Test 3 of a valid TG

4.7 TG Global Validation

As we discussed in Chapter 2, protocol syntax errors such as transmit interferences, looping, deadlocks and incompatibility may arise when two (or more) entities communicate with each other. We may derive a shuffled TG to represent the interactions of the entities by the restricted shuffling of their respective TGs. The same syntax checking techniques can then be applied to the integrated TG to ensure that it is well structured, non-deterministic and free of deadlocks.

Def.23 A global state of two communicating TGs is a composite state [S1,S2], where S1 and S2 are the states of the communicating entities. The global state may also include the state of the channel between the TGs for the purpose of validation.

Def.24 A realizable state of two communicating TGs is a global state which can be reached from [SØ,SØ'] where SØ and SØ' are the starting states of the TGs. A receive action of a TG cannot occur unless a corresponding send action at the other communicating TG has already occurred.

Static structure validation techniques for a local environment of the previous section can be applied to global validation. Incompatibility can also be checked from the restricted shuffling operation. Algorithms can be used to list all the loopings, static deadlocks and incompatibilities by exhaustive checking of all the possible transitions between global states.

However, TG global validation techniques do suffer from the state explosion problem when the number of states involved is relatively large. For two communicating entities with states numbering n1 and n2 respectively, the number of global interacting states after restricted shuffling is bounded by the Cartesian product of n1 and n2. The state explosion problem is even more serious when we have to model complex channel states such as loss and out of order. A validation automaton model is introduced in Chapter 5 to perform automatic global validations in a clear and concise manner.

4.8 Structured Verification

The following step-wise validation techniques can be used to verify the protocol correctness of the TG protocol model. First, induction rules can be conveniently used to prove protocol properties of validation independent parts directly from the transmission grammar.

Local validation can be applied to check the syntax of the TG to see if it is well structured and global validation can be applied to reveal sophisticated protocol syntax errors.

By the arbitrary shuffle operation, the validation independent part TGs can then be integrated into a new TG while still keeping the action sequences and the verified protocol properties of the individual validation independent part TGs. We may add some semantics to the integrated TG and apply the TG for the purpose of automatic implementation.

From the required hierarchical design of protocol TGs, the complexity of verification can be further alleviated by way of structured verification; that is, proving the properties of one layer by utilizing the known (proved) TG properties of its lower layers.

Induction proof

We will demonstrate a protocol proof example from TG specification. The example is similar to the sending part of G2 in Chapter 3. The purpose of this example is to demonstrate that the TG specification can contain certain protocol "semantics" properties. We can prove that it will generate the following language L. (GsSn and AsFs are two atomic symbols.) We use the symbol "#" as the meaning of "the number of occurrences".

L={w|w∈{GsSn,AsFs}* and (the # of GsSn's in w = the # of AsFs's in w) and (for all x,y such that w=xy with x≠∈, the # of GsSn's in x >the # of AsFs's in x)

Properties of L:

(1) L is the language of all sentences w in {GsSn,AsFs}* such that if one scans w from left to right, he will always have encountered at least the same number (if not more) of occurrences of GsSn's as AsFs's and at the end of the scan the # of GsSn's must equal to the # of AsFs's.

(2) If GsSn = (and AsFs =) then L is equivalent to the set of balanced parenthesis strings, i.e., the set {(),()(),(()),((())),....}. Therefore we call the string generated by L a <u>balanced</u> string.

EXAMPLE 4.5

Claim If G2 is a CFG2 for L, where

```
G2=(Vn,Vt,P,<st>)
Vn={<st>,<s>}
Vt={GsSn,AsFs}
P:
  <st> ::= <s> | ∈
  <s> ::= GsSn <st> AsFs <st>
```

then L(G2)=L

Proof

We will prove by induction on n, the length of sentence w, that for all $n \geq 1$, <st> => w if and only if w is a sentence in L; i.e., (the # of GsSn's in w = the # of AsFs's in w) and (for all x,y such that w=xy with $x \neq \epsilon$, the # of GsSn's in x $\geq$ the # of AsFs's in x)

(a) (<=) prove if w is balanced then w is a string in L(G2).

For n=Ø, i.e., w=∈ and w∈L(G2) is trivially true.

For n>Ø, let the hypothesis hold for all k<n where k is the length of the string.

If w is a string whose length is n, then w is the form

w = GsSn w1 AsFs w2

where w1 and w2 are balanced strings and may be null. Since w is of length n, w1 and w2 must have length less than n, therefore w1 and w2 can be generated by G (by assumption). i.e.

<st> => w1, <st> => w2

But from the production rules, we observed that:

```
<st> ::= <s>
<s> ::= GsSn <st> AsFs <st>
```

so we have <st> => w

From the induction rule we know (a) holds for all n e N.

(b) (=>) prove if w is a string in L(G2) then w is balanced.

For n=Ø, <st> ::= ∈ and ∈ is a balanced string, therefore the hypothesis holds.

For n>Ø, let the stings derived from <st> be balanced if they are of length less than n. Now, consider a derivation of a string w of length n and w = GsSn w1 AsFs w2

where w1 and w2 are strings in L(G2). Since w1 and w2 must have length less than n, therefore w1, w2 are balanced strings(by assumption). Now, w = GsSn w1 AsFs w2 is balanced if w1 and w2 are both balanced.

From the induction rule, hypothesis (b) holds for all n ∈ N.

CHAPTER 5

GLOBAL VALIDATION TECHNIQUES

5.1 Global Validation Overview

Techniques for global validation of computer network protocols have progressed significantly in the past two years [Sunshine78a,b]. Part of the increased success of recent efforts has involved the ability to automate some steps of the protocol analysis, particularly the checking of the global state space. A global state is the state of two communicating entities plus messages flowing in the transmission channel between them. Transitions from one global state to another are derived from the state transitions of the individual entities, by including all realizable transitions of either entity, given the state of the transmission medium.

Hajek [78, 79] and West [78b] have developed programs which interactively generate all the reachable global states from an initial start state of the system and have identified the occurrences of a number of errors such as deadlocks and incompatibility (receive incompleteness). The algorithm is essentially a tree generation scheme which works as follows: (1) first generate all possible transitions given the initial state to derive a number of new global states, and (2) repeat the process for each of the newly generated states until no new states can be generated. (Stop the exploration for transitions that lead to already generated states.) West has validated the call establishment part of the CCITT X.21 protocol and uncovered a number of incompleteness (incompatibility) errors. Hajek has designed a protocol verification program (APPROVER) which has revealed a number of errors in some published protocols.

In Chapter 4, we have shown that the local grammar description of a communication entity allows us to automate reachability analysis and to derive a protocol with correct program (TG) structure. The reachability analysis is also well-suited to validate the structure for global state transitions. It can detect protocol syntax errors such as deadlocks, incompatibility and infinite idle looping.

Although the global state models are able to model control aspects of protocols well when the numbers of global states are relatively small, the major difficulty of this technique is "state explosion". The cardinality of the composite states between two communicating entities is equal to (the cardinality of states in one entity) x (the cardinality of states in the other entity) x (the cardinality of states in the transmission medium). The global state explosion problem arises when the number of states involved is relatively large. The explosion problem is especially serious when we have to model a large number of sequence spaces, multiple outstanding sends and random delays. The difficulty is that these characteristics introduce non-polynomial complexity. For the case of 10 messages outstanding with random delay, the number of possible arrivals at the receiving entity is 10!. This explosion is large enough to make a program with an execution time of seconds become an execution time of years. This level of complexity certainly makes it impractical to generate and check all reachable states in reasonable time and storage.

We will introduce the validation automaton (VA) model using a special validation language to handle the state explosion problem. Several important types of protocols have been successfully validated by the VA model in this chapter. Reduction techniques can be used in the VA model to simplify the validation process for the protocols.

We shall use the abbreviations "VA" to represent "validation automaton" and "VAs" to represent "validation automata" throughout the rest of the dissertation.

5.2 Validation Automaton Model

The validation automaton model is designed to describe the interaction and interdependency of communicating entities. Using the va model, we first specify a VA for each of the communicating entities and then derive a global va from the VAs of the entities to represent the global state transitions. This validation model allows global state transitions in the global va to be represented in a simple and clear manner. It can conveniently model complex channel characteristics such as loss, disorder and multiple outstanding sends to reveal protocol syntax errors. The idea is to "view" the channels between the communicating entities as a set of queues and use actions in each VA to manipulate the queues. The actions are designed to have the capability of reducing the number of global states. We will first restrict ourselves to the case that only two entities are communicating and then expand the model to handle more than two entities.

As shown in Fig. 21, a full-duplex channel between two communicating entities A and B is represented by two sets of acknowledgment and message queues, one for each entity. The convention we use for naming a channel queue is to put the name of the queue first, followed by the name of the entity for which the queued elements are destined. (The two names are separated by a "_".) For example, the name ACK_B means the acknowledgment queue which contains acknowledgment elements flowing toward entity B; MSG_A means the message queue which contains message elements flowing toward entity A. For each validation automaton, we specify validation actions which manipulate the queues or change some relevant state information. The set of validation actions which can be used to manipulate the queues will be presented in the next section.

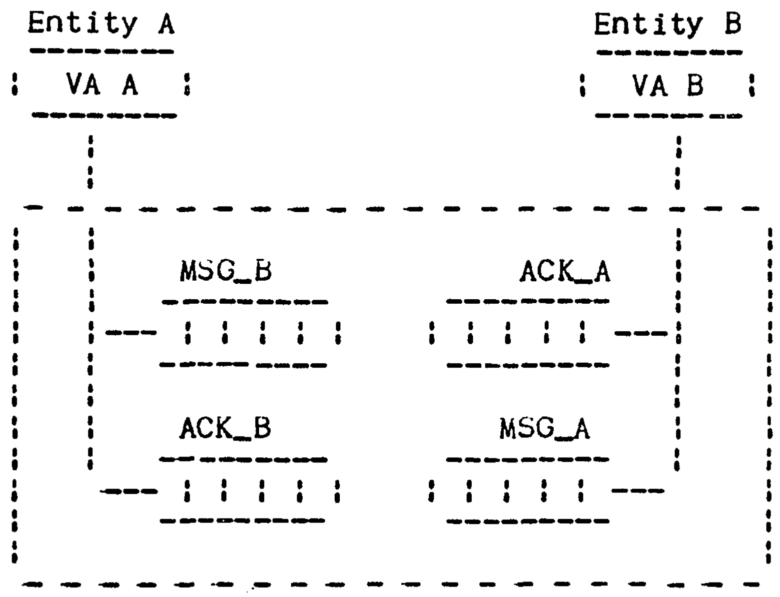


Fig. 21 Channels represented by a set of queues

A global validation state of two communicating entities A and B can be represented as the state information of both entities plus the channel represented by channel queues. For two-entity interaction, we can represent the global state by the following matrix format (excluding the interfaces with higher-layer entities).

$$\begin{array}{r}\text{Entity A}\\ \text{State}\\ \text{Information}\end{array} \begin{Bmatrix} \text{MSG\_B} & \text{ACK\_A} \\ \text{ACK\_B} & \text{MSG\_A} \end{Bmatrix} \begin{array}{l}\text{Entity B}\\ \text{State}\\ \text{Information}\end{array}$$

Each entity has a column in the matrix which includes an acknowledgment queue and a message queue of elements flowing toward the other entity. Since the matrix already has queues of channel information to represent the interactions, the state information usually is not required in the representation.

In order to include the interactions with higher-layer entities of entities A and B, the channel queues can be represented by a two-row matrix with multiple columns. The queues in the top row contain messages flowing to the immediate right entity and acknowledgments flowing to the immediate left entity. The queues in the bottom row have the opposite direction of flow to the top row. The channel queues of the communicating entities A and B are shown in Fig. 22, including the interfaces to higher-layer entities A1 and B1. To prevent confusion arising from using the same queue and destination names at different entities, we may include the source entity in the queue name. For example, ACK_A(B) means that the acknowledgment queue has elements originating from source entity B and destined for A.

Entity A1	Entity A		Entity B		Entity B1
MSG_A	ACK_A1	MSG_B	ACK_A	MSG_B1	ACK_B
ACK_A	MSG_A1	ACK_B	MSG_A	ACK_B1	MSG_B

Fig. 22 Channel queues including higher-layer interfaces.

As we can see in Fig. 22, each row in a channel matrix represents the message transfer one direction and the acknowledgment transfer in the other direction. We therefore need only one row to represent the global state of a half-duplex channel.

5.3 Validation Actions

The following is a list of the actions which are useful to manipulate messages in the channel queue. Each action is usually followed by a queue name and a message name as its operand. The convention is to put a "." to separate the names of the action, queue and message.

(1) Queue (Q) -- This action inserts the specified message into the specified queue in a first-in-first-out (FIFO) manner (i.e., it puts the message at the end of the queue).

(2) Priority Queue (P) -- This action inserts the specified message into the specified queue in a last-in-first-out (LIFO) manner (i.e., it puts the message at the front of the queue). It is the same as a PUSH operation in a stack structure.

(3) Fetch (F) -- This action deletes the specified message from the specified queue in a random manner. The action cannot "occur" during the global state generation if there is no message in the queue.

(4) Dequeue (D) -- This action deletes a message from the top of the specified queue (in a FIFO manner).

(5) POp (O) -- This action deletes a specified message from the bottom of the specified queue (in a LIFO manner).

(6) Clear (C) -- This action deletes all of the messages from the specified queue (i.e., clears the queue).

(7) Empty (E) -- This action tests whether the specified queue is empty. The empty action should appear as a head action symbol in an alternate part of a grammar rule in the VA. All the actions in the alternate part cannot "occur" during global state generation if the queue is not empty (i.e., the empty condition is not true).

(8) Non-empty (N) -- This action tests whether the specified queue is non-empty. The non-empty action should appear as a head action symbol in an alternate part of a grammar rule in the VA. All the actions in the alternate part cannot "occur" during the global state generation if the queue is empty (i.e., the non-empty condition is not true).

We can use a special symbol "*" to represent all the queues in an entity. For example C.*A means clearing all the queues of entity A.

From a protocol specified in TG model, we can easily derive its corresponding VA by changing all the terminal actions of the TG to the corresponding validation actions. These eight validation actions are adequate to model all the global state changes. In general, a send action puts a message into a queue for a period of time and a receive action deletes a message from a queue. Buffer size may be included in the validation automaton and can be checked automatically during the global VA generation.

An example of the validation automaton model is shown below for a simple point-to-point protocol between entities A and B. The interfaces with higher-layer entities have been omitted in the example.

The TGs for a simple protocol:

```
<idle> ::= "create.message" "send.MSG" <pending>
  <pending> ::= "receive.NACK" "retransmit" <pending>
              |"receive.ACK" <idle>
```

<u>SenderA</u>

```
<wait> ::= "receive.MSG" <consume> <wait>
  <consume> ::= "send.ACK" | "busy" "send.NACK"
```

<u>ReceiverB</u>

The corresponding VAs for A and B are:

```
<idle> ::= Q.MSG_B.MSG <pending>
  <pending> ::= F.ACK_A.NACK Q.MSG_B.MSG <pending>
             | F.ACK_A.ACK <idle>
```

<u>VA of SenderA</u>

```
<wait> ::= F.MSG_B.MSG <consume> <wait>
  <consume> ::= Q.ACK_A.ACK | Q.ACK_A.NACK
```

<u>VA of ReceiverB</u>

The global VA for the local VAs is shown in Fig. 23. Note that we usually put lower case letters to represent elements in the acknowldgment queues, and state 5 changes to state 2 by performing indivisible actions F.ACK_A.NACK and Q.MSG_B.MSG.

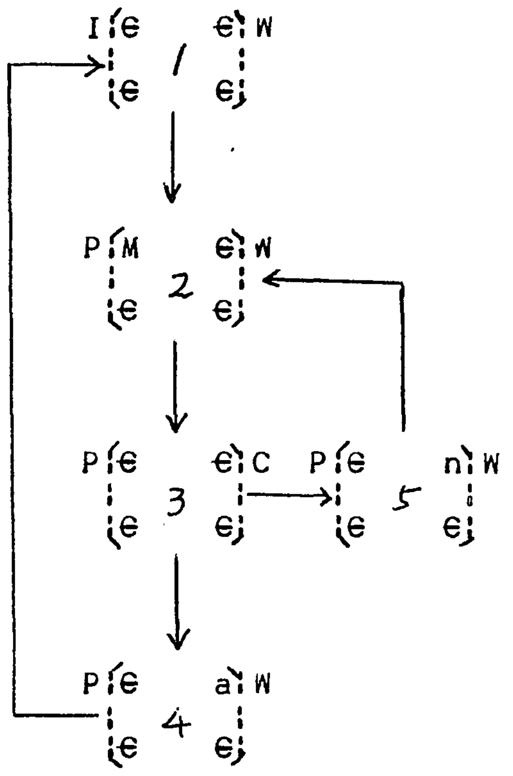


Fig. 23 Global VA of SenderA and ReceiverB

As we have discussed earlier, the major difficulty of this global validation model is "state explosion". Before using the model to represent and validate the global transitions of more complex protocols, we must investigate basic techniques for alleviating the global state explosion problem.

5.4 Reductions of States and Transitions

In the following, we present useful rules for reducing the number of states and transitions (arrows) in a global validation automaton, which can alleviate the global state explosion problem. The basic idea of the reduction is to remove all the states and transitions which are _irrelevant_ or _redundant_ for the purpose of validation.

5.4.1 General Reductions

Rule 1: It is not needed to model non-existent problems.

The validation problem checklist for a protocol can be derived according to the method described in Chapter 2. The validation problems have to be included in the VAs for the global validation but it is not needed to model those problems that do not exist in our design.

Rule 2: A sequence of actions within a communication entity should be combined into an indivisible action [Hajek78] or an atomic action [Sunshine78b] as long as it is known to be indivisible from the designer's point of view.

The validation automaton represented by the validation actions is already a simplified model because actions that are irrelevant to the global interactions are not included. The idea of combining actions together further reduces the number of local states in the VA, and thus reduces the number of global states in the global VA. In order to validate the actual design, the validation result should be the same even we combine actions together. During the process of the global state generation, any action can be performed without delay if it does not depend on the current content of channel queues. For a validation automaton specified in a grammar form, there is an algorithm available to combine the grammar rules [Wirth76]. We can use a similar technique to derive the indivisible actions.

Rule 3: Any global transition that causes local state changes for both of the two communicating entities cannot occur in the global VA model.

This simple rule helps us reduce transitions of simultaneous state changes for both of the entities [Sunshine78b]. Although the composite transition resulting from simultaneous transitions is perfectly legal, it can be represented by a sequence of the two individual transitions at either end.

5.4.2 Homogeneous Reductions

The TGs of two communicating entities are said to be homogeneous if one is a mirror image of the other (by mirror image we mean that the TGs or validation automata of these two entities are identical except that the directions of the messages are opposite). Two global states are said to be symmetric [Hajek78] if they are identical except for switching entity states and message directions in the channel queues.

Rule 4: If the TGs of two communicating entities are homogeneous, it is not needed to trace the extra symmetric global states in the process of global validation.

A large number of real-world protocols have homogeneous TGs among the communicating entities. We can reduce up to half of the global states in the global validation automaton by excluding the redundant symmetric states. The reason that we need to model only one of the symmetric states is that the states have the same "pattern" of transitions and the investigation of any one of them will serve our validation purpose of finding syntax errors.

Two symmetric states always have opposite directions of message flow. The following steps may be used to derive the symmetric state for any given global state: (1) interchange the two rows in the matrix; (2) reverse the sequence of both rows; and (3) change the names for both the state information and the elements of one entity to the corresponding names used by the other entity.

The checking of symmetric states can be done by a computer program if we use hashing techniques to check the elements of the state matrix.

<u>Rule 4.1</u>: If the channel between two homogeneous entities is half-duplex, it need validate the message flow for one direction only; that is, the sending part of one entity and the receiving part of the other.

This rule is obvious because of the symmetric property of the protocols for each direction of message flow.

<u>Rule 4.2</u>: If the channel between two homogeneous entities is full-duplex and the message flow in one direction does not interfere with the message flow in the other direction, only one direction of message flow need be checked in the global validation process.

The global transitions for both directions of message flow in the previous full-duplex channels are simply the arbitrary shuffle product of the global VAs for both of the half-duplex channels. This property will be shown in the example of Fig. 31 later in this chapter.

5.4.3 Retransmission Reductions

A message is said to be _fully retransmittable_ if it satisfies the following two conditions: (1) the message has a retransmission mechanism or is a response to a retransmittable message, and (2) the retransmit has the original message as its content or part of its content so that the semantics of the original message will be correctly delivered.

Rule 5: It is not necessary to model loss of a message (including acknowledgment message) and its retransmission in the validation automaton if the message is fully retransmittable.

Thus a fully retransmittable message can simply be modeled to remain in its channel queue in the global VA until it is fetched by the receiver. The second restriction of a fully retransmittable message is important and cannot be taken for granted. For example, if a retransmit is an acknowledgment in a piggybacked message, it should have its original acknowledgment as its content or part of its content. (The echo protocol, which echoes the current status of the number of messages received, is an example of having the original acknowledgment as part of its content.) Without this restriction, the retransmit cannot be assumed to represent its original content, and interference errors might happen if the incorrect assumption is made. If a message is not fully retransmittable, we have to model all the distinct retransmits and their losses in the global VA.

5.4.4 Multiple Outstanding Send and Disorder Reductions

Rule 6: If, at a global state, an entity has the state information about acceptable sequence(s) of incoming message(s) from the other entity, the following simplifications can be made for the transitions from the global state: (1) multiple outstanding sends can then be modeled by a FIFO queue and received at the receiving site in the corresponding FIFO sequence; (2) no delay message (with an old sequence number) need be modeled; and (3) non-acceptable messages in the queue can be directly cleared and need not be dequeued.

The first simplification of Rule 6 is significant for protocols that allow multiple outstanding sends. As we explained before, the number of possible arrivals at the receiving entity will be 10! for 10 outstanding message sends. In many cases, we simply cannot afford to have such a non-polynomial complexity in the algorithm for generating the global state transitions. Accepting the outstanding sends in a FIFO sequence, on the other hand, means that there will be just one possible arrival sequence that need be checked from the validation point of view.

The modeling of delayed messages from a previous or current connection session is very tedious and complex. The second simplification of Rule 6 is of course important since it removes the requirement of considering them. The third simplification eliminates the need for modeling transitions in a global VA to dequeue non-acceptable elements from message queues.

As we explained in Cnapter 2, application layer protocols have the guarantee of an input sequence which is the original output sequence from the sender. This property is equivalent to having the sequence information about acceptable messages all the time. Therefore, the validation of all the application layers can be greatly simplified and no delayed messages need be modeled. The connection management protocol of TCP can also use this rule as we shall explain in Case 4 of the next section.

If entities A and B are communicating through a channel with variable delay and each entity does <u>not</u> have knowledge about the expected sequence of the incoming messages from the other entity, then we must moael the delay messages. The following rule can help us reduce the number of states required for modeling delays in the global VA [Sunshine78b].

Rule 7: All the delay messages M of entity A can be represented as Mx if they have the same validation effect and it is superfluous to distinguish them. In general, a mesage M is labelled according to the following notations for communicating entities A and B:

Ma	:	stands for a current message sent out from entity A.
Mb	:	stands for a current message sent out from entity B.
Mx	:	stands for a delay message sent out from entity A.
My	:	stands for a delay message sent out from entity B.

Rule 8: Delay messages may appear in a channel queue one message at a time and should be dequeued immediately.

Rule 9: If a channel queue has semantically homogeneous elements, the queue can be modeled by a stack which consists of identical elements.

If the data or control messages in any channel queue are homogeneous, their fetching sequences at the receiving site are irrelevant to the validation.

Rule 10: The elements in an acknowledgment or message queue should be fetched individually.

This rule helps us reduce the number of transitions in a global VA by eliminating the transition of the fetching of multiple elements in a queue. Although multiple acknowledgment or message delivery is perfectly legal and possible with respect to the actual implementation, it can be represented by a sequence of single receptions.

5.5 Global Validation Examples

In order to apply the VA model and various reduction rules described in the previous sections, we consider in this section four validation examples. For simplicity, we assume the perfect error detection of messages (i.e., we can always detect the message error and damage). We also assume that the protocols are operated under normal conditions without hardware failures.

(1) Case 1: Hajek's echoing protocol

This simple and special protocol introduced in [Lynch68, Hajek78] is homogeneous for entities A and B, and we only show the protocol program for A in Fig. 24. It uses an implicit timeout scheme and sends out a message when the timeout occurs. It uses a simple half-duplex channel for a point-to-point connection and allows single outstanding send. It is also assumed to always have a ready-to-be-sent message at each end.

```
EntityA:  % Echoing protocol
begin     % <msgBtoA>::= <verifyBtoA> <altBtoA>
          % <databtoA> <checkBtoA>
  altdA := altsA := Ø; gotackA := true;
  everfetchedA := false;
  while true do   % until doomsday
  begin   % begin receiver
    bufferinA := msgBtoA;    % got empty or invalid
                             % or   valid input
    msgBtoA := empty;   % message "removed" from
                        %B-to-A channel head
    if checkBtoA is valid then
    begin gotackA := verifyBtoA;
      if altBtoA ≠ altda then
      begin queueinA := dataBtoA;    % accepted
        altdA := 1 - altdA;
      end;
    end;   % end receiver; begin sender
    if gotackA = altsA then
    begin if everfetchedA then delete (queueoutA);
      altsA := 1 - altsA;
      bufferoutA := altdA & altsA & queueoutA &
                    checkAtoB;
      everfetchedA := true;
    end;
    msgAtoB := bufferoutA;    %sent
    % end sender
  end while;
end;
```

Fig. 24 Hajek's echoing protocol program

Since an echoing acknowledgment is used in the protocol, it does not have the piggyback interference problem. Both the message and the acknowledgment are fully retransmittable. According to reduction rule 5, it is not necessary to model the message loss and retransmission in the validation automaton for the protocol. We derive the VAs and the global VA in the following to reveal potential syntax errors.

```
<A> ::= (F.MSG_A.B Q.ACK_B.b) Q.MSG_B.A <A'>
  <A'> ::= (F.MSG_A.B Q.ACK_B.b)(F.ACK_A.a Q.MSG_B.A) <A'>
```

By following rules 2,3 and 5 we can derive the global validation automaton as shown in Fig. 25. By following rule 4, we can derive the simplified global VA as shown in Fig. 26. The global VA demonstrates several important correctness properties: there is no deadlock state and the entities are compatible.

The original validation scheme in [Hajek78] did not apply the reduction rules and therefore has more states and transitions than the global VA. However, the major simplification is possible only if the messages are fully retransmittable.

$$\begin{tikzcd}
 & \begin{Bmatrix} \epsilon & \epsilon \\ \epsilon & \epsilon \end{Bmatrix} \arrow[dl] \arrow[dr] & \\
\begin{Bmatrix} A & \epsilon \\ \epsilon & \epsilon \end{Bmatrix} \arrow[dr] \arrow[dd] & & \begin{Bmatrix} \epsilon & \epsilon \\ \epsilon & B \end{Bmatrix} \arrow[dl] \arrow[dd] \\
 & \begin{Bmatrix} A & \epsilon \\ \epsilon & B \end{Bmatrix} \arrow[dl] \arrow[dr] & \\
\begin{Bmatrix} \epsilon & a \\ \epsilon & B \end{Bmatrix} \arrow[rr, shift right] & & \begin{Bmatrix} A & \epsilon \\ b & \epsilon \end{Bmatrix} \arrow[ll, shift right]
\end{tikzcd}$$

Fig. 25 Global validation automaton for case 1.

$$\begin{tikzcd}
 & \begin{Bmatrix} \epsilon & \epsilon \\ \epsilon & \epsilon \end{Bmatrix} \arrow[dl] \arrow[dr] & \\
\begin{Bmatrix} A & \epsilon \\ \epsilon & \epsilon \end{Bmatrix} \arrow[rr, no head] \arrow[ddd] \arrow[dr] & & \begin{Bmatrix} \epsilon & \epsilon \\ \epsilon & B \end{Bmatrix} \\
 & \begin{Bmatrix} A & \epsilon \\ \epsilon & B \end{Bmatrix} \arrow[ddl] \arrow[ddr] & \\
 & & \\
\begin{Bmatrix} \epsilon & a \\ \epsilon & B \end{Bmatrix} \arrow[rr, no head] & & \begin{Bmatrix} A & \epsilon \\ b & \epsilon \end{Bmatrix}
\end{tikzcd}$$

Fig. 26 Simplified global VA of Fig. 25.

(2) Case 2: This protocol is similar to the protocol of case 1, but it has an explicit timeout retransmission mechanism and the messages are not always ready-to-be-sent. The transmission grammar for the protocol is:

```
<st> ::= "receive data" <st> | "send data" <wait>
  <wait> ::= "receive data" <wait> | "acknowledged" <st>
               | "retransmit" <wait>
```

Because of the full retransmission and echo acknowledgment, we need not model the retransmission and loss for the protocol. The TG for entity A can be rewritten as the following validation automaton:

```
<st> ::= F.MSG_A.B Q.ACK_B.b <st> | Q.MSG_B.A <wait>
  <wait> ::= F.MSG_A.B Q.ACK_B.b <wait> | F.ACK_A.a <st>
```

By following rules 2,3 and 5 we can derive the global validation automaton as shown in Fig. 27. By following rule 4 we can further simplify the global VA to be the one shown in Fig. 28.

$$\begin{tikzcd}[ampersand replacement=\&]
 \& \left\{\begin{matrix} \epsilon & \epsilon \\ \epsilon & \epsilon \end{matrix}\right\} \arrow[dl] \arrow[dr] \& \\
\left\{\begin{matrix} A & \epsilon \\ \epsilon & \epsilon \end{matrix}\right\} \arrow[dr] \arrow[dddd, bend right] \& \& \left\{\begin{matrix} \epsilon & \epsilon \\ \epsilon & B \end{matrix}\right\} \arrow[dl] \arrow[dddd, bend left] \\
 \& \left\{\begin{matrix} A & \epsilon \\ \epsilon & B \end{matrix}\right\} \arrow[dl] \arrow[dr] \& \\
\left\{\begin{matrix} A & \epsilon \\ b & \epsilon \end{matrix}\right\} \arrow[uu] \arrow[dr] \& \& \left\{\begin{matrix} \epsilon & a \\ \epsilon & B \end{matrix}\right\} \arrow[uu] \arrow[dl] \\
 \& \left\{\begin{matrix} \epsilon & a \\ b & \epsilon \end{matrix}\right\} \arrow[dl] \arrow[dr] \& \\
\left\{\begin{matrix} \epsilon & a \\ \epsilon & \epsilon \end{matrix}\right\} \arrow[dr] \arrow[uurr, bend right] \& \& \left\{\begin{matrix} \epsilon & \epsilon \\ b & \epsilon \end{matrix}\right\} \arrow[dl] \arrow[uull, bend left] \\
 \& \left\{\begin{matrix} \epsilon & \epsilon \\ \epsilon & \epsilon \end{matrix}\right\} \&
\end{tikzcd}$$

Fig. 27 Global validation automaton for case 2.

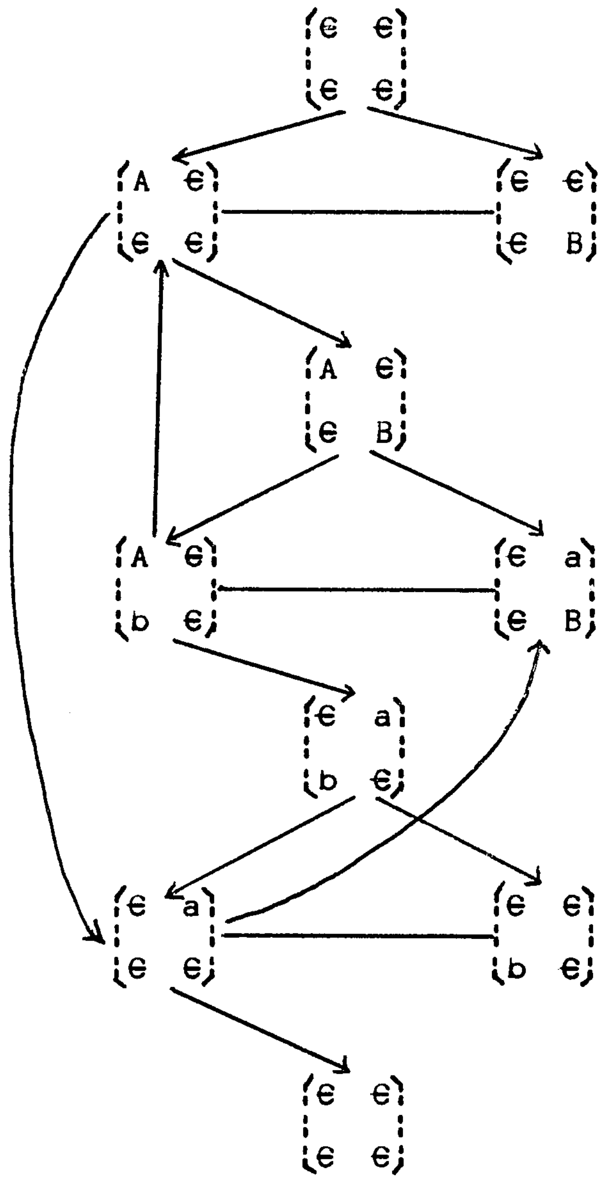


Fig. 28 Simplified global VA of Fig. 27.

The global state diagram for case 2 protocol demonstrates several aspects of protocol correctness. The entities are compatible and there is no deadlock in the global state diagram.

Since the protocol in case 2 is homogeneous and the message flow in one direction does not interfere with the message flow in the other direction, we can apply rule 4.2 to simplify the model by checking only one direction of message flow. We then need only to model the global validation automaton for sendA (send VIP of entity A) and receivB (receive VIP of entity B). The global VA for sendB and receiveA is for the other direction of message flow. The global VA has therefore been separated into two validation independent parts as shown in Fig. 29 and each part models one direction of message flow.

The VAs for sendA and receiveB are shown below:

```
<sendA> ::= Q.MSG_B.A <waitA>
  <waitA> ::= F.ACK_A.a <sendA>
<receiveB> ::= F.MSG_B.A Q.ACK_A.a <receiveB>
```

```
          Entity A      :    Entity B
                        :
                        :
   ---------------------:-------------------
  :    -----------      :     -----------    :
  :   : sendA      :    :    : receiveB  :   :
  :    -----------      :     -----------    :
   ---------------------:--------------------
                        :
   ---------------------:--------------------
  :    -----------      :     -----------    :
  :   : receiveA   :    :    :  sendB    :   :
  :    -----------      :     -----------    :
   ---------------------:--------------------
```

Fig. 29 Global validation independent parts.

We can derive the global VA of sendA and receiveB, A1, as shown in the upper left part of Fig. 30, and the global VA of sendB and receiveA, A2, is shown in the upper right part of Fig. 30. The global VA of entity A and B is the arbitrary shuffle product of A1 and A2, and is shown in the lower part of Fig. 30. It is the same as the global VA of Fig. 25 which represents the global interactions of the full-duplex channel.

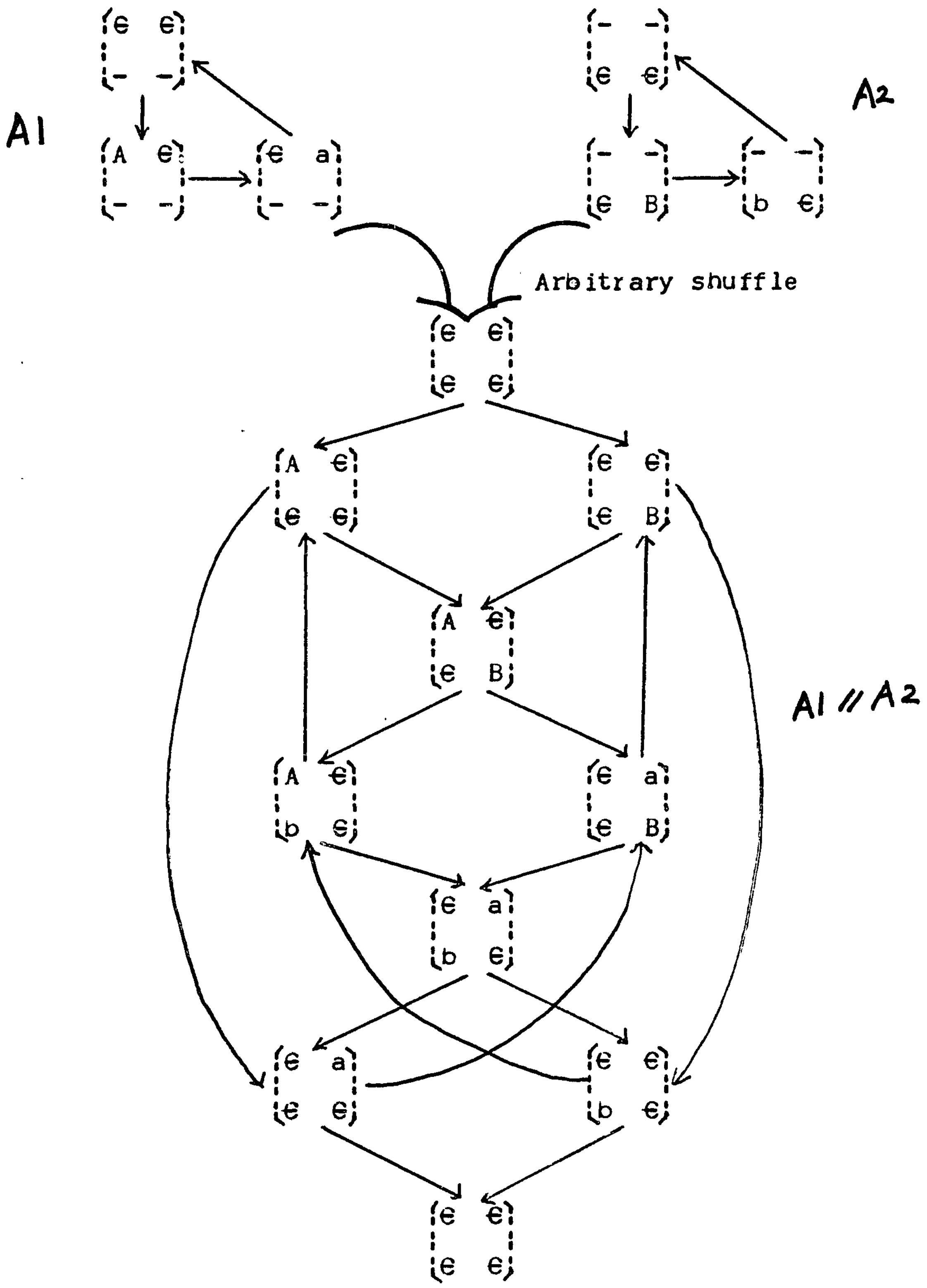


Fig. 30 Global arbitrary shuffle operation

Specificly, we can formally write the global VA for sendA and receiveB as automaton A1:

A1 = (K1,f1,[ϵ ϵ],F1)

where K1 = { [ϵ ϵ],[A ϵ],[ϵ a] }
F1 = { [ϵ ϵ] }

and f1 is defined as follows:

```
f1 ( [ϵ ϵ] ) = ( [A ϵ] )
f1 ( [A ϵ] ) = ( [ϵ a] )
f1 ( [ϵ a] ) = ( [ϵ ϵ] ),
```

The global VA for sendB and receiveA is written as automaton A2:

A2 = (K2,f2,[ϵ ϵ],Fa)

where K2 = { [ϵ ϵ],[ϵ B],[b ϵ] }
F2 = { [ϵ ϵ] }

and f2 is defined as follows:

```
f2 ( [ϵ ϵ] ) = ( [ϵ B] )
f2 ( [ϵ B] ) = ( [b ϵ] )
f2 ( [b ϵ] ) = ( [ϵ ϵ] ),
```

The global VA for entity A and B is therefore A1//A2:

A1//A2 = (K1 x K2,f3,[[ϵ ϵ],[ϵ ϵ]],F3)

```
where   K1 x K2 = ( [[ϵ ϵ][ϵ ϵ]],[[ϵ ϵ][ϵ B]],[[ϵ ϵ][b ϵ]],
                    [[A ϵ][ϵ ϵ]],[[A ϵ][ϵ B]],[[A ϵ][b ϵ]],
                    [[ϵ a][ϵ ϵ]],[[ϵ a][ϵ B]],[[ϵ a][b ϵ]] )
        F3 = ( [[ϵ ϵ][ϵ ϵ]] )
```

and f3 is defined as follows:

```
          f3 ( [[ϵ ϵ],qi] ) = ( [[A ϵ],qi] )
          f3 ( [[A ϵ],qi] ) = ( [[ϵ a],qi] )
          f3 ( [[ϵ a],qi] ) = ( [[ϵ ϵ],qi] )      ∀ qi ϵ K2

   and    f3 ( [pi,[ϵ ϵ]] ) = ( [pi,[ϵ B]] )
          f3 ( [pi,[ϵ B]] ) = ( [pi,[b ϵ]] )
          f3 ( [pi,[b ϵ]] ) = ( [pi,[ϵ ϵ]] )      ∀ pi ϵ K1
```

(3) Case 3: Multiple outstanding sends

As we have discussed in Chapter 3, some powerful protocols have a feature to allow several sending packets outstanding. This type of protocol can be modeled by a type-2 transmission grammar or a push-down automaton. The protocol in this case is homogeneous and full-duplex. It is validation independent for each direction of message flow and the elements in each queue are homogeneous. From reduction rules 3, 4, 4.1, 5 and 9, we can use the following validation automata for sendA and receiveB.

<sendA> ::= Q.MSG_B.A <sendA> F.ACK_A.a <sendA> | ϵ

<receiveB> ::= F.MSG_B.A Q.ACK_A.a <receiveB> | ϵ

From these VAs we can get the global VA for the direction of message flow from entity A to entity B as shown in Fig. 30. Because of the boundness of protocols, the number of outstanding messages has to be limited in the protocol. The figure shows that entity A can have at most n messages outstanding at any time. (We have to use a context-sensitive grammar to specify the global VA if the number of outstanding messages is unbounded.) The figure also shows that (1) when a message is sent from entity A, the state moves downward one level; (2) when an acknowledgment is received by entity A, the state moves upward one level;

(3) when a message is received by entity B, the state moves one step to the right; and (4) the total number of elements in both queues is equal to the number of outstanding messages, and the cardinality is the same for all the states at the same level.

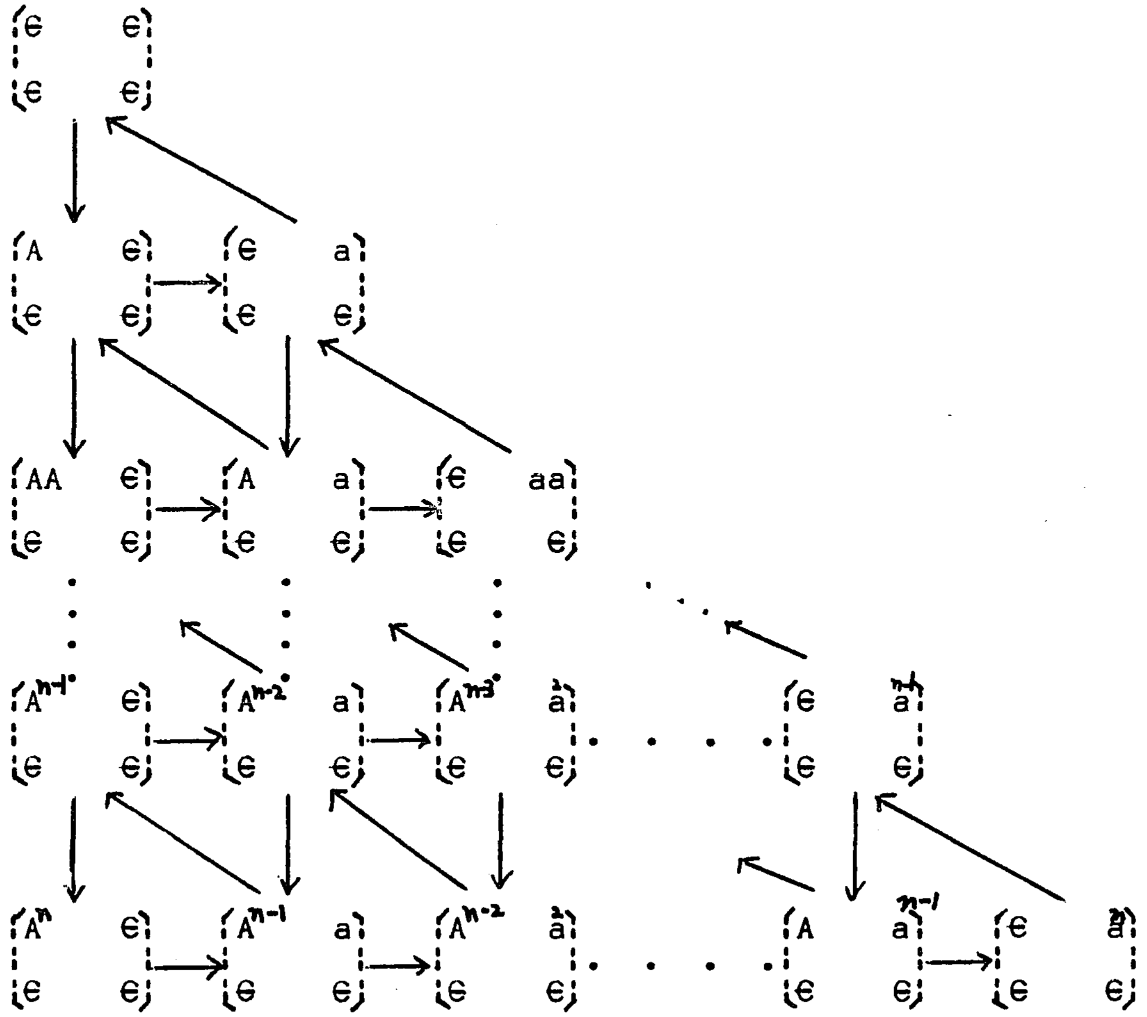


Fig. 31 Global VA for one-way message flow of case 3.

The global VA in Fig. 31 also demonstrates several aspects of protocol correctness for a half-duplex channel. There is no deadlock state and the entities are compatible. The protocol will eventually terminate at the state on the top level after all the messages have been sent and acknowledged.

The following context-sensitive grammar can be used to specify the global VA which defines all the global action sequences for an unbound number of outstanding sends:

```
<Exchange> ::= <Qm><Fm><Qa><Fa><Exchange> | ε
<X><Qm> ::= <Qm><X>  ∀ <X> e (<Fm>,<Qa>,<Fa>)
<Y><Fm> ::= <Fm><Y>  ∀ <Y> e (<Qa>,<Fa>)
<Z><Qa> ::= <Qa><Z>  ∀ <Z> e (<Fa>)
<Qm> ::= Q.MSG_B.A
<Fm> ::= F.MSG_B.A
<Qa> ::= Q.ACK_A.a
<Fa> ::= F.ACK_A.a
```

The global VA for the full-duplex channel can also be derived by the arbitrary shuffle operation of the global VIPs. The shuffled global VA will be correct because it preserves the action sequences and the verified properties of the individual global VAs.

(4) Case 4: Connection setup for variable delay channels.

Since a logical connection between two entities of the host/host layer in a computer network requires significant resources, it is desirable to maintain a connection only if the entities are communicating. This requires mechanisms for opening a connection when needed.

In a packet switching network, the mechanisms should also handle delay and failure. When a connection exists, a sequence checking mechanism can be used to reject duplicate messages from a previous closed connection or the current connection. If no connection exists, then no duplicate from a previous closed connection should cause a connection to be opened and duplicate data accepted by the next layer. The sequence checking mechanism used when a connection exists cannot be used if no connection exists, because the receiver does not have the required sequence information to check the incoming message. We can use a three-way-handshake procedure to protect against the interference of packets in an old connection by not allowing data to be passed to the next layer until the successful exchange of three messages [Cerf77,Fletcher78]. It works as follows:

Assume that an entity A wishes to communicate with an entity B in a full-duplex channel and <SYN> is the control packet to open a connection.

(1) A sends B a <SYN> packet with an initial sequence number a.
(2) B acknowledges the receipt of the sequence number a in a <SYN-ACK> packet which contains its own initial sequence number b.
(3) A in a third packet acknowledges receipt of B's initial sequence number b.

The reason for the three-way-handshake can be seen by assuming that the initial message from A is an old <SYN>x packet with sequence number x. B has no way of knowing this, so it responds <SYN-ACK>b with sequence number b and acknowledges the packet x. However, when A gets B's acknowledgment of the old opening (packet x), it can recognize that B is not acknowledging a valid sequence number. It can then reply with a reset signal <RST>x to break the connection rather than an acknowledgment of B's initial sequence number, b.

There are many subtleties with the three-way-handshake procedures, such as what happens if any of the three messages gets lost, if both sides receive old opening packets, or if both A and B try to open simultaneously. We would also like to know if this procedure will lead to deadlock or incorrect establishment. In order to formally study the three-way-handshake protocol, we first specify a TG for the TCP connection protocol as shown in Fig. 32. We can then derive the validation automata for both A and B from the TG, and check the global VA of the VAs.

The VAs in Fig. 33 illustrate local state changes of entities A and B, but contain neither the interfaces with higher-layer entities nor actions which are irrelevant to the state changes. Notice that the VAs are homogeneous and are very similar to the current version of connection management protocols of the TCP in ARPANET. The transmission grammar describing TCP has been shown in Chapter 3.

To model A, we have to model old messages for the channel where variable message delays are possible. The message elements are <RST>, <SYN-ACK> and <SYN> which are represented as R, A and S respectively in Fig. 33. <SYN> is a control message used at the initiation of a connection to indicate where the sequence numbering will start. <SYN-ACK> is a control message to acknowledge a received <SYN>. <RST> is a reset control message, indicating that the receiver should delete the current connection without further interactions. The receiver can determine, based on the sequence and acknowledgment fields of the incoming message, whether it should honor the reset command or ignore it.

```
<closedA> ::= <receiveA> ¦ <sendA>
            ¦ "RCV.<rst>" <closedA>
            ¦ <receive.old.synack> <closedA>
  <receiveA> ::= <receive.syn> <wait.ack>
    <receive.syn> ::= "RCV.<syn>" "RSND.<synack>"
    <wait.ack> ::= "RCV.<ack>" ¦ "RCV.<rst>" "clear"
                   <closedA>
  <sendA> ::= "RSND.<syn>" <wait.synack>
    <wait.synack> ::= <receive.synack> ¦ <collision>
                    ¦ <receive.old.synack> <wait.synack>
                    ¦ <syn.rejection> <closedA>
      <receive.synack> ::= "RCV.<synack>" "RSND.<ack>"
      <collision> ::= "RCV.<syn>" "clear" "RSND.<rst>"
      <syn.rejection> ::= "RCV.<rst>" "clear"
  <receive.old.synack> ::= "RCV.<old.synack>" "RSND.<rst>"
```

Fig. 32 TG for TCP connection protocols

```
<closedA> ::= <receiveA> | <sendA>
            | F.MSG_A.R <closedA>
            | <receive.old.synack> <closedA>
  <receiveA> ::= <receive.syn> <wait.ack>
               | Q.MSG_A.Sy <receive.old.syn> <wait.rejection>
    <receive.syn> ::= F.MSG_A.Sb Q.MSG_B.Aa Q.SN_A.a
                      Q.ACK_B.b Q.RN_A.b (F.MSG_A.R)
    <wait.ack> ::= F.ACK_A.a
    <receive.old.syn> ::= F.MSG_A.Sy Q.MSG_B.Aa Q.SN_A.a
                          Q.ACK_B.y Q.RN_A.y (F.MSG_A.R)
    <wait.rejection> ::= F.MSG_B.Ry C.*A <closedA>
  <sendA> ::= Q.MSG_B.Sa Q.SN_A.a <wait.synack>
    <wait.synack> ::= <receive.synack> | <collision>
                    | <receive.old.synack> <wait.synack>
                    | <syn.rejection> <closedA>
      <receive.synack> ::= F.MSG_A.Ab F.ACK_A.a
                           Q.ACK_B.b Q.RN_A.b
      <collision> ::= <current.collision> <closedA>
                    | Q.MSG_A.Sy <old.collision> <closedA>
        <current.collision> ::= F.MSG_A.Sb C.*A Q.MSG_B.R
        <old.collision> ::= F.MSG_A.Sy C.*A
                            (F.ACK_A.a Q.ACK_A.x)
                            (F.RN_B.a Q.RN_B.x)
                            (N.MSG_A.a E.ACK_A Q.MSG_B.R)
      <syn.rejection> ::= F.MSG_A.R C.*A <closedA>
  <receive.old.synack> ::= F.MSG_A.Ab F.ACK_A.x Q.MSG_B.Rx
```

Fig. 33 Local VAs for TCP connection protocols
(Continued on the next page)

```
<closedB> ::= <receiveB> ¦ <sendB>
            ¦ F.MSG_B.R <closedB>
            ¦ <receive.old.synack> <closedB>
  <receiveB> ::= <receive.syn> <wait.ack>
               ¦ Q.MSG_B.Sx <receive.old.syn> <wait.rejection>
    <receive.syn> ::= F.MSG_B.Sa Q.MSG_A.Ab Q.SN_B.b
                      Q.ACK_A.a Q.RN_B.a (F.MSG_A.R)
    <wait.ack> ::= F.ACK_B.b
    <receive.old.syn> ::= F.MSG_B.Sx Q.MSG_A.Ab Q.SN_B.b
                          Q.ACK_A.x Q.RN_B.x (F.MSG_B.R)
    <wait.rejection> ::= F.MSG_B.Rx C.*B <closedB>
  <sendB> ::= Q.MSG_A.Sb Q.SN_B.b <wait.synack>
    <wait.synack> ::= <receive.synack> ¦ <collision>
                    ¦ <receive.old.synack> <wait.synack>
                    ¦ <syn.rejection> <closedB>
      <receive.synack> ::= F.MSG_B.Aa F.ACK_B.b
                           Q.ACK_A.a Q.RN_B.a
      <collision> ::= <current.collision> <closedB>
                    ¦ Q.MSG_B.Sx <old.collision> <closedB>
        <current.collision> ::= F.MSG_B.Sa C.*B Q.MSG_A.R
        <old.collision> ::= F.MSG_B.Sx C.*B
                            (F.ACK_B.b Q.ACK_B.y)
                            (F.RN_A.b Q.RN_A.y)
                            (N.MSG_B E.ACK_B Q.MSG_A.R)
      <syn.rejection> ::= F.MSG_B.R C.*B <closedB>
  <receive.old.synack> ::= F.MSG_B.Aa F.ACK_B.y Q.MSG_A.Ry
```

Fig. 33 Local VAs for TCP connection protocols

The VAs in Fig. 33 are modeled to reduce the number of states and transitions in their global VA. The sequence numbers are labeled according to reduction rule 7, and actions are combined into an indivisible action. Retransmissions need not be modeled according to rule 6.

The VAs in Fig. 33 also contain other specialized reduction techniques which are suitable for the connection part of TCP. We model the state information of the send sequence number (SN) and the expected receive sequence number (RN) as queues in the VAs. We can represent the global state of A and B by the following format.

```
SN_A{MSG_B    ACK_A}RN_B
    :              :
RN_A{ACK_B    MSG_A}SN_B
```

SN_A is the queue which has the send sequence number of entity A and RN_A is the queue which has the expected receive sequence number of entity A. RN_B and SN_B have similar meanings. The sequence information is an important factor during global state transitions. From the VAs we can derive the global VA as shown in Fig. 39. We explain the global state diagram of Fig. 39 by the following cases of transition traces.

The normal case of the three-way-handshake is shown in Fig. 34. This figure should be interpreted in the following way. Entity A has four channel queues, MSG_B, ACK_B, RN_A and SN_A. Entity B has four queues, MSG_A, ACK_A, RN_B and SN_A. The empty sign (ε) indicates that the queue is empty. Messages of the first row flow from left to right while acknowledgments flow from right to left. Starting from initial closing states at both VAs, A begins by sending a <SYN>a (Sa as its abbreviation) indicating that it will use sequence numbers starting with sequence number a. B then takes the message from queue DATA_B, sends a <SYN-ACK>b (Ab as its abbreviation) and acknowledges the <SYN> received from A by sending acknowledgment a in ACK_A. In the next state, A fetches acknowledgment a and <SYN-ACK>b from the queues and responds with acknowledgment b in queue ACK_B. After B fetches acknowledgment b from ACK_B, the correct connection will be established.

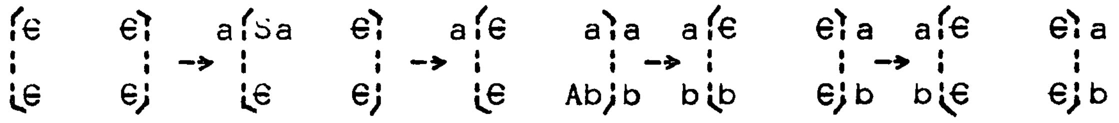

e e a Sa e a e a a a e e a a e e a
e e e e e Ab b b b e b b e e b

Fig. 34 Basic 3-way handshake for connection

Simultaneous opening will cause collision. When an entity detects collision it clears state information and sends a <RST> (R as its abbreviation) to reset the other side as shown in Fig. 35.

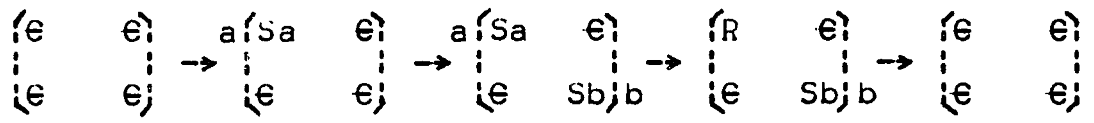


Fig. 35 A recovery from simultaneous openings

The principle reason for the three-way-handshake is to prevent old duplicate connection openings from causing an interference error. Two simple cases of recovery from an old (duplicate) <SYN> message are shown in Fig. 36 and 37.

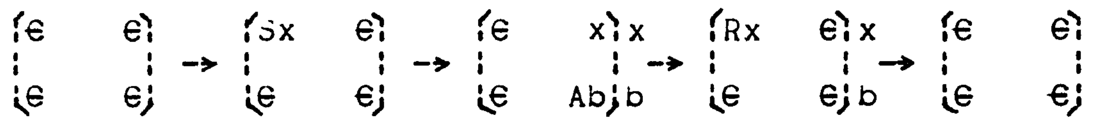


Fig. 36 A recovery from an old <SYN>x

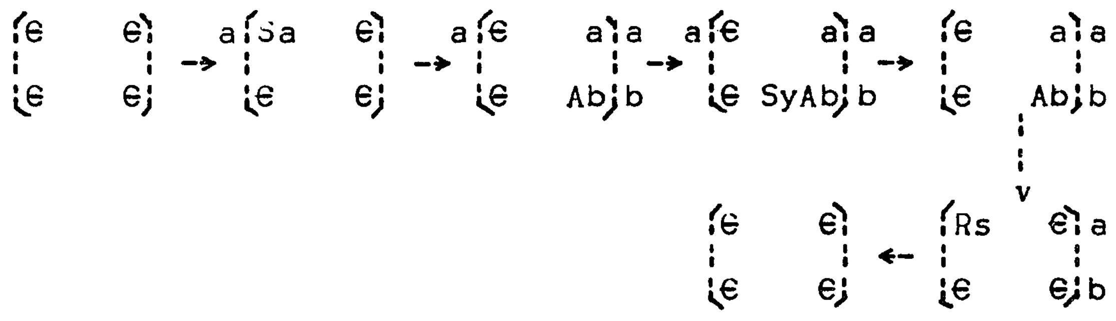


Fig. 37 A recovery from an old <SYN>y

A complicated case of recovery from two old <SYN> duplicates is shown in Fig. 38.

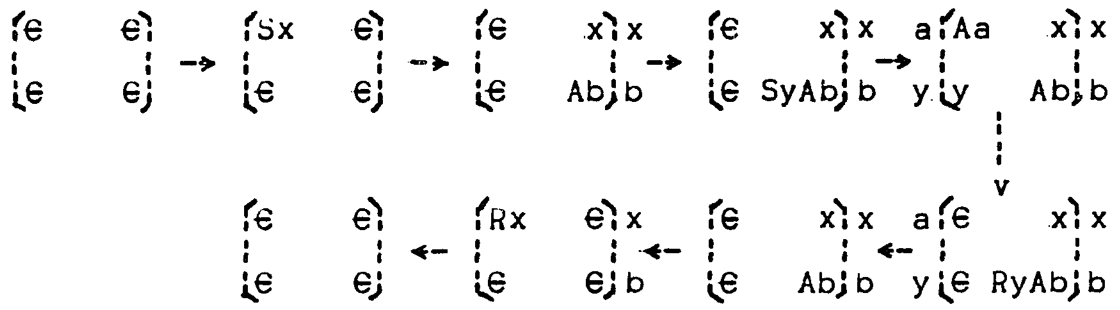


Fig. 38 A recovery from old <SYN>x and <SYN>y

Automatic techniques can be used to read the VAs of A and B and generate the global VA. Fig. 39 shows the global VA which contains all the transitions we want to validate following the reduction rules of this chapter. The global VA model demonstrates several aspects of protocol correctness: (1) there is no deadlock; (2) all the states except the terminate state can reach the initial state; and (3) the only terminate state is the state where both A and B are at the correctly synchronized local states. These results show the sufficiency of the three-way-handshake connection technique. Those global states which have queues containing more than 2 elements are omitted in Fig. 39 to cut down the size of the figure.

Three messages:

S—<SYN>
R—<RST>
A—<SYN-ACK>

Sequence numbers:

Ma—Current msg from A
Mb—Current msg from B
Mx—Old msg from A
My—Old msg from B

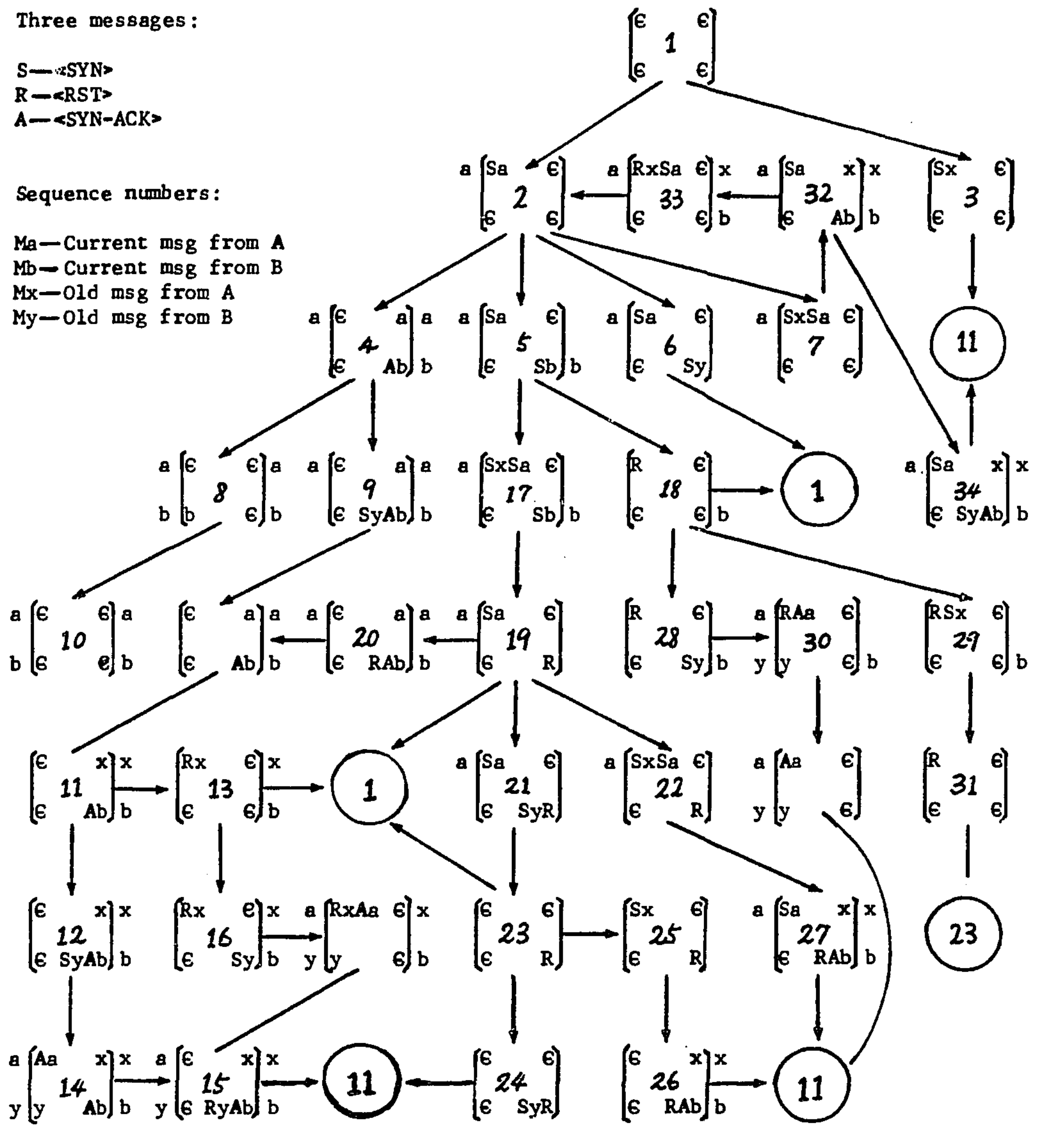


Fig.39 Global VA for TCP connection setup

The validation system in Appendix A can also be used to automate the reachability analysis of the global VA in Fig. 39. Fig. 40 is the output of the reachability analysis by the system. It shows that the proper terminating state of the global VA is state 10 as shown in Fig. 39. It also shows that there is no deadlock, incompatibility or looping in the global VA. The format of the output of Fig. 40 is the same as the output of local validations.

```
<1> ::= 1 <2> ; 1 <3> .
   |<2> ::= 2 <4> ; 2 <5> ; 2 6 <1> ; 2 7 <32> .
   |  |<4> ::= 4 8 10 ; 4 9 <11> .
   |  |  <11> ::= 11 12 14 15 <11> ; 11 <13> .
   |  |     <13> ::= 13 <1> ; 13 16 15 <11> .
   |  |<5> ::= 5 <17> ; 5 <18> .
   |  |  |<17> ::= 17 <19> .
   |  |  |  <19> ::= 19 <1> ; 19 20 <11> ;
   |  |  |                19 21 <23> ; 19 22 27 <11> .
   |  |  |     <23> ::= 23 <1> ; 23 24 <11> ;
   |  |  |                23 25 26 <11> .
   |  |  |<18> ::= 18 <1> ; 18 28 30 <11>
   |  |               ; 18 29 31 <23> .
   |  |<32> ::= 32 33 <2> ; 32 34 <11> .
   |<3> ::= 3 <11> .
 ------------------------------------------------------
  PROPER TERMINATION ANALYSIS
  ***************************
  THE TERMINATE SYMBOL IS: 10
  NO LEFT RECURSION
  REACHABILITY ANALYSIS
  *********************
  NO UNREACHABLE NONTERMINAL
  ALL NONTERMINALS CAN TERMINATE PROPERLY.
  ACTION SEQUENCE TESTING
  ***********************
 <1>
 1 2 4 8 10 .
  THE SYNTAX OF THE ABOVE ACTION
                   SEQUENCE IS: CORRECT
  INDENTED CYCLES LISTING
  ***********************

<1> ::= 1 <2>
   |<2> ::= 2 <4>
   |    <4> ::= 4 9 <11>
   |        <11> ::= 11 <13>
   |            <13> ::= 13 <1>
   |
   |<2> ::= 2 <5>
   |   |<5> ::= 5 <17>
   |   |    <17> ::= 17 <19>
   |   |       |<19> ::= 19 <1>
   |   |       |
   |   |       |<19> ::= 19 20 <11>
   |   |       |    <11> ::= 11 <13>
   |   |       |        <13> ::= 13 <1>
   |   |       |
   |   |       |<19> ::= 19 21 <23>
   |   |       |    <23> ::= 23 <1>
   |   |       |
   |    |      |    |<23> ::= 23 24 <11>
   |    |      |    |   <11> ::= 11 <13>
   |    |      |    |       <13> ::= 13 <1>
   |    |      |    |
   |    |      |    |<23> ::= 23 25 26 <11>
   |    |      |       <11> ::= 11 <13>
   |    |      |           <13> ::= 13 <1>
   |    |      |
   |    |      |<19> ::= 19 22 27 <11>
   |    |          <11> ::= 11 <13>
   |    |              <13> ::= 13 <1>
   |    |
   |    |<5> ::= 5 <18>
   |       |<18> ::= 18 <1>
   |       |
   |       |<18> ::= 18 28 30 <11>
   |       |   <11> ::= 11 <13>
   |       |       <13> ::= 13 <1>
   |       |
   |       |<18> ::= 18 29 31 <23>
   |           |<23> ::= 23 <1>
   |           |
   |           |<23> ::= 23 24 <11>
   |           |    <11> ::= 11 <13>
   |           |        <13> ::= 13 <1>
   |           |
   |           |<23> ::= 23 25 26 <11>
   |                <11> ::= 11 <13>
   |                    <13> ::= 13 <1>
   |
   | <2> ::= 2 6 <1>
   |
   | <2> ::= 2 7 <32>
        <32> ::= 32 34 <11>
            <11> ::= 11 <13>
                <13> ::= 13 <1>

<1> ::= 1 <3>
    <3> ::= 3 <11>
        <11> ::= 11 <13>
            <13> ::= 13 <1>

<2> ::= 2 7 <32>
    <32> ::= 32 33 <2>

<11> ::= 11 12 14 15 <11>

<11> ::= 11 <13>
    <13> ::= 13 16 15 <11>
```

Fig. 40 Validation of the global VA in Fig. 39

CHAPTER 6

PROTOCOL IMPLEMENTATIONS

We know that a transmission grammar is a control sequence specification of a protocol to regulate communication between communicating entities. The purpose of this chapter is to generate an algorithm that will encode the transmission grammar description of any protocol into a form that is suitable for automatic software/hardware implementations. Through the grammar model, the "syntax" and "semantics" of various protocols can be specified and then be executed by either software or hardware. This chapter starts with the discussion of a design and implementation methodology for a basic data transfer protocol. The discussion then leads to an idea of syntax-directed protocol implementations. A framework for automatic software/hardware implementations of protocols has also been constructed.

6.1 A Protocol Design Methodology

In this section, we present a step-wise methodology for protocol design and implementation using our generalized TG model. A typical protocol at the communication processor layer in a packet switching network is used as an illustrative example.

In general the entity at the communication processor layer (e.g., the interface message processor (IMP) in the ARPANET [Heart70]) mainly consists of two validation independent parts (VIPs), i.e. the sender and the receiver. We will define several sets of terminal actions at different levels of abstraction for both parts in order to design the TGs for the the VIPs of the sender and receiver. The TG integration techniques are then applied to combine the TGs.

(1) Step 1: Define the message grammar.

Since transmission actions have message units as their operands, we need to define the message grammar in order to complete the protocol specification. The message grammar enables us not only to represent the hierarchical structure of the message format, but to implement the automatic syntax parsing and error handling. We feel that the generalized TG specification is especially suitable for the design of higher-layer protocols. Fig. 41 shows the message grammar for the communication processor entity.

CONSTANTS:

```
  letter.size = (MAXIMUM   OF CHARACTERS IN <letter>)
  message.size = (MAXIMUM   OF CHARACTERS IN <message>)
  timeout = 100 msec
  local.hosts = (SET OF LOCAL HOST NUMBERS)
  ack = (CONSTANT REPRESENTING ACK)
  channel.i: (CHANNEL NUMBER REPRESENTING RECEIVE PORT)
```

VARIABLES:

```
  user.no: (24 BIT USER NO UNIQUELY IDENTIFY ING EACH USER)
  host.no: (8 BIT HOST IDENTIFY NO FOR user.no)
  size,source,dest,link.no,msg.no: INTEGER
  send.channel: (CHANNEL NUMBER REPRESENTING SEND PORT)
  complete.flag: BOOLEAN
```

MESSAGE GRAMMAR:

```
<message(B)>  ::=  <msg header> <msg text>
  <msg header>  ::=  <msg leader> size
      <msg leader>  ::=  source dest link.no msg.no
  <msg text>  ::= <text>
    <text> ::= (character)*


<packet> ::= <ACK> | <local packet> | <routing packet>
  <ACK> ::= ack <packet header>
    <packet header>  ::=  msg.no source dest link.no
                          packet.no
  <local packet> ::= <local host> <packet header> <letter>
    <local host> ::= ANY LOCAL HOST
    <letter(B)> ::= (character)*
  <routing packet> ::= dest <packet header> <letter>
```

FIG. 41 Message grammar for IMP

(2) Step 2: Define the abstract action grammar.

Fig. 42 shows the action grammar for the communication processor entity at an abstract level. By the induction rule, we can prove that in the sender part, <send> will always generate the action sequences that have an equal number of <send packet> and "acknowledgment". Both VIP parts can be easily understood due to the simplicity of the model.

```
<send VIP> ::= <send> <send VIP>
  <send> ::= <send packet> <wait ACK> ¦ ϵ
    <send packet> ::= "send routing packet"
                    ¦  "send host packet"
    <wait ACK> ::= "acknowledgment"
                 ¦  "retransmit" <wait ACK>
                 ¦  <send> <wait ACK>

<receive VIP> ::= <receive> <receive VIP>
                ¦  "send message to host" <receive VIP>
  <receive> ::= "receive packet" ¦ "receive message"
```

Fig. 42 Action grammar for IMP entity at an abstract level

(3) Step 3: Validate protocol properties

We can check the protocol correctness requirement for this architecture and use both local and global validation techniques to validate the required correctness properties. The validation automata for the protocol are easier to construct from the abstract TG. The global VA of the VAs can also be constructed, which is similar to the structure of example 3 in Chapter 5.

(4) Step 4: Refinement of the action grammar.

For a more detailed description of the protocol, we can define the following list of actions (see Table 4) for both sender and receiver parts, and then use the substitution operation to substitute non-terminals for the terminals of the model in Fig. 42 while describing these new non-terminals by the terminal actions of the refined (detailed) level. The substituted grammar is shown in Fig. 43.

Table 4 Actions of IMP protocol at a more detailed level

Messages and Buffers:

Ui: packet Unit
Pb,ACKb,Rb: Packet buffer, ACK buffer and
Reassembly buffer

Sender:
Ss: "allocate Space Pb for sender"
Gs: "Generate host message Ui, put in Pb"
Sn: "Send host message Ui Never transmitted before"
As: "find Acknowledgment for an outstanding Ui in ACKb"
Ro: "get Routing Ui not destined here"
To: "Time Out for Ui"
Ss: "Send Ui out lower level channel"
Rt: "Reset the time for Ui"
St: "Set the time for Ui"
Fs: "Free Pb, ACKb space for Ui"

Receiver:
Rn: "Receive Host message"
Ra: "Receive Acknowledgment for Ui"
Rl: "Receive Ui destined here"
Rr: "Receive Ui not destined here"
Ar: "transmit Acknowledgment out to lower level channel"
Cr: "Consume Ui in Pb and put it in Rb"
Fu: "Free Pb space Unit for Ui"
Sh: "Send all Ui's in Rb to host"
Fr: "Free Receive buffer Rb"

```
<send VIP> ::= <send> <send VIP>
 <send> ::= <send packet> <wait ACK> ¦ ϵ
   <send packet> ::=    <send routing packet>
                      ¦ <send host packet>
     <send routing packet> ::= Ss Ro Sn St Ss
     <send host packet> ::= Ss Gs Sn St Ss

   <wait ACK> ::= <acknowledgment>
                ¦ <retransmit> <wait ACK>
                ¦ <send> <wait ACK>
     <acknowledgment> ::= As Rt Fs
     <retransmit> ::= To Rt St Ss

<receive VIP> ::= <receive> <receive VIP>
                ¦ <send message to host> <receive VIP>
  <receive> ::= <receive packet> ¦ <receive message>
    <receive packet> ::= <receive ACK> ¦ <receive local>
                       ¦ <receive routing>
      <receive ACK> ::= Ra
      <receive local> ::= Rl Ar Cr Fu
      <receive routing> ::= Rr Ar
    <receive message> ::= Rh
  <send message to host> ::= Sh Fr
```

FIG. 43 Action grammar of IMP at a more detailed level

(5) Step 5: Integrate VIP action grammars.

To integrate the VIP TGs, we can use the arbitrary snuffle operation [Teng78a] to combine these two logically independent parts together. The arbitrary shuffle operation of two languages is similar to an arbitrary shuffling of two decks of cards (each deck of cards corresponds to a sentence of a distinct language). In order to ensure the integrity of the data and variables of both parts, we should keep a set of variables for each part or store the previous values of variables if the incoming action belongs to a different VIP. We may also combine a cluster of actions in each VIP into an indivisible action, thus preventing the development of a race condition between the VIPs.

For the purpose of simplicity, we will shuffle the VIP TGs of Fig. 42. In this case, we assume that each terminal in Fig. 42 is a cluster of actions from Fig. 43 which must execute together. Furthermore, we assume that the actions have the following priority sequences: "receive packet", "receive message", "acknowledgment", "retransmit", "send routing packet", "send host packet" and "send message to host". We can thus easily combine the two TGs in Fig. 42 to give the combined TG for the protocol, as shown in Fig. 44.

```
<TG> ::= <receive> <TG> | <send> <TG>
 <receive> ::= "receive packet" | "receive message"
 <send> ::= <receive> <send> | "send packet" <wait ACK>
          | "send message to host" | ε
   <wait ACK> ::= <receive> <wait ACK>
                | "acknowledgment"
                | "retransmit" <wait ACK>
                | <send> <wait ACK>
```

FIG. 44 Integrated action grammar of Fig. 42

The integrated action grammar of Fig. 44 is shown in Fig. 45 at a more detailed level by applying the substitution operation directly to the TG of Fig. 44 (i.e. Step 3). The same could be achieved if we had directly applied the shuffle operation to the VIP TGs of Fig. 43 (i.e., Step 44); however, the resulting combined TG would be much more complicated than that of Fig. 45.

```
<TG> ::= <receive> <TG> ¦ <send> <TG>
  <receive> ::= <receive packet> ¦ <receive message>
    <receive packet> ::= <receive ACK> ¦<receive local>
                       ¦ <receive routing>
      <receive ACK> ::= Ra
      <receive local> ::= Rl Ar Cr Fu
      <receive routing> ::= Rr Ar
    <receive message> ::= Rh
  <send> ::= <receive> <send>
           ¦ <send packet> <wait ACK>
           ¦ <send message to host> ¦ ϵ
    <send packet> ::= <send routing packet>
                    ¦ <send host packet>
      <send routing packet> ::= Ss Ro Sn St Ss
      <send host packet> ::= Ss Gs Sn St Ss
    <wait ACK> ::= <receive> <wait ACK>
                 ¦ <acknowledgment>
                 ¦ <retransmit> <wait ACK>
                 ¦ <send> <wait ACK>
      <acknowledgment> ::= As Rt Fs
      <retransmit> ::= To Rt St Ss
    <send message to host(OH)> ::= Sh Fr
```

FIG. 45 Action grammar of Fig. 44 at a detailed level

(6) Step 6: Syntax-directed implementation.

For some time, it has been recognized that a context-free grammar is a good meta-language for the syntactic description and specification of a programming language. Various top-down and bottom-up parsing techniques [Gries71] have been utilized for programming language compilers. Deterministic top-down parsing appears to have a variety of attractive features for the recognition of TG action sequences and message structures, and could be applied to parse both the message grammar and the action grammar.

The transmission grammar description of communication protocols therefore leads naturally to syntax-directed protocol program implementations. In the next section, we present the general idea of protocol implementations.

6.2 Syntax-Directed Implementations

When we discuss protocol implementations, we usually have an architectural block diagram of the protocol in our mind. The block diagram usually consists of I/O units, queues, registers, buffers, data paths and other hardware components.

Some actions such as receive or timeout can be viewed as "events". The protocol is a parser for event sequences during execution and each event may trigger the execution of a cluster of actions.

6.2.1 Implementation Specifications

There are a few differences between an abstract protocol specification and its actual implementation specification. The implementation specification should be more specific about implementation details than the abstract specification. For example, an acknowledgment action in an abstract specification may be implemented by some combination of the following schemes: (1) positive acknowledgment and retransmission, (2) negative acknowledgment, (3) piggybacked acknowledgment, (4) multiple acknowledgment, or (5) queued acknowledgment. The architecture detail is not important for the abstract specification but is necessary for the implementation specification.

For example, the abstract specification of the data exchange protocols of IMP and TCP are similar but their implementation specifications are quite different. TCP uses a window scheme for flow control, allows for advance message receptions, and uses a piggybacked acknowledgment scheme. For comparison purposes, we have constructed a detailed specification for the data exchange protocol of TCP. Table 5 is a list of actions for both sender and receiver parts. Fig. 46 is the action grammar specification.

Table 5 Detailed actions of TCP data exchanges

Sender

Bs:	"allocate Buffer for sender's letter"
Gs:	"Generate letter"
Qg:	"Queue Generate"
Fg:	"Fetch Generate queue"
Sn:	"Send segment Never sent before"
Ss:	"Send segment out a lower channel"
Fa:	"Fetch Acknowledgment queue"
AFs:	"find Full Acknowledgment for the outstanding segments"
APs:	"find Part Acknowledgment for the outstanding segments"
QTt:	"Queue Timed outstanding segment"
Dt:	"Dequeue (Reset) timed outstanding segment"
FTt:	"Fetch Timed outstanding segment queue (Timeout)"
Fs:	"Free retransmission space"

Receiver

Ru:	"Receive Unacceptable segment"
Rl:	"Receive segment overlap the Left side of receive window"
Rm:	"receive segment within the Middle of recieve window"
Ra:	"Receive Acknowledgment for sender"
Qa:	"Queue Acknowledgment"
DQn:	"Dequeue Queue New send queue"
DQTt:	"Dequeue Queue Timed outstanding queue"
Ar:	"transmit Acknowledgment to lower entity"
Cr:	"Consume receive segment into window"
Fu:	"Free the segment unit"
Qp:	"Queue received letter to Process"
Fp:	"Fetch received letter to Process"
Sp:	"Send letter to process"
Fw:	"Free left part of acknowledged receive Window"

```
<send VIP> ::= <send> <send VIP>
  <send> ::= <send segment> <wait ACK>
    <send segment> ::=  Fg Sn Ss QTt
    <wait ACK> ::= <full ACK> ¦ <part ACK> <wait ACK>
                 ¦ <retransmit> <wait ACK>
                 ¦ <send segment> <wait ACK>
      <full ACK> ::= Fa AFs Dt Fs (Ap)
      <part ACK> ::= Fa APs (Dt Fs)
      <retransmit> ::= FTt Ss QTt

<receive VIP> ::= <receive> <receive VIP>
                ¦ <send letter to process> <receive VIP>
  <receive> ::= <receive segment> <Ar>
              ¦ "receive process calls"
    <receive segment> ::= <receive unacceptable>
                        ¦ <receive acceptable> Ra (Qa)
      <receive unacceptable> ::= Ru
      <receive acceptable> ::= "receive control segment"
                             ¦ <receive data segment>
        <receive data segment> ::= "receive duplicate"
                                 ¦ <receive left window>
                                 ¦ <receive middle window>
          <receive left window> ::= Rl Cr (Qp)
          <receive middle window> ::= Rm Cr
    <Ar> ::= <piggyback ACK> ¦ "send ACK segment"
      <piggyback ACK> ::= DQn ¦ DQTt
  <send letter to process> ::= Fp Sp Fw
```

Fig. 46 Action grammar of TCP data exchange protocols

6.2.2 Deterministic Parsing Algorithm

Abstract specifications of protocols usually are in the form of a nondeterministic grammar, but corresponding implementation specifications should be in deterministic form for automatic implementation. A nondeterministic algorithm is permitted to contain statements that specify selecting one of several choices and to contain redundant statements for better human understanding. In a nondeterministic algorithm, the device executing the algorithm is supposed to make the correct choice at each such step (the choice that will lead to success if any choice will). Such an algorithm is not suitable for an efficient computer implementation since without further directions in the algorithm, the computer will have no way of knowing which choice to make.

It is always possible, however, to modify a nondeterministic algorithm so that each place where a choice is required, directions are provided which cause each choice to be tried successively. Once a nondeterministic algorithm is converted to deterministic algorithm it can be executed in a straightforward way.

The conversion of a nondeterministic algorithm to a deterministic algorithm usually increases the bookkeeping in the algorithm, since the deterministic algorithm must keep track of all the various choices made so that it can go back to try the alternate choices.
Since this additional bookkeeping obscures the basic structure of the algorithm, many algorithms concerning with parsing are best presented in nondeterministic form.

6.2.3 Software/Hardware Implementations

We can use the recursive decent technique to write a top-down parsing program for a protocol from its transmission grammar specification. In the next section is presented the idea of the general protocol system to automate the process of protocol construction from a validated implementation specification. Automatic hardware implementations of protocols will be the topic of Section 6.4.

6.3 The General Protocol System

Instead of composing a specific program for each protocol TG that arises, we may construct a single, general parsing program. Individual protocol grammars are then fed to the general program to make the general program act like the protocol. The General Protocol System (GPS) is a translator from transmission grammars into parser-driving data structures. The translator could be an interpreter to interpret the actions and data structure or a compiler to compile the TG into a self-controlled program. For simplicity, we assume the translator to be an interpreter in the following discussion. The translator accepts the BNF productions of a message grammar and an action grammar, converts them into the desired data structures to be stored in the translator program, and then interpretes the TG stored in the data structure as shown in Fig. 47.

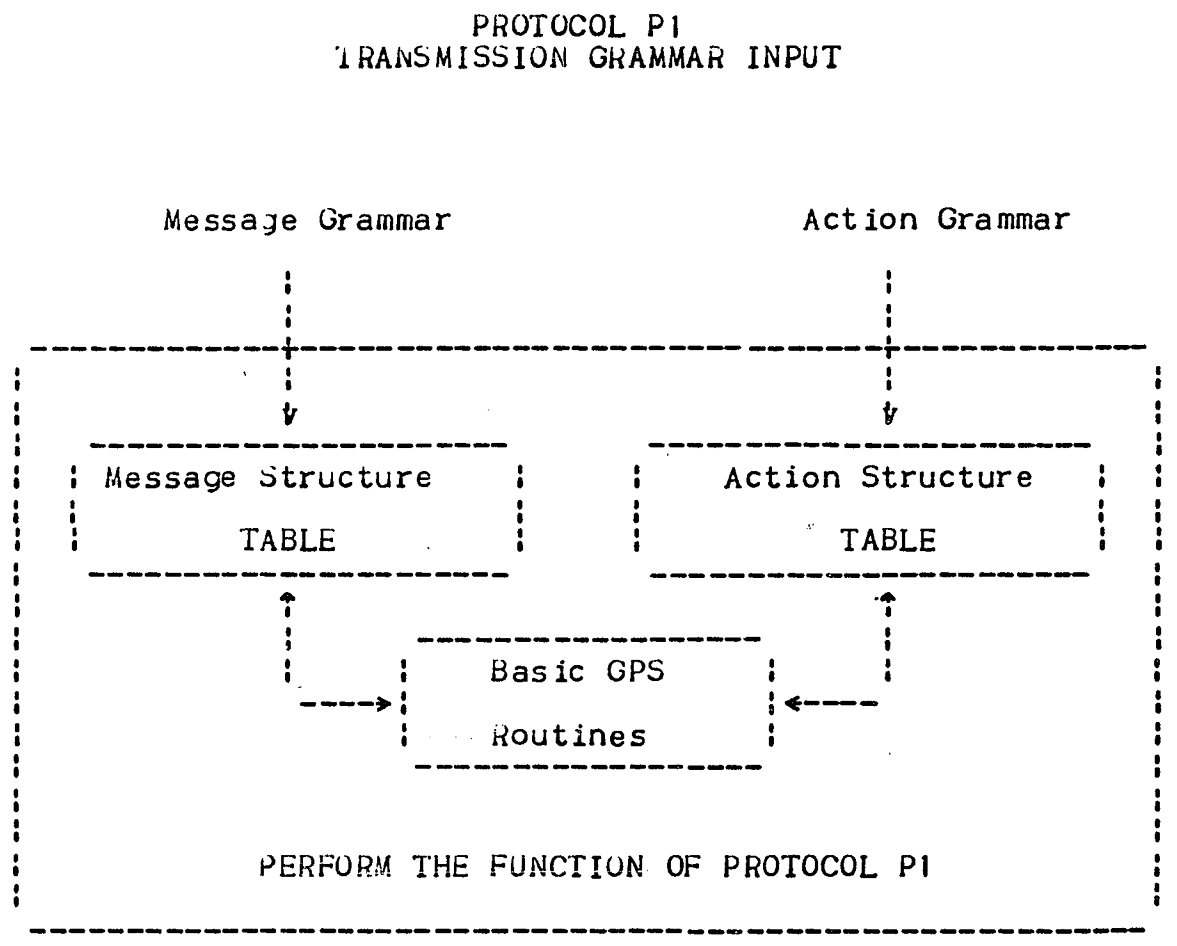


Fig. 47. Architecture of the general protocol system

The GPS is both a language and a computer program. As a language, it can unambiguously describe communication protocols. As a computer program, it can interpret and execute the protocol described in the GPS language.

The GPS strictly follows the rules of the simple top-down parsing method, and it is straightforward if the underlying TG is deterministic; that is, if the sentences of the TG can be parsed with one symbol of lookahead and without backtracking.

However, the transmission grammar described in previous chapters only defines the syntax of the protocol. For the purpose of automatic implementation, we need to relate the semantics of the protocol to the TG.

6.3.1 Actions of GPS Language

In the following, a subset of actions for the GPS (see Table 6) is introduced to describe the semantics of the TG in Fig. 45. GPS actions are designed to be very powerful and flexible so that they can easily describe the semantics of the protocol. Each GPS action carries with it information falling into two categories:

a) Operation - The "operation" of an action is a "verb" suggestive of the task the action accomplishes.

b) Operands - The operands may be thought of as the arguments used in calls on subroutines and we underline the operands that are expected to <u>return</u> values or message units. The number of operands each action has is not fixed. When operands are not explicitly provided, the GPS assumes no such operands to exist.

In Table 6, five types of GPS actions are shown which are of particular interest for describing the semantics of communication protocols.

Table 6 Actions of the General Protocol System

OPERATION	OPERANDS
ALLOCATE	<u>ptr</u>,size
FREE	ptr,size
ROUTE	<u>send channel</u>,destination
SEND	msg unit,send port,ptr,size
RECEIVE	<u>msg unit</u>,receive port,<u>ptr</u>,<u>size</u>
SEGMENT	segment no,ptr1,size1,<u>ptr2</u>,<u>size2</u>, segment size,<u>complete flag</u>
REASS	segment no,ptr1,size1,ptr2,size2, segment size,<u>complete flag</u>
QUEUE	name,key,ptr,size,wait time, other operands
DEPART	name,key
LOOKUP	name,key,<u>exist flag</u>
FETCH	name,<u>key</u>,<u>ptr</u>,<u>size</u>,<u>other operands</u>
CLEAR	names
CONDITION	operand 1,operand 2,relation operator

(1) Space management actions

<u>ALLOCATE</u> (ALC) - This operation specifies the requested buffer size. A pointer (ptr) specifying the beginning of the allocated buffer is returned.

<u>FREE</u> (FRE) - This operation returns the buffer specified by the buffer descriptor (ptr,size) to the GPS.

(2) Send and receive actions

<u>ROUTE</u> (RUT) - This operation requests the routing algorithm to return the send channel from the given destination.

<u>SEND</u> (SND) - This operation sends the msg unit specified in the message grammar (such as <letter(B)> in Fig. 41) out of the send channel. The msg unit is sent out (generated) according to its syntax definition in the message grammar in the following ways: a) The GPS will send out the values of the variables and constants which are defined as the terminals of the msg unit until a buffer indicator is encountered. (A buffer indicator is enclosed as (B) inside a non-terminal, such as <letter(B)>.) b) When the buffer indicator is encountered, the GPS will start sending out the msg unit from the buffer area defined by the ptr and size.

<u>RECEIVE</u> (RCV) - This operation receives and checks the syntax of the input of the receive channel according to the syntax of the msg unit. The GPS will store all the received values into the variables of the msg unit. When the buffer indicator is encountered, the GPS will also store the msg unit into the buffer area defined by the ptr and size.

If the specified channel has no input or the input is not the specified initial terminal of the alternate in the msg unit grammar rules, the GPS will try the next alternate of the action grammar.

(3) Segmentation and reassembly actions

SEGMENT (SEG) - This operation gets the buffer descriptor (ptr2,size2) of the packet space (with its packet # equal to the segment no.) from the known message buffer descriptor (ptr1,size1) and regular packet size (segment size). When the complete flag is on, it means the last piece of packet in the message has been taken out.

REASS (RAS) - This operation reassembles the packet at (ptr2,size2) into the message reassembly buffer (ptr1,size1). It is similar to the SEGMENT action.

(4) Event synchronization actions

The following six queues are of particular interest in our design example.

A	Acknowledgment queue
T	outstanding packets Time out queue
SR	Send Routing packet queue
SH	Send Host packet queue
OH	Output to Host queue
S	Space allocation queue

<u>QUEUE</u> (QUE) - This operation creates an entry in the queue specified in the name operand, with key operand as index and the remaining operands as information to be queued.

<u>DEQUEUE</u> (DEQ) - This operation deletes an entry in a FIFO manner from the queue specified in the name operand.

<u>LOOKUP</u> (LKP) - This operation checks if an entry with the key specified in the key operand is in the queue specified in the name operand.

<u>FETCH</u> (FCH) - This operation retrieves the information from the queue specified in the name operand in a FIFO manner, and puts the information in the appropriate variables specified in the operands. If the queue is empty, the GPS will try the next alternate action.

<u>CLEAR</u> (CLR) - This operation resets and clears all the queue names specified and is usually used for error recovery.

(5) Condition testing action

CONDITION (CND) - This operation compares the specified variable 1 and variable 2 with the relation (e.g., Equal, Not Equal, Greater or Less) specified in the relation operator, and if the condition is not satisfied, the GPS will take the next alternate in the action grammar.

There are many other important operations that should be included in the GPS. For example, a UNIQUE operation may be needed to get a unique initial sequence number for a new connection. Some other functions may also be needed to handle the semantics of flow control mechanisms and interprocess communication primitives [Walden72].

Fig. 48 shows both the syntax and semantics of the action grammar for the protocol at the communication processor layer. This figure is obtained by applying the substitution operation to the action grammar of Fig. 45. Note that the ALLOCATE operation in Table 6 is not used, because we assume that it is specified at the interprocess communication layer (i. e. Host/Host).

```
<TG> ::= <receive> <TG> | <send> <TG>
  <receive> ::= <receive packet> | <receive message>
    <receive packet> ::= <receive ACK> | <receive local> | <receive routing>
      <receive ACK> ::= <Ra>
        <Ra> ::=  "RCV      <ACK>,channel.i"
                  "LKP      T,<packet header>,exist"
                  <check1> [check if it acknowledges an outstanding packet]
          <check1> ::=  "CND      exist,Ø,NE"
                        "QUE      A,<packet header>"
                        | ϵ
      <receive local> ::= <Rl> <Ar> <Cr> <Fr>
        <Rl> ::= "RCV     <local packet>,channel.i,ptr,size"
        <Ar> ::= "SND     <ACK>,channel.i"
        <Cr> ::= "FCH     S,<msg leader>,ptr1,size1"
                 "RAS     packet.no,ptr1,size1,ptr,size,letter.size,
                          complete.flag"
                <check2> [check if all the message packets has received]
          <check2> ::= "CND    complete.flag,TRUE,E"
                       "QUE    OH,<msg leader>,ptr1,size1"
                       | ϵ
        <Fr> ::= "FRE     ptr,size"
      <receive routing> ::= <Rr> <Ar>
        <Rr> ::= "RCV     <routing packet>,channel.i,ptr,size"
                 "QUE     SR,<packet header>,ptr,size"
    <receive message> ::= <Rh>
      <Rh> ::= "RCV     <message>,channel.i,ptr,size"
               "QUE     SH,<msg header>,ptr,size,,1"
  <send> ::= <receive> <send> | <send packet> <wait ACK> |
             <send message to host> | ϵ
    <send packet> ::= <send routing packet> | <send host packet>
      <send routing packet> ::= <Ss Ro Ns> <St> <Ts>
        <Ss Ro Ns> ::= "FCH    SR,<packet header>,ptr,size"
        <St> ::= "QUE     T,<packet header>,ptr,size,timeout"
        <Ts> ::= "RUT     send.channel,dest"
                 "SND     <routing packet>,send.channel,ptr,size"
      <send host packet> ::= <Ss Gs Ns> <St> <Ts>
        <Ss Gs Ns> ::= "FCH    SH,<msg header>,ptr1,size1,packet.no"
                       "SEG    packet.no,ptr1,size1,ptr,size,letter.size,
                               complete.flag"
                               <check3>
                               [check if all the message packet has generated]
          <check3> ::= "CND    complete.flag,TRUE,NE"
                       "QUE    SH,<msg header>,ptr1,size1,,packet.no+1"
                       | ϵ
    <wait ACK> ::= <receive> <wait ACK> | <acknowledgment> |
                   <retransmit> <wait ACK> | <send> <wait ACK>
      <acknowledgment> ::= <As> <Rt> <Fs>
        <As> ::= "FCH     A,<packet header>"
        <Rt> ::= "DEQ     T,<packet header>,ptr,size"
        <Fs> ::= "FRE     ptr,size"
      <retransmit> ::= <Rs Rt> <St> <Ts>
        <Rs Rt> ::= "FCH    T,<packet header>,ptr,size"
    <send message to host> ::= <Sh> <Cb>
      <Sh> ::= "FCH     OH,<msg leader>,ptr,size"
               "PUT     send.channel,dest"
               "SND     <message>,send.channel,ptr,size"
      <Cb> ::= "FRE     ptr,size"
               "DEQ     S,<msg leader>"
```

Fig.48 GPS action specification for IMP protocols

The action grammar of Fig. 48 and the message grammar of Fig. 41 form a transmission grammar which contains enough semantics and can be directly translated by the GPS into executable data structures.

Automatic techniques can be applied to indent and format the grammar rules of the TG, and to obtain a listing showing the derivation of the final hierarchically structured document. (e.g., the listing in Fig. 48). We can then use validation techniques to check for possible syntax or logical errors, make modifications on the listing, and directly input the neatly formatted TG in the case of automatic implementations.

The GPS can check the syntax errors of the GPS actions. Such an error could be a missing operand of the msg unit for the SEND operation, or a specified operand that has different data type than expected.

6.3.2 Another GPS example

The master control part of the Binary Synchronization Communication (BSC) protocols is shown as another example of GPS action specification. Fig. 49 shows an abstract specification and Fig. 50 shows a GPS implementation specification.

```
<contention> ::= "bidding" <outcome>
  <outcome> ::= "refuse" <contention>
          | "busy" "enquire" <outcome> | "master" <trans>
          | "clear" <outcome> | "timeout" "enquire" <outcome>
    <trans> ::= "generate" <send> <trans>
             | "end trans" <contention>
      <send> ::= "send msg" <wait ACk>
        <wait ACk> ::= "acknowledgment"
                    | "retransmit" <wait ACk>
```

Fig. 49 Abstract specification for BSC protocols

```
<contention> ::= <bidding> <outcome>
  <bidding> ::= "RECEIVE       <enq>,U1"
                "SEND          <enq>,M2"
                <QTt>
    <QTt> ::=   "QUEUE          T',,,3"
  <outcome> ::= <refuse> <contention>
         | <busy> <enquire> <outcome> | <master> <trans>
         | <clear> <outcome> | <timeout> <enquire> <outcome>
    <refuse> ::= "RECEIVE       <nack>,M2"
                 "SEND          <nack>,U1"
    <busy> ::=   "RECEIVE       <wbt>,M2"
    <enquire> ::="SEND          <enq>,M2"
                 <QTt>
    <master> ::= "RECEIVE       <ack>,M2"
                 "DEQUEUE       T"
                 "SEND          <ack>,U1"
    <timeout> ::="DEPART        T"
    <clear> ::=  "RECEIVE       ,M2"
    <trans> ::= <generate> <send> <trans>
              | <end trans> <contention>
      <generate> ::= <Gs>
                 ::= "RECEIVE       ptr,U1"
      <send> ::= <send msg> <wait ACK>
        <send msg> ::= <Ss> <QTt>
          <Ss> ::= "SEND            <msg>,M2,ptr"
        <wait ACK> ::= <acknowledgment>
                     | <retransmit> <wait ACK>
          <acknowledgment> ::= <As> <Dt>
            <As> ::= "RECEIVE       <ack>,M2"
            <Dt> ::= "DEQUEUE        T"
          <retransmit> ::= <Dt> <Qs> <QTt> | <Ap> <QTt>
            <Dt> ::= "DEPART        T"
            <Qs> ::= "SEND          <enq>,M2"
            <Ap> ::= "RECEIVE       <ack.old>,M2"
      <end trans> ::="RECEIVE       <eot>,U1"
                     "SEND          <eot>,M2"
                     "RECEIVE       <eot>,M2"
```

Fig. 50 A GPS action specification for BSC protocols

6.3.3 Architectures and Data Structures

We can construct a list structure to represent the grammar rules in main memory [Gries71,Wirth76]. The grammar is assumed to be represented in the form of a deterministic set of rules. It is translated into the appropriate list structure instead of an executable program structure. The data type definition for the GPS is shown in Fig. 51. Each terminal or non-terminal symbol in the right part (corresponding to a node in the list structure) consists of the three components SYMBOL, ALTERNATE and SUCCESSOR, where

1. SYMBOL (SYM) represents two variants: one for terminal and one for non-terminal symbols. If the node represents a terminal, SYM is the symbol name in some internal form. If it represents a terminal GPS action, its internal form should be able to inform the GPS of its operation and operands. If the node represents a non-terminal, SYM is then a pointer to an entry in the hash table (hash[Ø..prime] in Fig. 51) which contains the index of the array structure (K[Ø..maxheader] in Fig. 51) representing the corresponding non-terminal symbol. Both variants contain two pointers, alternate and successor.
2. ALTERNATE (ALT) points to the first symbol of the next alternate in the right part following the one in which

the node occurs (NIL if none). This is only for the first symbol in the right part.

3. SUCCESSOR (SUC) points to the next symbol in the right part (NIL if none).

In addition, each non-terminal in the left part is represented by a header node that contains the name of the non-terminal symbol and a pointer to the first symbol in its first right part.

A hash table is used to speed up the table lookup of the non-terminals for both constructing a structure from a TG and the actual parsing of events.

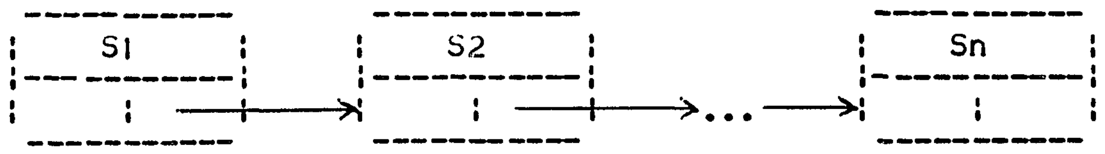
The resulting data type definition represented by PASCAL is as follows:

```
const
  maxheader = 58;
      /* max number of nonterminals */
  prime = 997;
      /* prime number for hashing table size */
type
  tblrange = 0..maxheader;
  alpha = packed array[1..20] of char;
  pointer = ^node;
  node = record
           suc, alt: pointer;
           case terminal: boolean of
                true: (tsym: alpha);
                false: (nsym: tblrange)
         end;
  header = record
             sym: alpha;
             entry: pointer
           end;
var
  k: array[tblrange] of header;
  hash: packed array[0..prime] of -1..maxheader;
```

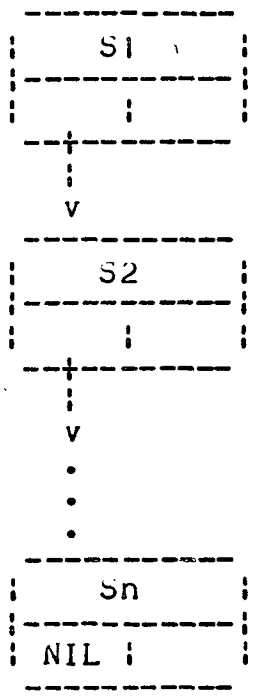
Fig. 51 Data structure of the GPS

The translation rules from grammars into data structures are straightforward.

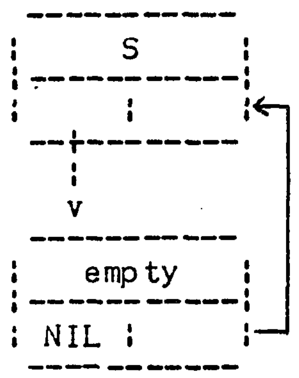
1. A sequence of symbol S1 S2... Sn of a right part is translated into the following list of data nodes:



2. The list of alternative right parts of a grammar rule is translated into the following data structure:



3. A loop is translated into the following data structure:



We can also represent extra components in the node for the analysis of iteration (loop) of strings, for the data type descriptor of the terminal in the message grammar, and for the buffer indicator specification of the non-terminal in the message grammar.

The approach to parsing using syntax and data structure can be used to automate protocol construction, thereby reducing the possibility of implementation errors from a validated TG specification. Its powerful actions provide many of the built-in semantics handling capabilities to simplify the protocol design. It can provide the flexibility to extend or modify protocols by changing syntactic and semantic constructs, without extensive coding and testing efforts. A more ambitious scheme even allows several protocols to be dynamically stored in the translator (either for security reasons or for different network protocols) and to perform proper actions depending on which protocol the input message belongs to. A possible application is to store both X.25 and SNA protocols in the GPS.

The TG of higher-level protocols, defined in a machine independent form for a host system, can be portable to other host systems, thereby reducing the cost and time of protocol implementation.

6.3.4 GPS Parsing Techniques

A sentence (action sequence) of a language (protocol) is a sequence of receive, timeout or synchronization events occurring during communication. After constructing the data structure of a protocol, the GPS is ready to check (parse) the action sequences (sentences) by following (interpreting) the protocol (data structure). The parsing of an action sequence consists of a repeated statement describing the transition from one node to the next node. The interpretation procedure is activated recursively.

A simplified parsing program is shown as part of the validation system program listed in Appendix A. It can work on a non-deterministic grammar which contains no choices between several alternative non-terminal symbols and contains no left recursion. More sophisticated parsers which operate on less restrictive classes of context-free grammars may be easily derived from it.

It is also feasible to convert a non-deterministic context-free grammar to a deterministic push-down automaton. The automaton is a table of state transitions with stack manipulation capabilities and may be executed more efficiently by avoiding recursive procedure calls.

The GPS is still in its early design stage and more work is needed. We have not yet addressed the semantics of the GPS such as: (1) routing algorithm, (2) flow control, (3) failure recovery and (4) contention control. A fully implemented GPS should have a more powerful "protocol language" to simplify both the semantics representations and processings.

6.4 Hardware Implementations

Recent advances of hardware technology in the field of microprocessors and large-scale integration of circuitry have made it possible to have hardware implementation of protocols. Hardware implementation of protocols could reduce the overhead of the host computers, and can also be used to implement the specialized protocol actions in a cost-effective way.

Theoretically speaking, every algorithm which can be implemented by software can also be implemented by hardware. There are two approaches to hardware implementation of protocols: hardwired circuits and microprogramming.

(A) Hardwired Circuits

Hill [Hill73] designed a formal model, A Hardware Programming Language (AHPL), to describe the hardware design of digital computers. AHPL includes operations which satisfy the constraints imposed by available hardware. Every AHPL step written down by the designer will represent some action or some already specified hardware elements. The precise correspondence between an AHPL statement and its hardware realization has also been established.

It is feasible to automate the hardware circuit generation from TG implementation specifications. From the implementation specification of a protocol, we can translate to AHPL description of hardware. The AHPL compiler can go through the hardware descriptions to construct the data part of protocol hardware, including register groups, register sizes, bus connections, decoders and clocks. There are several special representation techniques of AHPL for GPS actions, but we will not discuss the detailed correspondence between the GPS actions and AHPL statements in this dissertation. It will be left as part of our future research.

(B) Microprogramming

The control sequence (written in AHPL) can be stored in a memory rather than being hardwired. This implementation is called microprogramming. It is one level more detailed and one level closer to the hardware than machine language programming. A control store is used to hold the microprogram.

A microprogrammed protocol machine performs communication actions as it reads successive locations in the control store. The individual steps of protocol actions can be controlled by the microprogram. The transfer of data between registers, the sequence and timing of communication events, the selection of operands of GPS operations, the specification of GPS action functions; all these and more are under the control of microprogram. The hardware control signals are produced directly from the microprogram.

The attractive features of microprogrammed protocol machines are the flexibility in design, testing and modification. We can tailor the processor's performance to meet particular needs of GPS actions. We can define very powerful protocol instructions which move time-consuming operations from software to firmware. Since protocols perform mostly specialized operations, the firmware can be designed to execute the operations efficiently. Many of the protocol operations can be executed in the same cycle, thereby providing a high degree of parallelism. Hardware decoding techniques may also handle interrupts in parallel. This provides dramatic new processing power to protocol machines.

Another area of application is _protocol emulation_ which is, in a way, an extension of one protocol architecture (say, SNA) to cover another protocol architecture (say, X.25). A protocol machine of one architecture can perform functions for another protocol architectures by changing the microprogram in the control store.

We can see various implementation schemes of this chapter in Fig. 52.

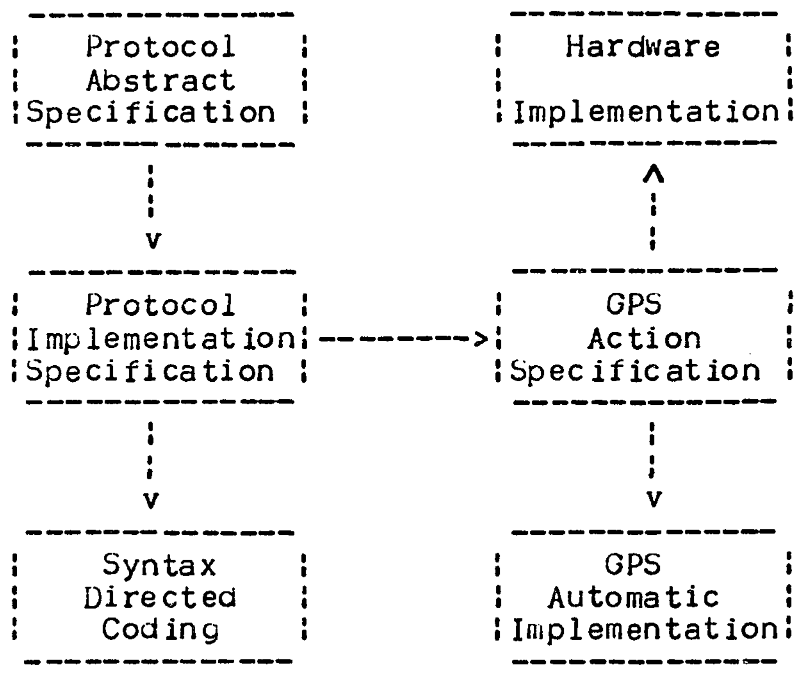

Protocol Abstract Specification
Hardware Implementation
Protocol Implementation Specification
GPS Action Specification
Syntax Directed Coding
GPS Automatic Implementation

Fig. 52 Protocol implementation schemes

It is well known that the direct use of a BNF to generate a language processor may yield a very inefficient algorithm; the amount of lookahead and/or backup is fairly great. However, optimization techniques are available to reduce an LR(k) grammar to an LR(1) grammar and to be a deterministic algorithm. The discussion of optimization techniques is not presented because this subject has already been extensively covered in the literature.

If we can represent a TG with a deterministic algorithm, the parser for the TG can run fairly fast. There are mechanical techniques to help the construction of a deterministic algorithm for a given context-free grammar. Once the algorithm is constructed, it can be easily implemented by hardware circuits.

CHAPTER 7

SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH

In previous chapters of this dissertation, we have addressed various issues for constructing protocol programs that are correct, modifiable and maintainable. Our research goal was primarily directed towards practical techniques that enable these programs to be implemented in an effective way and at acceptable cost. In order to reach this goal, the Transmission Grammar (TG) model has been introduced for a systematic construction of communication protocols.

In this chapter we shall summarize the significant features of the TG model and the main result of this research. We shall also indicate some topics for further research.

7.1 Significant Features of the TG Model

During the research of the transmission grammar model, attention was focused on extending and integrating formal language and software engineering techniques to produce a comprehensive design methodology for communication protocols. As a result of this effort, new concepts and techniques were developed, some of which are listed below.

1. This research has produced a general and comprehensive design methodology for the construction of complicated protocols. Prior research on network protocols has focused mostly on protocol verification. Based on the TG model, we developed a systematic and structured methodology for the specification, validation and implementation of protocols. The TG model has been applied to various phases of protocol software life cycle. The step-wise TG specification allows the protocol designer to specify protocol program modules in a well-structured manner. In addition, the specified grammar structure is kept so simple that automatic validation and implementation can be easily carried out.

2. The TG model is capable of specifying protocols which are more complicated than those modeled by finite state automata. We illustrated structured TG specification of

protocols which may have strictly context-free and context-senistive characteristics. The TG model can also contain necessary redundancies for better human readability, as compared with the FSA model. Using the TG model, we demonstrated a step-wise refinement technique for protocol specification and protocol documentation. The model was also shown to have the flexibility of describing complicated protocols with different degrees of details for validation and implementation purposes.

3. Local validation techniques has been developed and used to reveal the TG structure errors. Grammar notations were used to formally define the following TG errors: (1) lelt-recursion, (2) non-deterministic grammar, (3) undefined non-terminal, (4) superfluous rule and (5) improper termination. We also outlined the corresponding algorithms in locating these structure errors. Local validation does not take into consideration global timing and interactions between communicating entities, and is a relatively easy task when compared with global validation. Local validation can reveal most of the "simple" protocol errors before global validation.

4. Global validation techniques have also been developed to check the timing and interactions between communicating

entities and to reveal protocol syntax errors. We have presented the validation automaton (VA) model which can comprehensively represent the interdependency between communicating entities. Several complexity reduction rules were summarized to reduce the number of states and transitions during the automatic validation process. A validation checklist concept was also introduced to provide a systematic scheme for examining the fundamental correctness problems that may cause protocol syntax errors.

5. One of the most significant features of the TG model is its grammar integration operations. We have introduced the use of the arbitrary shuffle and substitution operations of formal languages to construct inherently correct protocols from validated components and/or independent parts. The properties and the algorithms of these operations have been also shown in this dissertation.

6. We have generalized the TG model to represent the hierarchical structure of message formats. Using this generalized specification, we have proposed the structure of a protocol recognizer for automatic software/hardware implementation of protocols. We have also outlined the

structure of such a recognizer, the General Protocol System. Other syntax-directed implementation techniques also were studied in this dissertation.

These features combine to make our design and construction schemes very efficient and useful for constructing correct protocol programs. Thus, it would seem that this research was rather successful in accomplishing its original goal of constructing correct, modifiable and maintainable protocols.

7.2 Suggested Research Problems

The following is a list of future research projects which follows from the work in this dissertation.

(1) In Chapter 2, we studied the fundamental protocol correctness problems and the derivation of a protocol validation checklist. We restricted ourselves to those correctness problems that may cause protocol syntax errors. Further research could be done in developing a protocol verification checklist for the complete verification of protocol semantics.

(2) A complex computer network is bounded to have failures of its components. Computer network protocols must include "graceful" failure recovery schemes that are transparent to the users, execute rapidly and require little manual work at the time of failure. In Chapter 3 and 4, we have demonstrated TG specification techniques for the purpose of protocol validation. We could further expand the TG protocol specification to include fault-tolerant schemes for reliability validation.

(3) A global validation technique, using the validation automaton model, has been addressed in Chapter 5. It would be worthwhile to investigate the possibility of automating the derivation of a validation automaton from a TG specification. We could also implement an efficient global validation system to generate global state transitions in nicely formatted listing. Hashing techniques could be included in the validation system to check symmetric states during the state generation. We might also expand channel queue notations to represent interactions between more than two entities. These investigations should yield additional refinements to the current global validation techniques.

(4) In the field of protocol implementation, work could be done in developing a protocol language designed along the idea of syntax-directed protocol implementations. The

protocol language can be viewed as a set of actions for the General Protocol System. An actual implementation of the General Protocol System might take debugging or error-recovery into consideration. The optimization of transmission grammars also seems to be useful for reducing the look-ahead during protocol program execution.

The most obvious "next-step" is, however, the implementation and verification of some real-world communication protocols by using the techniques developed in this dissertation.

Because of the drastic improvement of computer and communication technologies, distributed processing has been an important research area in recent years. Reliable protocol constructions are crucial in assuring correct synchronization in any distributed system. We expect our view of the communication protocol problems and their solutions will contribute to clarify the poorly understood field of distributed systems.

APPENDIX

VALIDATION SYSTEM PROGRAM LISTING

```
const
  maxprtln = 80;            /* max print line */
  maxheader = 40;     /* max number of nonterminals */
  maxtop = 41;      /* max pointer to next header */
  prime = 997;    /* prime number for hashing table size */
  free = -1;
type
  tokentype = (terminal,nonterminal,alternate,period,
               separator,repeatri,repeatlf,empty,equal);
  tblrange = 0..maxheader;
  alpha = packed array[1..20] of char;
  pointer = ^node;
  node = record
           suc, alt: pointer;
           case terminal: boolean of
                true: (tsym: alpha);
                false: (nsym: tblrange)
         end;
  header = record
             sym: alpha;
             entry: pointer
           end;
  index = 0..prime;
  /* transitive closure matrix type */
  matrix = packed array[0..maxheader, 0..maxheader] of
  boolean;
```

```
var
  ascii,j,kl:integer;
  inpfg : boolean;
  pgmeof : boolean;                 /* program eof flag */
  /* to print listing of program */
  prtln : array [1..maxprtln] of char;    /*output buffer */
  alt:char;
  prtlnptr : 0..maxprtln;             /* output buffer ptr */
  prtlncnt : integer;                 /* line # counter */
  prtlnfg : boolean;                  /* print line now? */
  prtlnum : boolean;                  /* print line number? */
  i: integer;
  chars,fill,cr,lf:char;
  symbol'type: tokentype;
  topheader,n: 0..maxtop;
  n, currnsym: tblrange;
  top:index;
  n1: tblrange; /* rule number */
  n2: 0..10; /* alternate number */
  a,p: pointer;
  sym: alpha;
  ok: boolean;
  k: array[tblrange] of header;
  hash: packed array[0..prime] of -1..maxheader;
  found: boolean;
  first, firstall: matrix;
  within, withinall: matrix;
  t: 1..10;  /* index of terminet symbols */
  terminet: array[1..10] of alpha;
  altcnt: array[0..maxheader] of 1..10;
  firstflag: boolean;
  lastflag: packed array[0..maxheader] of boolean;
  stop,newmark: boolean;
  mark: array [0..maxheader] of boolean;
```

```
procedure prtcnr;
var
  i : 1..maxprtln;
¦begin
¦
¦ /* is there a line to be output? (this depends on
¦  prtlnfg) if so, print the line up to the buffer
¦  printer (prtlnptp) print line with either line numbers
¦  or not (prtlnum). then reset all values.          */
¦
¦ if prtlnfg
¦ then
¦ ¦begin
¦ ¦ if prtlnum
¦ ¦ then
¦ ¦ ¦begin
¦ ¦ ¦ prtlncnt := prtlncnt + 1;
¦ ¦ ¦ write('   ',prtlncnt:5,'     ');
¦ ¦ ¦end
¦ ¦ else write('             ');
¦ ¦ for i := 1 to prtlnptr do
¦ ¦ ¦begin
¦ ¦ ¦ if prtln[i] in ['a'..'z']
¦ ¦ ¦ then
¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ ascii := ord(prtln[i]) + 32;
¦ ¦ ¦ ¦ prtln[i] := chr(ascii);
¦ ¦ ¦ ¦end;
¦ ¦ ¦ write(prtln[i]:1);
¦ ¦ ¦end;
¦ ¦ writeln;
¦ ¦ prtlnptr := Ø;
¦ ¦ prtlnfg := false;
¦ ¦ prtlnum := true;
¦ ¦end;
¦
¦ /* buffer(prtln) the characters as they are input. print
¦  the buffer whenever the end of a line is encountered on
¦  the input stream or when the buffer gets full, which
¦  depends on maxprtln.                                  */
¦
¦ prtlnptr := prtlnptr + 1;
¦ if chars = ','
¦ then prtln[prtlnptr] := chr(124)
¦ else       prtln[prtlnptr] := chars;
```

```
 ¦ if inpfg
 ¦ then
 ¦  if eoln(input)
 ¦  then prtlnfg := true;
 ¦ if prtlnptr = maxprtln
 ¦ then
 ¦ ¦begin
 ¦ ¦ prtlnfg := true;
 ¦ ¦ prtlnum := false;
 ¦ ¦end;
 ¦end;
procedure readnxchr;
 ¦begin
 ¦ if eof(input)
 ¦ then pgmeof := false
 ¦ else
 ¦ ¦begin
 ¦ ¦ read(input,chars);
 ¦ ¦ prtchr;
 ¦ ¦end;
 ¦end;
procedure prtsym;
 ¦begin
 ¦ i := 1;
 ¦ ¦repeat
 ¦ ¦ chars := sym[i];
 ¦ ¦ prtchr;
 ¦ ¦ i := i + 1;
 ¦ ¦until (chars = ' ') or (i > 20);
 ¦ if (i>20)
 ¦ then
 ¦ ¦begin
 ¦ ¦ chars := ' '; prtchr;
 ¦ ¦end;
 ¦end;
procedure prtrule;
 ¦begin
 ¦ inpfg := false;
 ¦ sym := k[nl].sym;
 ¦ i := 1;
 ¦ ¦repeat
 ¦ ¦ chars := sym[i];
 ¦ ¦ prtchr;
 ¦ ¦ i := i + 1;
 ¦ ¦until (chars=' ') or (i>20);
 ¦ if (i>20)
 ¦ then
 ¦ ¦begin
 ¦ ¦ chars := ' '; prtchr;
 ¦ ¦end;
¦ chars := ':'; prtchr;
¦ chars := ':'; prtchr;
¦ chars := '='; prtchr;
¦ chars := ' '; prtchr;
¦ a := k[n1].entry;
¦ ¦loop
¦ ¦ n2 := n2 - 1;
¦ ¦exit if (n2 = Ø);
¦ ¦ a := a^.alt;
¦ ¦end;
¦ ¦repeat
¦ ¦ if a^.terminal
¦ ¦ then sym := a^.tsym
¦ ¦ else sym := k[a^.nsym].sym;
¦ ¦ a := a^.suc;
¦ ¦ prtsym;
¦ ¦until (a = nil);
¦ prtlnfg := true;
¦ chars := fill;
¦ prtchr;
¦end;
```

```
procedure cycles;
var
  levelflg: boolean; /* flag to mark start level of
                        prtpath */
  used: array [Ø..maxheader] of boolean;
  currptr: record /* pointer to next node in the trace of
                   current rule */
             alt: 1..1Ø;
             suc:pointer
           end;
  ruleptr: /* ptrs for current traces of the rules */
  array[Ø..maxheader] of record
                           alt: 1..1Ø;
                           suc:pointer
                         end;
  i,k1: integer;
  initptr: pointer; /* ptr to the initial entry of current
                       rule */
  n,root:tblrange;
  path: array[Ø..maxtop] of record
                              rule: tblrange;
                              alt: 1..1Ø
                            end;
  startlevel,level: Ø..maxtop;
  procedure prtpath;
  var
    node: Ø..maxtop;
    indent: tblrange;
  |begin /* list one cycle (path) */
  | for node := startlevel to level do
  | |begin
  | | n1 := path[node].rule;
  | | n2 := path[node].alt;
  | | chars := ' ';
  | | indent := node;
  | | while (indent>Ø) do
  | | |begin /* output indentation */
  | | | prtchr; prtchr; prtchr;
  | | | indent := indent - 1;
  | | |end;
  | | prtrule;
  | |end;
  | chars := fill;
  | prtlnfg := true;
  | prtchr;
  |end;
```

```
¦begin
¦ /* indented cycles listing */
¦ writeln;
¦ writeln('                   indented cycles listing');
¦ writeln('                   ***********************');
¦ writeln;
¦ levelflg := true; /* initialize flag */
¦ n := topneader - 1; /* n is the number of
¦                          nonterminals */
¦ for root := Ø to n do
¦ ¦begin /* see if root can reach itself */
¦ ¦ if witninall[root,root]
¦ ¦ then
¦ ¦ ¦begin /* root can reach itself */
¦ ¦ ¦ ¦begin /* initialize tree for root */
¦ ¦ ¦ ¦ level := Ø;
¦ ¦ ¦ ¦ initptr := k[root].entry;
¦ ¦ ¦ ¦ patn[Ø].rule := root;
¦ ¦ ¦ ¦ patn[Ø].alt := 1;
¦ ¦ ¦ ¦ i := root;
¦ ¦ ¦ ¦
¦ ¦ ¦ ¦ for k1 := root to n do
¦ ¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ ¦ used[k1] := false;
¦ ¦ ¦ ¦ ¦ /* initialize ruleptr to the start of the
¦ ¦ ¦ ¦ ¦   first alt */
¦ ¦ ¦ ¦ ¦ ruleptr[k1].suc := k[k1].entry;
¦ ¦ ¦ ¦ ¦ ruleptr[k1].alt := 1;
¦ ¦ ¦ ¦ ¦end;
¦ ¦ ¦ ¦end;
¦ ¦ ¦
¦ ¦ ¦ currptr := ruleptr[i];
¦ ¦ ¦ ¦repeat /* creat tree for all the paths from root */
¦ ¦ ¦ ¦ j := -1;
¦ ¦ ¦ ¦ while ((currptr.alt<=altcnt[i]) and (j=-1)) do
¦ ¦ ¦ ¦ ¦begin  /* locate next nonterminal */
¦ ¦ ¦ ¦ ¦ while ((currptr.suc<>nil) and (j = -1)) do
¦ ¦ ¦ ¦ ¦ if currptr.suc^.terminal
¦ ¦ ¦ ¦ ¦ then currptr.suc := currptr.suc^.suc
¦ ¦ ¦ ¦ ¦ else
¦ ¦ ¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ ¦ ¦ j := currptr.suc^.nsym;
¦ ¦ ¦ ¦ ¦ ¦ currptr.suc := currptr.suc^.suc;
¦ ¦ ¦ ¦ ¦ ¦end;
¦ ¦ ¦ ¦ ¦ if (j = -1) /* currptr.suc = nil */
¦ ¦ ¦ ¦ ¦ then
¦ ¦ ¦ ¦ ¦  if (currptr.alt <= altcnt[i])
¦ ¦ ¦ ¦ ¦  then
```

```
! ! ! ! !    !begin
! ! ! ! !    ! currptr.suc := initptr;
! ! ! ! !    ! for k1 := 1 to currptr.alt do
! ! ! ! !    ! currptr.suc := currptr.suc^.alt;
! ! ! ! !    ! currptr.alt := currptr.alt + 1;
! ! ! ! !    !end;
! ! ! ! !end;
! ! ! ! if j<>-1
! ! ! ! then /* check reachability */
! ! ! ! /* we know root within+ i and i within j, if j
! ! ! !  within+ root then we can extend the path */
! ! ! !  if (witninall[j,root] and (not(used[j])))
! ! ! !  then
! ! ! !   !begin /* extend path to j */
! ! ! !   ! if (levelflg)
! ! ! !   ! then
! ! ! !   ! !begin /* startlevel is the starting trace
! ! ! !   ! !         level after last cycle printed */
! ! ! !   ! ! startlevel := level;
! ! ! !   ! ! levelflg := false;
! ! ! !   ! !end;
! ! ! !   ! path[level].alt := currptr.alt;
! ! ! !   ! if (currptr.suc = nil)
! ! ! !   ! then
! ! ! !   ! !begin
! ! ! !   ! ! currptr.suc := initptr;
! ! ! !   ! ! for k1 := 1 to currptr.alt do
! ! ! !   ! ! currptr.suc := currptr.suc^.alt;
! ! ! !   ! ! currptr.alt := currptr.alt + 1;
! ! ! !   ! !end;
! ! ! !   ! ruleptr[i] := currptr;
! ! ! !   ! if (j=root)
! ! ! !   ! then
! ! ! !   ! !begin
! ! ! !   ! ! prtpath; /* after printing path, reset flag
! ! ! !   ! !              to store next trace */
! ! ! !   ! ! levelflg := true;
! ! ! !   ! !end
! ! ! !   ! else
! ! ! !   ! !begin
! ! ! !   ! ! used[j] := true;
! ! ! !   ! ! initptr := k[j].entry; /* extend new
! ! ! !   ! !                            entry */
! ! ! !   ! ! level := level + 1;
! ! ! !   ! ! path[level].rule := j;
! ! ! !   ! ! currptr := ruleptr[j];
! ! ! !   ! ! i := j;
! ! ! !   ! !end;
! ! ! !   !end;
```

```
¦ ¦ ¦ ¦ if (currptr.alt > altcnt[i])
¦ ¦ ¦ ¦ then
¦ ¦ ¦ ¦ ¦begin /* backtrack in tree resetting ruleptr
¦ ¦ ¦ ¦ ¦       and used */
¦ ¦ ¦ ¦ ¦ ruleptr[i].alt := 1;
¦ ¦ ¦ ¦ ¦ ruleptr[i].suc := initptr;
¦ ¦ ¦ ¦ ¦ used[i] := false;
¦ ¦ ¦ ¦ ¦ levelflg := true;
¦ ¦ ¦ ¦ ¦ /* reset flag for the new try */
¦ ¦ ¦ ¦ ¦ if (level > 0)
¦ ¦ ¦ ¦ ¦ then
¦ ¦ ¦ ¦ ¦ ¦begin /* backup one rule */
¦ ¦ ¦ ¦ ¦ ¦ level := level -1;
¦ ¦ ¦ ¦ ¦ ¦ i := path[level].rule;
¦ ¦ ¦ ¦ ¦ ¦ initptr := k[i].entry; /* backtrack old
¦ ¦ ¦ ¦ ¦ ¦                           entry */
¦ ¦ ¦ ¦ ¦ ¦ currptr := ruleptr[i];
¦ ¦ ¦ ¦ ¦ ¦end;
¦ ¦ ¦ ¦ ¦end;
¦ ¦ ¦ ¦until ((j=-1) and (level = 0)
¦ ¦ ¦          and (currptr.suc=nil));
¦ ¦ ¦end;
¦ ¦ /* end of generating all cycles from root */
¦ ¦ used[root] := true; /* mark used */
¦ ¦end;
¦ /* try next nonterminal as the root */
¦ /* end of main loop */
¦end;
```

```
procedure getsym;
 procedure getterm;
  ¦begin
  ¦ sym := '                    ';
  ¦ symbol'type := terminal;
  ¦ sym := '                    ';
  ¦ i := 1;
  ¦ while not(chars= ' ') and not(eof(input)) do
  ¦ ¦begin
  ¦ ¦ if i <= 20
  ¦ ¦ then
  ¦ ¦ ¦begin
  ¦ ¦ ¦ sym[i] := chars;
  ¦ ¦ ¦ i := i+1;
  ¦ ¦ ¦end;
  ¦ ¦ readnxchr;
  ¦ ¦end;
  ¦end;
¦begin
¦ while (chars = ' ') and not(eof(input)) do
¦ ¦begin
¦ ¦ readnxchr;
¦ ¦
¦ ¦end;
¦ if not(eof(input))
¦ then
¦ ¦begin
¦ ¦  case chars of
¦ '<':
¦     ¦begin
¦     ¦ symbol'type := nonterminal;
¦     ¦ sym := '                    ';
¦     ¦ i := 1;
¦     ¦ while not(chars  = ' ') and not(eof(input)) do
¦     ¦ ¦begin
¦     ¦ ¦ if i <= 20
¦     ¦ ¦ then
¦     ¦ ¦ ¦begin
¦     ¦ ¦ ¦ sym[i] := chars;
¦     ¦ ¦ ¦ i := i + 1;
¦     ¦ ¦ ¦end;
¦     ¦ ¦ readnxchr;
¦     ¦ ¦end
¦     ¦end;
¦
¦ '*':
¦     ¦begin
¦     ¦ sym := '*                   ';
¦     ¦ symbol'type := empty;
¦     ¦end;
```

```
 ¦ ':':
 ¦      ¦begin
 ¦      ¦ readnxchr; readnxchr; symbol'type := equal;
 ¦      ¦end;
 ¦ ',': symbol'type := alternate;
 ¦ '[': symbol'type := repeatlf;
 ¦ ']': symbol'type := repeatri;
 ¦ '-':
 ¦      ¦begin
 ¦      ¦ while (chars = '-') do readnxchr;
 ¦      ¦ symbol'type := separator;
 ¦      ¦end;
 ¦ '.': symbol'type := period;
 ¦ '"': getterm;
others: getterm
      end;
     readnxchr;
    end;
  end /*getsym*/;
 procedure find(s: alpha);
   /* locate nonterminal symbol s in list. if not present,
    insert it */
   /* use hashing technique to speed up table lookup */
 var
   d,h1: index;
   h2: integer;
 ¦begin
 ¦ h2 := 0; d := 1;
 ¦ found := false;
 ¦ for i := 1 to 5 do
 ¦ h2 := ord(s[i]) + h2*128;
 ¦ h1 := h2 mod prime; /* hash function */
 ¦ ¦repeat
 ¦ ¦ if hash[h1] = -1
 ¦ ¦ then
 ¦ ¦ ¦begin /* insert */
 ¦ ¦ ¦ found := true;
 ¦ ¦ ¦ hash[h1] := topheader;
 ¦ ¦ ¦ h := topheader;
 ¦ ¦ ¦ k[h].sym := s;
 ¦ ¦ ¦ k[h].entry := nil;
 ¦ ¦ ¦ topheader := topheader + 1;
 ¦ ¦ ¦end
 ¦ ¦ else
 ¦ ¦  if (k[hash[h1]].sym = s)
 ¦ ¦  then
```

```
! !    !begin
! !    ! found := true;
! !    ! h := hash[hl];
! !    !end
! !   else
! !    !begin /* collision */
! !    ! hl := hl + d;
! !    ! d := d + 2;
! !    ! if hl >= prime
! !    ! then hl := hl - prime;
! !    ! if d = prime
! !    ! then
! !    ! !begin
! !    ! ! writeln('          ****x table overflow !!');
! !    ! ! ok := false;
! !    ! !end
! !    !end
! !until found;
!end;
procedure error;
!begin
! writeln;
! writeln ('incorrect syntax'); ok := false
!end/*error*/;
procedure term(var p,q,r: pointer);
var
   a,b,c: pointer;
   procedure factor(var p,q: pointer);
   var
     a,b: pointer; h: tblrange;
  !begin
  ! if symbol'type in [terminal,nonterminal,empty]
  ! then
  ! !begin /*symbol*/
  ! ! new(a);
  ! ! if symbol'type = nonterminal
  ! ! then
  ! ! !begin /*nonterminal*/
  ! ! ! find(sym,h);
  ! ! ! a^.terminal := false;
  ! ! ! if firstflag
  ! ! ! then/* construct relation matrix while parsing */
  ! ! ! first [currnsym, h] := true;
  ! ! ! within [currnsym, h] := true;
  ! ! ! a^.nsym := h
  ! ! !end
```

```
 ¦ ¦ else
 ¦ ¦ ¦begin /*terminal*/
 ¦ ¦ ¦ a^.terminal := true;
 ¦ ¦ ¦ if symbol'type=empty
 ¦ ¦ ¦ then a^.tsym := '*                    '
 ¦ ¦ ¦ else a^.tsym := sym;
 ¦ ¦ ¦end;
 ¦ ¦ firstflag := false;
 ¦ ¦ p := a; q := a; getsym
 ¦ ¦end
 ¦ else
 ¦  if symbol'type = repeatlf
 ¦  then
 ¦   ¦begin
 ¦   ¦ getsym; term(p,a,b); b^.suc := p; new(b);
 ¦   ¦ b^.terminal := true;
 ¦   ¦ b^.tsym := '*                    ';
 ¦   ¦ a^.alt := b; q := b;
 ¦   ¦ if symbol'type = repeatri
 ¦   ¦ then getsym
 ¦   ¦ else error
 ¦   ¦end
 ¦  else error
 ¦end /*factor*/;
¦begin
¦ factor(p,a); q := a;
¦ while symbol'type in
¦ [terminal,nonterminal,repeatlf,empty] do
¦ ¦begin
¦ ¦ factor (a^.suc, b); b^.alt :=nil ; a := b
¦ ¦end;
¦ r := a
¦end /*term*/;
```

```
procedure expression(var p,q: pointer);
var
  a,b,c: pointer;
¦begin
¦ firstflag := true;
¦ term(p,a,c); c^.suc := nil;
¦ if (lastflag[h])
¦ then /* trace possible terminate symbols */
¦  if (c^.terminal)
¦  then
¦   ¦begin
¦   ¦ if (c^.tsym <> terminet[t])
¦   ¦ then
¦   ¦ ¦begin
¦   ¦ ¦ terminet[t] := c^.tsym;
¦   ¦ ¦ t := t + 1;
¦   ¦ ¦end
¦   ¦end
¦  else
¦   lastflag[c^.nsym] := true;
¦   /* set flag for terminate trace */
¦ while symbol'type = alternate do
¦ ¦begin
¦ ¦ getsym;
¦ ¦ firstflag := true;
¦ ¦ term(a^.alt,b,c); c^.suc := nil; a := b;
¦ ¦ altcnt[currnsym] := altcnt[currnsym] + 1;
¦ ¦ if (lastflag[h])
¦ ¦ then
¦ ¦  if (c^.terminal)
¦ ¦  then
¦ ¦   ¦begin
¦ ¦   ¦ if (c^.tsym <> terminet[t])
¦ ¦   ¦ then
¦ ¦   ¦ ¦begin
¦ ¦   ¦ ¦ terminet[t] := c^.tsym;
¦ ¦   ¦ ¦ t := t + 1;
¦ ¦   ¦ ¦end
¦ ¦   ¦end
¦ ¦  else
¦ ¦   lastflag[c^.nsym] := true;
¦ ¦   /* set for terminate trace */
¦ ¦end;
¦ q := a
¦end /*expression*/;
```

```
procedure parse(goal: tblrange; var match: boolean);
var
  s: pointer;
¦begin
¦ s := k[goal].entry;
¦ ¦repeat
¦ ¦ if s^.terminal
¦ ¦ then
¦ ¦ ¦begin
¦ ¦ ¦ if s^.tsym = sym
¦ ¦ ¦ then
¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ match := true; getsym
¦ ¦ ¦ ¦end
¦ ¦ ¦ else match := (s^.tsym = '*                    ')
¦ ¦ ¦end
¦ ¦ else parse(s^.nsym, match);
¦ ¦ if match
¦ ¦ then s := s^.suc
¦ ¦ else s := s^.alt
¦ ¦until s =nil
¦end /*parse*/;
```

```
¦begin /*productions*/
¦
¦ /* initialization */
¦ inpfg := true;
¦ for j := 1 to prime do
¦ hash[j] := -1;
¦ prtlncnt := Ø;
¦ prtlnfg := false;
¦ prtlnum := true;
¦ prtlnptr := Ø;
¦
¦ /* special characters for nice output */
¦ lf := chr(1Ø);
¦ fill := chr(127);
¦ cr := chr(13);
¦ alt := chr(124);
¦
¦ readnxchr;
¦ topheader := Ø;
¦ getsym;
¦ t := 1;
¦
¦ /* initialize lastflag for terminate trace */
¦ lastflag[Ø] := true;
¦ for i:= 1 to maxheader do
¦ lastflag[i] := false;
¦
¦ /* initialize matrixes */
¦ for i := Ø to maxheader do
¦ for j := Ø to maxheader do
¦ first [i,j] := false;
¦ within := first;
¦ while symbol'type <> separator do
¦ ¦begin /* construct data structure for transmission
¦ ¦        grammar rules. the date structuer can be used
¦ ¦        to parse action sequence, to validate tg and
¦ ¦        to list cycles */
¦ ¦
¦ ¦ find (sym, h);
¦ ¦ currnsym := h;
¦ ¦ altcnt[currnsym] := 1;
¦ ¦ getsym;
¦ ¦ if symbol'type = equal
¦ ¦ then getsym
¦ ¦ else error;
¦ ¦ expression (k[h].entry,p); p^.alt := nil;
¦ ¦ if symbol'type <> period
¦ ¦ then error;
¦ ¦ getsym;
¦ ¦end;
```

```
/* start protocol validation */
h := Ø;  ok := true; /* check whether all symbols are
                      defined */
while h <> topheader do
|begin
| if k[h].entry = nil
| then
| |begin
| | writeln('            undefined symbol --> ',
| |         k[h].sym);
| | ok := false
| |end;
| h := h + 1;
|end;

/* proper termination analysis */
writeln;
writeln('               proper termination analysis');
writeln('               ***************************');
writeln;
if (t > 2)
then
writeln('               ** warning ** more than one',
        ' exits');

writeln;
write('              the terminet symbol is: ');
for i := 1 to (t - 1) do
write(terminet[i]:20);
writeln;

n := topheader - 1;
/* relation matrix within and first are constructed during
 prasing for efficiency, we construct transitive closure
 of relations first and within */
firstall := first;  withinall := within;
for i := Ø to n do
for j := Ø to n do
|begin
| if firstall [j,i]
| then
| for k1 := Ø to n do
| firstall [j,k1] := firstall [j,k1] or firstall [i,k1];
| if withinall [j,i]
| then
| for k1 := Ø to n do
| withinall [j,k1] := withinall [j,k1] or
| withinall [i,k1];
|end;
```

```
¦ /* left recursion checking */
¦ for i := Ø to n do
¦ if firstall [i,i]
¦ then
¦ ¦begin
¦ ¦ writeln ('                 left recursion --> ',k[i].sym);
¦ ¦ ok := false;
¦ ¦end;
¦ if ok
¦ then writeln('                 no left recursion');
¦
¦ writeln;
¦ writeln('                 reachability analysis');
¦ writeln('                 *********************');
¦ writeln;
¦
¦ /* reachability analysis */
¦ for i := 1 to n do
¦ if not(withinall[Ø,i])
¦ then
¦ ¦begin
¦ ¦ writeln('                 unreachable nonterminal ---> ',
¦ ¦         k[i].sym:2Ø);
¦ ¦ ok := false;
¦ ¦end;
¦ if (ok)
¦ then writeln('                 no unreachable nonterminal');
¦
¦ /* check for <u> ::= t for  some t in vt+ */
¦ for i := Ø to n do
¦ mark[i] := false;
¦ if (ok)
¦ then
¦ ¦begin /* nonterminal termination checking */
¦ ¦ ¦repeat /* check all  nonterminals to see if can
¦ ¦ ¦         derive a terminal string */
¦ ¦ ¦ newmark := false;
¦ ¦ ¦ for i := Ø to n do
¦ ¦ ¦ if not(mark[i])
¦ ¦ ¦ then
¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ p := k[i].entry;
¦ ¦ ¦ ¦ stop := false;
¦ ¦ ¦ ¦ while
¦ ¦ ¦ ¦ ((p^.suc<>nil) and (not(stop)))
¦ ¦ ¦ ¦ do
¦ ¦ ¦ ¦ if (p^.terminal)
¦ ¦ ¦ ¦ then
¦ ¦ ¦ ¦ p := p^.suc
¦ ¦ ¦ ¦ else
¦ ¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ ¦ if (mark[p^.nsym])
¦ ¦ ¦ ¦ ¦ then
¦ ¦ ¦ ¦ ¦ p := p^.suc
¦ ¦ ¦ ¦ ¦ else
¦ ¦ ¦ ¦ ¦ stop := true
¦ ¦ ¦ ¦ ¦end;
¦ ¦ ¦ ¦ if not(stop)
¦ ¦ ¦ ¦ then
¦ ¦ ¦ ¦ ¦begin
¦ ¦ ¦ ¦ ¦ mark[i] := true;
¦ ¦ ¦ ¦ ¦ newmark := true;
¦ ¦ ¦ ¦ ¦end;
¦ ¦ ¦ ¦end;
¦ ¦ ¦ stop := false;
¦ ¦ ¦ i := 1;
¦ ¦ ¦ while (i < n) and (mark[i]) do
¦ ¦ ¦ i := i + 1;
¦ ¦ ¦ if (i = n)
¦ ¦ ¦ then newmark := false;
¦ ¦ ¦until
¦ ¦   ((stop) or (not(newmark)));
¦ ¦ if not(newmark)
¦ ¦ then
¦ ¦ writeln('                 all nonterminals can terminate',
¦ ¦         ' properly.')
¦ ¦ else
¦ ¦ for i := Ø to n do
¦ ¦ if not(mark[i])
¦ ¦ then
¦ ¦ writeln('                 the nonterminal',k[i].sym:2Ø,
¦ ¦         'can not terminate properly'); writeln;
¦ ¦end;
¦ if not(ok)
¦ then goto 99;
¦ writeln;
¦ writeln('                 action sequence testing');
¦ writeln('                 ************************');
¦ writeln;
¦ /*goal symbol*/
¦ getsym; find(sym,h);
¦ /*sentences*/
¦ getsym;
¦ while not(eof(input)) do
¦ ¦begin
¦ ¦ parse(h,ok);
¦ ¦ if (symbol'type <> period)
¦ ¦ then ok := false;
¦ ¦ getsym;
¦ ¦ writeln('            the syntax of the above action');
¦ ¦ write('                                    sequence is: ');
¦ ¦ if ok
¦ ¦ then
¦ ¦ writeln('correct')
¦ ¦ else writeln('incorrect');
¦ ¦ ok := true;
¦ ¦end;
¦ if (ok)
¦ then cycles; /* list all the cycles */
99:
¦end.
```

BIBILIOGRAPHY

[Bochmann75] Bochmann, G. V., "Logical Verification and Implementation of Protocols," Proc. Fourth Data Communication Symp., Quebec City, Canada, Oct. 1975, pp. 7.15-20.

[Bochmann77a] Bochmann, G. V., and Gecsei, J., "A Unified Method for the Specification and Verification of Protocols," Proc. IFIP Congress, Toronto, 1977, pp. 229-234.

[Bochmann77b] Bochmann, G. V., and Chung, R. J., "A Formalized Specification of HDLC Classes of Procedures," Proc. NTC, Dec. 1977, pp. 03A:2.1-2.11.

[Bochmann78] Bochmann, G. V., "Finite State Description of Communication Protocols," Proc. of the Computer Network Protocols Symposium, Liege, Belgium, Feb. 1978, pp. F3/1-11.

[Caine75] Caine, S. H., and Gordon, E. K., "PDL: A Tool for Software Design," Proc. National Computer Conference, Vol. 44, 1975, pp. 271-276.

[Cerf77] Cerf, V., "Specification of Internet Transmission Control Program," TCP Version 2, Mar. 1977.

[Chen75] Chen, R. C., "Representation of Process Synchronization," Proc. ACM SIGCOMM/SIGOPS Interprocess Communication Workshop, Santa Monica, Calif., Mar. 1975, pp. 37-42.

[Clark78] Clark, D. D., Pogran, K. T. and Reed, D. P., "An Introduction to Local Area Networks," Proc. IEEE, Vol. 66, No. 11, Nov. 1978, pp. 1497-1516.

[Coway78] Coway, R., A Primer on Disciplined Programming, Workshop Publishers, Cambridge, Mass., 1978.

[Davies73] Davies, D. W., and Barber, D. L. A., Communication Networks for Computers, Wiley, New York, 1973, pp 383-384.

[Danthine78] Danthine, A., and Bremer, J.,"Modelling and Verification of End-to-End Transport Protocols," Computer Networks, Vol. 2, No. 4/5, Oct. 1978, pp. 381-395.

[Eilenberg74] Eilenberg, S., Automata, Languages, and Machines, Academic Press, New York, Vol. A, 1974, pp. 19-21.

[Ellis77a] Ellis, C. A., "A Robust Algorithm for Updating Duplicated Databases," Proc. Second Berkely Workshop on Distributed Data Management and Computer Networks, May 1977, pp. 146-158.

[Ellis77b] Ellis, D. J., "Formal Specifications for Packet Communication Systems," MIT/LCS/TR-189, MIT Laboratory for Computer Science, Nov. 1977, 138pp.

[Feinler78] Feinler, E., Postel, J., ARPANET Protocol Handbook, Network Information Center, Stanford Research Institute, Menlo Park, CA, Jan. 1978.

[Fletcher78] Fletcher, J. G., Watson, R. W., "Mechanisms for a Reliable Timer-Based Protocol," Computer Networks, Vol. 2, No. 4/5, Oct. 1978, pp. 271-290.

[Fraser76] Fraser, A. G., "The Present Status and Future Trends in Computer/Communication Technology," IEEE Communications Society Magazine, Vol. 14, No. 5, Sept. 1976, pp. 10-19.

[Glass79] Glass, R. L., Software Reliability Guidebook, Prentice Hall, New Jersey, 1979.

[Good77] Good, D. I., "Constructing Verified and Reliable Communications Processing Systems," Software Engineering Notes, Vol. 2, No. 5, Oct. 1977, pp. 8-13.

[Gouda76a] Gouda, M. G., and Manning, E. G., "Protocol Machines, a Concise Formal Model and its Automatic Implementation," Proc. International Conference on Computer Communication, Toronto, Aug. 1976, pp. 346-350.

[Gouda76b] Gouda, M. G., and Manning, E. G., "On the modelling, Analysis an Design of Protocols-A special class of software structures," Proc. 2nd International Conference on Software Engineering, San Francisco, Calif., Oct. 1976, pp. 256-262.

[Gries71] Gries, D., Compiler Construction for Digital Computers, John Wiley & Sons, New York, 1971.

[Hajek78] Hajek, J., "Automatically Verified Data Transfer Protocols," Proc. ICCC 1978, Kyato, Japan, Sept. 1978, pp. 749-756. Also see progress reports in Computer Communication Review, Jan. 1978 and Jan. 1979.

[Hajek79] Hajek, J., "Protocols Verified by APPROVER," Computer Communication Review, ACM-SIGCOMM, January 1979, pp. 32-34.

[Harangozo77] Harangozo, J. "An Approach to Describing a Data Link Level Protocol with a Formal Language," Proc. Fifth Data Communication Symposium, Snowbird, Utah, Sept. 1977, pp. 4.37-49.

[Harangozo78] Harangozo, J., "Protocol Definitions with Formal Grammars," Proc. Symp. Computer Network Protocols, Liege, Belgium, Feb. 1978, pp. F6.1-10.

[Heart70] Heart, F., Kahn, R., and Ornstein, S., "The Interface Message Processor for the ARPA Computer Network," Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N. J. pp. 551-567

[Hill73] Hill, F. J. and Peterson, G. R., Digital Systems: Hardware Organization and Design, John Wiley and Sons, New York, 1973.

[Hopcroft69] Hopcroft, J. E., and Ullman J. D., Formal Languages and their Relation to Automata, Addison-Wesley, Massachusetts, 1969.

[ISO78] Reference Model of Open Systems Architecture (Version 3), TC9/SC16 N117, International Organization for Standarization, Nov. 1978.

[Kleinrock76] Kleinrock, L., "ARPANER Lessons," Proc. ICCC, Philadelphia, PA, June, 1976, pp. 20.1-6.

[Kleinrock78] Kleinrock, L., "Principles and Lessons in Packet Communications," Proc. IEEE, Vol. 66, No. 11, Nov. 1978, pp. 1320-1329.

[Liu78] Liu, M. T., "Distributed Loop Computer Networks," in Advances in Computers, Vol. 17, pp. 163-221.

(M. C. Yovits, editor), Academic Press, New York, 1978.

[Lynch68] Lynch, W. C., "Reliable Full-Duplex File Transmission over Half-Duplex Telephone Lines," Commun. ACM, June 1968, pp. 407-410.

[McKenzie72] Mckenzie, A., "Host/Host Protocol for the ARPA Network," NIC# 8246, Jan. 1972; Published in ARPANET Protocol Handbook, NIC#7104, Rev. 1, Apr 1976. pp. 41-47.

[Merlin76a] Merlin, P. M., "A Methodology for the Design and Implementation of Communication Protocols," IEEE Trans. Commun., Vol. COM-24, June 1976, pp. 614-621.

[Merlin76b] Merlin, P.M., and Farber, D. J., "Recoverability for Communication Protocols-Implications of a Theoretical Study," IEEE Trans. Commun., Vol. COM-24, No. 9, Sept. 1976, pp. 1036-1043.

[Merlin79] Merlin, P. M., "Specification and Validation of Protocols", IEEE Trans. Commun., Vol. COM-27, No. 11, Nov. 1979, pp.1671-1680.

[Naur63] Naur, P., ed., "Report on the Algorithmic Language ALGOL60," Commun. ACM, Vol.6, Jan. 1963, pp. 1-17.

[Nayer75] Nayer, W., "A Loop-free Adaptive Routing Algorithm for Packet Switched Networks," Proc. Fouth Data Communication Symposium, Quebec City, Canada, Oct. 1975, pp. 7.9-14.

[Parnas75] Parnas, D. l., "The Influence of Software Structure on Reliability," International Conference on Reliable Software, Los Angeles, 1975. pp. 358-362.

[Postel74] Postel, J. B., "A Graph Model Analysis of Computer Communication Protocols," Ph.D. Dissertation, UCLA ENG-7410, Jan. 1974, 184pp.

[Ray76] Ray, W. S., and Cerf, V. G., "Compatibility or Chaos in Comminication," Datamation, Mar. 1976, pp. 50-55.

[Raubold76] Raubold, E. and Haenle, J., "A Method of Deadlock-free Resource Allocation and Flow Control in Packet Networks," Proc. ICCC, Toronto, Canada, Aug. 1976, pp.483-487.

[Segall78] Segall, A., Merlin, P. M. and Gakkager, R. G., "A Recoverable Protocol for Loop-free Distributed Routing," Proc. ICC, Toronto, Canada, June 1978, pp. 3.5.1-5.

[SNA78] SNA Format and Protocol Reference Manual: Architecture Logic, SC30-3112-01, IBM Corp., June 1978.

[Stenning76] Stenning, N. V., "A Data-Transfer Protocol," Computer Networks, Vol. 1, No. 2, Sept. 1976, pp. 99-110.

[Sundstrom77] Sundstrom, R. J., "Formal Definition of IBM's System Network Architecture," Proc. NTC, Los Angeles, Dec. 1977, paper 3A.1.

[Sunshine75] Sunshine, C. A., "Interprocess Communication Protocols for Computer Networks," Stanford Univ., Digital Syst. Lab., Tech. Report 105, Dec. 1975, 158pp. (Ph.D. Thesis).

[Sunshine76] Sunshine, C. A., "Survey of Communication Protocol Verification Techniques," Symposium on Computer Networks : Trends and Appl., Gaithersburg, Nov. 1976, pp. 24-26.

[Sunshine78a] Sunshine, C. A., "Survey of Protocol Definition and Verification Techniques," Proc. Computer Network Protocols Symp., Liege, Belgium, Feb. 1978, Paper F1.

[Sunshine78b] Sunshine, C. A., Dalal, Y. K., "Connection Management in Transport Protocols," Computer Networks, Vol. 2, No. 6, Dec. 1978, pp. 454-473.

[Sunshine79] Sunshine, C. A., "Formal Techniques for Protocol Specification and Verification," IEEE Computer Society Magazine, Vol. 12, No. 9, Sept. 1979, pp. 20-27.

[TCP79] Transmission Control Protocol, TCP (Version 4), USC/ISI, (J. B. Postel, editor), Feb. 1979.

[Teng78a] Teng, A. Y., and Liu, M. T., "A Formal Approach to the Design and Implementation of Network Communication Protocols," Proc. COMPSAC 78, Nov. 1978, pp. 722-727.

[Teng78b] Teng, A. Y., and Liu, M. T., "A Formal Model for Automatic Implementations and Logical Validation of Network Communication Protocols," Proc. Computer Networking Symposium, Gaithersburg, Maryland, Dec. 1978. pp. 114-123.

[Teng79] Teng, A. Y., "Protocol Constructions for Communication Networks, (abstract)" Proc. ACM Computer Science Conference, Dayton, Ohio, Feb. 1979, p. 38.

[Walden72] Walden, D. C., "A System for Interprocess Communication in a Resource Sharing Computer Network," Commun. ACM, Vol. 15, Apr. 1972. pp. 221-230.

[Warshall62] Warshall, S., "A Theorem on Boolean Matrix," JACM 9, Jan. 1962, pp. 11-12.

[West78a] West, C. H., and Zafiropulo, P., "Automated Validation of a Communication Protocol: the CCITT X.21 recommendation," IBM J. Res. Develop., Jan. 1978, Vol. 22, No. 1, pp. 60-71.

[West78b] West, C. H., "A General Technique for Communications Protocol Validation," IBM J. Res. Develop., 22, July 1978, pp. 393-404.

[Wirth71] Wirth, N., "Program Development by Step-wise Refinement," Commun. ACM, Vol. 14, No. 4, April 1971, pp. 221-227.

[Wirth76] Wirth, N., Algorithm ± Data Structures ≡ Programes, Prentice-Hall, New Jersey, 1976.

[Zafiropulo78] Zafiropulo, P., "Design Rules for Producing Logically Complete Two-Process Interactions and Communications Protocols," Proc. COMPSAC 78, Nov. 1978, pp. 680-685: