INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Umbaugh, Lloyd David

AUTOMATED TECHNIQUES FOR SPECIFICATION AND VALIDATION OF COMMUNICATIONS PROTOCOLS

The Ohio State University

University Microfilms International

Copyright 1983 by Umbaugh, Lloyd David All Rights Reserved
AUTOMATED TECHNIQUES FOR SPECIFICATION AND VALIDATION
OF COMMUNICATIONS PROTOCOLS

DISSERTATION

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

By

Lloyd David Umbaugh, BS, M.S.

* * * *

The Ohio State University
1983

Reading Committee:

Ming T. Liu
Sandra A. Mamrak
Jayashree Ramanathan

Approved By

Ming T. Liu
Adviser
Department of Computer
and Information Science
This dissertation is dedicated to my wife, Loretta, with grateful thanks for her love and support over the years. Her confidence and encouragement have been of immeasurable value during this time of study and research.
ACKNOWLEDGMENTS

I gratefully acknowledge the help of many people who have made this work possible. Hua-An Lin and Wael Hilal Bahaa-El-Din have spent many hours in discussion sharing their ideas and commenting on mine. Charles J. Graff has been a source of continuing support. Sandra A. Mamrak, Ming T. Liu, Wael Hilal Bahaa-El-Din, K. C. Fung and Tamer M. Ozsu all participated in and contributed extensively to the work on multi-destination protocols for packet radio networks reported on in chapter 5. The facilities and staff of the IRCC/CIS Computer Laboratory were very valuable both for the computational capability and the text processing facilities used. The members of my reading committee, Dr. S. A. Mamrak and Dr. J. Ramanathan, have contributed many helpful suggestions regarding both the content and structure of this dissertation. Finally, my advisor, Dr. M. T. Liu has been a source of help, encouragement, direction and wisdom. I have valued, most of all, his friendship and example of industry.

This research was supported by US Army CECOM, CENCOMS, Fort Monmouth, New Jersey under Contract No. DAAK81-K-0104. The views, opinions and/or findings contained in this dissertation are those of the author and should not be construed as an official Department of the Army position, policy or decision.
VITA

November 23, 1935 . . Born - South Bend, Indiana

1958 . . . . . . . . BS, United States Military Academy, West Point, New York

1958-1978 . . . . Officer, United States Army

1978 . . . . . . . . Teaching Assistant, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1979-1980 . . . . Administrative Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1979 . . . . . . . . M.S., The Ohio State University, Columbus, Ohio

1981-1983 . . . . Research Assistant, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

PUBLICATIONS

"Adaptive Multi-Destination Protocols for Packet Radio Networks," with Ming T. Liu, Record ICC81, pp. 73.2.1-6, June 1981.


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Architecture. Professor Ming T. Liu

Studies in Computer Operating Systems.
Professor Sandra A. Mamrak

Studies in Computer Programming Languages.
Professor Jayashree Ramanathan
<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. INTERRODUCTION TO PROTOCOL VALIDATION</td>
<td>1</td>
</tr>
<tr>
<td>The Protocol Designer Needs Help.</td>
<td>3</td>
</tr>
<tr>
<td>Approaches to Protocol Validation</td>
<td>6</td>
</tr>
<tr>
<td>Objectives of the Multi-Destination</td>
<td>8</td>
</tr>
<tr>
<td>Protocol Validation System</td>
<td>11</td>
</tr>
<tr>
<td>Description and Organization of this Work</td>
<td></td>
</tr>
<tr>
<td>II. PROTOCOLS, SPECIFICATION AND VALIDATION</td>
<td>16</td>
</tr>
<tr>
<td>Protocol Architecture</td>
<td>16</td>
</tr>
<tr>
<td>Specification of the Protocol Layer</td>
<td>22</td>
</tr>
<tr>
<td>Validation of the Peer-Peer Protocol</td>
<td>31</td>
</tr>
<tr>
<td>III. SPECIFICATION AND VALIDATION OF THE X.21 INTERFACE</td>
<td>52</td>
</tr>
<tr>
<td>IBM Zurich Validation</td>
<td>52</td>
</tr>
<tr>
<td>UCLA Validation</td>
<td>54</td>
</tr>
<tr>
<td>Transmission Grammar (TG) Validation</td>
<td>56</td>
</tr>
<tr>
<td>DTE/DCE State Diagrams</td>
<td>61</td>
</tr>
<tr>
<td>Automated Validation of the Validation Automaton</td>
<td>73</td>
</tr>
<tr>
<td>Comparisons</td>
<td>75</td>
</tr>
<tr>
<td>Conclusion</td>
<td>80</td>
</tr>
<tr>
<td>VI. VALIDATION OF THE TRANSMISSION CONTROL PROTOCOL</td>
<td>82</td>
</tr>
</tbody>
</table>
# Table of Contents

The Transmission Control Protocol ........................................ 83
The Transmission Grammar Model ........................................... 92
Validation Results .................................................................. 107
Summary ............................................................................. 114

V. CONNECTION FREE MULTI-DESTINATION PROTOCOLS
 IN PACKET RADIO NETWORKS .................................................. 117

Preliminary Definitions ......................................................... 118
PRNET Model ....................................................................... 119
Multi-Destination Addressing and Routing ............................... 131
Multi-Destination Protocols .................................................... 146
Intermediate-Level Protocols .................................................. 153

VI. AUTOMATED VALIDATION OF MULTI-DESTINATION
 PROTOCOLS ........................................................................... 161

Protocol Design and Validation .............................................. 162
Specific Advances Necessary .................................................. 167
Organization of the Automated Validation Program .................. 173
Illustrative Example ................................................................ 179

VII. SUMMARY AND CONCLUSIONS ........................................... 185

Summary ............................................................................. 185
Areas for Future Research ..................................................... 190
Conclusions ........................................................................ 192

APPENDIXES

A. GRAMMAR PRODUCTION RULES, X.21 ............................ 196
B. VALIDATION PROGRAM FOR MULTI-DESTINATION PROTOCOLS

C. MULTI-VALID OUTPUT FOR SIMPLE TWO-PARTY PROTOCOL . 237

BIBLIOGRAPHY ........................................................................ 244
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Error States of the Validation Automaton</td>
<td>73</td>
</tr>
<tr>
<td>2. Various Types of Protocol and Available Function</td>
<td>154</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Seven-layer Reference Model and Peer-to-Peer Protocol Interrelation.</td>
<td>20</td>
</tr>
<tr>
<td>2.</td>
<td>Layer Service Hierarchy</td>
<td>21</td>
</tr>
<tr>
<td>3.</td>
<td>Illustrative Protocol Modeled as a Finite State Graph.</td>
<td>35</td>
</tr>
<tr>
<td>4.</td>
<td>Petri Net Model of Illustrative Example.</td>
<td>39</td>
</tr>
<tr>
<td>5.</td>
<td>Token Machine for Illustrative Example.</td>
<td>40</td>
</tr>
<tr>
<td>6.</td>
<td>Transmission Grammar Model of Illustrative Example.</td>
<td>44</td>
</tr>
<tr>
<td>7.</td>
<td>VA State Matrix Channel Queues.</td>
<td>44</td>
</tr>
<tr>
<td>8.</td>
<td>General Global State Matrix for Protocol Validation.</td>
<td>46</td>
</tr>
<tr>
<td>9.</td>
<td>VA Model for Illustrative Example.</td>
<td>48</td>
</tr>
<tr>
<td>10.</td>
<td>Global VA for Illustrative Example.</td>
<td>51</td>
</tr>
<tr>
<td>11.</td>
<td>Call Establishment Phase for Circuit Switched Service.</td>
<td>58</td>
</tr>
<tr>
<td>12.</td>
<td>Clearing Phase and Quiescent States for Circuit Switched Service.</td>
<td>59</td>
</tr>
<tr>
<td>13.</td>
<td>DTE State Diagram</td>
<td>63</td>
</tr>
<tr>
<td>14.</td>
<td>DCE State Diagram</td>
<td>64</td>
</tr>
<tr>
<td>15.</td>
<td>TCP Header Format</td>
<td>85</td>
</tr>
<tr>
<td>16.</td>
<td>TCP Connection State Diagram</td>
<td>88</td>
</tr>
<tr>
<td>17.</td>
<td>Typical PRNET</td>
<td>123</td>
</tr>
<tr>
<td>18.</td>
<td>Message Formats for Protocol Layers</td>
<td>126</td>
</tr>
<tr>
<td>19.</td>
<td>Address Field Formats</td>
<td>135</td>
</tr>
<tr>
<td>Number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>20</td>
<td>Sample Addressing Scheme Implementation</td>
<td>133</td>
</tr>
<tr>
<td>21</td>
<td>Example of Distributed Branch Routing</td>
<td>144</td>
</tr>
<tr>
<td>22</td>
<td>Adaptive Multi-Destination Protocol</td>
<td>169</td>
</tr>
<tr>
<td>23</td>
<td>Global State Matrix for Three-Party Protocol</td>
<td>165</td>
</tr>
<tr>
<td>24</td>
<td>Structure of a VA_node</td>
<td>176</td>
</tr>
<tr>
<td>25</td>
<td>Multi-Valid Output for Simple Example</td>
<td>181</td>
</tr>
</tbody>
</table>
1. INTRODUCTION TO MULTI-DESTINATION PROTOCOL VALIDATION

Computer communications is a topic which has greatly increased in importance. Although communication with a computer via a remote terminal is a capability dating back to the very earliest days of the electronic digital computer, for a long time most computer communications was accomplished by someone walking up to the machine, placing punched cards in a card reader and pressing the appropriate button or buttons. Today, computer communications has become much more complex. Not only are computers serving hundreds of remote terminals, as in airline reservation systems, but oftentimes the remote terminals are themselves computers. It is common to find computers linked together by a wide variety of means. Computers are accessed by dial up connections through the telephone switched network; they are connected together via leased or privately owned telephone grade voice frequency lines and via high bandwidth lines using terrestrial cable, microwave links, satellite links and fiber optic transmission cables. Computer networks are being built consisting of only a few computers or ranging into the hundreds. They cover very limited geographical areas, perhaps just one room, and extended geographical areas which may span continents.
or extend world wide. Communication may be broadcast via radio or over a bus accessible to all the computers connected to it. They may be routed from source to destination through intermediate store and forward nodes in a mesh network or forwarded from one node to another around a loop network. When the complex of interconnections so far described is inadequate the networks themselves are connected together and internetwork communications becomes possible.

Into this recipe for chaos comes the protocol designer to attempt to provide the rules needed to bring order from chaos. Communications protocols are simply rules and definitions which prescribe the manner in which communications takes place, the meaning of what is communicated and what communications is appropriate (or permissible) under what prescribed conditions. At the very lowest level, computer communications protocols prescribe the physical medium over which information is to be transmitted and received and how that information is to be physically represented on the transmission medium. At higher levels, they attempt to overcome inherent unreliability in lower levels, to control the flow of communications and to provide mechanisms for addressing, routing and delivery. At still higher levels, protocols may provide for the communication of objects of a higher level of abstraction such as letters, records, files or transactions. In short, one might say that the purpose of protocols is to make communication with and
among computers easier and more useful.

Protocol specification has primarily been accomplished by use of natural language text augmented with some formal technique or by some formal technique augmented by natural language. Surveys of protocol specification techniques can be found in [2, 8, 43, 62]. The two main categories of formal protocol description are based on programming languages and on automata (or state transition models). Hybrid techniques combining programming language techniques with various forms of automata are also finding increased use. The programming language techniques can obviously make use of a wide variety of existing programming languages or pseudo languages. Automata based techniques also employ a wide variety of representations. Some examples of both programming language protocol descriptions and automata based protocol specifications will be discussed in the next chapter.

1.1. The Protocol Designer Needs Help

The task of the protocol designer is an important one. The successful operation of any computerized system based on communication depends on correct operation of the communications system. That depends, among other things, on correctly designed and implemented communications protocols. If correct protocol design were an easy task this dissertation would be unnecessary.
Unfortunately, protocol design based on traditional informal techniques is far more likely to produce incorrect than correct protocols. Correct design of any system is a significant problem in itself. For communications protocols, the problem is complicated by the fact that protocols are implemented by two or more entities operating in parallel. The entities operate remotely from each other and are able to coordinate their activities only by exchanging messages over some, usually unreliable, communications medium.

Two approaches open to the protocol designer are to focus on defining the allowable message sequences which can be exchanged over the transmission medium or to define the order in which each individual entity can send and receive messages. The result of both approaches should be the same, but the different viewpoints can lead to different results.

In the first, or centralized design, approach the designer risks specifying a message interchange which is not realizable by the activities of two loosely coordinated communicating entities or is realizable only by entities whose actions can produce other sequences of message interchange outside those specified. The second approach which focuses on the design of the individual entities can result in unforeseen events for which the design provides no response.

The difficulties described in the preceding paragraph lead to
an incomplete specification of the protocol. The protocol entities implemented from incompletely specified protocols are almost certain to be defective. They may produce unwanted results and may not include proper responses for some actions which can arise in use. Such incorrect implementations will lead to communications failure. Another serious difficulty is the unforeseen deadlock state. Two or more communicating entities, waiting for some action by other entities which will never occur, are deadlocked. Defects of the type described have been found in real protocol designs.

The problem, then, is to provide some way to improve the likelihood that the protocol designer will produce a complete protocol specification, free from the possibility of deadlock. The most straightforward approach is to check the protocol specification in some way to detect errors. The term validation [62] has been used for correctness (partial) testing of protocols. Validation systems which allow the designer to check for many of the more serious design defects can be of great help in producing more correct protocol designs.
1.2. Approaches to Protocol Validation

While much work has been done in the area of protocol specification and validation and many techniques have been applied, none of the results has been entirely satisfactory. Protocol validation techniques are based on some sort of formal specification. Most such specifications fit into one of two categories -- programming language based specifications or those based on state transition diagrams.

Validation techniques for protocols specified using programming language specifications are those common to program correctness proof. While program correctness proofs can be helpful in increasing one's confidence that an algorithm or program is correct, they are quite difficult in practice. The amount of effort which must be expended in a correctness proof is usually far greater than that required to develop the algorithm or program. The insight needed to develop program proofs is quite considerable and frequently beyond the reach of people not well versed in mathematics. Correctness proof does have the advantage that it can show that a protocol design performs the desired function as well as showing the absence of certain defects.

Validation of protocols specified by state transition techniques employs a procedure somewhat akin to symbolic execution of a program. The procedure is to model the global state of the
communications system representing the communicating entities and the messages in transit between them. A graph of all the global states reachable from the initial global state is produced. It includes the global states as nodes and directed arcs connecting each state to those directly reachable from it. Differences in such validation techniques arise from the variety of specification techniques used. A number of these formal specification techniques will be described in the next chapter. They include finite state graphs, Petri nets, and formal language techniques. They suffer from several basic deficiencies.

A number of problems are common to the state transition class of protocol specification techniques. The most serious is the combinatorial explosion of the state space of the global model as the complexity of the individual entities increases or as the model of communications medium becomes more complex. Also, for complex real protocols, the graphical techniques become too complicated for a complete representation. All of the state transition techniques are unsuited for modeling variables which can take on large numbers of values.

Most published work in the area of protocol validation has presented some such specification system and a validation approach based upon it demonstrated on a simple protocol of limited or chiefly academic interest. A system with the power to tackle
meaningful protocols is sorely needed.

1.3. Objectives of the Multi-destination Protocol Validation System

The purpose of this dissertation is to explore one possible way of providing the formal specification and validation that will make possible the correct, clear and unambiguous specification of network communications protocols.

The system presented is a powerful, efficient and flexible tool which is capable of validating real communications protocols. It is capable of validation not only of single-destination (or two-party) protocols, but also of validation of multi-destination (or multi-party) protocols. For that reason we call the system Multi-Valid.

The design objectives for Multi-Valid can be broken down into several major goals.

1. To base the validation system on a specification technique which is able to concisely represent complex communications protocols in a clear understandable form.

2. To provide the specification in a form which can easily be processed by computers.

3. To automate the validation process.

4. To include the capability to automatically validate multi-destination protocols.

5. To design the automated procedure in such a way that it is flexible and adaptable to improvements in modeling and validation techniques.

The Multi-Valid system is based on a formal grammar
specification technique called Transmission Grammar (TG) [67]. This technique provides a concise specification of exactly what events are expected and the appropriate response for each possible state of a communicating entity. Unlike graphical specification techniques, it can model complex protocols without presenting the user with a snarl of criss-crossing lines.

An advantage of formal grammars is that a computer can easily accept a formal grammar specification as input for processing. A program which can parse such a grammar specification and construct a data structure to represent that grammar is already available [67].

No validation scheme could be considered satisfactory unless it could automate the validation process. The importance of that objective may not be apparent to someone who has not attempted a protocol validation, but it was one of our most essential goals. The importance of this goal was recognized early in this research. An early validation effort described in chapter 3 demonstrated very clearly that the process of validation was far too time consuming and error prone if done by hand. It may be possible (though arduous) to perform validation on simple protocols. As protocols become more complex, the effort grows beyond human capability. The automated validation program for Multi-Valid is described in chapter 6.

Because multi-destination protocols are more complex than
two-party protocols, a multi-party validation system is both more urgently needed and more difficult to provide. The Multi-Valid program avoids unnecessary complexity where possible while providing a flexible system able to handle up to nine communicating entities. It is easily adaptable to validation of n-party protocols for n greater than nine.

Great pains were taken to design the Multi-Valid program in such a way that it can be easily revised to account for new discoveries in validation techniques. The reasoning for this emphasis stems from the previously mentioned combinatorial state space explosion problem. It is essential that all possible measures be taken to control the complexity of a protocol model. The Multi-Valid program incorporates several features to reduce the size of the model. As new techniques are discovered, they can be incorporated into the program.

The Multi-Valid system meets the established goals. Beyond theoretical goals, it has been tested in actual practical validation of real protocols. Description of those validations is included in chapters 3 and 4.
1.4. Description and Organization of This Work

This dissertation describes the development of an automated protocol validation system based on the use of a formal grammar specification. The format of the grammar (called Transmission Grammar) is based on [67]. The advantages of the Transmission Grammar (TG) model—a clear, machine parseable specification, with an automated analysis tool—made it an attractive candidate for development into a useful validation tool. The TG model, as a state transition based modeling technique, shares the basic deficiencies referred to in section 1.2. The problems—state-space explosion, difficulty in representing variable values, requirement for fixed topology and fixed number of communicating parties—are discussed in more detail in section 2.3.1. If the inherent limitation of the TG model could be overcome, if an automated tool could be developed to construct the Validation Automaton (VA), and if it could be extended to multi-destination protocols, then it could be useful to the designer of some of the most complex protocols. The following chapters show that this has been done.

The TG model has been used to validate two communications protocols. The X.21 recommendation of the International Telegraph and Telephone Consultative Committee [9] was validated first. Following that effort, the connection establishment procedure of the Transmission Control Protocol (TCP) was validated. A number of
lessons were learned about the use of the TG model for specification and validation of protocols, and specific results were obtained regarding the protocols subject to validation. Of perhaps greater importance was the demonstration that the problems inherent in the TG model could be overcome and useful results could be obtained.

Chapter 2 describes a number of state transition techniques for protocol specification and validation. The Transmission Grammar (TG) model is described and illustrated in detail. The use of the TG model is further illustrated in chapter 3 by a description of the validation of the X.21 recommendation [9]. The TG validation of that interface is compared with two other validation techniques which have been applied to the same specification. The validation of X.21 showed that Multi-Valid could be useful for real protocols and formed a basis of understanding of the techniques of modeling a complex protocol which was important for extension of the effort to more difficult tasks. The validation of X.21 had been expected to be a straightforward application of procedures described in [67]. It turned out, however, that even for that interface protocol we faced the problem of modeling variable information. The problem was solved successfully.

A more severe test of Multi-Valid as an effective validation technique is presented in chapter 4. The connection establishment procedure for the Transmission Control Protocol (TCP) is far more
complex than that of X.21. Specification of TCP required facing the problem of modeling variables with a large value space. The TCP segment includes both a sequence number and an acknowledgment number each of which is 32 bits long. Both fields are important in modeling the TCP connection establishment procedure. Chapter 4 shows that ways can be found to overcome the problem of specifying such a value space and that a protocol as complex as TCP can be modeled by a TG sufficiently compact that the state space of the resulting state reachability graph remains tractable. The solution to other difficult modeling problems is also demonstrated. One such is the representation of an entity state which depends on several variables rather than one state variable. The automated program which is such a vital part of Multi-Valid had not been completed in time to apply it to the TCP validation. The extensive manual effort involved in that validation strongly emphasized the need for an automated tool. The techniques developed to model sequence numbers and complex entity states extended the capabilities of the TG model as a protocol specification and validation tool.

Chapter 5 discusses a particular network environment in which multi-destination communications can be effective. The protocols required for multi-destination communications are seen to involve increased complexity. In the areas of routing, addressing, connection management and reliability, multi-destination protocols
are more complex than their two party counterparts. Even so, the various functions of the multi-destination protocols can be separated into hierarchical layers and divided into independent functions within layers as can single-destination protocols. Such division is vital to control complexity in modeling and validation.

Chapter 6 describes the design of a program to automate the construction of a Validation Automaton to validate multi-destination protocols. The emphasis in the design of that program is on flexibility. The problem of controlling complexity in protocol models is deserving of further research. As new techniques are defined the program should be adaptable so that modifications can be made readily.

No claim of absolute correctness is made for the program. It has been designed with care and tested. No amount of testing, however, can guarantee the absence of errors in a computer program. Testing can only increase our confidence in the program's correctness. Inspection of programs is also a useful tactic. Though one might convince oneself through reading a sequence of program statements that they perform in a certain way, one can rarely (if ever) be certain that some subtlety of execution or data has not been overlooked which will cause the program to behave in some unexpected way. The same sort of problem exist with proofs of program correctness. A proof is convincing only to the extent that
a person can read it and conclude that it proves what it purports to
prove. Given the complexity of program proofs, it requires a rare
self-confidence to wager a large sum that no error of logic will
ever be found in such a proof. Testing and proofs can provide
increased confidence in the correctness of a program. They can not
provide unqualified assurance of correctness. The general technique
for protocol validation should be evident from a careful reading of
the program (see appendix B) and from the description provided in
chapter 6. Should an error appear in use the program should be as
easy to modify to correct the error as to adapt it to new validation
techniques.

The conclusions of chapter 7 summarize the contribution to the
art and science of protocol Validation made by this research.
2. PROTOCOLS, SPECIFICATION AND VALIDATION

This chapter introduces computer communications protocols, describes a number of techniques for protocol specification and validation. The Transmission Grammar (TG) and Validation Automaton (VA) used in this work are described and illustrated.

2.1. Protocol Architecture

The idea of subdividing large, complex tasks into smaller, more manageable units and of building complex structures on top of or using simpler building blocks is not new. In the area of software engineering the terms top down design, modular programming, structured programming, and step-wise refinement refer to such techniques. In the area of operating system design, Dijkstra first proposed a hierarchical software system in 1968 [15]. The T.H.E. system built an abstract machine on top of the physical machine and through a hierarchical system of layers, it built up a computer operating system. Each layer of the system used the services provided by the next lower layer. Services of lower layers
were available only indirectly through the higher level abstraction provided by intervening layers.

The structuring of protocols into a hierarchical system of levels, or layers seems quite natural and appropriate for two reasons. First, a divide and conquer approach seems essential to control the complexity of the problem. Second, simple, rudimentary communications services tend to be constructed first. More advanced services can then be constructed more easily using the simpler services as a base. Given a system constructed in such a hierarchical fashion, another advantage accrues. As in well structured modular programming it is possible to change the implementation of a module (or level) without affecting the other modules so long as the interface between modules remains constant. In fact, a higher level protocol layer might make use of the services of more than one protocol at the next lower level providing their service interfaces were compatible.

2.1.1. The ISO OSI Reference Model

A number of terms (layer, interface, service, protocol) have been used with little or no definition or explanation of their relationship one to another. These terms will be explained in terms of a particular protocol hierarchy or architecture — that of the ISO Reference Model for Open Systems Interconnection [31]. The International Organization for Standardization (ISO) has been developing a protocol architectural model which would
provide a standard for the exchange of information among terminals, computers, people, networks, processes and so on, that are "open" to one another for this purpose by virtue of the applicable standards. [31]

In addition to the International Organization for Standardization, the International Telegraph and Telephone Consultative Committee (CCITT) and the European Computer Manufacturers Association (ECMA) and many national standards bodies have been participating in the effort to develop the Reference Model of Open Systems Interconnection (RM/OSI). Though many existing protocol architectures vary from the seven layer division of the RM/OSI, the view of network architecture as a layered structure has become essentially universal. The wider adoption of the standardized terminology and partitioning of functions should strikingly resolve much of the confusion of conflicting terms and definitions.

The reference model is divided into seven layers, each of which is shown in Figure 1. At the lowest level, the physical layer provides the mechanical, electrical, functional and procedural details necessary to connect data link entities by means of bit transmission through the physical layer. Whether the bit transmission within the physical layer is achieved by a single data-circuit or by the concatenation of two or more data-circuits is hidden from the data link layer and higher layers. At the top of the hierarchy lies the applications layer which directly provides services to the users of the OSI environment. The applications
layer represents to the user the entire architectural structure which serves the purpose of providing an access path between users of the network.

It has been pointed out that the simpler lower level communications services tend to be developed first. Hence, it should not be surprising that the ISO committee was able to draw on more practical experience and demonstrably successful protocol designs at the lower level layers and that the functions to be performed are defined in more detail for lower level protocol layers. The design of a protocol layer involves three aspects. The first of these involves the interface to the adjacent lower layer. The specification of a protocol layer describes this interface in terms of the services provided by the lower layer. For the data link layer in the RM/OSI the physical layer provides a physical connection, bit transmission and identification of the data circuit and of its endpoints, as well as notification to the data link layer of detected fault conditions. Characteristics of the services will also be specified. These may be such things as the delivery of bits in the same order as they were transmitted, error rate, availability, transmission rate and transit delay. This data link layer/physical layer interface exists between each physical connection endpoint and a data link entity at that endpoint. A physical connection may have two endpoints or more for a multiple
<table>
<thead>
<tr>
<th>Layer</th>
<th>Application</th>
<th>Presentation</th>
<th>Session</th>
<th>Transport</th>
<th>Network</th>
<th>Data Link</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;-&gt;</td>
<td>&lt;-</td>
<td></td>
<td>&lt;----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

physical media for interconnection

Figure 1: Seven-layer Reference Model and Peer-to-Peer Protocol Interrelation
endpoint connection.

The second aspect of a protocol layer design which must be specified is that of the peer-to-peer protocol interrelation. A peer entity is an entity that is in the same layer as another reference entity. The interrelation between an n-entity and its peer entity at the endpoint of an n-connection is carried out using the n-1 service in accord with the rules of the n-level protocol.

The information exchange of the n-level protocol forms the basis for the third aspect of the protocol — the n-service. The n-service is the raison d'être of the n-protocol layer and provides the interface to the n+1 layer. These three aspects of the n-layer are depicted in Figure 2.
2.2. Specification of the Protocol Layer

The purpose of a protocol specification is to convey an understanding of the function and design of the protocol layer. Chief among the intended users of the protocol specification is the protocol implementor. For the purpose of guiding the implementation the protocol specification must be particularly clear and detailed regarding the interaction between the peer entities. It can often be the case that protocol endpoint entities are implemented by different people, perhaps in different hardware and software systems. For that reason, it is vital that the meaning of each event and the appropriate response to that event be entirely clear. At the same time, it is desirable for the implementor to have freedom regarding details of the implementation. In other words, the protocol specification should specify what an action or response should be but not how to achieve it. The implementor can then provide the required functions in the manner most appropriate for the hardware and software systems in which the entity is implemented.

The services provided by a protocol layer are usually specified in terms of a number of service primitives along with a number of parameters for the primitives. These primitives are frequently expressed in the form of parameterized procedure calls (see eg. [45, 50]). This is not intended to imply that the
implementation will necessarily be in the form of procedure calls. The interface can range from very simple to quite complex. The implementation becomes particularly complex when adjacent layers are implemented in separate pieces of equipment. If, for example, the transport layer is implemented in a host computer and the network layer is implemented in a front end processor or some sort of network interface processor, then a particular protocol must exist to provide communications between the two pieces of equipment. Service requests and responses are transmitted via this inter-equipment protocol. For such situations it might seem more appropriate to depict the interface as an interface layer rather than, as is done in Figure 2, as a line dividing two layers. Other implementations may find adjacent layers in the same user memory space communicating via shared memory or via procedure calls. One layer might be considered a system service and its service invoked via system calls. None of these considerations are the concern of the protocol layer designer and should be ignored by the protocol layer specification. The specification may contain suggestions for implementation techniques, but such suggestions should be considered as helpful comments rather than as directions.

The protocol specification for an n-layer usually will define the actions of an n-entity. One can find exceptions to this usual procedure, viz. the specification for CCITT Recommendation X.21 [9]
and the CCITT Recommendation X.25 [10]. These are, in fact, interface specifications rather than protocol specifications. They define the signals appearing at an interface between entities rather than define the actions of the entities to produce the interface signals. The circuit switched service and packet switched service provided for by the X.21 and X.25 recommendations respectively are usually thought of as being provided by a common carrier. The recommendations provide a standard interface between the user customer and the carrier. This might explain the decision to define the interface using a single state diagram rather than to specify the actions of the communicating entities separately. The introductory chapter mentioned that such an approach to protocol design could result in a design such that no implementation would satisfy the specification. Some of the problems which are discussed in the next chapter arise in part from this difficulty.

An n-layer protocol specification should define:

- the n-entity response to requests for service from the n + 1 user entity
- the requests for service placed on the n - 1 service
- the response to n - 1 service events and
- the delivery of appropriate status signals and information to the n + 1 user entity to provide the n-service.

Informal protocol description techniques have been used to design a majority of existing protocols but not without producing protocols
with some unintended and undesirable behavior. Formal techniques for specification and validation of communication protocols are needed to provide clarity in the specification and to provide assurance that the protocol designs will function as desired.

Consider the problem of two separate implementations of some protocol, perhaps in different networks which, although implemented separately, were at some time interconnected. The variety of messages which can flow between entities in two such implementations of the same protocol is staggering, but each must act on and react to each message in a manner entirely predictable from the protocol specification. Should the specification be ambiguous in some respect, we should not be surprised by differences in the implementations. The result could be fairly innocuous or quite serious. Compromise of network security, loss of messages, or disruption of network traffic by flooding the network with retransmitted duplicate messages are examples of possible serious consequences. Informal techniques used in the past to specify and design protocols have been beset with many of the same problems which led to the development of more formal specification and design techniques for other software systems.

Formal techniques for specification of network communication protocols attempt to satisfy several goals:

1. To provide such clear, unambiguous definition of the protocol that implementors can produce compatible
implementations from the specification alone.

2. To enable validation that a protocol defined by the specification satisfies general requirements for completeness, absence of deadlock, and progress or termination.

3. To provide such clear, unambiguous definition of the protocol as to allow verification that implementations meet their specification.

The ambiguity inherent in informal specification techniques is well known. The use of such formal methods as state diagrams attempts to overcome the ambiguity problem. Formal models are, however, not without weakness as a protocol specification tool. A model may be unambiguous without being easy to understand. Furthermore, what is specified may be clear and unambiguous, but it is possible that some important details may remain unspecified. What is unspecified, or the fact that the specification is incomplete, is usually hard to discern. Completeness of the specification is not easily achieved either with formal or with informal techniques. This has been shown to be particularly true for attempts to specify the interface between two protocol entities with a single state diagram as is done in the X.21 and X.25 interface recommendations [9, 10]. Validation of protocol specifications can, among other things, reveal such incompleteness. Most of the problems found in recent validation efforts on the X.21 interface [55, 70, 71] were errors of incomplete specification.

Validation can also reveal other problems in protocol design.
Specifically, these include deadlock and livelock conditions, instability, and communications channel overflow. By deadlock we refer to a condition in which some or all protocol entities cannot make further progress. The protocol system stops functioning. An example of such a situation is a case of two communicating processes each waiting for some communication from the other before proceeding. If the communications medium is empty (or at least void of the required communication) the processes are deadlocked. Livelock refers to a situation in which the two processes may still be active, but in which their action results in no effective communications progress. Instability is a term used to describe the tendency of a protocol not to return to a synchronized state if its operation is disturbed. Overflow refers to the capacity of the communications channel which interconnects the communicating protocol entities.

Many validation techniques to detect these problems use some form of reachability analysis in which the possible conditions of the network are generated from some starting condition. A condition from which neither protocol entity can proceed represents deadlock. A cyclic sequence of conditions with no way out is typical of livelock. Instability is manifested by a sequence of actions which take the network to an unsynchronized condition and fail to return to "normal" operation. On the contrary, a stable system will return
to "normal" after some finite number of transitions no matter in which possible state it is started. A series of transitions which leads to a condition in which the capacity of the communications medium is exceeded overflows the channel. Such reachability analyses also reveal incompleteness in the specification of the protocol being validated. A condition which makes available in the communications medium a message for one of the protocol entities without any specified action to receive that message reflects incomplete specification. More will be said about validation in later sections. Protocol validation systems rely on some formal specification technique to clearly and unambiguously model the protocol to be validated. Such formal specification techniques are the next topic for discussion.

Programming languages obviously have the ability to express the functions of a protocol. They were developed with the intention of being able to precisely represent an algorithm. Protocols are one type of algorithm. The body of research in the area of Software Engineering is applicable to the design of computer communications protocols using programming languages as the specification tool. However, little work has been done to specifically apply the techniques of Software Engineering to protocols. Because a protocol specified in a programming language is quite near to an implementation the techniques of software testing as well as program
proving are applicable. Both of these have their shortcomings. Testing is not able to exercise all the paths through a program and cannot guarantee the absence of errors. Program proving is a long and arduous task with the effort to prove the program far exceeding that required to develop it in the first place. Because of its resemblance to an implementation, a protocol specification in a programming language will almost certainly include unessential or implementation dependent features in addition to those required to perform the required functions. This is the most serious deficiency of programming language specification. The ability to readily handle variables, allowing them to take on widely varying values is one of the major advantages of this technique as compared with automata based techniques. This latter ability facilitates the representation of such things as sequence numbers and timers which are difficult to represent with state transition models. Bochmann [3], Hajek [26] and Stenning [61] have used a Pascal-like protocol specification, Krogdahl [34] has used an Algol-like language. A good overview of this approach can be found in [25].

State machine models (automata) of one sort or another are particularly effective at representing the control structure of protocols. The response to events such as user calls, time outs, and message receptions usually involves little processing and is easily modeled by the transitions of a state machine model. Such
models, with the events forming the inputs and outputs of the state machine, are natural choices for describing a protocol entity. However, to completely model complex protocols the number of states and events becomes unworkably large for the state machine model. To completely model a protocol using sequence numbers there must be a separate set of states and events for each possible sequence number [47].

A seemingly ever growing number of representations of state machines and variations on the theme are being employed for protocol specification. These include: state transition graphs [6, 75, 13, 22, 56], Petri nets [41, 42, 11, 46], and various non graphic representations such as state transition tables, regular grammars [27, 28] and context free grammars [64, 65] and regular expressions [4, 29]. Various modifications of Petri nets have been used as well. Logical predicates were add to resolve conflicts in [46] resulting in the Petri Net with Predicates (PNP). The Time Petri Net (TPN) [40, 41] allows representation of timing knowledge in a Petri net-like model by specifying a range of time during which a transition can fire after being enabled. The UCLA Graph [47] modifies the allowable topology of the Petri net-like graph, adds logical expressions on the input and output arcs which control initiation of a node and passage of control over output arcs. The SARA Graph Model of Behavior (GMB) adds data graphs and
interpretations to the UCLA control graphs and has been used as a protocol specification tool [54, 55]. Bochmann [7] incorporates finite state transition diagrams with a set of program variables in the general transition model. Enabling Predicates which depend on the values of the program variables govern the execution of transitions within the finite state transition diagrams. The Format and Protocol Language (FAFL) is an augmentation of the programming language PL/I to give it specialized capabilities to represent finite state machines [59].

2.3. Validation of the Peer-Peer Protocol

Unambiguous specification is important to provide confidence in a specified protocol, but it is not sufficient. The protocol must also be shown to be, in some sense, correct. The process of verification, or validation, is intended to show correctness of a protocol. The two terms, verification and validation are often used interchangeably. This work will follow the example of Sunshine [62] in using the term verification to mean testing and confirming that a protocol layer correctly provides the service required by its service specification. That is, n-layer peer entities implemented according to the specification interacting through the underlying communication system as abstracted by the (n - 1)-layer implement the service specification. Verification implies validation which
refers to the more limited testing and confirming that the protocol satisfies a number of general requirements that are essential to all, or nearly all, protocols. The list of general requirements is agreed upon by most authors [6, 43, 21, 62] with minor variations.

Most protocols should exhibit these characteristics:

1. **FREEDOM FROM DEADLOCK**: Each system state allows for progress to some other state.

2. **ABSENCE OF TEMPO-BLOCKING LOOPS**: All looping paths provide some productive communications operation.

3. **ABSENCE OF LIVELOCK**: Tempo-blocking loops, if any, provide some exit to paths on which productive communications operation is possible.

4. **FREEDOM FROM STARVATION**: Each process will eventually acquire the resources it needs.

5. **RECOVERY FROM FAILURE**: A protocol will return to normal operation within some finite number of steps (or after some finite time).

6. **STABILITY**: From an abnormal (or unsynchronized) state, a protocol will return to a normal state after a finite number of steps (or after some finite time).

2.3.1. Global Reachability Graphs

The most widely used validation technique, and the one emphasized in this work, is based on state transition modeling technique. It involves developing a model of global system states and exhaustively generating the graph, or finite state automaton, which represents all of the global states and global transitions reachable from a starting state. Merlin [43], who emphasizes that some specification and validation techniques are more suited to one
class of protocol while other classes require different techniques, points out that this technique (reachability analysis) is limited. The parties, or entities, modeled must be representable by finite automata. The number of parties to the protocol and their interconnection topology must be fixed or at least limited. In practice, the number of parties (entities) must be quite small and the entities must be relatively simple. As the entities become more complex and more states are required in their finite state automaton representation, the number of states in the global model increases exponentially. This phenomenon is known as the "state-space explosion." State space explosion is the main disadvantage of reachability analysis as a validation tool. Despite this serious limitation, the technique has been used quite effectively to validate real protocols. The chief advantage of the technique lies in the possibility of automating the generation of the global state reachability graph and automatic testing for some of the required properties. As a verification tool, however, it provides little help in determining whether the specified protocol service is provided. While it might be possible to verify that the service is provided by examining the reachability graph, this appears to be an arduous task and its automation has not yet been demonstrated.

As mentioned previously, state transition models are not practical for representation of protocols for which complex state
information must be modeled. States which must account for some particular sequence number having been sent or received, as an example, can theoretically be modeled with a distinct state to account for each possible sequence number value. If the number of sequence number values is large, such an approach is impractical. Remembering the global state space explosion problem, the reader will recognize that validation by construction of the global reachability graph is ruled out for even a moderately wide range of sequence number values. Nonetheless, it is possible to construct a partial model of such a protocol using state transition representations and use the model for validation of the protocol. This is demonstrated in chapter 4. By restricting the range of sequence number values used in the model, important characteristics of a protocol are modeled and validated.

A simple example will be used to illustrate the reachability analysis using several state transition models, specifically:
finite state graphs, Petri nets, and formal grammars.

2.3.2. Finite State Graph Model

Figure 2 on page 21 illustrates the relationship between peer entities in a protocol layer. It is possible that the functions of the protocol layer can be subdivided among several entities at the protocol endpoints. For example, a full duplex communications link might be represented by one sending and one receiving entity at each end. Such subdivision of responsibility makes each entity simpler
and is of considerable value in modeling and validation of the protocol since it reduces the state space. The protocol may be asymmetric as in the case of a sending entity communicating with a peer receiving entity at the opposite end, or it may be symmetrical with identical (or perhaps mirror image) entities at each end.

Figure 3: Illustrative Protocol Modeled as a Finite State Graph

A finite state graph to represent a protocol specification consists of a set of nodes connected by directed arcs. The nodes are usually drawn as circles. Nodes are labeled and represent the
states of the entity. Arcs indicate the possible state to state transitions which are defined for the entity. Labels on the arcs indicate protocol actions which are associated with the state transitions indicated by the arcs.

The finite state graph shown in Figure 3 represents the protocol entity which will serve as the example for this and the next two sections. No particular useful protocol is intended by the example and the reader should not attempt to associate any particular meaning with the messages exchanged by the protocol. Three possible messages can be exchanged. These are represented by the digits 1, 2, and 3. Six states of the machine are labeled S, A, B, C, D, and E. The arcs, or transitions, are labeled with the communication actions of sending a message (eg. S.1) and receiving a message (eg. R.2). The entity is initialized at the S (Start) state and can (nondeterministically) make a transition to either state A or state B. Associated with the former transition is the action (event) of sending message 1. Associated with the latter transition is the action of receiving message 2. Only one transition is defined from each of the states A and B. Those transitions take the entity to state C from which it returns to the start state via an intermediate state, either D or E.

The global system state depends on the states of the two communicating peer entities and on the state of the transmission
medium. It also depends on the characteristics of the transmission medium being modeled. Those characteristics include the number of message types and the number of undelivered messages the communications medium can store. A perfect transmission medium which neither loses nor reorders its messages will ordinarily have a smaller set of states than a medium which allows permutation of the order of its messages. The validation model may limit the number of undelivered messages for one of two reasons. First, the communications medium being modeled may be limited in the number of messages it can hold. If a protocol entity attempts to insert a new message into a communications medium which has already reached its storage capacity, this communications line overflow can be detected by the validation process. Second, it is often necessary to limit the state space to be explored by limiting the number of transmitted but unreceived messages in the validation model. Such a limitation prevents complete exploration of the global state space. Some error in the protocol which only appears with more messages in transit than the limit will go undiscovered with such a validation. The trade-off is a small enough global state space that the validation is, at least, practicable.

The communications medium can be modeled by a finite automaton [6] or message queues may be represented by regular expressions [5]. The use of message queues was introduced in [66, 72]. Because the
global state used to validate a protocol modeled as a state transition graph can be the same as that used for a formal language model, illustration of the global reachability graph for the example will be postponed to the section on formal language representation (see Figure 10 in section 2.3.4).

2.3.3. Petri Net Model

The Petri net is also a graphical model. Figure 4 shows the example protocol as a Petri net. The six entity states of the finite state graph are represented by the six places labeled S, A, B, D, and E in the Petri net. Initialization of the Petri net model is provided by the initial marking. For the example, place S is marked with one token; none of the remaining places are marked. In addition to the entity state places, places representing transmission medium states are included in the Petri net model. Places la, 2a, and 3a represent the availability of messages from the corresponding peer entity. Messages sent to the corresponding peer entity are represented by places labeled with the "b" suffix. Actions (or transitions) corresponding to each of the transitions of the finite state graph connect source places to destination places. A transition is enabled (can fire) when each of its input (or source) places is marked with at least one token. The firing of a transition removes one token from the marking of each of its input places and adds one token to the marking of each of the transition's output places.
Figure 4: Petri Net Model of Illustrative Example

The reader should be able to visualize with ease the way a pair of peer entity Petri net models can be joined at their joint communications places to form a global protocol model. The global state of the model is represented by its marking. A global finite state graph showing the possible transitions from one marking to another is commonly called a token machine. The token machine for the illustrative example is shown in Figure 5.

This token machine provides the graph of all states reachable from the starting state and can illustrate some of the capabilities
Figure 5: Token Machine for Illustrative Example
of the technique for validation of a protocol. Notice that,
beginning in state <SS> (both Petri nets with place S marked with
one token), the token machine can, after two transitions, move to
state <AlblaA>. From the new state, no transition arcs define
transitions to other state. Marking of 1a and 1b indicate
availability to both peer entities of message 1. From place A only
message 2 can be received. This illustrates an incomplete
specification of the protocol. The specification permits the
collision of both entities attempting to send 1, but it provides no
means for reception. The designer must examine the intent of the
protocol to determine whether the collision should be allowable, and
if so, provide appropriate transitions for reception of the event.
Should the collision be undesirable, the designer must then redesign
to avoid its occurrence or provide for reception of the messages and
design a sequence of transitions to reject or ignore the messages.

Examination of the token machine shows that both 3a and 3b
places can be marked with a token concurrently. That is, message 3
can be in transit to both ends at the same time. The concurrent
marking of 2a and 3a in one state and 2b and 3b in another show two
messages, 2 and 3, in transit to the same destination at the same
time. Should two copies of the same message have been in transit to
the same destination, a more complex model of the communications
medium would have been required. The simple model provided in the
illustrative example does not permit the communications medium to lose, damage, or reorder messages. A more complex model would be required to account for the characteristics of such a communications medium.

2.3.4. Formal Grammar Model

The heart of a formal grammar is a set of production rules which, using a vocabulary of non-terminal and terminal symbols, defines a derivation for all of the sentences in a language. Since only sentences which can be derived by the production rules are in the language, the formal grammar also defines the language. To use a formal language for specifying a peer to peer communications protocol, the objective is to define the language of all protocol action sequences which make up correct operation of the protocol. A well known theorem of formal language theory holds that the class of languages which can be specified by finite automata recognizers is exactly the class of languages which can be specified by a regular (or type 3) grammar [30]. This means that any protocol entity which can be specified by a finite state diagram can be specified by a regular grammar and vice versa. The grammar production rules used here will be those of a regular grammar.

A particular form of regular grammar for specification of protocols was called the Transmission Grammar (TG) by Teng [67]. Terminal symbols in the grammar represent protocol actions such as the sending or receiving of a message. The non-terminals are
equivalent to the states of a finite state diagram. Production rules (also called action rules) of the Transmission Grammar (TG) look like this:

\[
\text{<nonterminal> ::= terminal string <nonterminal> .}
\]

Non-terminals are readily recognizable in that they are set off by angle brackets (< >). The meaning of the production rule is that the entity in the state specified by the left hand non-terminal may take the action specified by the terminal string and enter the state specified by the right hand non-terminal. More than one production (equivalent to a transition in the finite state diagram) may be defined for a non-terminal.

\[
\text{<Left_non-terminal> ::= action1 <lst_non-terminal> , action2 <2nd_non-terminal> .}
\]

is shorthand for

\[
\text{<Left_non-terminal> ::= action1 <lst_non-terminal> .}
\]
\[
\text{<Left_non-terminal> ::= action2 <2nd_non-terminal> .}
\]

The relationship between a TG specification and the finite state diagram specification should be made clear by comparing Figure 6 which gives a TG specification for the illustrative example protocol with Figure 3.

The specification of Figure 6 is quite compact as compared with the finite state graph and Petri net specifications of Figures 3 and 4. It also has the advantage that it can easily serve as input to a computer program for automated analysis.
\[\begin{align*}
\langle S \rangle &::= \text{Send}.1 \langle A \rangle, \text{Receive}.1 \langle B \rangle. \\
\langle A \rangle &::= \text{Receive}.2 \langle C \rangle. \\
\langle C \rangle &::= \text{Send}.3 \langle D \rangle, \text{Receive}.3 \langle E \rangle. \\
\langle D \rangle &::= \text{Receive}.3 \langle S \rangle. \\
\langle E \rangle &::= \text{Send}.3 \langle S \rangle. \\
\langle B \rangle &::= \text{Send}.2 \langle C \rangle.
\end{align*}\]

Figure 6: Transmission Grammar Model of Illustrative Example

![Diagram of VA State Matrix Channel Queues]

Figure 7: VA State Matrix Channel Queues

The TG specifies only the peer entity. In order to validate the protocol using reachability analysis, a model of the global state including the transmission medium and both communicating
entities must be formed. Teng suggested a matrix of queues to model
the transmission medium [67]. The full-duplex channel between two
communicating entities, A and B would be represented as shown in
Figure 7. The two queues in the left hand column represent messages
(in the top queue) and acknowledgments (in the bottom queue) flowing
toward entity B on the right. Similar queues with acknowledgments
and messages destined for entity A are in the right hand column.
 Such a model might have some advantages for modeling a protocol in
which acknowledgments flow via some alternate channel or in which
acknowledgments are a distinct message type, separate from all
others. Few communications systems provide a separate
acknowledgment channel. More commonly, acknowledgments and other
control information is incorporated with other messages.
Piggybacked acknowledgment is a term frequently applied. For such
communications systems, the modeling of a separate acknowledgment
channel can distort the validation process. This distortion occurs
because the model makes possible the reception of an acknowledgment
as an isolated event even though the real system might not permit
such an isolated event. To see how this is so, consider the case of
entity A in a state which anticipates receipt of an acknowledgment.
One action is provided for an acknowledgment received with no other
information in the message; another action is specified for an
acknowledgment received along with some other information. If
Acknowledgments are modeled as separate entries in an acknowledgment queue. Entity A has no way to decide which action is appropriate if the acknowledgment queue contains the expected acknowledgment and the message queue contains the other expected information unless it is not possible for the other information to be received sans acknowledgment. A more useful model of global system state is one proposed by West [72]. Figure 8 shows a two by two matrix for entity A communicating with entity B via a full-duplex channel. At Matrix position (A,A) is the state information for entity A. In the opposite corner, position (B,B), is the state information for entity B. The (A,B) position of the matrix has the queue of messages from entity A destined for entity B while the opposite direction of message flow is modeled by the channel in position (B,A).

\[
\begin{array}{c|c|}
\text{column} & A & B \\
\hline
\text{row} & A \_\text{State} & \text{Msg}_B \\
A & & \\
B & \text{Msg}_A & B \_\text{State} \\
\end{array}
\]

Figure 8: General Global State Matrix for Protocol Validation

For validation purposes, it makes sense to use operations on the message queues as terminal symbols in the Transmission Grammar (TG). Rather than send a message we should call for it to be enqueued. Rather than receive a message it should be dequeued. Teng proposed a number of validation actions on the message queues,
some of which will be used in this work. Those actions are listed along with one letter abbreviations as follows:

- **Queue (Q):** Insert the specified message into the specified queue in a first-in-first-out (FIFO) manner (i.e., put the message at the tail of the queue).

- **Fetch (F):** Delete one instance of the specified message from any position in the queue. The action is possible only if at least one instance of the specified message is contained in the queue.

- **Dequeue (D):** Delete the specified message from the front of the specified queue. The action is possible only if the specified message is at the front of the specified queue.

- **Pop (O):** Delete the specified message from the end of the specified queue. The action is possible only if the specified message is at the end of the specified queue.

- **Clear (C):** Delete all of the messages from the specified queue.

- **Empty (E):** Test whether the specified queue is empty. The actions in an alternate production rule are possible only if the queue is empty if this action is part of the alternate production. The empty action should be the first action symbol in an alternate production rule of which it is a part.

- **Non-empty (N):** Tests whether the specified queue is non-empty. The actions in an alternate production rule are possible only if the queue contains at least one message if this action is part of the alternate production. As with the empty action symbol, non-empty should be the first action symbol in an alternate production rule of which it is a part.

We will use the term VA to refer to the grammar representation of a peer entity using these validation actions in place of communications actions such as send and receive. The reachability graph produced from peer entities and communications channels modeled in such a way is called a **global** Validation Automaton (VA).
The VA corresponding to Figure 6 is shown below.

\[
\begin{align*}
&S ::= Q.B.1 \langle A \rangle, D.A.1 \langle B \rangle. \\
&A ::= D.A.2 \langle C \rangle. \\
&C ::= Q.B.3 \langle D \rangle, D.A.3 \langle E \rangle. \\
&D ::= D.A.3 \langle S \rangle. \\
&E ::= Q.B.3 \langle S \rangle. \\
&B ::= Q.B.2 \langle C \rangle.
\end{align*}
\]

Figure 9: VA Model of Illustrative Example

Figure 9 depicts the Validation Automaton for entity A. The structure of terminal symbols is an action symbol, followed by a queue identification followed by identification of the message specified. The three elements are separated by periods. The Fetch operation can be used in the Validation Automaton model of a protocol based on a communications medium which reorders messages. The use of the fetch operation also partially models the effects of message loss and retransmission without explicit deletion and re-enqueuing actions being specified.

The Validation Automaton for entity B is a mirror image of that for A. Mirror image VAs are identical except for the source and destination of messages. Where entity A enqueues its messages in queue B and dequeues messages from queue A, entity B enqueues in queue A and dequeues from B.

Construction of the global VA proceeds from the beginning global state with both entities in their initial states and both queues empty:
New states are added as descendants of a state in the global VA by applying VA action rules of one of the entities at a time. For entity A in state S the VA allows Q.B.1 <A> or D.A.1 <B>. Since the second of the two alternatives is possible only with message 1 in queue A, only one state can be added to the global VA by applying rules from VA A:

The mirror image action rule from entity B adds a symmetrical state to the global VA producing the following structure:

The initial state has been fully explored and no other global state is immediately reachable from the initial state by action of just one of the entities. States modeling the results of simultaneous
actions by two entities are reached in the global VA by two successive actions by first one, then the other entity. Each of the mirror image descendants of the initial state can produce two new states as shown below.

```
A   1
 e  S
|
/|
A e
| e B
|   |
| l A
```

The states produced by the other global VA state are symmetrical to these two. That is, the state with both queues empty has entity A in state B with entity B in state A. The global state shown as the right hand side descendant is symmetrical to itself. It is a descendant of both global states in the global VA level above. The full global VA is shown in Figure 10. The global states are numbered from 1 through 18 for reference. The state numbered 5 has no descendants. It is indicative of the incompleteness of the specification discussed in subsection 2.3.3 on the Petri net.
Figure 10: Global VA for Illustrative Example
3. SPECIFICATION AND VALIDATION OF THE X.21 INTERFACE

In order to further illustrate the use of Transmission Grammar (TG) specification for modeling and validating communications protocols, and to demonstrate the usefulness of the VA for validation of actual protocols, we will describe the validation of an interface specification which we undertook. That effort provided considerable insight into the use of formal grammar as a specification and validation technique. We share that insight with the reader and compare the global validation technique we used with two other techniques in the remainder of this chapter.

Recent accounts have been published of the use of two other state transition techniques to validate the CCITT X.21 recommendation [9].

3.1. IBM Zurich Validation

The first reported work on X.21 validation was that of the IBM Zurich Laboratory [71]. The validation technique used by West and Zafiropulo is based on a communications system modeled as
processes. Each process is represented in the model by a state transition diagram, a directed graph whose nodes represent process states and whose arcs represent transitions between the states. Those arcs which represent transitions associated with the exchange of information between processes are labeled with the name of the unit of information, called an event, and the name of the process to or from which the information exchange takes place. The communication medium is modeled as a set of simplex channels interconnecting pairs of processes. Channels are FIFO queues with capacity limited to some maximum number of events. The total system state is described by the state of each of the processes and the states of all the channels. (The state of a channel is an ordered list of the events it contains). The validation procedure has been programmed in APL and generates the complete set of possible system states by a series of perturbations from the initial system state. Each perturbation changes the state of just one process and the channel involved in any communication associated with the transition allowing the perturbation.
3.2. UCLA Validation

Inspired by the work of West and Zafiropulo, a group of investigators at UCLA undertook the modeling and verification of the X.21 communication protocol using a graph model and UCLA’s SARA design aid system to model the X.21 protocol [54, 55]. (SARA stands for Systems Architect’s Apprentice.) The UCLA control flow model used in SARA is a Petri net like model which models events (transitions) by nodes and precedence relations among events as directed arcs. The control graph is related to a state diagram in the following way: States of the state diagram are represented as arcs in the control graph. Transitions of the state diagram are represented as nodes in the control graph. Initiation of a node is determined by the activation of some or all of its input arcs depending on an input logical expression. A node’s input logical expression is an arbitrary function of its input arcs using "AND" and "OR" operators. Which output arcs are activated by a node’s initiation depends on a logical expression relating the output arcs.

As in Petri nets, the state of the control graph is represented by the distribution of tokens. Unlike the Petri net, control graphs have no places to hold the tokens which are placed directly on the control arcs. Tokens are placed to establish some initial state. They are absorbed from arcs which enable the initiation of a node and placed on output arcs as described by the output logic. Both
initiation of nodes and placing of output tokens may be nondeterministic. Arcs need not simply connect pairs of nodes, but can originate at more than one node and terminate at multiple nodes. This makes for a somewhat different configuration from the Petri net which was discussed in section 2.3.3. Petri net arcs can originate at only one node and can terminate at only one node. The use of input and output logical expressions is also an addition to the original Petri net idea.

From the control graph, a computation flow graph can be generated which represents all states (and sequences of states) reachable from the initial control state. The flow graph is the UCLA Control Graph equivalent of the Petri net token machine discussed in section 2.3.3. Control graphs are augmented with data graphs and interpretations to form a complete SARA Graph Model of Behavior (GMB). Data graphs and interpretations provide for definition of data sets and data transformations linked to initiation of control nodes and effecting decisions in the flow of control. Data and interpretation domains make available abstractions which significantly reduce the number of control states while providing enough information to permit partially automated validation. SARA includes an automated Control Flow Analyzer (CFA) which accepts descriptions of the control graph, and interpretation as input. The CFA transforms the input control graph into an
equivalent set of Transformation Expressions (TE). The TEs are similar to productions of a phrase structure grammar. The CFA first reduces the set of TEs. From the reduced set of TEs it builds a computation flow graph.

3.3. Transmission Grammar (TG) Validation

The TG model forms a representation of state diagram models as used by the IBM Zurich group. Where Zafiropulo assigns an integer to each message interchanged between processes, TG defines message formats via grammar production rules called message format rules. These rules consist of a message symbol and the message string derived from that symbol. Through a process of refinement the message format can be defined at any level of detail desired down to the bit level.

The TG system is similar to the Zurich system in the representation of the transmission medium as a matrix of channels. Using the TG automated validation system, one constructs an integrated TG to reflect the interaction of the communicating entities. Teng describes this integrated TG as the result of shuffling the individual grammars to obtain a complete mixing of the grammar symbols and then restricting the results to eliminate all non-realizable transitions, those which involve receiving a message before it has been sent.
Teng describes this restricted shuffle making clear the end result, but he did not provide a procedure for obtaining it. The result is equivalent to that obtained by applying the technique referred to by West [72] as state perturbation.
ANNEX 1
(to Recommendation X.21)

Interface signalling state diagrams

Figure 11: Call Establishment Phase for Circuit Switched Service
Figure 12: Clearing Phase and Quiescent States for Circuit Switched Service

Note 1. — The condition of the R circuit is for further study.
Note 2. — The DTE may be able to enter the controlled not ready state from states other than ready, this is for further study.
In the CCITT X.21 recommendation the DTE/DCE (Data Terminal Equipment/Data Circuit-Terminating Equipment) interface [9] is specified by a textual description, a state diagram, and timing diagrams and time limit tables. The state diagrams of the specification can be seen in Figures 11 and 12.

Four conductors, or interchange circuits provide the X.21 interface. Circuit T, Transmit, carries a signal established by the DTE to the DCE. R, Receive, carries signals from the DCE to the DTE. C, Control, and I, Indication, carry voltage levels respectively to and from the DCE.

West and Zafiropulo developed individual state diagrams for the DTE and DCE to define the separate communicating processes necessary for their protocol validation procedure. Estrin, et al. adopted these diagrams for the representation of the DTE and DCE processes for their work at UCLA. West and Zafiropulo pointed out that different pairs of processes could be derived from the interface diagram which would represent the interface. They chose the safest (in their opinion) approach for developing specifications for the individual entities which were served by the interface. They included each state and each transition of the CCITT X.21 interface finite state diagram in the finite state diagram for both of the entities. In examining the diagrams of [9] it seems apparent that a transition from one state to another in the interface need not
require a change of state by both the DTE and DCE. For example, the call request state need not involve a change of state by the DCE from Ready. That state involves a change of the DTE from ready to DTE state 2, call request, and a change of state on the DTE to DCE circuits from 1,OFF to 0,ON. The change in signals should result in a transition of the DCE to DCE state 3 with a transition on the DCE signalling circuits from 1,OFF to +,OFF. In eliminating such intermediate states, one must be cautious that necessary intermediate transitions are not left out. Transitions from states 3 to 4 and 4 to 5 are both labeled as DTE driven and involve changes in the DTE signal circuits from 0,ON to IA5,ON to 1,ON. If an intermediate state is not provided in the DCE, then that process must accept a signal change directly from 0,ON to 1,ON which may not be desirable.

3.4. DTE/DCE State Diagrams

Figures 13 and 14 show state diagrams developed using the state saving ideas of the previous paragraph. This reduces computation time in the automated validation. Figure 13 shows the derived state diagrams for the DTE; the DCE is shown in Figure 14. Bracketed signals within the state nodes indicate the [T,C] circuit signals for the DTE diagram and the [R,I] signals for the DCE diagram. These are the signals sent by the entity in whose diagram they
appear. The received signals which drive those particular transitions appear adjacent to transition arcs.

In Figure 14 the detected change in signals from the DTE, [1, OFF] to [0, ON], drives the DCE.1 to DCE.3 transition. The incoming call signal, [BEL, OFF], can be accepted by the DTE without a change in state. Neither diagram includes a state 15; the joint state DTE.2_DCE.8 represents the call collision state, state 15 in the X.21 diagram. We have chosen to specify CALL SELECTION, CALL PROGRESS, CALLED LINE IDENTIFICATION and CALLING LINE IDENTIFICATION by specific identifiers (SEL for the selection signal, for example). The X.21 interface signalling state diagrams merely indicated that ASCII (or International Alphabet IA5) characters were permitted. Using a specific signal avoids ambiguity between CALL PROGRESS and CALLED LINE IDENTIFICATION.

3.4.1. Construction of the Joint State Diagram

Up to this point, the TC/VA validation procedure has paralleled that of the other two groups except for the choice of state diagrams. At this point the UCLA group constructed a different representation of the DTE and DCE entities by converting the state diagrams to UCLA control graphs supplemented by a data graph to represent the IA5 signal. The IBM group assigned integer values to each signal (event) and labeled transitions in the state diagram with positive or negative integers, indicating that a given event is received or transmitted, as described in [74]. Each group was then
Figure 13: DTE State Diagram
Figure 14: DCE State Diagram
able to feed a representation of its model to an automated validation tool to obtain extensive analysis with specific error indications. Our automated validation program, as it existed at the time of our effort to validate the X.21 protocol, accepted a (TG) representation of a single validation automaton (VA). It was necessary to construct the joint VA manually. In [72] West clearly described the perturbation technique achieving the restricted shuffle Teng called for. By applying that technique a global VA representation of the joint DTE/DCE was obtained in Transmission Grammar (TG) form.

The joint states of the VA were represented by a pair of numbers each representing the state of one of the entities. The initial nonterminal symbol of the grammar was \(<1_1^1>\). This places both the DTE and DCE in the READY state. The state of the DTE is the first number of the pair.

The TG is defined by a series of production rules as described and illustrated in the previous sections. The right hand side of each production rule is a series of one or more alternative productions. Each alternative begins with a terminal symbol. That symbol represents the signals on the interface during the joint state from which the production derives. For example, the state \(<1_1^1>\) produces the terminal symbol 1.OFF_1.OFF. The terminal is followed by one of the nonterminals, a joint state produced by a
transition from the left hand side of the production rule. From state \(<1\_1>\) the terminal symbol 1.OFF_1.OFF precedes each of the nonterminal alternative productions which represent the joint states possible with transitions from \(<1\_1>\). For example, from state one the DTE can make a transition to state two so that the first alternative production is

\(<1\_1> ::= 1.0FF_1.OFF <2\_1>\)

The rules for the perturbation technique allow only one state of the state pair to change at a time. Each of the alternative productions from a given state can be listed simply by listing those produced through change of state in the DTE, then listing those in the DCE. Because the communication medium is modeled as a perfect medium with unit capacity on each channel, no separate transitions are required to model changes in the channel state. The DTE has transitions to states two and eight in the call establishment phase.

We saw the example of the production involving the change to state two. The transition to state eight is DCE driven and requires detection of the BEL.OFF signal. It is not possible from state \(<1\_1>\) because the 1.OFF signal is being sent on the interface by the DCE. Transitions by the DTE to states 21 and 14 in the clearing and quiescent phase are allowed from \(<1\_1>\). West and Zafiropulo are correct that [9] was not intended to imply transitions from the ready state to states 16 or 19. Such transitions are ambiguous with
transitions to states 21 and 18 respectively. The DCE transitions from \(<1_1>\) lead to states 3 and 18. The perturbation technique requires that these new states be added to the set of states. Having expanded the state set from the first state, the technique is applied to yet another of the states in the set of states. A display terminal and a good screen editor are convenient tools which permit creation of the input file to the validation system program as the Transmission Grammar (TG) is developed. As each production rule is entered, the new states (non-terminals) included on the right hand side of the production rule can be listed on separate lines indented slightly from the non-terminal on the left hand side of the production rule. This creates a list of non-terminals which must be defined. One can simply work ones way down the list of undefined non-terminals adding the production rules to define them. For example, after completing the expansion of state \(<1_1>\) the display screen would contain these entries.

\[
<1_1> ::= \text{1.OFF,1.OFF} <2_1> , \\
\text{1.OFF,1.OFF} <14_1> , \\
\text{1.OFF,1.OFF} <21_1> , \\
\text{1.OFF,1.OFF} <1_8> , \\
\text{1.OFF,1.OFF} <1_18> .
\]

\[
<2_1> ::= \\
<14_1> ::= \\
<21_1> ::= \\
<1_8> ::= \\
<1_18> ::= \\
\]

The expansion of \(<2_1>\) would insert \(<2_3>\), \(<2_8>\), \(<2_18>\) and \(<16_1>\)
between the \(<2_1>\) production rule and the \(<14_1>\) ::= entry.

Some states of the global VA are reachable from more than one predecessor. For large grammars, such as this one, one may find it helpful to do some extra bookkeeping to check on whether a state is already included in the derived state set or should be listed for future expansion. As soon as the list of production rules is too long to display on one screen, a separate list of the state set becomes helpful.

While expanding states, a check is necessary to determine whether each of the individual states in a state pair can accept the signal being produced by its opposite entity state. Look at this example. One state produced from \(<4_3>\) is \(<5_3>\). In state five the DTE signals \([1, ON]\) while expecting to receive \([SYN, OFF]\) from the DCE. In state three the DCE is signalling \([+, OFF]\) while its only possible transition, to DCE.4, depends on receiving \([SEL, ON]\) from the DTE. State \(<5_3>\) is obviously deadlocked. An implementation could get into state \(<5_3>\) only if the DTE were to make the transition from state four to state five before the DCE recognized the \([SEL, ON]\) signal from state DTE.4. The inclusion of state \(<5_3>\) in the joint VA is an artifact of the modeling process and could occur only from failure of one or more elements of the interface. Since X.21 is intended for synchronous operation, the design of the DCE would require that it synchronize on the two or more SYN
characters which initialize the selection sequence sent by the DTE on the Transmit conductor. This illustrates an important aspect of using the Transmission Grammar to model a protocol previously specified using some other technique. One must not blindly translate a state diagram, for example, into a TG without understanding how the protocol really is intended to work. Subtleties such as this may be evident only from other portions of the original specification.

The erroneous state was included in the grammar. The production rule was defined for the erroneous state as producing the terminal symbol "error" preceded by the interface signals from the erroneous state. Since no non-terminal symbol is produced, no further progress beyond the erroneous state is modeled. For state <5_3> the production rule would be:

$$<5_3> ::= \text{ON+OFF error}$$

All error states have similar productions. The set of production rules can be found in appendix A.

3.4.2. Analysis of the DTE/DCE Interface Model

Discovery of errors in the interface using Transmission Grammar (TG) requires a sharp eyed careful human. All the errors were found during the manual process of constructing the joint VA. A total of 22 erroneous states were derived, 16 of them are anomalies of the type discussed regarding state <5_3>. At each pair of successive
transitions in the X.21 state diagram (Figures 13 and 14) having the same entity (DTE or DCE) responsible, the modeling technique allows that entity to advance out of synchronization with its opposite. Had these been modeled to produce non-terminals rather than terminal "error" states, this out of synchronization activity could have propagated further in the model to produce more error states. From state five to state twelve in the call establishment phase, the DCE is responsible for seven successive transitions. State \(<5_7>\) could have had successor states in the VA of \(<5_6B>, <5_10>, <5_6C>, \) and so on. In view of the fact that these error states did not represent any implementation intended by the X.21 recommendation, there seemed no point in propagating the error further and burdening the validation system program with extraneous states and transitions. To have done so would not have produced further enlightenment. Not all of these anomalies were confined to the call establishment phase. From states 21 and 14 the DTE can advance through state 1 to state 2 and the DCE has the possible sequence 18 \(- 1 - 8.\) The transitions by the DTE from 14 \(- 1 - 21\) as well as 21 \(- 1 - 14\) produce similar errors. The interested reader can examine the production rules of the TG in appendix A to see the sequences of productions leading to the errors.

Another kind of error has been referred to as collision. When the DTE attempts to assert CALL REQUEST while the DCE asserts
INCOMING CALL a call collision results. This particular collision was provided for in the interface design with the call request getting priority over the incoming call. Other possible collisions were not provided for by the designers. CALL REQUEST can collide with the DCE transition to state 18. CALL INDICATION can collide with either of the DTE transitions from ready to states 21 or 14. Finally, the DCE transition to state 18 can collide with the DTE transitions from state 1 to states 21 or 14.

State <2_18> is an error state since the DCE should expect to see 1,OFF from the DTE. It is not a deadlocked state. The DCE can return to ready. The DTE should interpret 0,OFF from the DCE as a clear indication signal and make the transition to DTE.20, clear confirmation. The new state <20_18> is also in error since the DCE does not expect to receive 0,OFF in state 18. The DCE signal is satisfactory to the DTE (having caused the transition to DTE.20) and, should the DCE go to ready, the DTE will interpret this as a transition to DCE.21. Though all the states <2_18>, <20_18>, <20_1> are error states, if a DCE implementation ignores illegal signals from the DTE a progression of transitions leads the interface to <1_1>. Though this might be a reasonable solution to that particular collision, it comes about accidentally rather than through design. It is not envisioned in the X.21 recommendation. If this condition is not recognized by the implementor the reaction
to this particular collision could be unpredictable.

The <21_8> collision state will trigger a clearing sequence in the process which had entered the call establishment phase. The DTE will be receiving illegal signals until the DCE reaches state 21. On the other hand, the <14_8> collision places a unique alternating 0/1 bit signal on T for which there is no provision in DCE.8. The signal to be expected on R is not specified in the X.21 recommendation. That signal is left for further study as is the possibility that the Controlled not ready state may be entered from other than the ready state. The <21_18> and <14_18> collisions give rise to conflicting signals. In both cases only transitions which return to the ready state are available. One possible transition from the <14_8> collision state is to DCE Clear indication. Since state 14 is not completely defined it is not clear whether this state of the interface is legal. The DCE, however, may not leave state 19 until the DTE CLEAR CONFIRMATION, 0,OFF, is detected. The DTE, on the other hand, should go to DTE.18 when it is put back on line. On returning to ready the DCE CLEAR INDICATION signal will be interpreted as DCE NOT READY. The strange result, <18_19> is a deadlocked state. Table 1. summarizes the error states found during construction of the global VA.
Table 1: Error States of the Validation Automaton.

<table>
<thead>
<tr>
<th>modeling anomalies</th>
<th>collisions</th>
<th>followed collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;5_3&gt;</td>
<td>&lt;14_8&gt;</td>
<td>&lt;14_19&gt;</td>
</tr>
<tr>
<td>&lt;14_21&gt;</td>
<td>&lt;21_18&gt;</td>
<td>&lt;20_18&gt;</td>
</tr>
<tr>
<td>&lt;2_14&gt;</td>
<td>&lt;14_18&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;2_21&gt;</td>
<td>&lt;21_8&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;11_13&gt;</td>
<td>&lt;2_18&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;6C_12&gt;</td>
<td>&lt;11_13&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;10_11&gt;</td>
<td>&lt;6B_6C&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;7_10&gt;</td>
<td>&lt;6A_6B&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;5_7&gt;</td>
<td>&lt;10big_11&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;18_8&gt;</td>
<td>&lt;2_21&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;9_6C&gt;</td>
<td>&lt;21_14&gt;</td>
<td></td>
</tr>
</tbody>
</table>

3.5. Automated Validation of the Validation Automaton.

Teng's validation system program [67] accepted as input a Transmission Grammar (TG) definition of a Validation Automaton (VA). It analyzed the VA and reported on proper termination and reachability. It also listed all possible cycles. If the input included test sequences of action symbols, the program reported on the correctness of the sequences.

Although the construction of the joint VA from the state diagrams of the two entities, DCE and DTE, via the perturbation technique was quite straightforward, it was also quite prone to human error. The validation system program reported numerous instances of duplicate definitions, undefined symbols, and
unreachable symbols through several iterations before a VA was achieved which was totally acceptable to the program. Even though the global VA construction was not automated, the program was invaluable in developing a well-structured TG description of the VA.

Once the bugs had been worked out the validation program provided assurance that the VA terminated properly, that there were no unreachable nonterminals and that all non-terminals could terminate properly. Examining some one hundred pages of indented cycles listing showed that all cycles of the VA began and ended in state \texttt{<1_1>} except for three minor cycles which involve transitions into and out of not ready states. This information about the cycles of the VA provided assurance that, except for the errors, the interface does progress toward productive transitions and can perform the functions for which it was designed.

The validation system program was vitally helpful in discovering and correcting mistakes in the construction of the global VA. During construction of the global VA a number of errors were discovered resulting from the incomplete specification of the interface by the X.21 recommendation. Finally, the automated analysis assured us that, aside from the errors discovered during construction of the VA, the basic design was sound. The output of the automated analysis program is far too long to include in this dissertation.
3.6. Comparisons.

The three modeling techniques described in this section are in the class generically classified as state transition models. The UCLA group's model is strikingly different in its use of the Petri net like control graph as its primary tool for modeling the control domain and its use of data graphs and interpretations to extend the capability of the Graph Model of Behavior (GMB).

In many respects the IBM and OSU models resemble each other. Both rely on finite state transition diagrams or formal grammar representations of finite state machines. The differences lie primarily in the way communications between the entities and the communications medium is modeled and in the automated analysis tools provided. Since the UCLA work was the first response to the Zurich group's published report and since the claim is made for "greater clarity than previous models" [54], a discussion of the comparative strengths and weaknesses of the UCLA and Zurich validation systems is in order. Then an examination of the TG/VA system as compared with the other two will be presented.
3.6.1. Graph Model of Behavior versus IBM Technique

The obvious difference of the Graph Model of Behavior (GMB) is the different graphical model. In the case of the validation of the X.21 recommendation an extra step was needed to translate from state transition diagrams to control graph. This would be unnecessary for a protocol initially specified using GMB. It is not clear which model is more natural or easier to use for initial design and specification of a protocol that may depend on the familiarity of the designer with one technique or the other. It should be noted that the UCLA group did not undertake to construct their control graph directly from the CCITT recommendation. Instead, they chose to translate the state diagrams of West and Zafiropulo. To be fair, we should point out that the original recommendation was in the form of a state diagram and specification of the individual DTE and DCE processes was undoubtedly easiest using a similar state diagram model.

Given a protocol specified using state diagrams, is there some advantage to be gained in translating it to GMB for validation? In the case of the X.21 recommendation the claim of greater strength and clarity of the GMB appears to be based on the introduction of an explicit synchronization model [55]. The function of this synchronization model is to prevent any second change on the circuits prior to the first change being received. The long list of
anomalous states discussed earlier resulted from allowing such changes in construction of the global VA. West and Zafiropulo detected and eliminated such behavior through limiting the allowed channel capacity to one. This type of error was allowed, however, in the case of DTE and DCE each simultaneously cycling from ready to not ready states and back again without detecting the other's not ready signal. Noting that this type of error could not occur in any real implementation West and Zafiropulo simply ignored it.

Razouk and Estrin do not illustrate their report with a graphical depiction of the control graph. The small portions of it which they do show (one or two nodes) have enough complexity to make one wonder whether the complete graphical representation would not be too complex to be easily understood. An attempt by the author to construct a picture of the control graph quickly resulted in frustration. The argument regarding the superiority of UCLA control graphs for modeling synchronous interfaces remains unconvincing. In the particular case of X.21 there seems to be little, if any, advantage. Both the UCLA and IBM models are less ambiguous than that provided by [9]. The reduction in ambiguity is a direct result of providing individual specifications for the DTE and DCE.

The GMB does provide the ability to represent complex data structures and decision making algorithms based on the interpretation of that data. It does so, however, at the expense of
fully automated verification. For X.21 this was not a particularly
important feature of GMB. It was used to model one group of
signals, the IA5 characters. Disambiguation of the resulting
computation states was not automated. The analysis given by [55] is
presented in a more formal style, but is no more persuasive than
that given by [71]. The simple expedient adopted in the TG/VA
validation of naming these signalling sequences uniquely seems
clear-cut and less bothersome.

3.6.2. How Does Transmission Grammar VA Modeling Stack Up?

As noted, the TG/VA model has little to distinguish it from the
IBM model. The major difference lies in the automated support
provided. The ability to automate analysis from just the
independent models of the communicating entities is vitally
important. The validation system program very effectively
discovered unreachable states, undefined states, and nonterminating
VAs but had no way of knowing that some possible joint states or
transitions were missing unless they led to one of the
aforementioned errors. An automated program to overcome this
shortcoming of the TG/VA validation system is described in chapter 6
and listed in appendix B.

The example of the X.21 recommendation shows us the inadequacy
of a single state diagram as a protocol specification because it may
not completely and unambiguously define the individual entities that
make up the protocol. No independent implementation of Data
Terminal Equipment (DTE) and Data Circuit Terminating Equipment (DCE) could achieve the specified interface without also introducing other interface states that were not specified by the single interface state diagram specification. That does not mean a single specification of the interface could not be complete. The global Validation Automaton is, in effect, a single specification that combines the joint state of each protocol entity.

The TG/VA system provided a number of capabilities the other systems lack. Its listing of all the cycles a tested VA can produce gives an opportunity for close scrutiny. That scrutiny is tedious, however. In one early version of the VA over 11,000 lines of output were produced at considerable cost in computation time. It is a feature that could be improved by making it optional as has been done in the Multi-Valid system described in chapter 6.

Another feature included in Teng's validation system program was the option of testing possible execution sequences. This gave the tester an opportunity to actually test the semantics of a protocol to determine if it did, in fact, implement a number of expected action sequences. If one or more action sequences which the designer expected to be possible are labeled as incorrect by the validation system, the designer can conclude that the protocol does not do what he intended. One is limited, however in the number of possible action sequences one might imagine and submit for
3.7. Conclusion

The findings of the TG/VA validation with regard to the recommendation X.21 agreed with the findings of the other research groups. The X.21 interface specification leaves undefined a number of possible interactions between any DTE/DCE pair which might be implemented from it.

Comparison of the three modeling and validation systems shows a need for additional automated support for the TG/VA system. It was not possible to judge from this validation effort whether the slight difference in communications channel modeling between Teng's channel matrix and that of West is of significance. Teng included separate channels for acknowledgments but acknowledgments are not a part of the X.21 interface. See the discussion of this point in the following chapter.

The author's personal opinion is that control flow graphs offer no advantage over state diagram or grammar representation of protocols. They add an extra step in most instances and during analysis are transformed to computation-flow graphs which are, in essence, state diagram models. The ability to represent some variables as data graphs and provide programming language algorithms to analyze the data helps control the state explosion problem.
inherent in transition models. It was not of real importance to the X.21 validation. For modeling of other protocols in which variables play a larger part the expressive ability of the data graph would be valuable. That additional value is reduced by the limitation on automation of the validation.

The use of representative values for variables can give the TG/VA system an ability to model the essential elements of a protocol while retaining the possibility of automated validation. This is an important technique employed first in the effort to validate the X.21 interface specification. It is employed in a more complex instance the validation of the Transmission Control Protocol (TCP) discussed in the next chapter. The test of the TG/VA system on a real protocol specification showed it to be a useful system which could be of great value to a protocol designer.
4. VALIDATION OF THE TRANSMISSION CONTROL PROTOCOL

This chapter discusses the application of Transmission Grammar (TG) to the specification and validation of a much more complex protocol than X.21. The Transmission Control Protocol (TCP) presented a number of challenging problems which had not been faced in a formal grammar protocol specification. The TG as a specification and validation tool has been described in detail. The next several sections describe the Transmission Control Protocol (TCP), discuss the development of a Transmission Grammar (TG) model for TCP, and describe the validation of TCP. The final two sections first outline the results of the validation effort and suggest some additions and changes to the DOD Standard Transmission Control Protocol specification document, and then summarize the finding and evaluate the validation procedures.
4.1. The Transmission Control Protocol (TCP)

The subject for this protocol validation is the DOD Standard Transmission Control Protocol [48]. The Deputy Undersecretary of Defense for Research and Engineering has declared the Transmission Control Protocol (TCP) to be a basis for DOD-wide inter-process communication protocol standardization. The standard is based on the ARPA TCP specification. TCP is intended to be a reliable host-to-host protocol in packet-switched computer communication networks. It is a connection oriented protocol meaning that a connection establishment procedure is used to establish certain connection parameters which are used to control the communication between two connected processes during the duration of the connection. It is intended to be part of a layered network architecture. With regard to the International Standards Organization Reference Model for Open Systems Interconnection (ISO-RM/OSI) [31] TCP is part of the transport layer. TCP depends on a lower level network layer for the actual transmission of TCP message units called segments. The service provided by that underlying layer may be a potentially unreliable datagram service. This means that segments passed to the datagram layer may not be delivered in the same sequence as they were sent and may not be delivered at all. This is not to say that the datagram service is malicious, but just that it contains no safeguards to prevent these results. The "DOD Standard Transmission
Control Protocol" [48] (we will call it the TCP Standard) cites the Internet Protocol [49] as a suitable lower adjacent protocol layer. A wide variety of underlying service layers could be suitable including even hardwired or circuit switched layers as opposed to packet switched networks.

The TCP is intended to provide a reliable service. It includes the appropriate acknowledgment and retransmission and error detection capabilities to achieve reliable communications. It also assumes responsibility for flow control between connected user processes and reordering of segments into the proper sequence prior to delivery to the user.

The specification also assumes underlying support for precedence, security classification and compartmentation of TCP segments such as that provided by Internet. These security and precedence issues are largely ignored in this model of the protocol. TCP is concerned with these items only insofar as insuring that accepted packets match the type of service for which the connection was established. An unmatched security classification affects the protocol behavior just as some other illegal packets might. Such illegal packets are lumped together in the model.

The format of a TCP segment passed to or received from a lower protocol layer is shown in Figure 15. The 16 bit source and destination port numbers serve to identify a particular TCP
Note that one tick mark represents one bit position.

**Figure 15: TCP Header Format**
The 32 bit segment sequence number (SEG_SEQ) is the number of the first data octet in this segment unless the SYN bit is on. The SYN and FIN are included in the sequence numbering as is each octet of data. If the ACK control bit is on, the 32 bit ACK field (SEG_ACK) contains the value of the next receive octet expected by the sending TCP. The 4 bit data offset field contains a count of the number of 32 bit words in the header. The 8 control bits are:

**URG:**
The Urgent Pointer field is significant

**ACK:**
The Acknowledgment field is significant

**EOL:**
End of Letter

**RST:**
Reset the connection

**SYN:**
Synchronize sequence numbers

**FIN:**
No more data from sender

The 16 bit window tells the destination TCP how many data octets the sending TCP is prepared to accept. The 16 bit check sum is computed over the header and data. It serves to provide a check on corruption of the segment during transmission. The 16 bit urgent
pointer is an offset count from the current segment sequence number. It points to the octet following urgent data. The variable length options field deals with details not important to this TG model for validation. The curious reader is referred to [48].

The operation of TCP is specified in terms of a series of states through which a connection progresses during its lifetime. The states are: LISTEN, SYN_SENT, SYN_RECEIVED, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, TIME_WAIT, CLOSE_WAIT, CLOSING, and CLOSED.

When a TCP reaches the CLOSED state, the connection ceases to exist. A TCP having reached the CLOSED state will not respond favorably to segments received from a distant TCP until it receives an OPEN command from its user. The state diagram of Figure 16 illustrates some of the state changes which occur together with the causing events and resulting action.

Each state change, or transition, is labeled with the cause above a horizontal line and the effect below the line. For example:

\[
\begin{array}{ll}
\text{passive OPEN} & \\
\text{create TCB} & \\
\end{array}
\]

The user issues an OPEN command and the TCP responds by opening a Transmission Control Block (TCB) and entering the LISTEN state where it is receptive to the initiation of a connection by some distant TCP(s). The cause of a transition can be a command from the higher level (user) protocol layer, receipt of particular segments from the distant TCP, or a timeout. It is also possible for a single
Figure 16: TCP Connection State Diagram
received segment to cause more than one transition. We will explain how later.

The user commands shown include active and passive OPEN, SEND, and CLOSE. Other user commands are also described in the TCP standard, such as RECEIVE, ABORT, and STATUS. The RECEIVE and STATUS commands do not affect the protocol interaction between the TCPs which are party to the connection. Both commands are of concern only to the interface between the user and TCP layers though the status of a connection and the segments received do indeed affect the TCP's response to its user. The ABORT command does result in TCP to TCP interaction. In all but the CLOSED, LISTEN, SYN-SENT and CLOSING states the ABORT causes a RST segment to be formed and sent to the distant TCP. The ABORT command destroys the connection so that, in effect, the TCP enters the CLOSED state.

The Transmission Control Block (TCB) which is created at the time a TCP is OPENed contains several variables that serve to identify and specify a particular connection. They aid housekeeping and user/TCP communication, recording the security classification, precedence, compartmentation information, local and remote socket numbers, and the like. Numerous variables relate to send and receive sequence numbers. In truth, the complete state of the TCP is determined by the values of these variables as well as by which of the 10 states shown in the state diagram the TCP is in. Since
each sequence number variable can have $2^{32}$ possible values and ten TCB variables have sequence number values, the total number of possible states is astounding. Clearly, it would not be practical to attempt to specify TCP by means of a grammar containing separate non-terminal symbols for the total number of possible states. Our solution to this problem will be presented in section 4.2.

The transition from TIME_WAIT to CLOSED is governed by a timer set according to the expected segment transit time to be sure the remote TCP received the acknowledgment of its connection termination request. The TCP standard also specifies another waiting time after starting up or following a crash in which memory of sequence numbers is lost. In order to avoid selecting an initial send sequence number which might duplicate an old segment remaining in the network from a previous incarnation of a connection, a TCP must delay for the maximum segment lifetime. The specification suggests this might be 2 minutes. As long as a TCP "remembers" sequence numbers used it need not delay. The delay is intended to allow old sequence numbers from a connection to be eliminated from the network before a new initial send sequence number (ISS) is picked to synchronize a new incarnation of the connection. Ordinarily this delay is unnecessary since some mechanism for generating the ISS assures its uniqueness. A 32 bit clock with a nearly 5 hour cycle is suggested for the purpose. The delay guards against a crash which also destroys the
memory of the ISS generator.

The "three-way handshake" procedure used to establish a connection is discussed in [63]. Typically a connection is established when one TCP LISTENing for the contact of another TCP actively seeking to synchronize the connection first responds to that synchronizing segment and then receives the acknowledgment to its own SYN. The protocol also provides for simultaneous attempts to initiate the connection from each end. The three-way handshake is used to prevent old duplicate SYN segments from establishing a false connection. This exchange of SYN controls and acknowledgments serves to synchronize the sequence numbers for the connection. The initial send sequence numbers selected by the TCPs are the segment sequence numbers of the SYN segments and the sequence numbers of the SYN control bits. Any data octets sent on the connection will have higher (modulo \(2^{32}\)) sequence numbers. Data may be sent in the same segments as the SYN and ACK but will not be processed for delivery to a user until the receiving TCP reaches the ESTABLISHED state. Data octets are counted to compute the RCV.NXT variable which indicates the next expected segment sequence number. RCV.NXT is used in the ACK number field of segments prepared for transmission. Acknowledgments are cumulative so that an acknowledgment number acknowledges receipt of all lower numbered octets.
4.2. The Transmission Grammar Model

The Transmission Grammar (TG) was described in chapters 2 and 3. It is used to describe the protocol between two entities. One production rule used to define a transmission grammar for TCP might be:

\[
\text{LISTEN} ::= \text{Q.MSG_B.A1 <SYN-SENT>}
\]

The non-terminal symbols in this example represent states in the TCP state diagram. Note that non-terminals are enclosed by < >. The terminal symbol Q.MSG_B.A1 represents the action of enqueueing message 1 from TCP A on the message queue with which we represent the transmission link between TCP A and TCP B. By examining the state diagram one can guess that message A1 represents a synchronization message. It bears TCP A's initial send sequence number (ISS). Figure 16 shows two other transitions from the LISTEN state. These could be represented by individual production rules. In practice, we list the alternative productions to the right of the ::= separating them by commas. Teng originally proposed the Transmission Grammar as a context free grammar. While such a grammar might be able to more concisely represent the actions of a protocol, experience has shown that it is more understandable and easier to develop a protocol model using production rules which restrict the TG to a regular grammar.
The use of a grammar form of representation rather than a finite state diagram has a number of advantages which are clearly illustrated by the case of TCP. One advantage is the ability to represent complex protocols compactly and understandably. The state diagram of Figure 16 is quite clear and understandable. If it were to include all of the transitions representing out of sequence segment delivery, RST segments transmitted in response to ABORT commands and arrival of unacceptable segments, it would be extremely cluttered and difficult to follow. Such a state diagram would still be incomplete since many additional states are required to represent the full operation of the TCP connection establishment procedure. The TG production rules are linear and can be expanded quite easily to represent additional complexity. Though they may not as easily communicate the overall operation of a protocol as a simple state diagram, TG models readily depict the full range of actions possible from a particular state of a protocol machine. The Transmission Grammar is also well suited for input to automated validation programs. The use of BNF grammars to define the syntax of programming languages gives us parsing techniques which make the grammar production rules easy to handle. Another attraction is the possibility of automated implementation of protocols specified in Transmission Grammar form. Automated implementation of a portion of IBM's System Network Architecture has already been reported by a
group using a different specification technique [52].

Construction of the TG model began by first attempting to define the messages to be included in the model. For validation purposes it is important to limit the detail at which messages are modeled, thereby limiting the number of message types included. It was desirable to include all those messages under one message type which have the same effect insofar as the TCP protocol is concerned. At the same time a representation for each message type which produced some unique transition was needed. If one wants to include precedence and security in the model, one need have only one message type to reflect mismatched security/compartment values or a lower precedence. The response to each of these is the same, sending a reset. Discovering that two messages produce the same TCP response by examining the DOD standard [48] is not easy. Once the TG production rules have been prepared, however, it is easier to see that a group of messages precedes the same non-terminal in the alternatives of some production rule. By examining other rules, common subgroups can be found which lead to a common state and produce the same action in each of the productions in which they appear. Any two or more message types which appear grouped together in each production for which any one of them appears can be combined into a single message type. This is an important procedure applicable to the specification and validation of other protocols.
It first became evident in our effort to validate TCP. The importance of reducing the number of message types modeled stems from the state space explosion problem. A smaller population of message types reduces the number of transitions as well as the number of possible states of the transmission medium.

In constructing the set of messages to be included in the TG model only six fields of the segment header were considered—sequence number, acknowledgment number, ACK, RST, SYN, and FIN. The message types sent by TCP A were numbered from A1 to A15 and likewise from B1 to B15 for those sent by TCP B. In the skeleton segments listed below, the six control fields appear from left to right in the order listed in the sentence two preceding this. For example, "a" represents the initial send sequence number of TCP A, "b" that of TCP B. Similarly, "aa", "aaa", ... and "bb", "bbb", ... represent successive sequence numbers of TCP A and TCP B, respectively. Because we are not including data transmission in the model the sequence number range can be limited. Only the SYN and FIN control bits (when set) are counted in the sequence number space. An invalid segment sequence number (SEG.SEQ) received at A is represented by x. An invalid SEG.SEQ received at B is designated by y. The fifteen messages from TCP A to TCP B are as follows:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>a.0.0.0.1.0</td>
<td>SYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>a.bb.1.0.1.0</td>
<td>SYN,ACK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>a.bb.1.0.0.0</td>
<td>ACK of SYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>0.xn.1.1.0.0</td>
<td>RST to invalid no ACK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Message String</td>
<td>TCP Action Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------------------</td>
<td>--------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>yn.0.0.1.0.0</td>
<td>RST response to ACK yn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>aa.bb.1.0.0.0</td>
<td>ACK of SYN,ACK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>aaa.bbb.1.0.0.1</td>
<td>FIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>aaa.bbb.1.0.0.0</td>
<td>ACK of FIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td>a.bb.1.1.0.0</td>
<td>RST from SYN-RCVD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td>aa.bb.1.1.0.0</td>
<td>RST from ESTAB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A11</td>
<td>aaa.bb.1.1.0.0</td>
<td>RST from FIN-WAIT_n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A12</td>
<td>a.a.bbb.1.1.0.0</td>
<td>RST from CLOSE-WAIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A13</td>
<td>0.bb.1.1.0.0</td>
<td>RST to SYN from CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A14</td>
<td>aa.0.0.1.0.0</td>
<td>RST to ACK from CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A15</td>
<td>aaa. 0.0.1.0.0</td>
<td>RST to FIN,ACK from CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This idea of abstracting the large sequence number space into a few values represented symbolically by sequences of letters makes possible the specification and validation of the essential part to the TCP connection establishment procedure. One might think of the technique as a more extended application of the idea used in the X.21 validation of chapter 3. In that validation the call progress signals, called line identification signals, and calling line identification signals were abstracted into three symbolic values. Here three symbolic values were used for each of two variable fields in the TCP segment.

With the list of messages it is then possible to generate the TG model (the production rules) by examining the TCP standard. Some of the production rules can be taken directly from the state diagram. As previously mentioned, however, the state diagram is a quite incomplete representation of TCP. One specific chapter of the standard provides the functional specification [48]. That chapter describes the header format, and defines the state names and
variables referenced. It also discusses sequence numbers and their use, and discusses establishment and closing of connections as well as precedence and security. One section covers communication of data between the TCPs and another section defines the commands which form the user/TCP interface. For deriving the TG production rules, the meat of the functional specification is in a section entitled "Event Processing." This section attempts to describe the TCP's response to each of these events: User calls of OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; arriving segment event; timeouts of USER TIMEOUT; and RETRANSMISSION TIMEOUT.

The user commands have not been included in the model except to show each of the protocol actions which result. Often the only activity associated with a user command takes place over the user/TCP interface. The OPEN command, for example, can result in the sending of a SYN segment if the TCP is CLOSED. Otherwise, the TCP returns "error: connection already exists". Such error messages are part of the service specification for the TCP layer but are not part of the TCP/TCP protocol. From the described reaction for each event and each state of the protocol, the production rules indicated were developed. The notation has already explained by example, that is, Q.MSG_B.A1 associated with sending a segment. While Q indicates "Add a message to the queue", F (as in F.MSG_B.A1) indicates "Fetch a message from the queue". The fetch operation can
occur only if the indicated message is available in the queue. The transition associated with such a fetch occurs only if the fetch can occur. The desired message need not be at the head of a queue to be fetched. In order to model a system which permitted no reordering of segments in the transmission medium, segments would have been obtained from the queue with the dequeue operation. D.MSG_A.Bl takes a SYN segment originated at TCP B from the head of the message queue from B to A. Productions with such operations can only occur when the specified message is at the head of the specified queue.

During construction of the global Validation Automaton (VA) described in the next section, it was discovered that a few segment arrival events occurred for which production rules had not been provided. Such discoveries allowed completion of the transmission grammar. In that way elements of the specification which had been overlooked in initially preparing the TG could be added to the specification. Incremental additions of this sort developed iteratively. New production rules were added and when applied, revealed new events as yet undefined in the grammar. In the end, a complete specification was achieved.

The transmission grammar model, if complete, specifies all the actions of a protocol to whatever level of detail the protocol may be modeled. One can check the grammar for a number of qualities to see that it is properly constructed. If, for example, some non-
terminal symbol remains undefined (has no production rule from which a terminal string can be derived) or if some symbol is unreachable from the starting symbol, then the grammar is not well structured and is in error. The automated validation program [67] developed by Teng, which was discussed in chapter 3, accepts a Transmission Grammar (TG) and validates its structure. The validation program also produces a listing of all the possible cycles through which the defined protocol can pass. This is useful for detecting livelock looping.

The more important part of protocol validation lies in building a model of the system which includes both communicating protocol entities as well as the transmission medium connecting them. The global states of the Validation Automaton are modeled by a two by two matrix, as described in chapter 2, in which the main diagonal elements contain entity state information. Off the diagonal, the entry in row 1 column 2 (row A column B) contains the message queue from entity 1 (A) destined for entity 2 (B) while row 2 column 1 (row B column A) queue holds messages from 2 (B) destined for 1 (A). If we call the CLOSED state 1 then the starting state of the VA with entities A and B in state 1 and empty channel queues would look like this:

```
/1 e\  
| 1. | we label this state 1.
\e  1/  (e signifies an empty queue)
```
This starting state is the starting point for construction of the validation automaton, using simplified state variables for the moment. The possible next states of the VA which can be achieved by productions of each TCP separately making a transition from state 1 with empty message queues are derived using the transmission grammar specification. No next state of the VA is derived by simultaneous transitions of more than one entity. Four next states can be derived as shown:

**Entity A Productions**

\[
\begin{align*}
\text{CLOSED} & ::= \\
Q \cdot \text{MSG}_B, A1 \text{ <SYN_SENT>} & \implies \\
\text{CLOSED} & ::= \text{LISTEN} \\
\end{align*}
\]

**Entity B Productions**

\[
\begin{align*}
\text{CLOSED} & ::= \\
Q \cdot \text{MSG}_A, B1 \text{ <SYN_SENT>} & \implies \\
\text{CLOSED} & ::= \text{LISTEN} \\
\end{align*}
\]

The technique for constructing the VA is called *perturbation* [72]. By examining, in turn, all possible ways each system state can be perturbed the total interaction domain of the system is produced. That interaction domain is represented by the Validation
Automaton. As the possible perturbations to one particular global VA state are completed, a potentially new set of states is added to the set of states in the VA. A new, unexplored state is selected from the state set and perturbed in all possible ways. The process continues until all states in the state set have been explored. At the time the TCP validation was accomplished [69] the automated validation tool, Multi-Valid, to accomplish the perturbations and construct the Validation Automaton (see chapter 6) was not yet available. The construction of the global VA for TCP connection establishment described in this section was accomplished manually.

During the construction of the VA it is possible to check for several errors. If no perturbations are possible for some state, two possibilities exist. One, the state is a final state in the system. Two, the state represents a deadlock. The model defined any return to the CLOSED state from any state other than LISTEN as a transition to the END state. With both TCP A and TCP B in the END state the VA is in a final state and no further perturbation is defined. No true deadlock is possible in this model since modeling the ABORT user command made available a transition to the END state from any TCP state without regard to channel contents. Any state of the VA from which the only possible perturbations result from ABORT commands should be regarded suspiciously. It represents an undesirable condition.
A state in which some message queue contains a message for which an appropriate fetch operation is not part of some production alternative points up an incomplete specification. This condition can be checked either on adding a message to a queue or when expanding a VA state. On a number of occasions it was necessary to reexamine the functional specification and add new production alternatives to the TG when several such states were discovered. The technique of constructing a protocol specification in parallel with exploration of a reachability graph is known as protocol synthesis [1].

Any communications medium has a storage capacity that limits the amount of information on it at any instant. An attempt to exceed this capacity may result in the loss of information being transferred between processes. The VA includes states which contain as many as four messages in a queue. West [72] discussed constraints used to limit transmissions in order to prevent overflow, and the use of bounded channel capacity to model timing constraints.

Modeling a protocol as large as TCP can produce a validation automaton (VA) of considerable size. Size prohibits reproducing more than the very minor portion we have included in this section. Some techniques can help keep the VA size down. When modeling using the Fetch (F) action, the order of messages within the queues is not
significant. If a perturbation creates a VA state with MSG_A queue contents of: A7,A8 and the VA state set already contains another state differing only in that the contents of MSG_A is A8,A7, then the two states should be considered to be identical and the new state should not be added to the state set. Limiting perturbations to change only one entity state reduces the number of transitions (but not the number of states since simultaneous transitions can be modeled by a sequence of single transitions.)

In chapter 2 we explained the need to perturb only one of two symmetrical global states because these states have the same pattern of transitions and the investigation of any one of them will serve the validation purpose of finding syntax errors. Two global VA states are symmetrical if interchanging the states of the entities and interchanging the queue contents of one will make it identical with the other. In the case of TCP we are dealing with identical protocols so that their TG models will be mirror images of each other. A production rule in one grammar will have a corresponding rule in the other differing only in the origin and destination of messages. Some VA states are symmetrical with themselves. The starting state, state 1, is an example. Only half of the VA states directly derivable from such self symmetrical states need be produced since each will have a symmetrical state produced by applying the equivalent production rule from the opposite grammar.
By not modeling message loss and retransmission specifically the model has been greatly simplified. Since fetch retrieves messages in any order, the effect of message loss and retransmission is achieved simply by allowing a message to remain in its channel until it is fetched by the receiver.

As we mentioned earlier in the chapter, the complete state of a TCP connection is composed of both its named state, eg. ESTABLISHED, and the values of the variables in its TCB. While building the Validation Automaton, it became apparent that some of the TCB information was necessary to make decisions on the acceptability of certain segments and the appropriate action to be taken. The sequence number and segment acknowledgment (SEG.SEQ and SEG.ACK) of received segments must be compared with the initial send sequence number (ISS), initial receive sequence number (IRS), send unacknowledged number (SND.UNA), send next (SND.NXT), and receive next (RCV.NXT). Since reestablishment of a connection was not modeled, ISS and IRS remain the same throughout the VA. Defining the non-terminals using all the necessary state information, or state variables, provided a more complete and accurate model. Included in the state information for each TCP were the variables: state, SND.UNA, SND.NXT, and RCV.NXT. The non-terminal symbol in the grammar developed to represent the total connection state consisted of these variables concatenated with periods separating
the symbolic representation of each variable.

Because it is possible to reach many states in the TCP state diagram via different sequences of transitions, it is possible to have differing TCB variable values for the same TCP state. A glance at Figure 16 will show that FIN-WAIT-1 can be reached by transitions from both SYN-RCVD and ESTABLISHED. In one case the transition might be from SYN-RCVD.b.bbb.aa to FIN-WAIT-1.b.bbb.aa, whereas the other might be from ESTABLISHED.bb.bbb.aa to FIN-WAIT-1.bb.bbb.aa. In both cases the FIN-WAIT-1 SND.NXT and RCV.NXT variables are "bbb" and "aaa" respectively. The SND.UNA variables are different. In the first case, a received segment with SEG.ACK of "bb" would be viewed as acknowledging TCP B's SYN while in the second case such a SEG.ACK field would be viewed as a duplicate acknowledgment. The idea of representing protocol states by a composition of variables is new and was developed specifically in response to a need in modeling TCP. It is a technique of general applicability which can find use for many protocols.

Modeling the TCP states more completely permitted production of more precisely modeled messages. The initial list of message segments showed the FIN segment as message A7 (aaa.bbb.ACK.FIN). The need for messages A7a and A7b, respectively aa.bbb.ACK.FIN and aa.bbb.ACK.FIN, was discovered during the course of global VA construction. Thus, the process of building the VA not only
revealed new message receptions to add to the production rules of
the TG, but additional states (or nonterminal symbols) and
additional messages as well.

A further kind of unanticipated TCP state resulted from the out
of sequence reception of segments. Because segments must be
processed by the TCP in sequence number order the TCP must queue
segments received with SEG.SEQ greater than RCV.NXT. Using the
FIN-WAIT-1 state again as an example suppose RCV.NXT is "aa". 
Receipt of aaa.bbb.ACK which we have called message A8, ACK of FIN,
might at first glance require a transition to FIN-WAIT-2. That is
the transition shown from FIN-WAIT-1 on receipt of ACK of FIN.
However, comparison of SEG.SEQ with RCV.NXT shows a SEG.SEQ one
greater than expected. The A8 segment must be queued for processing
in its proper sequence. A need now exists to invent a new TCP state
which represents the FIN-WAIT-1 state with an A8 segment queued for
future processing. Call it FIN-WAIT-1_A8.bb.bbb.aa. This new state
calls for production rules, one of which is:

<FIN-WAIT-1_A8.bb.bbb.aa> ::= 
F.MSG_B.A7a Q.MSG_A.B8 CLOSING END

As can be seen, the FIN causes a transition to CLOSING, then the ACK
leads to the END state instead of FIN-WAIT-2.

A restriction forced by the unwillingness to model reconnection
of closed connections required altering the response to a RST
received in SYN-RCVD state. The TCP standard calls for a return to
LISTEN if the connection was opened from that state. Had a
production been used which did that, a new SYN could be sent reusing
the ISS. The model went, instead, to END.

One other peculiarity which came to light during development of
the VA also involved FIN-WAIT-1. It is possible for a received FIN
segment to have a SEG.ACK which acknowledges a previously sent FIN.
The event processing section of the TCP standard calls for
processing the SEG.ACK before processing the FIN. The ACK causes
the TCP to enter the FIN-WAIT-2 state. Later, when the FIN is
processed the TCP is caused to enter the CLOSE-WAIT state. This is
an example of one segment causing two transitions in the state
diagram.

4.3. Validation Results

This validation effort produced results in two ways. First,
the detailed reading of the functional specification brought to
light areas which could be made clearer, more consistent or more
complete. Second, the actual validation process produced some
interesting results. In the following we first address suggestions
to improve the functional specification. Then specific defects in
the protocol as specified in the DOD standard will be addressed.
4.3.1. Suggested Additions to the DOD TCP Standard

Several areas in which the functional specification section in the TCP standard could be improved were noted. All the suggested improvements involve changes to the Event Processing sub-section (sub-section 3.9). One area involves TCP response to user calls which might disclose information to users not authorized to use the TCP. Another area involves minor changes to clarify the actions a TCP takes. The third involves actions which were omitted from the specification. The event requiring the action and the specific action to be taken were evident from other portions of the standard. Other event processing action omissions were not obvious and were determined from the Validation Automaton. These are covered in section 4.3.2.

Unauthorized Access: The response to SEND call from CLOSED state includes "if the user should not have access to such a connection, then return 'error: connection illegal for this process'". This response is appropriate for all states. The same comment applies to the CLOSE, RECEIVE, ABORT, and STATUS calls. Without such a consistent response, an unauthorized user could obtain information about the status of a connection or perhaps affect its operation.

Clarification of State Transition: In a number of instances, the specification neglects to specify the state transition which
should occur in response to some event. In most cases the next state is the same as the current state (a loop transition in the state diagram) or the CLOSED state. Transitions to CLOSED from some other state include the instruction "Delete TCB, return 'ok'.” Deletion of the TCB implies entering the closed state. In that case the specific instruction, "enter CLOSED state," is not essential. This action is appropriate in response to CLOSE and ABORT in all states except CLOSED. In the latter state the instruction could include the phrase, "remain in CLOSED state." The specification would be clearer if these transitions were stated explicitly in each instance.

Omitted Actions: Several events and their appropriate response are evident from other portions of the TCP standard, but are not included in the Event Processing sub-section. The first involves the passive open. Open calls from the user come in two flavors, active and passive. The Event Processing description for the OPEN call event does not specify the TCP response to a passive open. Adding the phrase, "if inactive, enter LISTEN state, and return," will clarify the specification.

The ACK field processing of a received segment from FIN-WAIT_1 state should provide the response "if ACK of our FIN enter FIN-WAIT_2 state". This event and action are evident from the state diagram.
The response to RST from SYN-RECEIVED state should include "enter CLOSED state" if the connection was initiated by an active OPEN. Note that modeling the different responses to RST from SYN-RCVD depending on active or passive opening require splitting SYN-RCVD into two separate states.

4.3.2 Obscure Event Processing

Not every event which can occur in operation of TCP is evident from examining the TCP Standard. Construction of a global Validation Automaton (VA), however, is sure to reveal every possible event reachable from the starting state. One could discover the appropriate action to take in response to nearly every event from reading the TCP standard. There were a few that were omitted. These events must be included in the TCP standard if it is to be a complete specification of TCP.

The response to a FIN is unspecifed for SYN-SENT and SYN-RCVD states. Receipt of a FIN is possible in both of these states as shown by the validation. The following paragraph describes one sequence which results in such a reception.

An interesting situation appears in the VA which results in reception of FIN in SYN-RCVD state. Consider what happens if TCP A, executing an active OPEN, sends SYN to TCP B in LISTEN. TCP B responds with SYN,ACK. At this point the VA state is:

\[/4 \quad e\] State 4 is SYN-SENT
\[1.1.2.1\] State 3A is SYN-RCVD
\[\backslash B2 \quad 3A/\]
Before the SYN,ACK is received at A, TCP B executes CLOSE and sends a FIN. Now TCP A receives the FIN. Perhaps the SYN,ACK was delayed or even lost and in need of retransmission. How does TCP A respond to the FIN? This event is not specified in the TCP standard.

Not having a SYN from TCP B, TCP A has not established an initial receive sequence number (IRS) for the connection. How is it to recognize whether the FIN is valid? The segment acknowledgment number (SEG.ACK) is a valid acknowledgment of the initial SYN segment so TCP A should advance its send unacknowledged variable (SND.UNA) to equal SEG.ACK. Perhaps the correct response to the FIN is to either discard it or hold it for processing after the connection is established; the latter is a good choice only if the valid ACK can be taken as assurance that a SYN,ACK can be expected. Once a SYN,ACK has been received TCP A can test the acceptability of the SEG.SEQ for the FIN segment.

If, instead, TCP A accepts the FIN as valid, it sends an acknowledgment and enters the CLOSE-WAIT state (state number 9). This is the action included in the validation model of TCP. The user is informed that the connection is being closed and responds with its own CLOSE command. TCP A sends a FIN and enters the CLOSING state (state number 10). Receipt of the delayed or retransmitted SYN at this point presents another problem. Again, the connection has never been properly synchronized so no good
criteria exist to choose among three alternatives. One possible action is to ignore the segment. Another choice is to send another acknowledgment of the FIN. Sending a reset is the third choice. We elected a choice in which the validation TG calls for sending an acknowledgment.

The chain of VA states from that point is shown below. The MSG_B queue holds a FIN (A7b) and two ACKs (A8 and A8a).

```
/10 A8,A7b,A8a\ /10 A8,A7b \ | 1.1.2.1.2.3.1.2 | ---> | 1.1.2.1.2.3.1.2.3 |
\e 6/ \e 7/ | V  \\_______________V
/10 A7b\ /10 e\ | 1.1.2.1.2.3.1.2.3.4 | ---> | 1.1.2.1.2.3.1.2.3 |
\e 7_A8/ \B8 8/ 
```

The A8 message is aaa.bbb.ACK. The A8a is aa.bbb.ACK. A8a, at the head of the MSG_B queue, was sent before the FIN, A7b (aa.bbb.ACK-FIN), and has a lower SEG.SEQ than A8. Remember that FIN control bits are counted in the sequence number space so that, after sending A7b, TCP A's SND.NXT variable must be set to aaa. The next message, A8, is sent with the aaa SEG.SEQ. On receipt of the A8a ACK, TCP B enters the FIN-WAIT-2 state. If the A8 ACK is received before the A7b FIN, TCP B must enter the FIN-WAIT-2_A8 state with the A8 queued for future processing. Finally the A7b FIN is received and TCP B enters the TIME-WAIT state (state 8). The A8
may be processed at this point. Since A8 is a duplicate acknowledgment, it is ignored. This appears to be a reasonable response to the unexpected FIN. Other alternatives should be considered and the best action should be added to the TCP standard.

4.3.3 Deadlock Findings

In section 4.2 we mentioned the need to examine states of the VA with the only possible perturbations due to ABORT commands. A number of such states appeared in the VA. Examination showed that each such state had been arrived at through the previous exercise of an ABORT or CLOSE command. One question arises from a possible sequence in which one end either closes or aborts while the TCP at that end is in SYN-SENT state. The first TCP sends either a FIN or RST segment to announce to the distant end that CLOSE or ABORT is in process. The other TCP, in SYN-SENT state, simply closes its end of the connection with no notification to the distant end. This appears to leave the connection half open with the distant TCP in the SYN-RCVD state. In the TG validation model the only way out of SYN-RCVD is for the user to issue CLOSE or ABORT. The situation is not so serious, however. Although the model provides for no further perturbations from the END state. The functional specification clearly prescribes a response to segments received with no TCB opened (connection is closed). There is a SYN,ACK segment en route to the closed TCP. The TCP which initially closed the connection would respond to the outstanding SYN,ACK with a RST segment from the
CLOSED state. The design behaves properly in this regard. If one end ABORTs, the other end should recognize a half open connection and abort also; which is exactly what occurs. Should the SYN.ACK segment be lost by the transmission medium a retransmission will occur following timeout by the TCP which originated the SYN,ACK until the RST response closes the connection completely.

4.4. Summary

Though extensive effort to classify and detail the proper response to every event is evident in the TCP standard, the attempt to convert that specification to a transmission grammar and, more importantly, to construct a validation automaton from that grammar model has revealed weaknesses in the specification. Validation techniques have, once again, been shown useful for exposing areas of incompleteness in a protocol specification.

The Transmission Grammar was an effective tool for specifying the Transmission Control Protocol. A number of interesting problems were presented which called for solution. The first involved representation of the sequence number (SEG.SEQ) and acknowledgment (SEG.ACK) fields of the TCP segments. The second problem was to incorporate Transmission Control Block (TCB) variable values into the total state of the TCP specification. A third problem resulted from the need to separately designate TCP states for which out-of-
sequence segments had been received and not yet processed.

The first problem is not unique to TCP. Specification and validation of a wide range of protocols will be limited by the difficulty of representing such variables using any state transition based technique. A generalization of the procedure we used for TCP can be stated as follows:

- Determine some limited range of variable value over which the characteristics of the protocol of interest can be modeled. (In TCP the connection establishment portion of the protocols operation was of interest. The connection could be established and ended with a range of three sequence number values.)

- Select some representation of the initial value of the variable range. This initial value representation can be some arbitrary value from the total address space, or some abstraction such as a mnemonic.

- Devise a representation for subsequent values in the limited range. If an arbitrary value was selected the subsequent values can be represented naturally. Otherwise any readily recognizable representation will be satisfactory.

If the total state of a protocol entity depends on the values of a number of variables, the transmission grammar specification for the protocol entity can be constructed with the non-terminal symbols consisting of concatenated sequences of variable value representations. Variable values can be represented as described above.

The solution to the problem of representing states for which an out-of-sequence segment is enqueued for later processing in sequence is more ad hoc. It is, nevertheless, applicable to other similar
situations so long as the number of messages enqueued is limited. The solution adopted was to concatenate a representation of the out-of-sequence segment to the end of the non-terminal representation for the state.

One further discovery regarding the Transmission Grammar (TG) model resulted from the TCP validation. The original representation of a global state consisting of both message and acknowledgment queues was found to be poorly suited for representation of protocols in general. A representation consisting of entity states and message queues only, provides a more suitable global state. In the latter representation, acknowledgments are represented as a particular type of message.

We must point out that the more serious errors in the TCP specification were discovered independently and corrected in a later edition of the DOD standard [50]. Most of the suggested clarifications were also included in the ninth edition of the standard.
5. CONNECTION FREE MULTI-DESTINATION PROTOCOLS
IN PACKET RADIO NETWORKS

This chapter presents an analysis of the state-of-the-art in connection-free multi-destination protocols and an examination of their applicability to Packet Radio Networks (PRNETs) in general. The intent is to examine some of the issues involved in design of multi-destination protocols, and thereby gain an understanding of the complexities which are faced in attempting to validate such protocols using the Transmission Grammar (TG) and Validation Automaton (VA).

The first section provides a set of definitions to establish a conceptual framework in which the protocol problem can be discussed. Section 5.2 presents the basic category of communication network considered in this chapter. Section 5.3 examines the problem of multi-destination addressing and routing in PRNETs. Section 5.4 compares connection-oriented versus connection-free protocols for the multi-destination environment. Section 5.5 deals with the feasibility of providing an intermediate-level protocol, combining the attributes of the connection-free protocol with the data
transmission reliability of the connection-oriented protocol for application to the PRNET.

5.1. Preliminary Definitions

In this section key terms used in the chapter are briefly defined.

5.1.1. Multi-Destination Routing and Addressing

Many applications require the ability to communicate identical information to more than one destination (e.g., conferencing, database updating, etc.). This capability could be achieved by sending multiple copies of the original information, one copy to each destination. The alternative, a multi-destination routing capability could be provided. Greater efficiency and low delay can be expected in PRNETs, due to the inherent broadcast mode of the radio channel.

It is useful to distinguish between multi-destination addressing, how the network selects the destinations of the message, and multi-destination routing, how the network selects the path(s) over which the message travels. The distinction is also made between the addressing mode, how the user identifies the intended recipients of a message, from the addressing implementation, how the network processes the message. The former is an interface between the user and the network, whereas the latter is a protocol within the network.
5.1.2. Connection-Free and Connection-Oriented Protocols

Connection-oriented protocols typically include an establishing phase, such as that of the Transmission Control Protocol (TCP) described in chapter 4, during which necessary parameters, such as sequence numbers, are synchronized at each end, and a data transfer phase followed by a clearing phase. Such connection-oriented (or virtual circuit, VC) protocols are in contrast to connection-free (or datagram, DG) protocols, which permit data transmission without establishing a connection. Thus, connection-free protocols appear as a very simple and efficient transport facility; however, they are inherently unreliable.

5.2. PRNET Model

A Packet Radio Network (PRNET) is a large network of three types of devices, terminals, repeaters, and stations. These devices have a common component called a packet radio unit (PRU) or packet radio (PR) for short.

Each packet radio (PR) consists of two parts. The radio section provides a half duplex transceiver. Transmission and reception both take place over the same frequency channel necessitating the half duplex mode of operation. The radio section is connected to and controlled by the digital section which is a microcomputer having both a microprocessor and storage for both data
and instructions. The digital section executes those parts of the PRNET communications protocol which are the responsibility of the PR. They include channel access control (time of transmission), error detection, and determination of what action to take on a received packet.

Access to the network is obtained via connection to the digital section of a PR. A packet, formatted with the proper addressing and control information in the header, is input into the network via a terminal over this connection. The digital section of the PR processes the packet in accordance with the protocol adding any necessary control information and providing address translation if necessary. The digital section causes the packet to be transmitted by the radio section. The packet will be received by each PR in the net within radio range provided the transmission was successful. A successful transmission depends, among other things, on whether no other radio in the net within range was transmitting at the same time. Many radios may hear the packet, but only those for which the routing algorithm requires action will store it for further action. Such radios will evaluate the header information to determine what further action is required.

Packets addressed to a local terminal will be processed for delivery and delivered over the connection. Otherwise, the packet will be retransmitted according to the routing algorithm. In this
fashion packets are passed from radio to radio in store-and-forward operation typical of packet switched networks until delivered to their final destination(s). Those PRs which participate in this forwarding operation will be referred to as repeaters. It is likely that all PRs in a network will be capable of functioning as repeaters; however, only selected ones will be designated to serve that function. We will not consider the criteria on which a PR is selected to serve as a repeater.

A terminal's role is one in which it accepts no packets destined for other users and consequently it only transmits packets that are originated by its users. A repeater is simply a packet radio that happens to be programmed to function in a repeater mode. If there is a need for clarification, a radio will be referred to as the packet radio at the terminal, or to the packet radio repeater (or simply repeater). A packet originating at a terminal proceeds through a series of one or more repeaters until it reaches its final destination. Thus, the set of repeaters forms a backbone of radio links in the network.

A station usually has a minicomputer connected to a packet radio. The station provides centralized network control. Functions of the station include determination of network connectivity, establishment of routing paths, and dynamic assignment of repeaters. In a large network, multiple stations would offer an advantage in
providing distribution of stations throughout the network, thus
decreasing the average number of hops from each PR to its nearest
station. Stationless PRNETS are also possible and are discussed by
Kahn [32] along with the functioning of a station or multiple
stations in a PRNET. A stationless net has some advantages, in that
distribution of control reduces the susceptibility of the network to
damage or failure at a key node in the network. For the purposes of
this chapter the network model assumed is the single station
network. Figure 17 shows a typical PRNET.

5.2.1. Protocol Layers for PRNET

Chapter 3 described the technique of designing computer
communications protocols as a series of layers and illustrated the
idea with the ISO Reference Model for Open Systems Interconnection
(ISO RM/OSI). This subsection will detail from the lowest layer
toward the outer layers the divisions chosen as a model for
discussion of connection-free and connection-oriented protocols in
this chapter. This division of function and responsibility into
horizontal layers makes for less complex peer entities to implement
the protocols. That means that validation will be less complex and
can help keep the state space within reasonable bounds.
Figure 17: Typical PRNET
5.2.1.1. Access Protocol

At the lowest level (save the physical radio transmission) of the network lies the Radio Channel Access Protocol. The function of this protocol is to share the common channel radio frequency among those PRs within "earshot". Techniques for accomplishing this include pure ALOHA, slotted ALOHA, and Carrier Sense Multiple Access (CSMA). These techniques will not be described here. Kahn [32] discusses these techniques for controlling the multiple access channel as well as discussing the nature of the problem.

5.2.1.2. Node-to-Node Protocol

The protocol governing communications between adjacent nodes along the route of transmission of a packet through the network is the Node-to-Node Protocol. The algorithm implementing this protocol is in the digital section of the PR. The functions of the Node-to-Node Protocol (which will be described in some detail later in the chapter) are the following: 1) routing determination, 2) addressing implementation, 3) hop-by-hop acknowledgment and retransmission on time-out. To the extent that these functions can be modeled as the work of independent entities, validation can be simplified.
5.2.1.3. Transport Protocol

Another protocol operates between the PR connected to the source of a packet and the PR connected to the destination device. We call this the Terminal-to-Terminal Protocol or the Transport Protocol. In the transport layer are the following functions: 1) assembly and disassembly, 2) message and packet sequencing, 3) connection management, 4) flow control, 5) end-to-end acknowledgment and retransmission on time-out. Within this layer it is also possible to model and validate independent functions. In chapter 4 we showed the validation of connection establishment and termination for the TCP as a separate function.

5.2.1.4. Higher-Level Protocols

Finally, above the transport level are Higher Level Protocols. These protocols have nothing to do with network communications, but use the transport facility provided by the transport level as a channel for accomplishing tasks of interest to processes in the terminal device or of interest to the device operator. Examples of such Higher Level Protocols are remote job entry protocols and file transfer protocols. These applications protocols will not be discussed further. Figure 18 shows typical message formats for the first three layers of protocol described above.
Figure 18: Message Formats for Protocol Layers
5.2.2. Previous Work on VC and DG Protocols

In the years since the pioneering work of ARPANET a considerable body of work has been published on computer communication protocols for both virtual circuit (VC) and Datagram (DG) service [51]. Much of the published work deals with protocols which have actually been implemented in operating networks. The host-to-host protocol of the ARPANET [14] provides a virtual communications circuit. Operators of public packet-switched networks predominantly offer virtual circuit service. By and large these networks provide network access based on the CCITT X.25 standard which provides a virtual call interface to the network. Datagram (DG) capability has been added to X.25 by CCITT [17]. Section 5.4 will look at the two as they apply to multi-destination communications.

5.2.2.1. Connection Free Protocols

Connection-free and Connection-oriented protocols were defined in section 5.1. Within the context of packet switched data communications networks (PSN) the existence of a virtual circuit between two entities allows them, in many respects, to communicate as though a physical connection existed between them. In particular, they should be assured that transmissions received over the VC originate with the entity at the distant end of the VC, and that messages are received in the order transmitted. Connection-free communications between two entities, on the other hand,
proceeds without benefit of control of the sequence of delivery of messages and without reference to any circuit identification. If a message happens to be delayed in the PSN allowing a later transmission to reach the destination ahead of the delayed message, the first arrival will be delivered first under connection free protocol. If the two messages are unrelated (particularly with regard to time) no harm should result, but if the sequence in which the messages were transmitted is of some importance to the recipient, then the receiving entity needs a means of resequencing arriving messages.

Datagram (DG) service is the most elementary form of packet switched service using a connection-free protocol (CFP). In fact, DG service is the most elementary form of packet switching. Pouzin [51] defines a datagram (DG) as

a packet of information which is carried to its destination without reference to any other packet, or prior setting of a data path. In other words, a DG is a self-contained packet, in terms of switching.

There are applications for which DG service is ideally suited. If the entities never require transmissions larger than a single packet and hold conversational interactive communications sessions such that neither entity sends an additional packet until it has received a response to its latest transmission, then DG is ideally suited to the task. Such an interchange (perhaps between a credit authorization terminal and a host computer) maintains
synchronization so that out of order packet delivery (discounting duplicates) is also avoided.

Datagram service provides for sending single addressed packets through the network without prior call set-up. The DG protocol provides no means to control the rate at which a sender generates messages for a particular receiver, though the network may protect itself from being flooded with more traffic than it can handle. Though the network may provide no synchronization services to match packet delivery rate with a destinations ability to handle it, that does not mean that the network is unusable for activities which require such synchronization. It is possible for a higher-level protocol to exist on top of the DG protocol which can provide such synchronization. Such a protocol can be defined only between one particular pair of communicating entities or may be more generally available for wider use.

5.2.2.2. Connection-Oriented Protocols

As was stated earlier, connection-oriented protocols in a packet switched network simulate some of the characteristics of a physical circuit. A virtual circuit (VC) exists between communicating entities by virtue of its having been set up by the VC protocol as a result of one or both of the entities following some established call set-up procedure which is a part of the VC protocol or by virtue of the network management having established the VC on a permanent basis (probably with some considerable charge
therefore). The network designers would like to guarantee that messages sent over a VC would be delivered in the order in which they were sent, without error and without the correspondents being concerned with the physical packet size in the network. To achieve this the VC protocol undertakes to establish an association (connection) between the communicating entities and not intermix traffic from other associations or VC with traffic on this circuit, to disassemble messages larger than the network packet size and reassemble the packets in correct order prior to delivery, to deliver messages in the order in which they were received from the originator, and to deliver only such messages as are introduced into the network without duplicating any. In addition to these features which distinguish VC from connection-free service, VC service typically provides facilities for controlling the flow of traffic on the VC and includes end-to-end error control which tends to make VC service more reliable than DG. While error control is possible within the context of a DG protocol, it is usually left for the user to add on top of DG. Flow control, on the other hand, just does not fit the concept of DG service.
5.3. Multi-Destination Addressing and Routing

How should one user of a computer network address a message to other users? How should the network select the route over which the message travels? The answer to these questions are fundamental in successful operation of a computer network. A good characterization of routing and how it relates to addressing and naming is given in [60]:

The name of a resource indicates what we seek, an address indicates where it is, and a route tells us how to get there.

Most of the work in addressing mechanisms and routing algorithms deals with single-destination messages for point-to-point packet-switched networks. The topic of multi-destination addressing and routing has, surprisingly, received little attention to date. This section first briefly surveys the literature on addressing and routing in computer networks, and then proposes various multi-destination addressing nodes and routing algorithms for use in PRNET.

5.3.1. Multi-Destination Addressing

For reasons of convenience and efficiency, it is desirable to provide a facility for addressing a message to multiple destinations. The multi-destination addressing capability can cut down on network traffic and user overhead by substituting a single transmission for several separately addressed messages. The
inherent broadcast mode of the radio channel also provides an opportunity for efficient multi-destination routing that is not present in point-to-point packet-switched networks. Routing algorithms also depend heavily on the address recognition mechanism used in the interface. For example, an optimal multi-destination routing is one for which the addressing mechanism is such that in as few transmissions as possible, those processes named as destinations in a multi-destination message are recognized by the interface hardware, and copy the message.

As pointed out in section 5.3, one should distinguish between the addressing modes (how the user identifies the intended recipients of a message) and the addressing implementation (how the network processes the message). The former is an interface between the user and the network, while the latter is a protocol within the network (belonging to level 2, the Node-to-Node protocol, according to the PRNET model presented in section 5.2). The principle of economy of means suggests that an all-purpose addressing mode with a single implementation technique would be most desirable. However, in many cases several addressing modes are required to provide the best efficiency levels and to ensure adequate network reliability. Four addressing modes were proposed in [37] to meet the need of PRNETs: 1) single-destination, 2) full-broadcast, 3) group-destination, and 4) multi-destination. Two criteria were used for
evaluating addressing capability: 1) overhead in terms of memory and processing power for name mapping, additional bits in the packet header, and decoding logic in the interface; and 2) total delivery time, defined as the time necessary to deliver a message to all its intended destinations.

Figure 19 is from [37] and shows the address field for each of the four addressing modes proposed. In each mode, the first two bits of the address field are used as mode indicators.

5.3.1.1. Single-Destination Addressing Mode

This is a degenerate case of multi-destination addressing in which there is one and only one destination. The addressing scheme is designed for a PRNET with up to 1,000 nodes. Ten bits are needed to uniquely identify the physical address (ID) of each node. Because a user should be able to communicate with other users or systems by logical name rather than strictly by physical ID, the station should perform the mapping, using a table-lookup. Partial mapping tables may exist at certain nodes to keep frequently used IDs. Figure 20 shows some sample logical names associated with physical IDs as they might appear in a PRNET serving a military organization.
<table>
<thead>
<tr>
<th>Field</th>
<th>Mode</th>
<th>Unique ID</th>
<th>No. of Bits</th>
<th>Total:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE-DESTINATION</td>
<td>OU</td>
<td>10</td>
<td>2</td>
<td>12 bits</td>
</tr>
<tr>
<td>FULL-BROADCAST</td>
<td>11</td>
<td></td>
<td>2</td>
<td>2 bits</td>
</tr>
<tr>
<td>GROUP-DESTINATION</td>
<td>01</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>MULTI-DESTINATION</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td>64</td>
</tr>
</tbody>
</table>

Figure 19: Address Field Formats
**FUNCTION:** Command Message

**Corps**

**Division**

**Brigade**

**Battalion**

Figure 20: Sample Addressing Scheme Implementation
5.3.1.2. Full-Broadcast Addressing Mode

This is a special case of multi-destination addressing in which a message is fully broadcast to all other nodes in the network. Thus the address field for this mode contains only two bits to denote its mode and nothing else.

5.3.1.3. Group-Destination Addressing Mode

In some environments, multi-destination messages are more likely to be sent to groups of nodes that are related by some positions in a functional/organizational hierarchy. For example, consider the case of command message flow in a military organization, as shown in Figure 20. Each node in this hierarchy can be uniquely identified by its level in the hierarchy and its unique ID within that level. Thus the third mode of addressing takes advantage of this functional/organizational relationship inherent in the organization, and is called group-destination addressing since the assignment of an ID in the address field represents its position in the hierarchy. There are several ways to utilize these group fields to provide maximum addressing flexibility:

- All members of a group up to level i are addressed by setting all level fields to 1's up to level i and all remaining fields to 0's. For example, to address all corps, division and battalion nodes shown in Figure 20, the address field would be set to

01 111111 11111 00000 00000 00000 00000 00000 00000.

- All members of a particular group at level i are addressed by specifying the unique group ID up to level i-1, filling in level i with 1's and setting all remaining fields to
zero. For example, all companies belonging to battalion B3 would be addressed as

01 (C1) (D2) (B3) 11111 000000 000000 000000 000000 000000

where the logical addresses in parentheses would appear as physical addresses in the actual address field.

- All members of a group below level i are addressed by specifying unique group ID's up to level i-1 and setting all remaining fields to 1's. For example, all battalions within division D2 and all companies within all those battalions can be simultaneously addressed using the following address field:

01 (C1) (D2) 111111 111111 111111 111111 111111 111111.

Since this particular hierarchy only has three levels, the value of the last 5 fields is ignored.

5.3.1.4. Multi-Destination Addressing Mode

In this mode, which is basically group-oriented, a set of individual members within a group can be selectively addressed in a single address field. In this case, 10 bits are used to uniquely identify a group and 64 bits are used as a bit-map to individually identify members of the group. For example, in figure 20, to send a multi-destination message to C1, C3, C5 and C6 in B3, the address field is (ML)10101100...0, where ML is the unique ID of group B3.

It should be noted that the group-destination and multi-destination addressing modes do not provide a general multi-addressing capability. For example, there is no way to form a single address field to broadcast to a division group and to all battalion groups. This requires two messages under this scheme. Similarly, there is no way to form a single address field to send a
message to an arbitrary set of destinations that are not related by membership in the same group. We believe that the need for this generality is small and the overhead too high.

The proposed scheme provides minimum total delivery time for all addressing modes if all multi-destination addressing is conceptualized within the defined group framework. For those multi-destination messages outside of the group framework, total delivery time will be increased in proportion to the number of messages greater than one which must be sent. We expect such messages to occur infrequently. We are thus willing to experience greater total delivery time in this case in order to reduce other aspects of addressing overhead.

5.3.2. Multi-destination Routing

The main features that distinguish the PRNETs from point-to-point packet-switched networks are: 1) the communication channel is shared dynamically among all devices in the network; and 2) devices use a broadcast mode of transmission so that packets can be transmitted to several devices simultaneously. Thus the inherent broadcast mode of the radio channel provides an opportunity for efficient multi-destination routing demanded by multi-destination messages.

As described in section 5.2, a PRNET includes three functional devices: terminals, repeaters, and stations. A terminal is a source and/or sink of information flow. A repeater is used to
provide area coverage for a cluster to route messages. The station provides centralized network control functions, including packet radio initialization, network connectivity monitoring, routing path establishment, directory management, etc.

For routing purposes, a cluster of terminals are dynamically assigned by the station to a nearby repeater as the first relay node. A packet sent by a terminal in the cluster is then relayed from repeater to repeater in a store-and-forward fashion, according to the routing path set up by the station, until it arrives at the final repeater(s), which then broadcasts it directly to its destination(s). Thus the set of repeaters forms the backbone of a PRNET as far as routing is concerned. In this model, which is taken from [35], it is assumed that a terminal can send a message directly to any terminal in the network, without being relayed through the station, if both terminals have been properly initialized.

There are two key objectives in developing routing algorithms for PRNETs. First, we must assure that a message launched into the network from a source will reach its final destination(s) with high probability. Second, we must guarantee that a large number of messages will be able to be transmitted through the network with a relatively small time delay. The first goal may be thought of as a reliability issue, whereas the second is an efficiency consideration.
As in the case of multi-destination addressing discussed in the previous subsections, we found that several routing algorithms are required to provide the best efficiency levels and to ensure adequate reliability [35]. Thus we proposed the following four routing algorithms to meet the need of PRNMTs, each providing different degrees of efficiency and reliability for various addressing modes.

5.3.2.1. Undirected Broadcast Routing

This scheme is useful for initializing the network and for performing routing without the management or direction of a station, and has been described in [19, 20, 32]. To implement undirected broadcast routing, every packet must have a unique packet identifier (UPI), and every repeater in the network keeps a short list of UPIs for packets previously relayed by it. If a repeater receives a packet whose UPI is already on its list, it will discard the packet; otherwise, it will accept and relay the packet. The use of a list of UPIs in each repeater prevents any packet from returning to portions of the network through which the packet or any of its copies has already passed. Thus in this scheme, each packet radiates away from the source radio as in a wavefront type of propagation, and the packet will eventually arrive at every other radio in the network so long as the network is not partitioned.

Undirected broadcast is thus best suited for routing full-broadcast messages and also good for routing group-destination
and multi-destination messages if the number of destinations is large. However, it is not an efficient mode of operation for routing single-destination messages, since it generates too many duplicate copies of messages. This scheme is very reliable as compared with other schemes to be given below, but it is not absolutely reliable in the sense that not all radios may actually receive every broadcast message and an end-to-end procedure must be invoked if reliable delivery is essential.

5.3.2.2. Broadcast Routing

This scheme uses shortest path routing and has been described in [20]. In this scheme, every repeater in the network keeps a table in the form of a distance vector. Repeater Ri will be given by the station the vector Di = (dij), where dij is the minimum number of hops from repeater Ri to repeater Rj. The destination repeater ID and the distance to it from the currently relaying repeater are used to determine the next next-hop repeaters that should accept the packet for relaying. In this case, a packet is accepted by a downstream repeater if the repeater is nearer in distance to the destination repeater than the currently relaying repeater. The last repeater in the route can identify itself by noting that its distance to the destination repeater is zero.

This algorithm is simple and reliable for routing single-destination messages, but it may generate many duplicate copies of messages along the path from source to destination. A
better scheme for routing single-destination messages is given next. Directed broadcast routing, however, is not suitable for routing other types of messages ('full-broadcast, group-destination and multi-destination), and requires periodic updating of the distance vectors by the station.

5.3.2.3. Source-Based Routing

This is a point-to-point routing procedure by which a packet originating at the source proceeds directly through a series of one or more repeaters until the packet reaches its destination. The routing information, consisting of an ordered set of repeater IDs, is determined by a station. The station is able to compute a nearly optimal route since it knows the current network connectivity and traffic. The scheme has been described by Kahn [32], who has suggested two methods of sending the point-to-point routing information. The first method is to distribute the information to the individual repeaters along the point-to-point route, and the second is to send it directly to the sender's packet radio. In the latter case, each packet originating from the sender will contain the entire set of repeater IDs in its header, thereby increasing overhead to the packet and reducing network efficiency.

This scheme is best suited for routing single-destination messages only. It generates no duplicates, but the sender has to ask the station for routing information before packets can be transmitted. As network conditions change, routes have to be
5.3.2.4. Distributed Branch Routing

In this scheme, every repeater in the network keeps a routing table containing information similar to ARPANET IMPS [39], i.e., routing information about the next hop for every possible destination. The table is updated periodically by the station after a radical change in topology or traffic. This scheme is designed mainly for routing group-destination and multi-destination messages, and works as follows.

When a group-destination or multi-destination packet is transmitted by the sender to its nearby repeater, the repeater generates one or more copies of the packet according to the destinations and then transmits these packets to the next-hop repeaters according to the routing table. This process is repeated by every intermediate repeater until the packet is sent to all its destinations. The procedure is best explained by an example shown in Figure 21.

In this figure, assume that terminal T11 (covered by repeater R1) sends a multi-destination packet to terminals T54 (by R5), T61 and T63 (by R6), T75 (by R7), and T92 (by R9). When the packet is transmitted to R1, it will generate two copies of the packet, one destined for T54, T61 and T63 and the other for T75, T92 and T93. These two packets are then relayed to next-hop repeaters R2 and R3, respectively. At R2, its packet will be relayed to R4, where two
Figure 21: Example of Distributed Branch Routing
copies of the packet are generated and transmitted to R5 and R6, respectively. These four packets are finally broadcast by R5, R6, R7 and R9 to terminals T54, T61, T63, T75, T92 and T94.

This scheme can also be used to route single-destination messages when the number of destinations reduces to one. However, it is not suitable for routing full-broadcast messages because of unnecessary overhead resulting from additional processing at repeaters.

From the above description, we see that there is no single routing algorithm that is most suitable for routing all types of messages. As in the case of addressing modes, we found that the fully comprehensive nature of the capability desired of the protocol demands multiple routing schemes, depending on the specific addressing mode. In summary, undirected broadcast routing is best suited for routing full-broadcast messages, directed broadcast routing and source-based routing are for single-destination messages, and distributed branch routing is for group-destination and multi-destination messages.

So long as interaction between the various routing modes is not a problem, the routing protocols can be modeled and validated by separate models for each routing mode. This is an application of the divide and conquer principle. Attempting to combine all of the various routing mechanisms into one entity specification might
produce a grammar with so many states that the resulting global validation automaton would be intractable. It might also be necessary to restrict the size of the network being validated as well as restricting the number and size of the address groups. In doing so, the protocol designer must recognize that some possible protocol interactions may not have been validated.

5.4. Multi-Destination Protocols

The previous section considered multi-destination addressing and routing in PRNETs. In particular, routing simply directs a multi-destination message efficiently to its destinations. However, this mechanism does not guarantee that a multi-destination message will arrive at every destination or that every destination will receive only one copy of the message and will discard any duplicated or damaged messages. In other words, a multi-destination protocol is still needed on top of multi-destination addressing and routing, if some levels of reliability are desired.

This section first briefly surveys the literature on multi-destination protocols. It then considers in turn acknowledgment mechanisms, connection-oriented multi-destination protocols and connection-free multi-destination protocols. Finally, a brief analysis and critique are given.
5.4.1. Previous Work

While packet switched network protocols for single destination communications have been described and discussed in some detail in the literature [12, 33, 58, 23, 68], a comparable body of work does not exist for multi-destination protocols. Work by Pardo and by Dalal was referenced in section 5.3. We have proposed addressing modes which were also described in section 5.3. An algorithmic description was presented in [37] for the node-to-node and terminal-to-terminal level protocols applicable to both single destination and multi-destination datagram service.

Pardo suggests the use of an acknowledgment bit map at the sender's site with retransmission on timeout to only those destinations present in the destination bit map but missing in the acknowledgment bit map. He also extends the idea of "connection" (i.e., an association between a sender process and a receiver process) in point-to-point protocols to the notion of an "association group" between a sender process and a set of receiver processes.

One of the touchy aspects of maintaining the correct control information and staying in synchronization on a virtual connection is establishing and updating sequence numbers used by sending and receiving processes to reorder packets and messages and to detect duplicates. Pardo points out that each distributed algorithm
consisting of \( N \) remote processes could have \((2^N-1-N)\) different next sequence numbers (<NSN) at each protocol site [44]. The likelihood that every process would want to exchange messages with every other process in the distributed system that Pardo was considering is infinitesimal. Nevertheless, the potential for uncontrolled growth of control information in the multi-destination environment is well worth remembering. Of course, a transmission grammar representation of such complex state information would be thinkable only for small numbers of processes.

Among the most inclusive references on the subject of packet radio communications, Kahn, et al. [32] discuss multi-destination communications, but primarily from the standpoint of broadcast routing. Protocols are not discussed.

5.4.2. Connection-Oriented Multi-Destination Protocols

The advantages of connection-oriented protocols are the increased reliability they can provide, the delivery of messages in correct sequence (i.e., delivery in the same sequence in which the messages were transmitted), the provision of flow control, and the ability to deliver messages larger than the physical packet size transmitted in the node-to-node layer. Virtual circuits also allow addressing information to be exchanged between source and destination as part of the connection setup procedure. As a result, packets flowing in the network can be addressed much more simply. Only the circuit identification and direction need be transmitted if
intermediate stages know of the connection routing. At any rate, it is not necessary to include both the logical address of source and destination entities and the physical address information needed for routing since the logical addresses can be supplied at the destination prior to delivery. This can result in considerable saving for long term use of a connection. What do these features mean in terms of multi-destination communications?

First of all, reliability of the VC depends on the end-to-end acknowledgment that is provided in a VC. The end-to-end acknowledgment is frequently tied closely to the flow control and sequence numbering system employed in the VC protocol. In the window mechanism system employed commonly for this purpose, an acknowledgment for a specific sequence number also acknowledges all preceding sequence numbers and advances the window allowing additional transmissions to flow from the source. In multi-destination communications we have not one destination sending End-to-End Acknowledgment (EEACK) to a source, but many destinations sending EEACK. The implications of this are several. Since all EEACKs converge on one network node, congestion is a likely result if the number of destinations is high. This congestion may require limitations on the use of multi-destination messages. If the EEACK is used only for reliability (to confirm delivery), then the acknowledgment bit map proposed by Pardo will suffice to allow
retransmission when required. If the sender must also remember the remaining window size for each destination, the situation becomes more complex. Here a bit map is needed for each packet sent. The bit map must be remembered until all destinations have acknowledged that packet or until the protocol "gives up". The highest sequence number which can be legally transmitted is the sequence number of the earliest acknowledgment bit map being retained for additional EEACKs plus the window size minus one. (Sequence number computation, of course, must be done modulo some limiting number to keep the size within reasonable bounds while attempting to avoid duplicating sequence numbers still alive in the network.) Again, large numbers of destinations are likely to cause problems. Not only is more memory needed to store the bit maps, but as the number of destinations goes up the probability rises that some EEACKs will be delayed sufficiently long to cause a halt in transmission to all destinations. This is another of the situations difficult to model with a grammar specification for single destination protocols. For multi-destination protocols, the complexity is much worse. The only hope for being able to validate such aspects of multi-destination protocols is to restrict the range of nodes and variable values in the model to a small subset of those possible.

This discussion has looked upon connection-oriented multi-destination protocols as being essentially one way connections with
data flowing only from one sender to multiple destination, but with control information also flowing in the other direction. One might also consider a virtual connection which behaved like a multi-party conference call in the switched circuit voice net, but that will not be considered in this work.

5.4.3. Connection-free Multi-destination Protocols

A connection-free (datagram based) multi-destination protocol is much simpler than any attempt at multi-destination virtual circuit protocol would be, since the datagram is a simple one way device which requires the maintenance of no control state information in the network. In particular, implementation of a multi-destination datagram in the PRNET environment involves little more than developing and implementing addressing and routing schemes, which were the subject of section 5.3. Beyond that, one needs the single destination datagram protocols in the underlying layers.

A suitable node-to-node protocol was described in [37]. That protocol provides a hop-by-hop acknowledgment with each repeater retaining a copy of messages it has switched until it hears the retransmission from the next repeater(s) in the route. For multi-destination packets a bit map can be used to record the hop-by-hop acknowledgments, since in general one to m of the n repeaters within range may be required by the routing algorithm to relay a packet. Such a protocol can provide reasonable reliability through the
retransmission on timeout at each stage. When a time limit or maximum number of retries is exceeded without an acknowledgment, the protocol gives up. A control packet should be returned to the sender advising of the status. The protocol can lose packets through failure of a repeater, loss of its buffer contents and state information. Control packets advising of failure to deliver can likewise become lost. A sender is not guaranteed that packets have been delivered to all destinations even though no failure is reported.

5.4.4. Analysis and Critique

One of the important conclusions from the discussion in the previous subsections is that while virtual circuit networks have some advantages over datagram networks in reliability of data transfer, datagram networks facilitate the use of multi-destination messages. These important points of comparison have not been fully analyzed in the literature.

As pointed out by McQuillan [38], multi-destination addressing is unwieldy in a virtual circuit network. It is both inefficient and difficult to control. He also suggests that multi-destination addressing should be implemented only for datagrams and that subscribers using these services must take responsibility for providing reliable transmission for the end user. We find no reason to disagree with that assessment. We would add that datagram protocols have much less state information and are, therefore, much
more amenable to validation.

For a datagram with many addresses the problem is simply to route it efficiently to the destinations. We will examine the possibility of augmenting the datagram service with reliability in the next section.

5.5. Intermediate-Level Protocols

The previous section considered two extreme cases of multi-destination protocols: connection-free (DG) and connection-oriented (VC). The former (DG) is simple to implement, efficient to operate, but unreliable in data transmission, while the latter (VC) is reliable, but both inefficient and difficult to control. A question naturally arises as to the feasibility of combining the attributes (simple and efficient) of the connection-free protocol with the data transmission reliability of the connection-oriented protocol.

This section describes two such intermediate-level protocols for use in PRNETs which were proposed in [35]. The first, called the Acknowledged Datagram (ADG), is obtained from the DG protocol by augmenting it with an EEACK mechanism. The second, called the Lettergram (LG), is also obtained from the DG protocol by augmenting it with an assembly-disassembly capability. Table 2 lists available functions provided for various types of protocols. This table suggests that an adaptive protocol can be designed to give a user
whatever level of service is needed for his applications.

Table 2: Various Type of Protocol and Available Function

<table>
<thead>
<tr>
<th>LAYERS OF PROTOCOL</th>
<th>AVAILABLE FUNCTION</th>
<th>TYPES OF PROTOCOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport Protocol (Terminal-to-Terminal)</td>
<td></td>
<td>DG</td>
</tr>
<tr>
<td>1) Connect. management</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>2) Flow control</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>3) Sequence control</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>4) Assembly/Disassembly</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>5) EEACK/Retransmission</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Node-to-Node Protocol (Repeater-to-Repeater)</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>1) Multi-destination addressing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2) Multi-destination routing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3) HHACK/Retransmission</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Channel Access Protocol</td>
<td>1) Radio channel access control</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.5.1. Acknowledged Datagram (ADG) Protocols

The Acknowledged Datagram (ADG) protocol is a datagram service augmented with an end-to-end acknowledgment (EEACK) mechanism. This type of protocol provides for best-effort-to-deliver reliability with little overhead. The acknowledgment mechanism uses both HHACK (in the Node-to-Node layer) and EEACK (in the Transport layer) and works as follows:

When a sender transmits a multi-destination message to its
nearby repeater, the EEACK mechanism is activated to receive an EEACK from each destination in the message. This message is stored in the wait-for-EEACK buffer and an EEACK bit-map is also set up to record received EEACK from each destination. If a timeout occurs, the message is retransmitted to only those destinations that have not returned their EEACK, up to a maximum number of retries (which is between 5 and 8 in ALOHANET [18]). After the maximum retries, a status (success or failure) report is given to the sender and the wait-for-EEACK buffer is cleared.

When the (first) nearby repeater receives the message, the HHACK mechanism is activated to receive an HHACK from each of the next-hop repeaters in the route. The message is stored in the wait-for-HHACK buffer and an HHACK bit-map is set up to record received HHACK from each of the next-hop repeaters in the route. In this case, a broadcast by the next-hop repeater can be taken as an HHACK (echo acknowledgment), so that no extra message is generated. Aside from this, the scheme for retransmission on timeout is exactly like EEACK described above. The HHACK mechanism is also activated by each intermediate repeater in the route. The last repeaters, since they will not receive HHACK, all have to listen for the EEACKs transmitted by the destinations as their acknowledgment messages.

The above protocol is reliable and can tell the sender exactly who did or did not receive the message. However, there is a good
deal of overhead associated with this protocol if the number of destinations specified in the message is large. To overcome this problem, we proposed an intended message-control message protocol, which does not use the EEACK mechanism (but does use the HHACK mechanism). However, the latter protocol is not as reliable as the former, since there is a small probability that some destinations could fail to receive both the intended and control message. Further, no provision is made to report failure to the user.

Consider how one might set about to validate the HHACK mechanism of the node-to-node protocol. To begin with, a separate state is required for each possible configuration of the HHACK bit map. If the number of messages in process at a time is greater than one, two or more bit maps must be maintained. Separate states for each combination of bit map configurations means that with \( n \) possible next-hop repeaters and \( m \) possible messages in process the repeater must be modeled with \( 2^{mn} \) separate states. In order to be able to validate the protocol, the designer must model it with a small \( n \) and perhaps allow only one message to be in process. By modeling one repeater transmitting datagrams to \( n \) receivers acknowledging the datagrams, the complexity of the model can be kept within reasonable bounds.
5.5.2. Lettergram (LG) Protocols

The Lettergram (LG) protocol is a datagram service augmented with the capability of disassembling a message larger than the network packet size at the sender and reassembling the packets in correct order prior to delivery at the destinations. Each packet associated with a message will be assigned a serial number at the sender, and will be treated as a separate packet and transmitted into the network using DG service. The number of packets in the message will appear in the header of every packet along with its serial number. At each destination of the multi-destination message, each arriving packet will be stored in a reassembly buffer. The destination can easily detect any missing packets in the message by examining the number of packets and the serial numbers of the packets received, and will ask for retransmission of any missing packets. When all the packets of the message have arrived, it will reassemble them into the original message prior to delivery to the receiver.

Note that this protocol does not use the EEACK mechanism, but does use HHack mechanism as described before. It is relatively reliable, since any retransmission request due to a missing packet in the message serves also as a negative EEACK for that packet. Note also that the intended message-control message protocol proposed in [37] is a special case of this protocol, when the number
of packets in a message is exactly two. The reliability of this protocol increases in proportion to the number of packets in a message, but so does overhead due to retransmission.

To reduce the overhead and to avoid traffic congestion at the sender due to retransmission, the message holder scheme proposed in [37] can be used here, too. This holder scheme works as follows.

A group of holder repeaters are assigned by the station during the initialization process. These repeaters must have a special buffer to store messages sent under the LG protocol. These message holders hold every message of this type in their buffer for a predetermined time interval (the holding period). They honor all requests they receive for retransmission of this message during this holding period. Each terminal in the network is assigned a nearest repeater holder by the station during its initialization process. Any retransmission request for the missing packets in the message will be made first to the nearest message holder if it is in operation; otherwise, the request is made to the original sender of the message.

The above holder scheme is expected to reduce congestion by distributing the nodes which will receive requests for retransmission of the missing packets. There is overhead incurred in the station as well as in the holder repeaters, but it is more desirable in many cases to incur this overhead than to tolerate the
expected congestion at the source.

5.5.3. Adaptive Multi-destination Protocols

So far four types of multi-destination protocols have been considered for use in PRNETs, each offering different capabilities and services as shown in Table 2. Each of these protocol function can be implemented as a program module and stored in the memory of the digital section of a packet radio (see section 5.2. If a modular design is adopted, these four types of protocol can be implemented as shown in Figure 22. The digital section can adaptively execute those parts of functions called for by a particular type of protocol. Such an adaptive protocol combines various features and capabilities to give a user whatever level of service is needed.
Figure 22: Adaptive Multi-Destination Protocol
6. AUTOMATED VALIDATION OF MULTI-DESTINATION PROTOCOLS

In chapter 2 we have discussed the need for protocol specification and validation using formal techniques. Application of state transition techniques and, in particular, the transmission grammar and validation automaton have been described. The actual use of the validation automaton for validation of a communications interface was demonstrated in the third chapter. Application of the VA to a more complex validation task and the techniques used to overcome the difficulties inherent in representation of more complex state information was the topic of the fourth chapter. In the preceding chapter the complexities of multi-destination communications were illustrated via a discussion of multi-destination protocols applicable to a particular environment, that of the packet radio net. The intent of this chapter is to show how the Transmission Grammar (TG) model can be generalized for the design of multi-destination protocols and to describe a computer program designed to automate the validation process. The task of constructing a validation automaton without the support of an
automated validation tool is sufficiently great to eliminate the technique as a useful validation tool for protocols of any complexity. An automated tool for testing whether a transmission grammar model of a protocol is well structured exists [67]. The following sections describe the development and expansion of tools and techniques necessary to convert the generalized Transmission Grammar (TG) to a realistically usable tool.

6.1. Protocol Design and Validation

From a communicating entity's viewpoint, a protocol simply consists of a set of rules that can be used to define the action sequences during the phases of communication. In using a TG model to specify a protocol the designer attempts to envision all possible actions taken by a protocol entity in a given state and to define the new state of the entity following each possible action. These action rules are written down in the form of formal grammar production rules. When action rules have been produced for each possible state of the entity, a Transmission Grammar has been defined which defines the language of all possible legal action sequences the entity can produce. Teng's TG specification system [67] provides an automated tool which the designer can use to evaluate the TG defined.

That tool points out to the designer any undefined states, that
is, any states which result from some action but for which no action rules are defined. It determines that some complete action sequence is possible and lists the final actions possible in the legal action sequences. It detects any states which have been defined, but which are unreachable from the starting state (often due to a typographical error). It can test some string(s) of action sequences the designer thinks are legal to verify whether they are legal in the defined language and likewise check illegal sequences to be sure they are not included in the language. Finally, it can list, for the designer to examine, all possible cycles of action rules beginning from the starting state. This portion of the design process and the help provided by the automated TG evaluation tool required little change to adapt it to multi-destination protocols.

To be certain that the design is valid The designer must proceed to a further step in the design process. That further step is the construction of a Validation Automaton (VA). The Validation Automaton (VA) combines into a global system model the TG models of the protocol entities with models of the communications media connecting the protocol entities. That global model (or VA) is a reachability graph whose nodes are global system states and whose directed arcs connect from one global state to another global state that is the immediate result of exercising one TG action rule of just one protocol entity. The work in validating the X.21 DTE/DCE
interface recommendation (chapter 3) and the TCP connection protocol (chapter 4) showed that manual construction of the VA was an exceedingly time consuming and error prone process. This led to the development of an automated VA construction tool.

Construction of the VA can reveal unexpected events such as the arrival of a message at a protocol entity. Discovery of such events which the designer overlooked in attempting to define all necessary communications actions for the entity, requires the designer to add new action rules (such as a rule to receive the message) to the TG for that entity. If the VA contains nodes (global states) with no exiting arcs, such nodes represent either desired final states of the system or undesirable deadlock possibilities in the system. If deadlock states are discovered, that requires the designer to change the TG specification of one or more protocol entities. After such an iterative process of protocol revision, validation, error correction and re-validation, eventually the designer has a complete, deadlock free protocol specification. Such a process is known as protocol synthesis.

The Validation Automaton state suggested by Teng was a two by two matrix. It could accommodate only two communicating entities. In [67] he extended the concept somewhat to allow user processes to be included in the model by the addition of queues for the service interface. The more general global state model used in chapters 3
and 4 is also a two by two matrix with the entity states included in the matrix and no separate acknowledgment queues (see Figure 8 on page 46). A simple extension to that model makes it suitable for validation of multi-destination protocols. West [72] proposed a model of the global system state for an n-party multi-destination protocol that is an n by n matrix. The n elements on the main diagonal of the matrix are the states of each of the n entities in that global state. The remaining matrix elements are the message queues which model the communication channels. The queue at row i, column j (i ≠ j) models the channel from entity i to entity j. Figure 23 shows how the matrix would represent the global state of a three-party protocol.

```
State.1            Queue.1_2            Queue.1_3
Queue.2_1          State.2             Queue.2_3
Queue.3_1          Queue.3_2           State.3
```

Figure 23: Global State Matrix for Three-Party Protocol

Such a matrix is simply an expansion of the VA state matrix used for single destination protocol validation. Each individual VA state matrix is simple and easy to understand though the number of elements in the matrix grows rapidly as n increases (by the square of n, obviously). The rapid increase in the number of VA states is easy to see from a simple example. Consider a protocol in which the communicating entities may each be in one of ten possible states.
Three distinct messages are defined and up to three of them (with no duplicates) may be queued in each of the communications channels. For the single destination, or two-party, protocol the number of possible VA states can be easily computed as the product of the number of possibilities for each of the four matrix elements. If the message order is significant each of the two message queues has 16 possible content states. The total is $10 \times 10 \times 16 \times 16$, or 25,600 possible states. This is the worst case limit since some entity states may be mutually exclusive and certain entity state combinations may exclude some message queue configurations. If the same sort of protocol is a three-party protocol, \(10^3\) entity state combinations are possible along with \(16^6\) possible configurations of the six message queues for a total of 16,777,216,000. This is, again, an upper limit, but the growth in the number of possible VA states for multi-destination protocols is seen to be quite dramatic. Clearly, it would be unthinkable to attempt VA construction without a computer program. It is just as clear that a VA construction program must be capable of using every technique available to limit the number of states which must be explored in the VA to achieve a useful validation.
6.2. Specific Advances Necessary

The specific work required to extend the Transmission Grammar — Validation Automaton system lies in two areas. The first is the extension of the current program to automate construction of the Validation Automaton (VA). The second is an expansion of the number of known techniques to limit the extent to which a full VA must be constructed. The next sections discuss these two areas of effort in reverse order. Understanding of the requirements for the automated program depends on understanding the ways we can limit the size of the reachability graph which must be constructed.

6.2.1. Controlling Complexity.

Much of what is known about paring the size of the VA we owe to [67]. Teng stated ten rules to reduce the number of states and transitions which must be modeled. Those most significant to this discussion are restated as follows:

a. **Symmetric states:** If the TG specification for the entities are the same (except that destinations for messages, if specified, are opposite) then we need explore the state transitions from only one of two symmetric states in the VA. Two global states are symmetric if the one can be transformed to the other by interchanging the values of the two state elements and interchanging the contents of the two message queues elements.

b. **Insignificance of message sequence:** Under some
circumstances the order of messages within a queue need not be considered significant. For example, a queue with MsgA,MsgB,MsgC would be considered to be the same as a queue containing MsgA,MsgC, MsgB.

  c. **Semantically equivalent messages:** If two or more messages in a channel queue are semantically equivalent, the order in which they are fetched at the receiving site is irrelevant to the validation.

Applying these rules to the construction of Validation Automata for multi-destination protocols can do much to control the size of the state space which must be explored to properly validate the protocol. Some of the rules require additional study in the multi-destination protocol context. How to define symmetric states for an n-party protocol, and whether the symmetric states rule can help when not all n entities are identical are two questions which are open to further study.

A search for additional rules to permit further reduction in the number of states which must be constructed is important to successful validation of complex protocols. Several promising possibilities suggest themselves for research. One such possibility is the idea that not all n entities need be modeled in order to adequately validate an n-party protocol. Suppose that the protocol being investigated were a broadcast protocol with one transmitting
entity and n minus one identical receiving entities. It seems likely that modeling the system as a three-party protocol (or perhaps even as a two-party protocol) could provide results which would be valid for n greater than three. Questions to be answered are: "What if any sorts of protocol errors might be missed by such a validation technique?" and "Under just what circumstances is such a rule valid?" Another possible reduction might be achieved by elimination of certain redundancies in the calculations involved in the global-state generation method of protocol validation. Other researchers (Rubin and West [57]) have suggested that the multiple transition paths leading from one global-state to another be replaced by one canonical sequence of transitions such that the sequence of transitions undergone by each entity separately is the same for all the transition paths. The canonical sequence is proposed between stable states. A stable state is defined as one in which all message queues are empty. Rubin and West suggest that substitution of canonical sequences for completely exercising all possible states of the system has only minor impact on the significance of validation results. It is necessary to determine the applicability of the technique to multi-destination protocols, to explore in greater detail the possible protocol errors which might be overlooked by using canonical sequences, and to determine whether canonical sequences can be applied between global-states
with non-empty message queues.

6.2.2. Automating the VA Construction.

The case for automating the VA construction process is clear cut. For multi-destination protocols a computer program to build the VA is not just important, it is essential. Even with a computer program, however, the need remains to use all available techniques to limit the size of the state space which must be explored. For that reason, the automated VA builder must be carefully designed. It must be a program which can be easily revised to adapt to new techniques as they are developed. It must include the means to recognize or to be advised which rules are applicable to the protocol being validated. It must be adaptable to the validation of protocols involving a varied number of entities (up to some reasonable limit). It must incorporate efficient data structures and efficient algorithms for manipulating those data structures.

One task which must be performed repeatedly during construction of a VA emphasizes many of the needed qualities just pointed out. Each time a new global-state is derived from some predecessor state in the graph, that new state must be added to the state set. Before a new state can be added to the state set a search for a duplicate (or perhaps symmetric) state in the state set must take place. Not only is the comparison of such complex matrices difficult, but also the number of states in the state set through which the search must take place can be quite large. Some representation of the
global-state which would speed the comparison (perhaps a hash function) and some storage structure which would minimize search time (perhaps a hash table) are needed. Following the initial design of the program, discovery of some new rule could make a different storage implementation or search algorithm advisable. The program must be designed in such a way that the remainder of the program be insulated from any changes in the implementation of the data structure for the VA or the search algorithm used. It is also necessary that the criteria for deciding on equality of global states be flexible. If symmetrical states are to be considered equivalent, one decision criterion is needed whereas another is required if symmetry is not to be considered. Likewise, order of messages in a queue may or may not be significant to a determination of equality between queues. On consideration of these requirements, it seems almost imperative that the program be designed using abstract data types.

6.2.3. Data Abstraction and Modifiable Programs

An abstract data type presents an object in terms of the operations defined on that object and the results of those operations. Information regarding the implementation of the object and the algorithms through which the operations are carried out is hidden from the user of the object [24]. A number of programming language developments [36, 53, 73] designed to facilitate data abstraction have sprung from the class construct of SIMULA 67 [16].
However, none of these languages is in widespread use. In order to provide a tool with extensive portability a language with widespread popularity and availability is needed. Pascal was selected for the purpose despite the fact that Pascal has no particular features to facilitate information hiding and data abstraction. Discipline in program development can avoid operating directly on the implementation of the data structure through other than the defined operations. This discipline is necessary because the only abstraction mechanism in PASCAL, the function or procedure, requires that the internal representation be available if some external representation is available. For example, an operation on some abstract object may be defined as a Pascal procedure or function. To invoke that operation, some instance of that object must be referenced. Even if the reference is via a pointer, the structure of the object must be defined externally to the function or procedure in order for the pointer to be available at the level invoking the operation.

Pascal is the implementation language for the TG validation tool which tests for well structured Transmission Grammars. Development of the VA builder in PASCAL has permitted integration of the two tools and made available the parsed structures developed by that original program.
6.3. Organization of the Automated Validation Program

The functions originally performed by Teng's validation program remain intact. Modifications have been made which provide for optionally omitting the testing of action sequences and the production of cycle listings. In both cases the program writes the question to the operator "Do you want action sequence testing (cycle listing)?" A "Y" response produces the function asked about while a "N" omits the function in question. When used for validating the TG for a single protocol entity, the action sequence testing could be omitted by not supplying an action sequence in the input. The original program would simply ignore the action sequence testing if it came to the end of file without reading the action sequence. A second, third, or n-th protocol TG follows the first TG in input to Multi-Valid. If the input contains no action sequence following a TG, the program would attempt to read the next protocol TG as an action sequence.

The production of a cycles listing was found to consume large amounts of computer time and consume large quantities of file space for a complex protocol (see chapter 3). The option to omit that function makes it possible to avoid the use of that time and file space when it is not warranted. Omitting the function also avoids the temptation to consume large amounts of paper printing the file.

The original program reads the input and builds data structures
to represent the TG action rules. Terminal symbols were originally parsed to no deeper level. The VA TG has terminal symbols with a three part structure. The first part of such a symbol is the action. Queue, Dequeue, and Fetch are examples of actions. The action part must be followed by a designation of the queue to be acted upon. The third part of a terminal symbol is the message part to be inserted in or removed from the designated queue. Some actions, such as Empty and Clear, do not require a message parameter. For those actions the third part of the action terminal symbol is omitted. A modification to the data structure used to represent the terminal symbols and an addition to the routines which parse the input, made the structure of action terminal symbols available to the VA building portion of the program. This modification has been added to Teng's original protocol validator.

6.3.1. Data Abstractions Used in Automated Validation Program

The central abstract object used in construction of the global reachability state graph is a structure called the VA. Operations defined on the VA are VA_initialize, VA_add, and VA_get. Because the Pascal block structure and scope rules provide no alternative, the actual data structure which makes up the VA is defined at the highest level of the program and is, therefore, accessible throughout the program. The operations are executed by a call to the procedure VA with the operation name passed as a parameter. The rule against operating on the data structures through other than the
defined operations can be easily violated. This is a serious
shortcoming of Pascal for use in the data abstraction programming
technique.

The VA_initialize operation accepts a global state as the
initial state and creates a VA containing just that one state which
is marked as explored. The term explored is used to indicate that a
global state has been made available for perturbation to develop new
reachable states which may be added to the VA. A global state is
made available by having been established as the initial state or
through the VA_get operation. VA_get returns the value TRUE in a
result parameter if any state in the VA has not been explored. A
result of FALSE is returned if each global state in the VA has been
explored. Along with a TRUE result, a previously unexplored global
state is returned in response to VA_get. The VA_add operation
simply adds a new global state to the VA state set. If the added
state was not previously a member of the state set, then it is not
marked explored and is available to be returned by some future
VA_get operation.

Internal to the VA is a VA_node object. Since the VA_node has
no operations defined, it is modified with primitive Pascal
operations within the VA. The VA_node includes a representation of
a global VA state, an explored marker, and pointers to other
VA_nodes. The structure of a VA_node is shown in Figure 24.
In order to speed searching for duplicate nodes in the VA, each node body is accessible through a hash table. The location within that table is stored as one field in the node. The node body itself is a pointer to a global state which is an array of matrix elements. Matrix elements are either queues or states. The global state is, itself, an abstract object. Operations to create a new global state, to modify a global state by replacing one of the matrix elements at a specified row and column, and to get a copy of the element (state or queue) at a specified row and column, allow manipulation of the global states. Another operation tests whether two global states are equal.

Operations on queues include all of the VA actions listed in section 2.3.4 as well as operations which create a new empty queue,
copy a queue into a new instance with the same content, and sequentially retrieve the messages in a queue returning a FALSE indication when the tail message is retrieved. A further operation which tests the equality of two queue objects provides a true indication when both queues contain identical messages in the same order. Operations are also provided to access the messages at the head and tail of a queue without altering the queue, as contrasted with the dequeue and pop operations which leave the queue one message shorter. These operations provide the tools used in the procedure which expands, or explores, a global state through perturbation to produce possibly new global states to be added to the VA.

A message is an object which can be tested against another object of type message with the operation Msg_equal. The Msg_equal function returns the values TRUE and FALSE. No other operations are defined for messages.

6.3.2. Operation of the Automated Validation Program

A procedure, expand, is defined to conduct the expansion of the state set by perturbation of previously unexplored global states. It uses the VA_add operation to add each new state to the VA. The algorithm to construct the VA can be simply outlined as follows: Create the initial global state; initialize the VA; expand the initial state; get a new, unexplored state; while the result is TRUE — expand the most recently gotten state and get another. When
result is no longer TRUE the process has been completed. The Pascal
instructions which implement this algorithm are the last lines of
the Multi-Valid program listed in appendix B.

The sequence of processing a global state to expand it includes
two major subdivisions of effort. In the first subdivision each of
the queues is examined in turn. The purpose of the examination is
to detect incompleteness. Testing of the queues determines whether
action rules for the state of the destination entity provide for
receipt of the queue's contents. Remember that the queue at row i,
column j contains messages destined for the entity whose state is
represented in the matrix element at row j, column j. Messages at
the head of the queue may be either dequeued or fetched, while
message beyond the head of the queue may be fetched. The procedures
D_match and F_match are used to make the test for, respectively, the
first and subsequent messages in each queue. These checks detect
incomplete specification. Notification of any errors found is
included in the output. The second subdivision of effort proceeds
dentity by entity from row one, column one through the entity in row
n, column n (for an n party protocol). The action rules for the
dentity i in the state indicated by row i, column i are tested for
applicability. Those requiring removal of a message from a queue
are applicable if the message is in the appropriate place in the
queue. Actions requiring insertion of a message into a queue are
always applicable so long as no limit has been placed on queue length and, if such a limit has been specified, the actions are applicable so long as they do not cause the specified limit to be exceeded. Each applicable action rule is applied to the global state being expanded to create a new global state to be added to the VA using the VA_add operation. When the last applicable action for the entity/state represented at row n, column n has been applied, the expansion of that global state is complete. Any global state for which the expansion produces no new states (that is, no action rule was applicable for any entity/state) is identified as a deadlock state.

6.4. Illustrative Example

The clearest way to explain the use of Multi-Valid is through use of a simple example. Figure 25 shows the analysis provided by Multi-Valid for two mirror image entities. The Transmission Grammars specify a protocol similar to that used for illustrative purposes in chapter 2. The entities are referred to as mirror images since they differ only in the directions in which they receive and send messages. The initial production rule (line 1) for the first TG provides for Queuing message "a" in the queue toward entity 2 and for Dequeuing message "a" from the queue from entity 2. The initial production rule (also labeled as line 1) for the second
TG provides for Queueing toward entity 1 and Dequeueing from entity 1. Unlike the example from chapter 2, this protocol does not return to the <start> state. Rather, it terminates in the <end> state with <end> producing the empty symbol represented by the asterisk (see line 6 in both TGs).

A number of points about preparation of the input for Multi-Valid may not be obvious from figure 25. Production alternatives must be separated by commas on input. Multi-Valid replaces the commas on output with the OR symbol (vertical bar, "|"). Each token in the input (non-terminal, terminal, "::=" symbol, comma or period) must be separated from the preceding and following token by a space or newline character. The TGs are separated by a sequence of one or more hyphens (minus signs). Action sequences, if present in the input, are separated by hyphens from the preceding and following TGs. An action sequence consists of the non-terminal from which the sequence is derived followed by the sequence of action (or terminal symbols) to be checked. For this example, an action sequence might be like the following:

```
<start>
D.2.a Q.2.b Q.2.d D.2.e *
```

A period terminates each production rule and the action sequences. At least one space should follow the last separator (hyphen) which terminates the input.

The constant, no_entities, which is defined on the first page
proper termination analysis
******************************************************************************

the terminate symbol is: *
no left recursion

reachability analysis
******************************************************************************

no unreachable nonterminal
al nonterminals can terminate properly.

Figure 25: Multi-Valid Output for Simple Example
of appendix B determines how many TG specifications the Multi-Valid program will attempt to process. The program must be recompiled with a new value of no_entities to change from a two-party protocol, as in this example, to an n-party protocol for n other than two.

The analysis provided for each TG in the example shows the results of termination analysis and reachability analysis. Any sentence in the language defined by these grammars must terminate with the symbol, "*". Each of the non-terminals can be reached by some sequence of productions. The terminal symbol, "*", can be reached from each of the non-terminals by some sequence of productions.

With regard to the local analysis performed on the individual TGs in figure 25, no error was found. That does not mean, however, that the protocols defined are perfect. The important step of global validation remains. The complete output of Multi-Valid showing the results of global validation is shown in appendix C. The construction of the global Validation Automaton (VA) begins with the initial state. Both entities are in the <start> state and the queues are empty. A total of 32 applications of production rules expand existing states. Some of the new states duplicate global states previously added to the VA. Only 24 total states are expanded (including the initial state) which means that eight of the 32 states duplicated previously derived global states. Productions
17 and 18 both produce the same state, for example. That state is the global terminal state for the protocol. Below the production numbered 17, is a state is marked as a deadlocked state. Since both entities have terminated in the empty symbol, "*", no further transitions are possible. In this instance the deadlock represents reaching the designed terminal state rather than an error.

Following the production labeled 4, Multi-Valid identifies an error state containing the message "a" in both queues with no production rules to receive the message. Both entities are in the <asc> state for which the only production rule seeks to Dequeue message "b". Not only is the protocol specification incomplete (in failing to provide for reception of message "a" in state <asc>), but it also leads to deadlock in this particular state. A similar global deadlock is displayed following production number 11.

Following productions 25 and 31, Multi-Valid reports errors while attempting to expand global states containing two messages in the same queue. If the communications channel being modeled permits more than one message in flight between entities and if the channel does not reorder messages, then this does not represent an error. If, however, it is possible for message "d" to arrive before message "b", then the specification is not complete.

It should be clear from this example how a protocol specification can be shown to be incorrect even though the TG
specifications of the individual entities are correct in the local analysis. Though the example is simple, it clearly demonstrates the ability of the Multi-Valid program.

The use of the automated validation program will greatly expand the usefulness of the Validation Automaton technique for validating protocols. It has been designed in such a way that it can be easily modified and adapted as new procedures are found to improve the validation process. The size and complexity of a validation automaton for a fairly complex protocol makes validation, using the techniques described in this thesis, impractical without such an automated program. The program can be used to test existing protocols, and as a tool to design new protocols through successive validation and refinement steps to achieve a protocol which is completely deadlock free and completely specified.
7. SUMMARY AND CONCLUSIONS

7.1. Summary

The task we set out upon was to provide a validation technique which would provide protocol designers a useful tool, a tool which would greatly increase confidence in the correctness of their protocol design. In the introductory chapter we pointed out the extensive growth in the use and importance of computer communications in our society. We emphasized the essentiality of correctly functioning protocols to the proper operation of communication networks. The importance of replacing informal, ad hoc, design methods is born out in the frequency with which errors are discovered in protocols designed by "seat-of-the-pants" methods. The programming language approach to protocol design was discarded because of the difficulty of program proof techniques. The complications for proofs of the correctness of systems of concurrent programs executing in parallel are far more severe than for simple non-parallel programs. The reachability state graph approach to protocol validation was recognized as having its own difficulties.
Nevertheless, we felt that proper automated tools would make possible the validation of complex real world protocols. We also felt that the importance of multi-destination communications warranted extending the capability of any system we developed to multi-party protocols.

In the second chapter, we described the formal grammar on which our specification and validation system is based. We compared the grammar with other state transition based specification techniques -- finite state machines, petri nets, and UCLA Graphs. We showed how these specification techniques could be used in protocol validation using a simple, but not real, protocol as an example.

In the third chapter, we discussed the validation of an actual protocol -- that between the Data Terminal Equipment and Data Circuit-Terminating Equipment. First, we showed that the TG model developed for individual protocol entities need not duplicate, state for state, an original centralized specification. This can result in a much simplified entity specification. The effect of simplification is magnified in the global VA. In addition to demonstrating that the formal grammar could effectively model the X.21 interface and be used as a basis to validate the protocol, that effort produced new techniques for modeling variable information as part of the message content. The problem appeared in the X.21 interface validation in a form so simple it was scarcely recognized.
In X.21, the problem was to represent variable information flowing between the entities. Such things as call selection signals, call progress signals, called line identification, and calling line identification needed to be defined. The exact form of such signals was not important to the validation. The set of signals could be partitioned into classes according to the response specified for the interface. The solution seemed obvious to us — invent a symbol in the grammar to represent each class of signal. More importantly, the X.21 validation demonstrated the importance of automating the construction of the reachability state graph. The amount of effort and the many errors made along the way during manual construction of the reachability state graph for the X.21 protocol were convincing evidence that automation of this process was essential if a useful tool were to be provided to the protocol designer.

A more complex protocol was discussed in chapter 4. The Transmission Control Protocol (TCP) presented a number of difficult challenges to the validator. These challenges in representation were overcome. Techniques were developed to represent the sequence numbers needed in a connection establishment and to represent complex protocol entity states. The sequence number problem is similar to the problem of variables in X.21 signals; an extension of the same approach proved to be effective. Here, each variable could assume many values which were significant to aspects of the protocol.
being validated. A single symbol to represent each variable would not be sufficient. The solution was to identify a range of values over which the variables would range within the context of the specific function of the protocol being modeled. Identification of a symbol to represent some arbitrary initial value in the range and definition of symbols to represent subsequent values through the range solved the problem.

In TCP, the state of each entity depended on other variables in addition to the state variable. A state naming convention was adopted in which the non-terminal symbols of the grammar were made up of a concatenation of those variables on which the total state depended. Representation of the TCB variables followed the procedure described in the preceding paragraph, while the state variable values were represented by state names taken from the DOD TCP standard.

The technique adopted to represent states of a TCP entity with out-of-sequence segments queued for later processing was to use the existing non-terminal representation with a code for the particular pending TCP segment catenated at the right.

The time consuming effort involved in generating the set of global states reachable during a TCP connection establishment reiterated the need for automation of that process. The errors found in this protocol specification also reemphasized the need for
formal specification and validation of communications protocols as well as demonstrating that the reachability state graph can be used to validate a real world protocol of considerable complexity. In other words, the problem of state space explosion does not render the technique useless.

In chapter 5 we discussed the usefulness of multi-destination protocols. In particular, we looked at the advantages of a packet radio net with its partial broadcast transmission for delivery of multi-destination packets. A number of addressing and routing techniques for use in such an environment were discussed. The protocols for multi-destination communications were seen to involve considerable complexity. Even so, that complexity could be controlled by the divide and conquer approach of structuring the protocol architecture into layers. Even the adaptive multi-destination protocol which incorporated the features of several levels of reliability was a layered design which served to subdivide the protocol.

Finally, chapter 6 described the automated validation tool through which an effective protocol validation system is made available. That automated tool eliminates the arduous effort involved in manual attempts at protocol validation. It also eliminates the human errors which are an inherent part of manual protocol validation. The structure and design of the Multi-Valid
program are described as are the reasons for using an abstract data
type programming paradigm. Aside from providing a validation tool
for protocol validation, the Multi-Valid program extends the
technique to multi-destination protocols.

7.2. Areas for Future Research

The most significant area of research which remains is the
investigation of additional techniques to limit the size of the
global VA. We have mentioned some possible approaches. With regard
to multi-destination protocols, it is not practical to include all
entities in a global VA for an n-party protocol where n is large.
Just as useful results can be obtained while limiting the number of
messages in the queues, we feel that by limiting the number of
entities included in the model we can make it practical to validate
multi-destination protocols. The extent to which possible protocol
 specification errors might be overlooked by such a practice is an
appropriate area for future investigation.

The technique of constructing non-terminal symbols in the
Transmission Grammar (TG) by concatenation of variable value
representations is effective. It is usually necessary, however, to
discover the entire set of such non-terminal symbols during the
process of constructing the global VA. As a previously unspecified
(in the TG) event is reported by Multi-Valid, a new grammar
production rule and new next state (non-terminal symbol) must be added to the grammar. It would be useful to be able to specify message fields in terms of the current variable values and to specify new variable values in terms of current values and message fields. Such a specification system would probably incorporate programming language like constructs into the grammar. Extensive revision of Multi-Valid would be necessary to incorporate such constructs. If automated validation could be retained under such a combined system, it would have a clear advantage. The specification process would be much easier. If a similar solution could be found to the problem of accounting for messages queued within an entity for delayed processing it would have similar advantages.

Though not strictly research items, some refinements to the Multi-Valid program are proposed. Multi-Valid does not currently identify symmetric states in the VA. Such a capability should be added and made an option for use when specified by the user. Another option should allow the user to specify that queues with identical contents be considered equivalent without regard to order of messages in the queues. While other refinements are possible, these two will most help Multi-Valid to use known techniques to control global VA size.
7.3. Conclusion

The specification and validation of protocols is not a simple task. The choice of specification technique has a great effect on the sort of details of the protocol which can be easily and efficiently modeled. The extent to which a specification represents only essential elements of the protocol and does not include implementation specific details is also greatly influenced by the kind of model selected. The Transmission Grammar and other state transition based models have certain restrictions. The difficulty in modeling variables such as sequence numbers or time out variables is one such restriction. The validation of the Transmission Control Protocol (TCP) discussed in chapter 4 showed that useful results could be obtained in validating a protocol having such variables. Nevertheless, the Transmission Grammar model constructed for the TCP falls far short of being satisfactory as a specification for the protocol. One of the purposes of a specification is to provide a clear and unambiguous definition from which implementations may be constructed. The limitations of the TG prevented specification of many details of the protocol which would be essential for implementation. The TG constructed covered only the establishment and termination of TCP connections. To model the data transfer function would have required modeling windows for flow control, and a much broader sequence number space than the three sequence number
values symbolically represented in the TG model. It seems unlikely that a way could be found around this problem in a purely grammar representation of the TCP protocol.

Development of the Multi-Valid program has extended the usefulness of the Transmission Grammar (TG) as a validation tool. By converting global VA construction from a tedious, error prone manual task to an efficient, automated process, we have converted the TG from a theoretically interesting validation techniques to a practically useful one. The manual construction of the global VA for TCP connection establishment required months of effort. With the Multi-Valid program, the time required can be measured in seconds. The Multi-Valid program will follow the specification of the TG accurately. The frequent errors which are a part of the manual construction and the endless cross checking to discover and correct such errors are a thing of the past.

The TG is no longer an untested technique. It has been demonstrated to be effective on two real world protocols. Though limitations remain, there is no longer reason to doubt that useful validation results can be obtained for complex protocol modeling problems.

A number of useful techniques have been developed to solve various problems arising in the process of protocol validation. Section 7.1 highlighted these techniques with reference to the
chapters in which they were presented. To reiterate briefly, the techniques accomplish the following things:

- Simplification of entity state diagrams derived from centralized specification.
- Representation of variable information in message content.
- Representation of variable information as part of the entity state.
- Incorporation of out-of-sequence message receipt in the entity state representation.
- Validation of multi-party protocols.

In the final analysis, the techniques just itemized permit the development of a TG model which is not of real use for protocol validation without the automated validation tool. Multi-Valid provides the tool needed to make the TG/VA model useful. It can be used to analyze existing protocols and to assist in synthesizing corrections for any errors found. Multi-Valid should also be valuable to the protocol designer. No longer must the protocol designer attempt to visualize each possible event and specify the appropriate response. He can now use Multi-Valid to find the unspecified events in a partially completed protocol specification. The designer can add responses to such events, run Multi-Valid again and continue the process with newly discovered errors. In the end, Multi-Valid will process the completed specification without reporting any errors. The designer will be able to rest with considerable confidence that he has completely specified the actions
of the protocol and that those actions will not lead to deadlock. That high level of confidence represents the achievement of the goal of this research effort.
APPENDIX A. GRAMMAR PRODUCTION RULES, X.21

<1_1> ::= 1.OFF_1.OFF <2_1> , 1.OFF_1.OFF <1_8> , 1.OFF_1.OFF <1_18> , 1.OFF_1.OFF <21_1> , 1.OFF_1.OFF <14_1> .
<2_1> ::= 0.ON_1.OFF <2_3> , 0.ON_1.OFF <2_8> , 0.ON_1.OFF <2_18> , 0.ON_1.OFF <16_1> .
<2_3> ::= 0.ON_+.OFF <4_3> , 0.ON_+.OFF <16_3> , 0.ON_+.OFF <2_19> .
<4_4> ::= Sel.ON_+OFF <4_4> , Sel.ON_+OFF <5_3> , Sel.ON_+OFF <16_3> , Sel.ON_+OFF <4_19> .
<5_4> ::= 1.ON_+.OFF <5_6A> , 1.ON_+.OFF <16_4> , 1.ON_+.OFF <5_19> .
<5_6A> ::= 1.ON_SYN.0FF <6A_6A> , 1.ON_SYN.0FF <5_7> , 1.ON_SYN.0FF <16_6A> , 1.ON_SYN.0FF <5_19> .
<6A_6A> ::= 1.ON_SYN.0FF <6A_7> , 1.ON_SYN.0FF <17_6A> , 1.ON_SYN.0FF <6A_19> .
<6A_7> ::= 1.ON_Prog.OFF <7_7> , 1.ON_Prog.OFF <6A_6B> , 1.ON_Prog.OFF <16_7> , 1.ON_Prog.OFF <6A_19> .
<7_7> ::= 1.ON_Prog.OFF <7_6B> , 1.ON_Prog.OFF <16_7> , 1.ON_Prog.OFF <7_19> .
<7_6B> ::= 1.ON_SYN.0FF <6B_5B> , 1.ON_SYN.0FF <7_10> , 1.ON_SYN.0FF <16_6B> , 1.ON_SYN.0FF <7_19> .
<6B_5B> ::= 1.ON_SYN.0FF <6B_10> , 1.ON_SYN.0FF <16_6B> , 1.ON_SYN.0FF <6B_19> .
1.0N_Id.OFF error.
<6A_6B> ::= 1.0N_SYN.OFF error.
<5_7> ::= 1.0N_Prog.OFF error.
<2_8> ::= 0.ON_BEL.OFF <2_4> , 0.ON_BEL.OFF <16_8> ,
0.ON_BEL.OFF <2_19> .
<16_3> ::= 0.OFF.+.OFF <16_17> , 0.OFF.+.OFF <16_21> ,
0.OFF.+.OFF <16_19> .
<16_19> ::= 0.OFF_+.OFF <16_21> .
<2_4> ::= 0.ON_+.OFF <4_4> , 0.ON_+.OFF <16_4> ,
0.ON_+.OFF <2_19> .
<21_8> ::= 0.OFF_BEL.OFF <l_8> , 0.OFF_BEL.OFF <16_17> ,
0.OFF_BEL.OFF <21_21> .
<1_8> ::= 1.OFF_BEL.OFF <9_8> , 1.OFF_BEL.OFF <21_17> ,
1.OFF_BEL.OFF <l_17> , 1.OFF_BEL.OFF <14_18> ,
1.OFF_BEL.OFF <1_19> .
<9_8> ::= 1.ON_BEL.OFF <9_10bis> , 1.ON_BEL.OFF <16_8> ,
1.ON_BEL.OFF <9_19> .
<9_10bis> ::= 1.ON_CId.OFF <10bis_10bis> , 1.ON_CId.OFF
<9_6C> , 1.ON_CId.OFF <16_10bis> ,
1.ON_CId.OFF <9_19> .
<10bis_10bis> ::= 1.ON_CId.OFF <10bis_6C> , 1.ON_CId.OFF
<16_10bis> ,
1.ON_CId.OFF <10bis_19> .
<10bis_6C> ::= 1.ON_SYN.OFF <6C_6C> , 1.ON_SYN.OFF
<10bis_11> , 1.ON_SYN.OFF <16_6C> ,
1.ON_SYN.OFF <10bis_19> .
<10bis_11> ::= 1.ON_1.OFF error.
<10bis_19> ::= 1.ON_0.OFF <20_19> .
<20_19> ::= 0.OFF_0.OFF <20_21> .
<20_21> ::= 0.OFF_1.OFF <l_21> .
<16_10bis> ::= 0.OFF-CId.OFF <16_17> , 0.OFF-CId.OFF
<16_21> , 0.OFF-CId.OFF <16_19> ,
0.OFF-CId.OFF <16_6C> .
<16_8> ::= 0.OFF_BEL.OFF <16_19> , 0.OFF_BEL.OFF <16_21> .
<9_19> ::= 1.ON_0.OFF <20_19> , 1.ON_0.OFF <16_19> .
<1_19> ::= 1.OFF_0.OFF error.
<21_17> ::= 0.OFF_0.OFF <21_21> .
<1_18> ::= 1.OFF_0.OFF <1_18> , 1.OFF_0.OFF <2_18> ,
1.OFF_0.OFF <1_19> , 1.OFF_0.OFF <14_18> ,
1.OFF_0.OFF <21_18> .
<18_18> ::= 1.OFF_0.OFF <18_1> .
<18_1> ::= 1.OFF_1.OFF <1_19> , 1.OFF_1.OFF <18_8> .
<18_8> ::= 1.OFF_BEL.OFF error.
<2_18> ::= 0.ON_0.OFF <2_18> , 0.ON_0.OFF <16_18> ,
0.ON_0.OFF <20_18> .
<16_18> ::= 0.OFF_0.OFF <16_17> , 0.OFF_0.OFF <16_21> .
<2_19> ::= 0.ON_0.OFF <20_19> .
<20_19> ::= 0.OFF_0.OFF <21_20> .
<21_20> ::= 0.OFF_1.OFF <21_21>.
<21_21> ::= 0.OFF_1.OFF <1_21>.
<1_21> ::= 1.OFF_1.OFF <1_1>, 1.OFF_1.OFF <2_21>.
<2_21> ::= 0.ON_1.OFF error.
<20_18> ::= 0.OFF_0.OFF error.
<9_6C> ::= 1.ON_SYN.OFF error.
<16_4> ::= 0.off.+off <16_17>, 0.off.+off <16_21>,
0.off.+off <16_19>.
<2_19> ::= 0.ON_0.OFF <20_19>, 0.ON_0.OFF <16_19>.
<4_19> ::= Sel.ON_0.OFF <20_19>, Sel.ON_0.OFF <5_19>,
Sel.ON_0.OFF <16_19>.
<5_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<16_6A> ::= 0.OFF_SYN.OFF <16_17>, 0.OFF_SYN.OFF <16_21>,
0.OFF_SYN.OFF <16_19>, 0.OFF_SYN.OFF <16_7>.
<16_7> ::= 0.OFF_Prog.OFF <16_17>, 0.OFF_Prog.OFF <16_21>,
0.OFF_Prog.OFF <16_6B>, 0.OFF_Prog.OFF <16_19>.
<6A_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<7_19> ::= 1.ON_0.OFF <20_19>,
1.ON_0.OFF <16_19>.
<16_6B> ::= 0.OFF_SYN.OFF <16_17>, 0.OFF_SYN.OFF <16_21>,
0.OFF_SYN.OFF <16_19>.
<16_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<16_10> ::= 0.OFF_Cid.OFF <16_17>, 0.OFF_Cid.OFF <16_21>,
0.OFF_Cid.OFF <16_19>.
<16_6C> ::= 0.OFF_SYN.OFF <16_17>, 0.OFF_SYN.OFF <16_21>,
0.OFF_SYN.OFF <16_19>.
<10_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<6C_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<16_11> ::= 0.OFF_1.OFF <16_17>, 0.OFF_1.OFF <16_21>,
0.OFF_1.OFF <16_19>.
<11_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<16_12> ::= 0.OFF_1.ON <16_17>, 0.OFF_1.ON <16_21>,
0.OFF_1.ON <16_19>.
<12_19> ::= 1.ON_0.OFF <20_19>, 1.ON_0.OFF <16_19>.
<13_19> ::= data.ON_0.OFF <20_19>, data.ON_0.OFF <16_19>.
<14_18> ::= 01.OFF_0.OFF error.
<16_1> ::= 0.OFF_1.OFF <16_21>.
APPENDIX B. VALIDATION PROGRAM FOR MULTI-DESTINATION PROTOCOLS

PROGRAM Validate(input,output);

LABEL 99;

CONST
maxprtln = 120;  { max print line }
maxheader = 120;  { max number of nonterminals }
max__entities = 9;  { will process transmission grammars for no
more }
{ than max__entities. }
maxtop = 121;  { max pointer to next header }
prime = 997;  { prime number for hashing table size }
free = -1;
no__entities = 2;  { how many protocol entities are modeled in VA }
va_prime = 1993;

TYPE
va_INDEX = 0..VA_prime;
err_type = (nofetch);  { Input to the VA_Error routine to flag }
{ appropriate message }
tokentype = (terminal,nonterminal,alternate,period,
separator,repeati,repeatlf,empty,equal);
tblrange = 0..maxheader;
tg_no = 1..no__entities;
action_type = (state,cc,e,nn,qq,d,f,pp,o);
alpha = PACKED ARRAY[1..20] OF char;
pointer = "node;
queue_type = tg_no;
q_rec_type = RECORD q_part: queue_type;
message: alpha
END;
tsym_type = RECORD symb: alpha
CASE action: action_type OF
  state: (term_state:alpha);
  cc,e,nn: (queue: queue_type);


qq,d,f,pp: (q_rec: q_rec_type)

END;

node = RECORD
  suc, alt: pointer;
CASE terminal: boolean OF
true: (tsym: tsym_type);
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;
q_type = ^q_element;
matrix_element = RECORD
  CASE state: boolean OF
true: (state_el: tblrange);
false: (queue_el: q_type);
END;

q_element = RECORD
  head,next,tail: q_type;
  msg: alpha;
END;

va_call = (va_add,va_get,va_initialize);
va_st_matrix = ARRAY[1..no_entities,1..no_entities] OF
matrix_element;
va_st_ref = ^va_st_matrix;
va_node_pointer = ^va_node;
va_node = RECORD
  node_hash: 0..va_prime;
  link,next,parent,l_child,r_sibling:
va_node_pointer;
  explored: boolean;
  va_node_body: va_st_ref;
END;

VAR
  VA_hash_tbl: PACKED ARRAY[va_index] OF VA_node_pointer;
  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 }
PROCEDURE Prtchr;

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 prtlncptr DO
BEGIN
{the following are a part of the original program as listed in Teng's Phd dissertation. They give incorrect output so I am commenting them out.
L David Umbaugh (6/23/81).}
{ if prtln[i] in ['a'..'z']
then
begin
ascii := ord(prtln[i]) + 32;
prtln[i] := chr(ascii);
end;
}
Write(prtljn[i]:1);
END;
WriteLn;
prtlncptr := 0;
prtlncfg := false;
prtlncnum := true;
END;

{ buffer(prtljn) 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 maxprtlnc. }

prtlncptr := prtlncptr + 1;
IF chars = ' '
THEN prtljn[prtlncptr] := Chr(124)
ELSE prtljn[prtlncptr] := chars;
IF inpfg
THEN
  IF Eoln(input)
  THEN prtlncfg := true;
IF prtlncptr = maxprtlnc
THEN
BEGIN
prtlncfg := true;
prtlncnum := false;
END;
END; PROCEDURE Readxchr;
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[tg_entity,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[tg_entity,nl].entry;
LOOP
n2 := n2 - 1;
EXIT IF (n2 = 0);
a := a^n.alt;
END;
REPEAT
IF a^terminal
THEN sym := a^tsym.symb
ELSE sym := k[rg_entity,a^nsym].sym;
a := a^suc;
Prtsym;
UNTIL (a = NIL);
prtlng := true;
chars := fill;
Prtchr;
END;

PROCEDURE Cycles;
VAR
levelfig: boolean; { flag to mark start level of prtpath }
used : ARRAY [0..maxheader] OF boolean;
currptr: RECORD { pointer to next node in the trace of
current rule }
  alt: 1..10;
suc: pointer
END;
rulptr: { ptrs for current traces of the rules }
ARRAY[0..maxheader] OF RECORD
  alt: 1..10;
suc:pointer
END;
i,k1: integer;
initptr: pointer; { ptr to the initial entry of current
rule }
n,root:tblrange;
path: ARRAY[0..maxtop] OF RECORD
  rule: tblrange;
  alt: 1..10
END;
startlevel,level: 0..maxtop;

PROCEDURE Prtpath;
VAR
  node: 0..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>0) DO
    BEGIN { output indentation }
      Prtchr; Prtchr; Prtchr;
  END;
END;

...
indent := indent -1;
END;
Prtrule;
END;
chars := fill;
prtlnfg := true;
Prtchr;
END; { prtpath }
BEGIN
{ indented cycles listing }
Writeln;
Writeln(’indented cycles listing’);
Writeln(’*******************’);
Writeln;
leveflfg := true; { initialize flag }
n := topheader - 1; { n is the number of nonterminals }
FOR root := 0 TO n DO
BEGIN { see if root can reach itself }
IF withinall[root,root]
THEN
BEGIN { root can reach itself }
BEGIN { initialize tree for root }
level := 0;
initptr := k[tg_entity,root].entry;
path[0].rule := root;
path[0].alt := 1;
i := root;
FORE kl := root TO n DO
BEGIN
used[kl] := false;
BEGIN { initialize repeptr to the start of the first alt }
ruleptr[kl].suc := k[tg_entity,kl].entry;
ruleptr[kl].alt := 1;
END;
END;
currptr := ruleptr[i];
REPEAT { create tree for all the paths from root }
j := -1;
WHILE ((currptr.alt=saltcnt[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;
currrptr.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 (withinall[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[tg_entity,j].entry; { extend
new entry

```plaintext
level := level +1;
path[level].rule := j;
currprtr := ruleptr[j];
i := j;
END;
END;

IF (currprtr.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[tg_entity,i].entry; { backtrack old entry }
    currprtr := ruleptr[i];
  END;
  END;
UNTIL ((j==1) AND (level == 0)
  AND (currprtr.suc=NIL));
{ end of generating all cycles from root }
used[root] := true; { mark used }
END;
{ try next nonterminal as the root }
{ end of main loop }
END; { cycles }
```

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;
Readnchr;
END;
END; \{ Getterm \}
BEGIN
WHILE (chars = ' ') AND NOT(Eof(input)) DO
BEGIN
Readnchr;
END;
END; { Getterm }
BEGIN
WHILE (chars = ' ') AND NOT(Eof(input)) DO
BEGIN
Readnchr;
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;
  Readnchr;
  END;
  END;
  '*':
  BEGIN
  sym := '*';
symbol_type := empty;
  END;
  ':':
  BEGIN
  symbol_type := empty;
  END;
  ':':
  BEGIN
  symbol_type := alternate;
  END;
  ':':
  BEGIN
  symbol_type := repeatlf;
  END;
  ':':
  BEGIN
  symbol_type := repeatri;
  END;
  BEGIN
  WHILE (chars = ' -') DO Readnchr;
symbol_type := separator;
END;
symbol_type period; "": Getterm;
OTHERS: Getterm
END;
Readnxchr;
END;
END { getsym };

PROCEDURE Find(s: alphabet; VAR h: integer);
{ 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,tg_entity] = -1
  THEN
    BEGIN { insert }
      found := true;
      hash[h1,tg_entity] := topheader;
      h := topheader;
      k[tg_entity,h].sym := s;
      k[tg_entity,h].entry := NIL;
      topheader := topheader + 1;
    END
  ELSE
    BEGIN { collision }
      h1 := h1 + d;
      d := d + 2;
      IF h1 >= prime
      THEN h1 := h1 - prime;
      IF d = prime
      THEN
        BEGIN
          Writeln(" **** table overflow !!");
          ok := false;
        END
    END
  END
IF (k[tg_entity,hash[h1,tg_entity]].sym = s)
THEN
  BEGIN
    found := true;
h := hash[h1,tg_entity];
  END
ELSE
  BEGIN { collision }
h1 := h1 + d;
d := d + 2;
IF h1 >= prime
THEN h1 := h1 - prime;
IF d = prime
THEN
  BEGIN
    Writeln(" **** table overflow !!");
    ok := false;
  END
END
E N D

E N D

U N T I L  f o u n d ;
E N D ;  \{  F i n d  \}  P R O C E D U R E  E r r o r ;
B E G I N
W r i t e l n ;
W r i t e l n  \{  ' i n c o r r e c t  s y n t a x '  \} ;  o k  : =  f a l s e
E N D ( e r r o r ) ;  P R O C E D U R E  T e r m ( V A R  p , q , r : p o i n t e r ) ;
V A R
  a , b , c : p o i n t e r ;
P R O C E D U R E  F a c t o r ( V A R  p , q : p o i n t e r ) ;
V A R
  a , b : p o i n t e r ;  h : t b l r a n g e ;
P R O C E D U R E  P a r s e _ t e r m ( V A R  a : p o i n t e r ; s y m : s y m b o l ) ;
B E G I N
  a ^ . t s y m . s y m b  :=  s y m ;
  I F  s y m [ 2 ]  = = ' . '  \{  A  p e r i o d  s e p a r a t e s  a c t i o n  c o d e  f r o m
  q u e e
  n o t a  t e r m i n a l  s t a t e . \}  \}
  T H E N
B E G I N \{  T e r m i n a l  s t a t e  \}
  a ^ . t s y m . a c t i o n  :=  s t a t e ;
  a ^ . t s y m . t e r m _ s t a t e  :=  s y m ;
E N D
  E L S E \{  S o m e  a c t i o n  o n  a  q u e u e  \}
  C A S E  s y m [ 1 ]  O F
    ' C ' :  B E G I N  a ^ . t s y m . a c t i o n  :=  c c ;  \{  N o t e :  W i t h  t h i s
    r a t h e r  \}
    a ^ . t s y m . q u e u e  :=  O r d ( s y m [ 3 ] ) - 4 8 ;  \{  i n e l e g e n t
    t e c h n i q u e  \}  \}
  E N D ;  \{  f o r  p a r s i n g  q u e u e  n u m b e r s  \}
    ' E ' :  B E G I N \{  t h e  n u m b e r  o f  e n t i t i e s  \}
    a ^ . t s y m . a c t i o n  :=  e ;  \{  m o d e l e d  i s  r e s t r i c t e d
    t h a n  t e n . \}  \}
    a ^ . t s y m . q u e u e  :=  O r d ( s y m [ 3 ] ) - 4 8 ;  \{  t o  l e s s
  E N D ;
    ' N ' :  B E G I N  a ^ . t s y m . a c t i o n  :=  n n ;
    a ^ . t s y m . q u e u e  :=  O r d ( s y m [ 3 ] ) - 4 8 ;
  E N D ;
    ' Q ' :  B E G I N  a ^ . t s y m . a c t i o n  :=  q q ;
    a ^ . t s y m . q _ r e c _ q _ p a r t  :=  O r d ( s y m [ 3 ] ) - 4 8 ;
  E N D ;
    ' D ' :  B E G I N  a ^ . t s y m . a c t i o n  :=  d ;
    a ^ . t s y m . q _ r e c _ q _ p a r t  :=  O r d ( s y m [ 3 ] ) - 4 8 ;
  E N D ;
    ' F ' :  B E G I N  a ^ . t s y m . a c t i o n  :=  f ;
    a ^ . t s y m . q _ r e c _ q _ p a r t  :=  O r d ( s y m [ 3 ] ) - 4 8 ;
'P': BEGIN a^ .tsym .action := pp;
   a^ .tsym .q_rec .q_part := Ord(sym[3])-48;
END;

'O': BEGIN a^ .tsym .action := o;
   a^ .tsym .q_rec .q_part := Ord(sym[3])-48;
END;

{ CASE }
CASE a^ .tsym .action OF
   qq,d,f,pp,o:
BEGIN
   a^ .tsym .q_rec .message := ' ';
   FOR i := 5 TO 20
   DO a^ .tsym .q_rec .message[i] := sym[i];
END;
END;
{ Parse_Term };
BEGIN { factor }
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^ .sym := '*'
      ELSE Parse_term(a,sym);
      END;
      firstflag := false;
p := a; q := a; Getsym
   END
ELSE
   IF symbol_type = repeatlf
   THEN
   BEGIN
   END
GETSYM; TERM(p,a,b); b^suc := p; New(b);

b^terminal := true;
b^tsym.symb := '*';
a^alt := b; q := b;
IF symbol_type = repeatri THEN Getsym
ELSE Error
END
ELSE Error
END {factor};
BEGIN {term}
Factor(p,a); q := a;
WHILE symbol_type IN [terminal, nonterminal, repeatl, 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.symb <> terminate[t]) THEN
BEGIN
  t := t + 1; {change from original progr. see next
  comment.}
  terminate[t] := c^tsym.symb;
  {Teng's original program initialized t to "1" and incremented after the pre-
   ceding assignment. I want to test against the old value since otherwise,
   terminate[t] is always undefined, and the comparison fails.}
  {
    t := t + 1;
  }
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[currsym] := altcnt[currsym] + 1;
  IF (lastflag[h])
  THEN
    IF (c\^~.terminal)
    THEN
      BEGIN
        IF (c\^~.tsym.symb <> terminate[t])
        THEN
          BEGIN
            t := t + 1;
            terminate[t] := c\^~.tsym.symb;
            \{ commenting out the post incrementation of "t" as above.\}
            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[tg_entity,goal].entry;
    REPEAT
      IF s\^~.terminal
      THEN
        BEGIN
          IF s\^~.tsym.symb = sym
          THEN
            BEGIN
              match := true; Getsym
            END
            ELSE match := (s\^~.tsym.symb = '""')
          END
        ELSE Parse(s\^~.nsym, match);
        IF match
        THEN s := s\^~.suc
ELSE s := s^alt
UNTIL s ~ NIL
END (parse); FUNCTION Queue_create: q_type;
VAR q : q_type;
BEGIN
New(q);
WITH q^ DO
BEGIN
head := q; { The head of the queue must point to itself}
next := NIL;
tail := NIL; { nil tail indicates empty WQueue}
Queue_create := q;
END { WITH q DO}
END; { Queue_create}

FUNCTION Msg_equal(msg1,msg2:alpha): boolean;
VAR test: boolean;
count: 0..20;
BEGIN
count := 0;
REPEAT count := count + 1;
test := msg1[count] = msg2[count];
UNTIL ((count = 20) OR NOT test);
Msg_equal := test; { return with result of test}
END; { Msg_equal}

PROCEDURE Queue_deq(VAR qu: q_type; VAR
out_msg: alpha; VAR result: boolean);
BEGIN
result := qu^.tail <> NIL;
IF result
THEN
BEGIN
out_msg := qu^.msg;
IF qu^.next = NIL
THEN
qu^.tail := NIL
ELSE
BEGIN
qu := qu^.next;
Dispose(qu^.head);
END;
END;
END; { Queue_deq}
FUNCTION Queue_equal(queue1, queue2: q_type): boolean;
VAR result1, result2, equal: boolean;
msg1, msg2: alpha;
BEGIN
Queue_deq(queue1, msg1, result1);
Queue_deq(queue2, msg2, result2);
WHILE (equal AND result1 AND result2) DO
BEGIN
  equal := Msg_equal(msg1, msg2);
  Queue_deq(queue1, msg1, result1);
  Queue_deq(queue2, msg2, result2);
END;
IF equal
  THEN equal := result1 = result2;
Queue_equal := equal; { return with result of test }
END; { Queue_equal}

PROCEDURE Queue_head(qu: q_type; VAR out_msg: alpha; VAR result: boolean);
{ returns the message at the head of the queue
  without altering the queue as contrasted with Deq. }
{ If queue is empty, result is false on return
  otherwise result is true.}
BEGIN
  IF qu^.tail = NIL { queue is empty}
  THEN
THEN
BEGIN
result := false;
out_msg := ' ';
END
ELSE
BEGIN
out_msg := qu^.msg;
result := true;
END
END; { Queue_head}

PROCEDURE Queue_tail(qu: q_type; VAR out_msg: alpha; VAR result: boolean);
{ returns the element at the end of the queue 
  without altering the queue. Result is false on return if the 
  queue is empty. }
BEGIN
result := qu^.tail <> NIL;
IF result
THEN
  out_msg := qu^.tail^.msg
ELSE
  out_msg := ' ';
END; { Queue_tail}

FUNCTION Queue_next(VAR qu: q_type; VAR out_msg: alpha): boolean;
VAR result: boolean;
{ Returns the message at Qu and on return, 
  Qu points to the next message in the queue. Thus, if Qu is 
  the 
  head of a queue on first invocation, successive invocations 
  return successive elements in the queue. Result is true on 
  return until the tail is returned on which invocation 
  result is 
  false on return and Qu is again the head of the queue. }
BEGIN
result := NOT (qu = qu^.tail);
IF result
THEN
BEGIN
  IF qu^.tail = NIL
  THEN
    BEGIN out_msg := ' ';
      result := false;
      END
    END
  END
END
ELSE
BEGIN
out_msg := qu^.msg;
qu := qu^.next;
END
ELSE { not result, ie. tail of queue.}
IF qu^.tail = nil
THEN
out_msg := nul_msg (* In case the queue was empty. *)
ELSE
BEGIN
out_msg := qu^.msg;
qu := qu^.head;
END;
Queue_next := result; { return the result. }
END; { Queue_next}

FUNCTION Queue_empty(qu: q_type): boolean;
BEGIN
Queue_empty := qu^.tail = nil; { true if empty queue}
END; { Queue_empty}

FUNCTION Queue_non(qu: q_type): boolean;
BEGIN
Queue_non := qu^.tail <> nil; { true if not empty queue}
END; { Queue_non}

FUNCTION Queue_clear(qu: q_type): q_type;
{ returns an empty queue}
BEGIN
IF qu^.tail = nil
THEN { qu is already empty}
Queue_clear := qu
ELSE
BEGIN
qu^.tail^.next := qu;
qu := qu^.tail;
WHILE qu <> qu^.tail DO WITH qu DO
BEGIN
head := next;
next := next^.next;;
Dispose(head);
END;
qu^.head := qu;
qu^.tail := nil;
qu^.next := nil;
Queue_clear := qu;
FUNCTION Queue_fetch(VAR qu: q_type; f_msg: alpha): boolean;
{ If f_msg is in the queue, returns true with
  f_msg removed from queue Qu on return.)
VAR q_hold: q_type;
  test_msg: alpha;
  m_result, q_result: boolean;
BEGIN
  q_hold := qu;
  q_result := Queue_non(qu); { can't fetch from an empty queue}
  IF q_result
  THEN
    BEGIN
      m_result := false;
      q_result := Queue_next(q_hold, test_msg);
      WHILE q_result DO
        BEGIN
          m_result := Msg_equal(f_msg, test_msg);
          IF NOT m_result
          THEN
            q_result := Queue_next(q_hold, test_msg)
          ELSE
            q_result := false;
          END;
      IF m_result
      THEN
        Queue_fetch := true
      ELSE
        Queue_fetch := Msg_equal(f_msg, test_msg)
    END
  ELSE
    Queue_fetch := false
END; { queue_fetch}

FUNCTION Queue_push(qu: q_type; p_msg: alpha): q_type;
{ adds p_msg at the HEAD of the queue and
  returns the resulting queue.}
VAR new_head: q_type;
BEGIN
  IF qu^.tail = NIL
  THEN
    BEGIN
      qu^.tail := qu;
      qu^.msg := p_msg;
      qu^.next := NIL;
      END

ELSE
BEGIN
new_head := Queue_create;
new_head^.msg := qu^.msg;
new_head^.tail := qu^.tail;
new_head^.next := qu^.next;
new_head^.head := qu;
qu^.next := new_head;
qu^.msg := p_msg;
END;
Queue_push := qu
END; { queue_fetch }

FUNCTION Queue_pop(VAR qu: q_type; VAR out_msg: alpha): boolean;
{ Returns the message at the tail of the queue
and on return, Qu is that message shorter, unless Qu was empty.
Result is false on return if Qu was empty. }
VAR temp_q: q_type;
result: boolean;
BEGIN
result := qu^.tail <> NIL;
IF result AND (qu^.tail <> qu)
THEN
BEGIN
temp_q := qu;
WHILE temp_q^.next <> qu^.tail DO
  temp_q := temp_q^.next;
qu^.tail := temp_q;
temp_q := qu^.next;
WHILE temp_q <> qu^.tail DO
  BEGIN
    temp_q^.tail := qu^.tail;
temp_q := temp_q^.next;
  END;
  temp_q^.tail := temp_q;
out_msg := temp_q^.next^.msg;
Dispose(temp_q^.next);
temp_q^.next := NIL;
END
ELSE
IF qu^.tail = qu
THEN
BEGIN
out_msg := qu^.msg;
qu^.tail := NIL;
END
ELSE out_msg := '';
Queue_pop := result;
END; { Queue_Pop}

FUNCTION Queue_add(qu: q_type; a_msg: alpha): q_type;
{ Inserts A_msg at the tail of the queue and
returns the new, longer queue.)
VAR new_q: q_type;
    extra_q: q_type;
BEGIN
    IF qu^ .tail = NIL
    THEN
        BEGIN
            qu^ .tail := qu;
            qu^ .msg := a_msg;
            qu^ .next := NIL;
        END
    ELSE
        BEGIN
            New(new_q);
            new_q^ .tail := new_q;
            new_q^ .next := NIL;
            new_q^ .head := qu;
            new_q^ .msg := a_msg;
            qu^ .tail^.next := new_q; (* link the new one into the end *)
            WHILE qu <> new_q DO
                BEGIN
                    qu^ .tail := new_q;
                    qu := qu^ .next;
                END;
            qu := qu^ .head;
        END;
    END;
    queue_add := qu
END; { Queue_Add}

PROCEDURE Va_error(err: err_type; tg_state: tblrange; tg_entity: tg_no;
    msg: alpha);
BEGIN
    CASE err OF
        nofetch: Writeln(\ error *** encountered message ', msg,
            ' while in state ',
            k[tg_entity,tg_state].sym,
            ' No receive for the message.\');
        OTHERS: Writeln(\ error *** Undefined error.\)
    END { CASE}
END; { VA_error}

FUNCTION F_match(tg_state: tblrange; row, col: tg_no; msg: alpha;
VAR
  to_state:tblrange): boolean;
VAR link,n_link: pointer;
  match: boolean;
BEGIN
match := false;
link := k[row,tg_state].entry;
REPEAT
  IF link^ .terminal
  THEN WITH link^ DO
  BEGIN
    n_link := link;
    IF tsym .action = f
    THEN
      IF tsym .q_rec .q_part = col
      THEN
        match := Msg_equal(msg,tsym .q_rec .message);
      END;
    IF NOT match
    THEN
      link := link^ .alt;
  UNTIL (n_link^ .alt = NIL) OR match;
F_match := match;
IF match
THEN
  BEGIN WITH link^ DO
  WHILE suc <> NIL DO
    BEGIN
      IF NOT suc^ .terminal
      THEN
        to_state := suc^ .nsym;
        suc := suc^ .suc;
    END;
  END;
END; { F_match }

FUNCTION D_match(tg_state: tblrange; row, col: tg_no; msg: alpha;
  to_state:tblrange): boolean;
VAR link,n_link: pointer;
  match: boolean;
BEGIN
match := false;
link := k[row,tg_state].entry;
REPEAT
  IF link^ .terminal
  THEN WITH link^ DO

BEGIN
  n_link := link;
  IF tsym.action = d THEN
    IF tsym.q_rec.q_part = col THEN
      match := Msg_equal(msg,tsym.q_rec.message);
    END;
  END;
  IF NOT match THEN
    link := link^alt;
  UNTIL (n_link^alt = • NIL) OR match;
D_match := match;
  IF match THEN
    BEGIN WITH link^ DO
      WHILE suc <> NIL DO
        BEGIN
          IF NOT suc^terminal THEN
            to_state := suc^nsym;
            suc := suc^suc;
          END;
        END;
      END;
    END; { D_match}
  END; {Va_st_get}

PROCEDURE Va_st_get(in_state:va_st_ref;row:integer;VAR out_elem: tblrange);
BEGIN
  out_elem := in_state^[row,row].state_el;
END; {VA_ST_Get}

PROCEDURE Va_st_qget(in_state:va_st_ref;row,col:integer;VAR out_elem:q_type);
BEGIN
  IF row <> col THEN out_elem := in_state^[row,col].queue_el ELSE out_elem := Queue_create;
END; {VA_ST_Qget}

PROCEDURE Va_st_modify(VAR in_state:va_st_ref;row,col:integer;
  state_in: tblrange; queue_in:q_type);
BEGIN
  IF row = col THEN in_state^[row,col].state_el := state_in
ELSE in_state~[row,col].queue_el := queue_in;
END; { VA_ST_modify }

PROCEDURE Va_st_create(no_entities: integer;VAR c_state:va_st_ref);
VAR row,col: tg_no;
BEGIN

New(c_state);
col := 1;
FOR row := 1 TO no_entities DO
BEGIN
WHILE col < row DO
BEGIN
  c_state~[row,col].state := false;
  c_state~[row,col].queue_el := Queue_create;
  col := col + 1;
END {WHILE col < row};

  c_state~[row,col].state := true; { .state_el remains undefined }

END { FOR row = 1 }
END { VA-ST_Create }; PROCEDURE Va_st_equal(no_entities: integer;state1,state2:va_st_ref;
result: boolean);
VAR
element1,element2: q_type;
index,index2: tg_no;
nulmsg: alpha;
BEGIN
result := true;
FOR index := 1 TO NO_entities DO
  IF result
  THEN
    result := state1~[index,index].state_el = state2~[index,index].state_el;
FOR index := 1 TO NO_entitites
  DO
    IF result
    THEN
      FOR index2 := 1 TO NO_entitites DO
        IF result
        THEN
          IF index <> index2
          THEN
            BEGIN
              Va_st_qget(state1,index,index2,element1);
              Va_st_qget(state2,index,index2,element2);
              result := Queue_equal(element1,element2);
            END;
          END;
      END {VA_ST_EQUAL};
PROCEDURE Va(VAR va_state: va_st_ref; VAR static_node: va_node_pointer;
operation: va_call; VAR result: boolean); FORWARD;

PROCEDURE Expand(va_state: va_st_ref);
  VAR t_hold,hold_queue,expand_queue: q_type;
  transit_state,expand_state: tblrange;
  abort,change_test,q_result,result: boolean;
  exp_element: alpha;
  new_state: va_st_ref;
  prod_link,successor: pointer;
BEGIN
  abort := FALSE;
  FOR va_st_row := 1 TO no_entities DO
    BEGIN
      Va_st_qget(va_state,va_st_row,expand_state);
      FOR va_st_col := 1 TO no_entities DO
        BEGIN
          IF va_st_row = va_st_col THEN { null statement }
        ELSE
        BEGIN
          Va_st_qget(va_state, va_st_row, va_st_col,
                      expand_queue);
          hold_queue := expand_queue;
          q_result := Queue_next(hold_queue,exp_element);  
          { Get message at head of queue }
          IF q_result OR NOT msg_equal(exp_element,nul_msg)
          THEN
            BEGIN
              IF D_match(expand_state, va_st_row, va_st_col,
                          exp_element,transit_state)
OR
F_match(expand_state, va_st_row, va_st_col,
ex_element, transit_state)

THEN
BEGIN
Va_st_create(no_entities,new_state);
new_state* := va_state*;
Va_st_modify(new_state,va_st_row,va_st_col,
expand_state,hold_queue);
Va_st_modify(new_state,va_st_row,va_st_row,
transit_state,hold_queue);
Va(new_state,static_node,va_add,result);
END
ELSE
Va_error(nofetch, expand_state, va_st_col,
ex_element);

END;
{ IF q_result }

WHILE q_result DO
BEGIN
q_result := Queue_next(hold_queue,exp_element);
IF
F_match(expand_state,va_st_row,va_st_col,
ex_element,transit_state)

THEN
BEGIN
Va_st_create(no_entities,new_state);
new_state* := va_state*;
t_hold := expand_queue;
q_result := Queue_fetch(t_hold,exp_element);
Va_st_modify(new_state,va_st_row,va_st_col,
transit_state,t_hold);
Va_st_modify(new_state,va_st_row,va_st_row,
transit_state,t_hold);
Va(new_state,static_node,va_add,result);
END
ELSE
Va_error(nofetch,expand_state,va_st_col,
ex_element);

END { WHILE };
END; { ELSE }

END; { FOR VA_ST_COL }

{ Incoming messages have all been checked for Appropriate receive actions }

{ Now exercise all send transitions for the state. }
prod_link := k[va_st_row,expand_state].entry;
WHILE prod_link <> NIL DO
    BEGIN
        sucessor := prod_link;
        change_test := false;
        VA_ST_create(NO_entitles,new_state);
        new_state^ := VA_state^;

        REPEAT
            IF sucessor^.terminal THEN
                CASE sucessor^.tsym.action OF
                cc: BEGIN
                    expand_queue := Queue_create;
                    Va_st__modify(new_state,va_st_row,
                    sucessor^.tsym.queue,
                    expand_state, expand_queue);
                    change_test := true;
                    END;
                e,nn: BEGIN
                    Va_st_qget(new_state, va_st_row,
                    sucessor^.tsym.queue,
                    abort := Queue_empty(expand_queue);
                    IF sucessor^.tsym.action = e THEN abort := NOT abort;
                    END;
                qq: BEGIN
                    Va_st_qget(new_state, va_st_row,
                    sucessor^.tsym.q_rec.q_part,
                    expand_queue);
                    hold_queue := Queue_add(expand_queue,
                    sucessor^.tsym.q_rec.message);
                    Va_st__modify(new_state, va_st_row,
                    sucessor^.tsym.q_rec.q_part,
                    expand_state, hold_queue);
                    change_test := true;
                    END;
                pp: BEGIN
                    Va_st_qget(new_state, va_st_row,
                    sucessor^.tsym.q_rec.q_part,
                    expand_queue);
                    hold_queue := Queue_push(hold_queue,
                    sucessor^.tsym.q_rec.message);
                    Va_st__modify(new_state,va_st_row,
successor^tsym^q_rec^q_part,
expand_state, hold_queue);
   change_test := true;
END;
OTHERS: { Should already have handled D and F }
   END { CASE Tsym.action }
ELSE { NOT successor^terminal }
   IF change_test Then
      Va_st_modify(new_state, va_st_row, va_st_row,
         successor^nsym, expand_queue);
      successor := successor^suc;
   UNTIL
      (successor = NIL) OR abort; { one alternate production
   processed }
      IF change_test AND NOT abort
      THEN
         Va(new_state, static_node, va_add, result); (* The VA_node
      created by this production alternative has been added *) (* to the
         VA *)
            prod_link := prod_link^.alt; { To process the next
      alternate production. }
      END; { WHILE Prod_Link not nil }
      END; { FOR VA_ST_row }
END { Expand };)
PROCEDURE Va;
{ Maintain and implement the Validation Automaton }
{ Accepts States to be added (adds if the new state is unique),
   gets }
{ unexplored state for exploration and returns true result with
   unexplored }
{ VA state unless no unexplored states remain, then returns
   false result. }
{ Operations are VA_ADD, VA_GET, and VA_initialize. }
CONST
   va_prime = 1993;
VAR
   temp_node, current_node: va_node_pointer;
   state: matrix_element;
   kk: va_index;
FUNCTION Va_find(va_state: va_st_ref; VAR node:
   va_node_pointer): boolean;
   VAR result, va_found: boolean;
      node_tmp: va_node_pointer;
FUNCTION Va_hash (va_state: va_st_ref): va_index;
   VAR state: tbrange;
      hash_count: va_index;
row, col: integer;
BEGIN (* VA_hash *)
hash_count := 0;
FOR row := 1 TO no_entities DO
BEGIN
    Va_st_get(va_state, row, state);
    hash_count := state + hash_count * maxtop;
END;
Va_hash := hash_count MOD va_prime;
END;
BEGIN (* VA_FIND *)
va_found := false;
kk := Va_hash(va_state);
IF va_hash_tbl[kk] = NIL THEN
BEGIN
    New(node);
    va_hash_tbl[kk] := node;
    WITH node DO BEGIN
        va_node_body := va_state;
        node_hash := kk;
        link := nil;
        next := nil;
        parent := nil;
        l_child := nil;
        r_sibling := nil;
        explored := FALSE;
    END;
END;
ELSE
BEGIN
    node := va_hash_tbl[kk];
    WHILE (node <> NIL) AND NOT va_found DO BEGIN
        Va_st_equal(no_entities, va_state,
                    node^va_node_body, va_found);
        IF NOT va_found THEN
            BEGIN
                node_tmp := node;
                node := node^link;
            END;
    END;
    IF node = NIL THEN BEGIN
        New(node);
        node_tmp^link := node;
WITH node^ DO BEGIN
    va_node_body := va_state;
    node_hash := kk;
    link := NIL;
    next := nil;
    parent := nil;
    l_child := nil;
    r_sibling := nil;
    explored := FALSE;
END;

node^.node_hash := kk;
END;

END;
Va_find := va_found;
END; { VA_FIND }
BEGIN ( VA )
result := false;
CASE operation OF
  va_add: IF Va_find(va_state, current_node)
    { TRUE if VA_State matches an preexistant state in VA }
    THEN result := true { The add was successful }
    ELSE BEGIN { Link the current node into the VA }
      current_node^.parent := static_node;
      IF static_node^.l_child =NIL
      THEN BEGIN
        static_node^.l_child := current_node;
        current_node^.next := static_node^.next;
        static_node^.next := current_node;
      END
      ELSE BEGIN
        temp_node := static_node^.l_child;
        WHILE temp_node^.r_sibling <> NIL DO
          temp_node := temp_node^.r_sibling;
        temp_node^.r_sibling := current_node;
        current_node^.next := temp_node^.next;
        WITH current_node DO BEGIN
          l_child := NIL;
          r_sibling := NIL;
        END;
        temp_node^.next := current_node;
      END;
  END;
va_get:
IF static_node^.next = NIL
THEN result := false
ELSE BEGIN
    static_node := static_node^.next;
    static_node^.explored := true;
    va_state := static_node^.va_node_body;
    result := true;
END;

va_initialize: { Put Starting State at root of VA }
BEGIN
    FOR kk := 0 TO VA_prime DO
        VA_hash_tbl[kk] := NIL; (* INITIALIZE *)

    result := Va_find(va_state, static_node);
    WITH static_node DO
        BEGIN
            link := NIL; next := NIL; parent := NIL;
            l_child := NIL; r_sibling := NIL;
            explored := true;
        END;
    END; { WITH DO }
END; { VA_initialize }
END; { CASE operation }
END; { VA }

BEGIN { productions }

{ initialization } nul_msg := ""; inpfg := true; FOR j := 1 TO prime DO
    FOR tg_entity := 1 TO no_entities DO
        hash[j,tg_entity] := -1;

{ special characters for nice output } lf := Chr(10); fill := Chr(127); cr := Chr(13); alt := Chr(124); Readnxchr; FOR tg_entity := 1 TO no_entities DO
    BEGIN
        prtlncut := 0;
        prtlng := false;
        prtlnum := true;
        prtlmptr := 0;

        topheader := 0;
        Getsym;
        t := 0;

        { initialize lastflag for terminate trace }
lastflag[0] := true;
FOR i := 1 TO maxheader DO
    lastflag[i] := false;

{ initialize matrixes }
FOR i := 0 TO maxheader DO
    FOR j := 0 TO maxheader DO
        first[i,j] := false;
    within := first;

WHILE symbol_type <> separator DO
    BEGIN { construct data structure for transmission
            grammar rules. The data structure 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[tg_entity,h].entry,p); p.alt := NIL;
    IF symbol_type <> period
    THEN Error;
    Getsym;
    END; { WHILE <> separator }

{ start protocol validation }
h := 0; ok := true; { check whether all symbols are
    defined }
WHILE h <> topheader DO
    BEGIN
        IF k[tg_entity,h].entry = NIL
        THEN BEGIN
            writeln('undefined symbol --> ',
                    k[tg_entity,h].sym);
            ok := false
            END; { WHILE <> topheader }
        h := h + 1;
        END;

{ proper termination analysis }
writeln;
writeln('proper termination analysis');
writeln('**************************');
WRITELN;
IF (t > 1)
THEN
  WRITELN('** warning ** more than one',
            'exists');

WRITELN;
WRITE('the terminate symbol is: ');
FOR i := 1 TO t DO
  WRITE(terminate[i]:20);
WRITELN;

n := topheader - 1;
{ relation matrix within and first are constructed during
  parsing for efficiency, we construct transitive closure
  of relations first and within }
firstall := first; withinall := within;
FOR i := 0 TO n DO
  FOR j := 0 TO n DO
    BEGIN
      IF firstall [j,i]
      THEN
        FOR k1 := 0 TO n DO
          firstall [j,k1] := firstall [j,k1] OR firstall [i,k1];
        IF withinall [j,i]
        THEN
          FOR k1 := 0 TO n DO
            withinall [j,k1] := withinall [j,k1] OR withinall [i,k1];
        END;
      END;
{ left recursion checking }
FOR i := 0 TO n DO
  IF firstall [i,i]
  THEN
    BEGIN
      WRITELN('left recursion -->',tg_entity,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[0,1])
THEN
BEGIN
WriteLn('unreachable nonterminal --> ',
k[tg_entity,i].sym:20);
ok := false;
END;

IF (ok)
THEN WriteLn('no unreachable nonterminal');

{ check for <u> := t for some t in vt+ }
FOR i := 0 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 := 0 TO n DO
  IF NOT(mark[i])
  THEN
    BEGIN
      p := k[tg_entity,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;
        END;
      IF NOT(stop)
      THEN
        BEGIN
          mark[i] := true;
          newmark := true;
        END;
    END;
  stop := false;
i := 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 := 0 TO n DO
        IF NOT (mark[i])
            THEN
                Writeln(' the nonterminal',k[tg_entity,i].sym:20,
                          ' can not terminate properly');
        Writeln;
END;
IF NOT(ok)
    THEN GOTO 99;
writeln(tty,'Do you want action sequence testing? Y or N.');
readln(tty); read(tty,chars);
IF chars = 'y'
    THEN
        BEGIN
            writeln;
            writeln(' action sequence testing');
            writeln( ' ***********************');
            writeln;
            { goal symbol}
            getsym; find(sym, h);
            { sentences}
            getsym;
            WHILE (symbol_type <> separator) AND 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;
end;
IF (ok) THEN
BEGIN
  write(tty,'Do you want cycles listing? Y or N.');
  readln(tty); read(tty,chars);
  IF chars = 'y' THEN
    Cycles;
  { list all the cycles }
END;
END { FOR tg_entity DO };

{ Initialize the VA by setting up the starting state and passing it to the VA } { with the initialize operation. }

va_st_create(no_entities,va_state); FOR va_st_row := 1 TO no_entities DO
  va_st_modify(va_state,va_st_row,va_st_row,0,empty_q);
va(va_state,static_node,va_initialize,result); Expand(va_state);
va(va_state,static_node,va_get,result); WHILE result DO BEGIN
  Expand(va_state);
  va(va_state,static_node,va_get,result);
END; 99; END.
APPENDIX C. MULTI-VALID OUTPUT FOR SIMPLE TWO-PARTY PROTOCOL

<start> ::= Q.2.a <asc> | D.2.a <arec>.
<asc> ::= D.2.b <ascab>.
<ascab> ::= Q.2.d <desc> | D.2.d <drec>.
<desc> ::= D.2.e <end>.
<drec> ::= Q.2.e <end>.
<end> ::= *.
<arec> ::= Q.2.b <ascab>.

proper termination analysis
********************************************************************************

the terminate symbol is: *
no left recursion

reachability analysis
********************************************************************************

no unreachable nonterminal
all nonterminals can terminate properly.

<start> ::= Q.1.a <asc> | D.1.a <arec>.
<asc> ::= D.1.b <ascab>.
<ascab> ::= Q.1.d <desc> | D.1.d <drec>.
<desc> ::= D.1.e <end>.
<drec> ::= Q.1.e <end>.
<end> ::= *.
<arec> ::= Q.1.b <ascab>.

proper termination analysis
********************************************************************************

the terminate symbol is: *
no left recursion

reachability analysis

******************************************************************************

no unreachable nonterminal
all nonterminals can terminate properly.

<start>  empty
empty  <start>
expanding this state

<asc>  a
empty  <start>
This state added
1   <start>  ::= Q.2.a  <asc>

<start>  empty
a  <asc>
This state added
2   <start>  ::= Q.1.a  <asc>

<asc>  a
empty  <start>
expanding this state

<asc>  a
a  <asc>
This state added
3   <start>  ::= Q.1.a  <asc>

<asc>  empty
empty  <arec>
This state added
4   <start>  ::= D.1.a  <arec>

<asc>  a
a  <asc>
expanding this state
Error *** encountered message a
while in state <asc>    No receive for the message.
Error *** encountered message a
while in state <asc>    No receive for the message.
****************************************************************************** DEADLOCKED STATE *********************
****************************************************************************** NO TRANSITIONS OUT OF ABOVE STATE *****
empty <asc> empty
empty <arec>
expanding this state

<asc> empty
b <ascab>
This state added
5 <arec> ::= Q.1.b <ascab>

<asc> empty
b <ascab>
expanding this state

<ascab> empty
empty <ascab>
This state added
6 <asc> ::= D.2.b <ascab>

<asc> empty
b d <desc>
This state added
7 <ascab> ::= Q.1.d <desc>

<ascab> empty
empty <ascab>
expanding this state

<desc> d
empty <ascab>
This state added
8 <ascab> ::= Q.2.d <desc>

<ascab> empty
d <desc>
This state added
9 <ascab> ::= Q.1.d <desc>

<desc> d
empty <ascab>
expanding this state

<desc> d
d <desc>
This state added
10 <ascab> ::= Q.1.d <desc>

<desc> empty
empty <drec>
This state added
11  <ascab> ::= D.1.d <drec>
    <desc> d
    d <desc>
    expanding this state
  Error *** encountered message d
  while in state <desc>  No receive for the message.
  Error *** encountered message d
  while in state <desc>  No receive for the message.
  **************** DEADLOCKED STATE ****************
  **************** NO TRANSITIONS OUT OF ABOVE STATE *****

    <desc> empty
    empty <drec>
    expanding this state

    <desc> empty
    e <end>
  This state added
12  <drec> ::= Q.1.e <end>

    <desc> empty
    e <end>
    expanding this state

    <end> empty
    empty <end>
  This state added
13  <desc> ::= D.2.e <end>

    <desc> empty
    e *
  This state added
14  <end> ::= *

    <end> empty
    empty <end>
    expanding this state

    * empty
    empty <end>
  This state added
15  <end> ::= *

    <end> empty
    empty *
This state added
16 \<end> ::= *

* empty
empty \<end>
expanding this state

* empty
empty *
This state added
17 \<end> ::= *

* empty
empty *
expanding this state

*************** DEADLOCKED STATE ***************

*************** NO TRANSITIONS OUT OF ABOVE STATE *****

\<end> empty
empty *
expanding this state

* empty
empty *
This state added
18 \<end> ::= *

\<desc> empty
e *
expanding this state

\<end> empty
empty *
This state added
19 \<desc> ::= D.2.e \<end>

\<ascab> empty
d \<desc>
expanding this state

\<desc> d
d \<desc>
This state added
20 \<ascab> ::= Q.2.d \<desc>

\<drec> empty
empty \<desc>
This state added
21  <ascab> ::= D.2.d <drec>
    <drec> empty
    empty <desc>
    expanding this state

    <end> e
    empty <desc>
This state added
22  <drec> ::= Q.2.e <end>
    <end> e
    empty <desc>
    expanding this state

    * e
    empty <desc>
This state added
23  <end> ::= *
    <end> empty
    empty <end>
This state added
24  <desc> ::= D.1.e <end>

    * e
    empty <desc>
    expanding this state

    * empty
    empty <end>
This state added
25  <desc> ::= D.1.e <end>

    <asc> empty
    b d <desc>
    expanding this state
Error *** encountered message d
while in state <asc>. No receive for the message.

    <ascab> empty
    d <desc>
This state added
26  <asc> ::= D.2.b <ascab>

    <start> empty
    a <asc>
expanding this state

<asc> a
a <asc>

This state added
27 <start> ::= Q.2.a <asc>

<arec> empty
empty <asc>

This state added
28 <start> ::= D.2.a <arec>

<arec> empty
empty <asc>

expanding this state

<ascab> b
empty <asc>

This state added
29 <arec> ::= Q.2.b <ascab>

<ascab> b
empty <asc>

expanding this state

<desc> b d
empty <asc>

This state added
30 <ascab> ::= Q.2.d <desc>

<ascab> empty
empty <ascab>

This state added
31 <asc> ::= D.1.b <ascab>

<desc> b d
empty <asc>

expanding this state

Error *** encountered message d
while in state <asc> No receive for the message.

<desc> d
empty <ascab>

This state added
32 <asc> ::= D.1.b <ascab>
8. BIBLIOGRAPHY


30 Hopcroft, John E. and Jeffry Ulman *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Massachusetts, 1969.


34 Krogdah, Stein "Verification of a Class of Link-level Protocols", *BIT* 18, 1978, 436-448.


61 Stenning, Norman V. "A Data Transfer Protocol", Computer Networks 1, 2, September 1978, 99-110.


