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.
Lin, Huai-An

A NEW METHODOLOGY FOR DESIGNING COMMUNICATION PROTOCOLS

The Ohio State University

University Microfilms International

Copyright 1983

by

Lin, Huai-An

All Rights Reserved
A NEW METHODOLOGY FOR DESIGNING COMMUNICATION PROTOCOLS

DISSERTATION

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

By
Huai-An Lin, B.S.E.E., M.S.

* * * * *

The Ohio State University

1983

Reading Committee: Approved By
Ming T. Liu
Jerome Rothstein
William F. Ogden

Adviser
Department of Computer
and Information Science
This dissertation is dedicated to my wife, Shih-Gee, with grateful thanks for her love and support over the years.
ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my graduate advisor, Professor Ming T. Liu for his guidance and encouragement during the many stages of this research.

A debt of gratitude is also owed to Professor Jerome Rothstein, Professor William F. Ogden and Professor Needham Soundararajan for their many helpful suggestions and comments on this dissertation. Thanks are also due to Charles J. Graff, who has been a source of continuing support.

I also gratefully acknowledge Dr. L. David Umbaugh, Wael Hilal Bahaa-El-Din, Ching S. Lu, Nien C. Liu and Ian Y. Chou for sharing their ideas and commenting on mine.

The work reported in this dissertation was supported in part by U.S. Army CECOM, CENCOMS, Fort Monmouth, New Jersey under Contract Nos. DAAK81-K-0104 and DAAB07-83-K-K542. 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

January 20, 1955 . . . . Born - Taipei, Taiwan, Republic of China

1977 . . . . . . . . B.S.E.E., Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

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

1980-1981 . . . . Graduate Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1981 . . . . . . . . M.S., The Ohio State University, Columbus, Ohio

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


Lin, Huai-An, Ming T. Liu and Charles J. Graff, "Verification of a Methodology for Designing Reliable Communication Protocols," Proceedings, the 8th Data Communications Symposium, October 1983 (to appear)

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Architecture and Computer Networking
Professor Ming T. Liu

Studies in Theoretical Computer Science
Professor Jerome Rothstein

Studies in Computer Programming Languages
Professor William F. Ogden
# TABLE OF CONTENTS

**DEDICATION** ................................................. ii  
**ACKNOWLEDGMENTS** .......................................... iii  
**VITA** ....................................................... iv  
**LIST OF TABLES** ............................................. ix  
**LIST OF FIGURES** ........................................... x  

**Chapter**  

I. **INTRODUCTION** ........................................ 1  

| Formal Techniques in Protocol | Specification and Verification | 3  
| Objectives and Main Results | 5  
| Organization of Dissertation | 10  

II. **COMMUNICATION PROTOCOLS: THEIR ARCHITECTURE,**  
**SPECIFICATION AND VERIFICATION** ...................... 13  

| Protocol Architecture | 14  
| Formal Techniques in Communication | Protocol Design | 20  
| Specification | 22  
| Verification | 23  

| Current Approaches to Formal Specification and Verification | 25  
| Finite State Automaton Models | 25  
| Formal Grammar Models | 32  
| Petri Net Models | 37  
| Programming Language Models | 38  
| Summary | 44  

III. **PRIORITY DRIVEN COMMUNICATION PROTOCOL DESIGN** . . . 46  

| A Generalized Validation Technique | 48  
| FSA Protocol Specification Model | 51  
| Enhanced FSA Protocol Specification Model | 60  

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intention Functions</td>
<td>62</td>
</tr>
<tr>
<td>Consistency</td>
<td>65</td>
</tr>
<tr>
<td>Reachability Analysis</td>
<td>67</td>
</tr>
<tr>
<td>Reachability Analysis Algorithm</td>
<td>69</td>
</tr>
<tr>
<td>Validating General Correctness Properties</td>
<td>71</td>
</tr>
<tr>
<td>Comparison of Validation Complexity</td>
<td>76</td>
</tr>
<tr>
<td>Relationship between Desirable Interaction Graph and the Individual Processes</td>
<td>78</td>
</tr>
<tr>
<td>Collisions</td>
<td>83</td>
</tr>
<tr>
<td>Synchronization Mechanism</td>
<td>87</td>
</tr>
<tr>
<td>Collision Detection</td>
<td>88</td>
</tr>
<tr>
<td>Collision Resolution</td>
<td>92</td>
</tr>
<tr>
<td>Application of PDCPD to X.21 Recommendation</td>
<td>96</td>
</tr>
<tr>
<td>The Specification of X.21 Recommendation</td>
<td>97</td>
</tr>
<tr>
<td>The Validation Result</td>
<td>101</td>
</tr>
<tr>
<td>Desirable Interaction Graph of X.21 Recommendation</td>
<td>102</td>
</tr>
<tr>
<td>Collision Resolution in X.21 Recommendation</td>
<td>106</td>
</tr>
<tr>
<td>Summary</td>
<td>111</td>
</tr>
<tr>
<td>IV. PROTOCOL SEMANTICS AND VERIFICATION OF CONCURRENT PROGRAMS</td>
<td>113</td>
</tr>
<tr>
<td>Programming Language Models for Protocol</td>
<td></td>
</tr>
<tr>
<td>Semantics: Principle and Practice</td>
<td>114</td>
</tr>
<tr>
<td>The Notation of CSP</td>
<td>118</td>
</tr>
<tr>
<td>Simple Commands</td>
<td>119</td>
</tr>
<tr>
<td>Structured Commands</td>
<td>122</td>
</tr>
<tr>
<td>Differences from Hoare's CSP</td>
<td>124</td>
</tr>
<tr>
<td>Current Verification Systems for CSP</td>
<td>125</td>
</tr>
<tr>
<td>Axiomatic System of Levin and Gries</td>
<td>127</td>
</tr>
<tr>
<td>Axiomatic System of Apt, Francez and DeRoever</td>
<td>129</td>
</tr>
<tr>
<td>Axiomatic System of Soundararajan</td>
<td>130</td>
</tr>
<tr>
<td>Proving CSP Programs Using</td>
<td></td>
</tr>
<tr>
<td>Input and Output Assertions</td>
<td>133</td>
</tr>
<tr>
<td>Reducing Complexity in Proving CSP Programs</td>
<td>135</td>
</tr>
<tr>
<td>Semantics of the Individual Processes</td>
<td>141</td>
</tr>
<tr>
<td>Rule for Parallel Composition</td>
<td>144</td>
</tr>
<tr>
<td>Other Useful Rules of Inference</td>
<td>151</td>
</tr>
<tr>
<td>Case Studies</td>
<td>152</td>
</tr>
<tr>
<td>Summary</td>
<td>166</td>
</tr>
<tr>
<td>V. FORMAL SPECIFICATION AND VERIFICATION OF PDCPD</td>
<td>168</td>
</tr>
<tr>
<td>Notations</td>
<td>169</td>
</tr>
<tr>
<td>Formal Specification of PDCPD</td>
<td>180</td>
</tr>
<tr>
<td>Formal Verification of PDCPD</td>
<td>194</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Comparison of Verification Complexity</td>
<td>7</td>
</tr>
<tr>
<td>2. Simple Commands of CSP</td>
<td>119</td>
</tr>
<tr>
<td>3. Structured Commands of CSP</td>
<td>122</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>Layered Protocol Organization</td>
<td>16</td>
</tr>
<tr>
<td>2.</td>
<td>The Seven-layer ISO Reference Model for Communication Protocols</td>
<td>19</td>
</tr>
<tr>
<td>4.</td>
<td>Global State Space of Alternating Bit Protocol Generated by Reachability Analysis Under the Empty Medium Abstraction</td>
<td>31</td>
</tr>
<tr>
<td>7.</td>
<td>Global State Space of Alternating Bit Protocol Generated by Reachability Analysis in Petri Net Model</td>
<td>40</td>
</tr>
<tr>
<td>10.</td>
<td>Validation of the Protocol in Figure 9 Based on State Perturbation Technique</td>
<td>56</td>
</tr>
<tr>
<td>11.</td>
<td>An Improved Connection Establishment Protocol</td>
<td>58</td>
</tr>
<tr>
<td>12.</td>
<td>Ambiguity in Figure 11</td>
<td>59</td>
</tr>
<tr>
<td>14.</td>
<td>Elements of $T_{@,1}$, $T_{@,2}$, $R_{@,1}$, $R_{@,2}$, $I_{@,1,2}$ and $I_{@,2,1}$, Where $@$ Is the Protocol Given in Figure 9</td>
<td>63</td>
</tr>
</tbody>
</table>
15. Elements of $T_{@,1}$, $T_{@,2}$, $R_{@,1}$, $R_{@,2}$, $I_{@,1,2}$, and $I_{@,2,1}$, where $@$ is the Protocol Given in Figure 11. ........................................ 64
16. DIG for the Protocol in Figure 9 ..................... 72
17. DIG for the Protocol in Figure 11 ..................... 73
18. DIG for the Protocol in Figure 13 ..................... 74
19. Execution Tree for the Protocol in Figure 11 .... 84
20. The Use of Logical Time in Collision Detection .. 91
21. The Use of Confirmation Signals in Collision Detection ............................................. 93
22. Specification of the DTE ............................... 99
23. Specification of the DCE ............................... 100
24. Specification of X.21. (a) Elements of $T$'s and $R$'s. (b) Definition of Intention Function ... 104
25. Desirable Interaction Graph of X.21 Recommendation 107
26. Priority Assignment of X.21 ......................... 108
27. An Example of Collision Resolution in X.21 .... 110
28. History Variables ...................................... 139
29. A Seemly Cyclic Proof Dependancy Among $is_{1,2}$, $os_{1,2}$, $is_{2,1}$, and $os_{2,1}$ ..................... 149
30. Desirable Interaction Graph of a Hypothetical Protocol .......................... 170
31. Conditions for $p_{1}$<@$p_{2}$ greet .......................... 175
32. Conditions in the Definition of Path$_{@,k^{*}}$ ........ 179
33. Specification of PDCPD ............................... 182
34. The Use of Varibale $h$-path .......................... 186
35. The Use of Boolean Guard in the Second Type of Guarded Command. ......................... 188
36. The Use of Boolean Guard in the Third Type of Guarded Command .............. 191
37. The Use of Boolean Guard in the Fourth Type of Guarded Command .............. 193
38. An Equivalent Terminating Specification of PDCPD .................. 195
39. Illustration for AS\_i(4,2) and AS\_i(4,3) .................. 202
1. INTRODUCTION

The revolutionary development in microelectronics and information processing has given economic incentive for distributed-processing computer networks. A computer network consists of a number of computers, terminals and devices linked together by a wide variety of communication media. Today, computers are accessible by dialing connections through telephone switched networks; they are joined by leased or privately owned telephone grade voice frequency lines and via high bandwidth lines using coaxial cables, twisted wires, optic fiber transmission cables, microwave links, satellite links and radio broadcasts. As a result of advances in packet-switching techniques [68], the last 15 years have seen the emergence of numerous computer networks. They range from a pair of connected processors to complicated interconnections in which hundreds of computers of various sizes and capabilities scattered over a wide geographical area are interconnected to each other and to thousands of terminals, often with various forms of special multiplexors and controllers in between [29].

The motivation to have distributed-processing computer networks is manifold. The most important factors influencing the trend
toward the use of computer networks include: the need to share valuable resources among a community of users, the desire to distribute processing capabilities throughout a number of processing units to attain higher cost effectiveness, and the wish to enhance system’s responsiveness and reliability. Depending on particular circumstances and applications, the communication between a pair of end users in a computer network may take the form of a terminal user invoking a remote application program, two application programs interacting with one another, one application program querying or updating remote files, and so forth.

One of the key aspects of computer networking is communication protocols, which provide rules governing all communications. Protocols are rules of procedure which prescribe the manner in which communication takes place, the meaning of data communicated and the correctness of a particular communication under some prescribed condition [55]. Depending on the level of operation, communication protocols are used to direct a computer to send a raw bit stream of data into the network, to group the bits into logical units, or to rectify problems when bits are garbled by transmission errors. They are also used to control the flow of communication and to provide mechanisms for addressing, routing and delivery. At some higher levels, protocols also provide services for transferring files among physically separated computers, enabling communication among
Incompatible terminals and ensuring security in data transmission. In short, communication protocols are important in computer networks that operate among geographically separated parts of a system, making use of the network's ability to carry messages.

1.1. Formal Techniques in Protocol Specification and Verification

Because communication protocols form the cornerstone upon which computer networks are built, the task of the protocol designer is an important one. With more and more computer networks in service, an increasingly interesting question is how to build communication protocols correctly. Correctness in protocols is a complicated problem because a protocol is implemented by two or more entities operating independently in parallel, and because these physically separated entities must coordinate their activities by exchanging messages over some, usually unreliable, communication media.

Due to the efforts of developing systematic expressions of communication protocols in the context of network architectures, it has been possible to represent protocols precisely and formally. The techniques used for formal specifications in the general context of software engineering [60] have been successfully applied to the area of formal protocol specifications. Surveys of development in this area can be found in [7, 19, 78]. Two major approaches to formal protocol specifications are based on state transition models.
[5, 17, 28, 35, 50, 83] and on programming language models [30, 77]. Hybrid combinations of these two approaches are also increasingly used [4, 6].

Once formal specifications have been completed, various testing or verification procedures can be used either to examine the protocols for the presence of various defects or to prove their absence. Verification procedures used by state transition models in order to check the logical correctness of a protocol consist of (1) modeling the global state of the communication system (i.e., the state of protocol entities and communication media), and (2) producing a graph of all global states reachable from some initial state. Verification procedures for protocols specified using programming language models are those common to program correctness proof [22, 23, 36, 49]. Bolstered by logic assertions and inductive reasoning rules, correctness proof has the advantage of showing that a protocol design performs its intended functions as well as demonstrating the absence of certain defects.

Compared to protocol specifications, the success of protocol verifications has been only moderate. All of the existing verification techniques suffer from the same difficulty — they can be applied only to protocols of limited complexity. The most serious problem common to state transition models is the combinatorial explosion of the global state space as the complexity
of individual protocol entities increases. In contrast, the proofs of correctness used with programming language models are quite difficult to develop. The steps involved in investigating the properties of the protocol place too much dependence on human ingenuity and intuition, making the amount of effort in proof usually far greater than that required to specify the protocol.

1.2. Objectives and Main Results

As noted above, the objectives of protocol verifications are to examine the design of communication protocols for the presence of various errors (or to prove their absence). According to Bochmann and Sunshine [7, 78], the overall verification problem can be divided into two categories: the verification of general correctness properties and the verification of properties that are particular to the individual protocols.

General correctness properties are those properties or characteristics that are essential to all, or nearly all, communication protocols. They are general because they are independent of the special function that a protocol is supposed to perform. Among those general properties are deadlock freeness and completeness (i.e., having provisions for all conditions that may arise). Freedom from livelock (cyclic system behavior without meaningful progress), freedom from overflow, stability and proper
termination also fall into this category.

State transition models are suitable for verifying general correctness properties because the presence of these properties can easily be detected by examining the global state graph, as mentioned in the last section. The difficulty of such a verification technique is the "state space explosion" — the number of global states increases exponentially with the number and complexity of protocol entities and communication media.

As observed in [30], even simple protocols can produce extremely complicated interactions, since they must cope with asynchronous protocol entities. However, from the designer's point of view, many of these potential interactions are unexpected and undesirable. To cope with this problem, we have obtained a new method for designing protocols, which is called Priority Driven Communication Protocol Design (PDCPD). PDCPD consists of an enhanced state transition model and a synchronization mechanism. The enhanced state transition model is a specification technique based on finite state automata. This new model exploits the coupling relationship between communicating protocol entities and defines the desirable interactions from the viewpoint of the protocol designer. In the actual execution of protocol, the communicating entities may fail to interact in the desired fashion. Once the failure conditions (i.e., deviating behavior) have been
discovered, the synchronization mechanism is used to correct such conditions. This mechanism endows the protocol entities with the ability to detect and recover from a failure condition dynamically during the execution of the protocol.

There are two advantages of this new approach. First, the state space explosion problem involved in protocol verification is greatly alleviated. A drastic reduction of verification complexity can be seen in the following example. Assume that a communication protocol consisting of n entities is specified by n finite state automata, each describing the behavior of an individual entity. Between every pair of entities there is a (logical) communication channel that provides transportation of messages. In addition, assume that (1) each automaton has p states; (2) each channel may contain up to m messages at a time; and (3) there are q different types of messages. The time and space complexity involved in verifying this protocol using traditional approaches and PDCPD is listed in Table 1.

<table>
<thead>
<tr>
<th>Traditional State Transition Models</th>
<th>PDCPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space: (0(n^2 \times (p^n \times q^m \times n^2)))</td>
<td>(0(n^*p^E))</td>
</tr>
<tr>
<td>Time: (0(n^2 \times (p^n \times q^m \times n^2)^2))</td>
<td>(0(n^<em>p^{2</em>n}))</td>
</tr>
</tbody>
</table>

Table 1: Comparison of verification complexity.
The second advantage is that the burden of handling failure conditions is removed from the protocol designer, since failures are corrected automatically by the synchronization mechanism. The designer can concentrate his attention on the functional aspects of the protocol, thereby reducing the design effort and the possibility of introducing design errors.

Programming language models are superior to state transition models in the area of handling variables and parameters. They also allow for verification of the full range of protocol properties. Verifying a communication protocol usually consists of two steps: (1) developing the verifications of individual processes; and (2) combining the individual proofs, showing they are interference-free and deriving the desired properties of the entire communication system. The difficulties, as noted in the previous section, are due to the dependence on human ingenuity and intuition. Specifically, in verifying an individual process, the effect of communication commands on the process is difficult to account for; and in proving the individual proofs are free from interference, the task can be arduous and tedious. These difficulties have restricted application of programming language models to the simple data transfer aspects of communication protocols, rather than the more complex control aspects.

In this regard, our effort has concentrated on reducing the
complexity in verification. Specifically, a new axiomatic system for verifying the correctness of concurrent programs is proposed. Because communication protocols are simply one type of concurrent program, this axiomatic system is readily applicable to protocol verification. The distinctive characteristic of this system is the use of input and output assertions constructed from history variables. In verifying a single process, the input and output assertions provide an easy rule for reasoning about the effect of each communication command, thereby reducing the complexity in proving the individual processes. Upon discovering the potential interference between the proofs of individual processes, the new system works on a process basis, rather than on a statement basis, as done by some existing techniques. The advantage of the new system can be seen from the following example. Assume that a concurrent program consists of \( n \) communicating sequential processes, and that each process consists of \( m \) input commands and \( m \) output commands with every other process. Then for the existing techniques, it will take \( 2^m m^2 C(n, 2) \) proofs, where \( C(m, n) \) denotes the \( n \)-combination of an \( m \)-element set, to ensure the satisfaction of cooperation between each pair of corresponding communication commands. On the other hand, the new axiomatic system needs only \( 2^m C(n, 2) \) consistency proofs to verify interference-freeness between the proofs of individual processes.
As mentioned at the beginning in this section, determining the correctness of a communication protocol includes the verification of the general correctness properties and the verification of the special functions of the protocol. PDCPD, being a state transition based method, is apt to assist the protocol designer in achieving general correctness properties. Conversely, the new axiomatic system is more capable of proving the special properties of concurrent programs as well as communication protocols. Together they (PDCPD and the axiomatic system) constitute a complete methodology for designing reliable communication protocols and form the backbone of this dissertation. Even though PDCPD has been tested on several real protocols, including CCITT X.21 Recommendation [10] to demonstrate its usefulness, its applicability to arbitrary protocols has been a challenge. To satisfy this curiosity, the axiomatic system has been applied to PDCPD. It has been shown that PDCPD is applicable to any protocol which operates in a certain environment.

1.3. Organization of Dissertation

This dissertation is concerned with the new methodology for designing communication protocols. Each chapter addresses a distinct topic of research or area of investigation. When combined, they provide a coherent view of the methodology that is useful for
the design of some of the most complex communication protocols.

Chapter 2 serves as a basis for this research. It contains a discussion of the layered architecture of communication protocols; in particular, it presents the International Standards Organization's (ISO) Reference Model of Open Systems Interconnection, to help the reader appreciate the value of such a protocol architecture. It describes a number of formal techniques for protocol specification and verification, and illustrates their usage by applying them to describe a simple communication protocol, the alternating bit protocol for data transferring. It also addresses the relationship between these existing techniques and the new protocol design methodology.

Chapter 3 presents PDCPD, emphasizing how PDCPD can help satisfy the general correctness properties described in the previous section. It describes the enhancement of a conventional state transition model and the principle of the synchronization mechanism. It also discusses the validation technique of PDCPD and its complexity. The use of PDCPD is further explained in this chapter by describing the verification of International Telegraph and Telephone Consultative Committee (CCITT) X.21 Recommendation. The verification of X.21 demonstrates that PDCPD is useful in dealing with real protocols and increases confidence of using PDCPD in coping with more complex protocol designs. The results, in
comparison with several existing reports on verifying X.21, demonstrate the superiority of PDCPD.

Chapter 4 discusses the new axiomatic system for verifying both concurrent programs as well as communication protocols. It first addresses the motivation for using such a system. Because Hoare's programming language, Communicating Sequential Processes (CSP), is used for specifying concurrent programs, it includes a brief description to CSP. Several existing axiomatic systems for CSP are reviewed, and their shortcomings in dealing with complex concurrent programs are discussed. This chapter presents the tools which are used by the new axiomatic system to overcome these shortcomings. A number of examples are also included to give the reader a feel for the new system.

Chapter 5 contains the formal specification and verification of PDCPD. The success in verifying the correctness of PDCPD demonstrates the applicability of the new axiomatic system to complex concurrent programs.

Finally, Chapter 6 summarizes the contribution made by this research to the state of art and science in protocol design. Directions for further research which may extend and enhance the results of this dissertation are also suggested.
This chapter introduces computer communication protocol architecture, defines the meanings of protocol specification and verification, and surveys several formal techniques for protocol specification and verification. These formal techniques will be applied to a simple protocol, the alternating bit protocol (ABP), in order to examine their strengths and weaknesses.

This chapter is organized as follows: In the first section the concept of the layered architecture of communication protocols is introduced. Specifically, informal introduction to computer networking using the International Standards Organization's (ISO) Reference Model of Open Systems Interconnection (OSI) is presented in order to explain concepts such as processes, protocols, interfaces and services. In the second section the need for formal techniques in communication protocol design is discussed. The meanings of specifications and verifications in the context of layered protocol architecture are clarified. In the third section several major formal approaches to specifying and verifying communication protocols are briefly surveyed. The models of
particular interest in this chapter include finite state automata, formal grammars, Petri nets and programming languages. Finally, the last section summarizes the contents of this chapter. The relationship between the formal models introduced in this chapter and the new protocol design methodology, which will be discussed thoroughly in the following chapters, is also pointed out.

2.1. Protocol Architecture

The idea of solving complex problems by decomposing them into smaller problems and by building complex structures out of simpler building blocks is not new. Applications of such an idea can be seen in every aspect of software development. For example, in the area of operating system design, most of the early operating systems consisted simply of one big program. As systems became larger and more comprehensive, this "brute force" approach became unmanageable. Eventually, it became clear that the extended machine approach could be applied to the design of operating systems. The extended machine approach builds an abstract machine on top of the physical machine. Through a hierarchical system of layers, an operating system with each layer using the services provided by the layer beneath it can be built. At least two advantages are acquired from such an approach. First, the divide and conquer nature of such an approach helps control the complexity of design. Second, given a system
constructed in such a hierarchical fashion, it is possible to change the implementation of a layer without affecting the other layers, as long as the interface between layers remains constant.

The same approach has been applied to the design of computer communication protocols. To reduce their design complexity, most computer networks are organized as a series of layers, each one built upon its immediate lower layer. The purpose of each layer is to provide a particular service to the layer above, shielding the higher layers from knowing the actual implementation of this service.

Figure 1 depicts the aforementioned layered architecture of communication protocols. A computer network is a collection of computing machines that communicate with one another by exchanging messages. Layer $n$ on one machine is called an $(n)$ process. The processes comprising the corresponding layer on different machines are called peer processes or communicating processes. The peer $(n)$ processes receive the $(n-1)$ services from the next lower layer (layer $n-1$) and they collectively, in turn, provide $(n)$ services to the processes in Layer $n+1$.

An $(n)$ process makes use of $(n-1)$ services to carry on a conversation or communication with the other peer $(n)$ processes. The rules and conventions used in this conversation are collectively called $(n)$ protocol. Between two adjacent layers is an interface,
Figure 1: Layered protocol organization
which defines certain primitive operations through them the upper layer can request services from the lower layer.

In reality, data or messages are not transmitted horizontally from one (n) process to another peer (n) process except in the lowest layer. Instead, data are passed vertically down the layers of the sending machine and continues up the layers of the receiving machine. Real horizontal communication occurs only in the lowest layer, which takes care of the data transmission over the physical media. However, each process appears to communicate as being horizontal with its peers, through the services provided by the layer immediately below it.

To illustrate the aforementioned concepts, a particular communication protocol architecture will now be introduced — that of the International Standards Organization’s (ISO) Reference Model for Open Systems Interconnection (OSI) [38]. The ISO has been developing a protocol architectural model that 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" [91]

The reference model is divided into seven layers, as shown in Figure 2. At the lowest layer, the physical layer provides the mechanical, electrical, functional and procedural details allowing the machine to send a raw bit stream into the network. The form of the bit transmission within the physical layer is hidden from the
higher layers. At the top of the hierarchy lies the **application layer** which directly provides services to the users of OSI environment.

In between are five intermediate protocol layers. The following are the purposes of these layers. The **data link layer** adds a frame structure to the raw bit stream between two machines. It also attempts to recover from communication errors due to transmission noises, loss and/or duplication. The **network layer** ensures data are correctly received at their destination and are in the proper order. The control of routing and congestion is also handled in this layer. The **transport layer** provides multiplexing services, handles addressing, and takes care of connection establishment and termination across the network. The **session layer** sets up and manages user-to-user connections. Finally, the **presentation layer** performs a variety of conversions of data to serve the purposes of text compression, data security, communication between incompatible terminals as well as file transfer. A more thorough discussion of ISO OSI model can be found in [79, 80, 91].

In addition to ISO, many national and international standards organizations have been participating in the development of the Reference Model for Open Systems Interconnection. Although many existing network protocol structures vary from the ISO's seven-layered Reference Model, the layered approach has become
Figure 2: The seven-layer ISO Reference Model for communication protocols
2.2. Formal Techniques in Communication Protocol Design

As protocols are developed, they must describe precisely an understanding of the function and design of the protocols in the context of the protocol architecture. In an early design phase, such descriptions provide a reference for cooperation among designers of various parts of a protocol. Later the protocol description is used to examine the logical correctness of the design. Finally, the description is used to guide the implementation of the protocol. In each phase, a protocol description must be clear and detailed regarding the interaction between the communicating processes. For example, in protocol implementation, protocol endpoint processes are often implemented by different people in different hardware and software environments. Therefore, it is important that the meaning of each activity involved in the protocol and the reaction of each process to that activity be clearly specified.

Although informal techniques using narrative descriptions and walk-throughs are useful for the aforementioned purposes, in practice they have proven to be inadequate. Formal techniques, therefore, have been adopted to provide clearness and to render confidence when designing communication protocols. Their value is essentially universal.
that they allow the protocol designer to describe the external behavior of a protocol system precisely without specifying its internal implementation. In particular, formal techniques attempt to satisfy the following goals:

1. To provide a precise description of the protocol system.

   They should leave no doubt as to the behavior of a system for each possible input.

2. To provide a means for checking logical correctness.

3. To provide a user interface that is easy to understand.

4. To separate what a protocol system does (function) from how it does (implementation). They should describe the behavior of a system without constraining the way it may be implemented.

As noted above, formal techniques are used in three design steps of communication protocols: specification, verification and implementation. Implementation relies on the particular hardware and software systems used; therefore, it is not considered in this dissertation. The words "specification" and "verification" are often used by people to refer to a variety of things. The goal in this section is to define these terms in the context of the layered protocol architecture which has been discussed in the previous section.
2.2.1. Specification

Specification refers to the description of an object. In the layered architecture of communication protocols, there are two types of specifications in Layer n of the protocol hierarchy [7, 78]:

- The (n) service specification describes the services which the communicating (n) processes provide for their (n+1) users (i.e., the (n+1) protocol layer). The services provided by a protocol layer are usually specified by a number of service primitives which describe the operation at the interface through which the services are provided.

- The (n) protocol specification describes the interaction among the (n) processes that comprise Layer n. A protocol specification should describe the operation of each process in response to commands from its users, messages from the other peer processes as well as internally initiated actions (e.g., timeouts).

Most formal techniques for specifying communication protocols concentrate on protocol specifications, rather than on service specifications. For formal specifications, several major models that are primarily used for protocol specifications will be presented in the next section.
2.2.2. Verification

To verify an object is to demonstrate that the object meets its specifications. According to this definition, verifying a communication protocol consists of two parts: (1) The protocol specification is analyzed to determine whether the combined operation of the communicating processes satisfies the service specification; and (2) the implementation of communicating processes is examined to see if it is consistent with the protocol specification.

As noted above, the concern is not with verifying the implementation of protocols because that involves consideration of the particular hardware and software systems used in the implementation.

The problem of verifying protocol specification against service specification can, in turn, be divided into two categories. In one category, verifications refer to the limited testing and confirmation that the protocol specification satisfies a number of general correctness properties which are essential to all, or nearly all, protocols. Verification of this category has often been termed "validation." In the other category, verification means demonstration that the protocol implemented according to the specifications provides the users with the prescribed services.

The list of general correctness properties in the first
category is agreed upon by many researchers [5, 27, 50, 78]. Most protocols should exhibit the following characteristics:

1. FREEDOM FROM DEADLOCK: Each protocol system state allows for progress to another state.

2. COMPLETENESS: The protocol is able to handle all conditions that may arise.

3. STABILITY: From an "abnormal" state, a protocol will eventually return to a normal state.

4. PROGRESS: The protocol does not exhibit cyclic behavior in which no meaningful communication takes place.

5. FREEDOM FROM OVERFLOW: The protocol will not allow more in-transit messages than the communication channels can handle.

6. TERMINATION: The protocol will arrive at the desired final state.

Previous experiments have already included verifying reliability in data transfer protocols, proper establishment and termination of connections in transport protocols, and copying files in file transfer protocols. Definitions of these features are the subject of service specifications.
2.3. Current Approaches to Formal Specification and Verification

During the last decade, many formal techniques have been used to specify and verify communication protocols. (For a list of bibliography, see [19].) In general, these formal techniques can be classified as either state transition techniques or programming language techniques. This section describes four major approaches, which includes finite state automata, formal grammars, Petri nets and programming languages.

2.3.1. Finite State Automaton Models

One of the most popular state transition models, including the one used in the new protocol design method (PDCPD, to be presented in Chapter 3), is based on modelling the behavior of communicating processes with finite state automata (FSA) [5, 8, 28, 66, 69, 70, 71, 75, 89, 90]. FSA models are motivated by the observation that protocols consist largely of relatively simple processing in response to numerous "events" such as commands (from the user), message arrival (from another peer process), etc. Hence, FSA with such events forming their transitions are a natural model for specifying communication protocols. The basic approach consists of specifying the protocol system as a number of finite state automata, each describing the behavior of a participating process.

In the following description of FSA models, the notation used
is adopted from [5]: The possible states of the process are denoted $s_i$. A transition of event $e$ between two states $s_i$ and $s_j$ is identified by $s_i \xrightarrow{e} s_j$.

Figure 3 illustrates the use of FSA in specifying a simple protocol, called the alternating bit protocol (ABP). This protocol is used by a pair of communicating processes to transmit data over an unreliable channel in the following way: Data are continuously transmitted from one process (called the transmitter) to the other (called the receiver). The units of data, each called a frame, are transmitted one at a time on an unreliable channel. During the operation, each data frame carries a parity bit, which alternates in value between successive data frames. Upon transmitting a data frame, the transmitter starts an internal clock. If the data frame is correctly received, the receiver returns to the transmitter an acknowledgement frame with the same parity as the received data frame had. No acknowledgement frame will be returned in case of damage or loss. If no reply (acknowledgement frame) has been received within a certain predetermined time interval, the clock times out. In this case, the transmitter assumes the transmitted data frame is damaged or lost and retransmits this data frame again. The next data frame will be transmitted only when an acknowledgement of the previous frame has been received before the clock times out.

One assumes that each of the communicating processes starts
Figure 3: Specification of alternating bit protocol using finite state automaton model
from its respective state 1. In Figure 3 events "new" and "use" denote the events of receiving a data frame from the user on the transmitter site and delivering a received frame to the user on the receiver site, respectively. Frame transmissions are distinguished from receptions by underlined symbols. (For example, $D_0$ and $D_0$ represent the events of transmitting and receiving data frame with parity 0, respectively.) Transmission errors (frame damage and loss) and timeout are denoted by E and T, respectively.

The validation techniques used by FSA models are all based on some form of reachability analysis in which all possible interactions of communicating processes are generated from some starting global state. A composite global state of the protocol system is a Cartesian product of the states of participating processes and the states of communication channels that transport messages (frames, data, packets, etc.) between processes. From the starting global state, new global states are generated by applying all possible transitions. The step of discovering new reachable global states are repeated until no new states can be generated.

Reachability analysis is readily applicable to examining general correctness properties which have been described in the previous section. A global state from which none of the peer processes can proceed represents a deadlock or a desired terminating state. A global state in which a communication channel contains a
message for one process without specifying action to receive that message indicates incompleteness in the protocol specification. Instability can be manifested by a sequence of transitions that brings the protocol system to an abnormal global state and fails to return to its normal operation.

More discussion on FSA specification models and their validation techniques will be made in Section 3.1. In order to assist the reader in understanding the process of validation, a reachability analysis technique that uses a so-called "empty medium abstraction" [5] is presented here. Under the assumption of this abstraction, a global state consists only of the states of the communicating processes. The communication channels are considered empty; i.e., no messages are in transit.

A global state space graph of the aforementioned alternating bit protocol generated by reachability analysis under the empty medium abstraction is given in Figure 4 [5]. Because the complete graph is quite complex, only half of it is shown. In this graph, the transition labeled $D_0$ stands for reliable transmission of a data frame with parity 0. To model damage or loss, $D_0^E$ represents the data frame that is damaged on transmission, while $D_0^L$ represents the data frame that is lost on transmission. To allow modelling simultaneous message transmissions in both directions, $D_0 \parallel A_0$ is used to stand for simultaneous reliable transmissions of $D_0$ and $A_0$. 
Similarly, $D_0 | | A_0^T$ represents that $D_0$ is transmitted correctly but $A_0$ is lost. The labels of the other transitions can be interpreted similarly.

One of the major advantages of FSA models is that the process of generating global state space can be easily automated. Several computer aided systems for this purpose have been developed [16, 65, 86, 88, 89]. FSA models are also particularly effective in representing the control structure of protocols. Events, such as user's commands, message receptions and timeouts, form transitions, thereby making FSA models a natural choice for describing the control aspect of protocol systems.

The main difficulty of the FSA models is the so-called "state space explosion" problem. As shown in Figure 4, even the simple alternating bit protocol requires a large number of global states in order to describe the interactions of the overall protocol system. As described before, a composite global state is a Cartesian product of the states of the individual processes and the communication channels. As the individual processes become more complex and more states are required in their finite state automaton representation, the number of global states increases exponentially, thereby severely limiting the usefulness of this approach.

Another limitation is inherent in the model's ability in representing the communication channels. Due to the complexity in
Figure 4: Global state space of the alternating bit protocol generated by reachability analysis under the empty medium abstraction [5]
reachability analysis, the validation is restricted to situations in which a certain number of messages are in transit at any given time. In the empty medium abstraction, one explores only situations in which no message is in transit. Consequently, some errors which appear with more in-transit messages than the limiting number will go undiscovered in such a validation.

A more general validation model based on FSA description will be discussed in Section 3.1.

2.3.2. Formal Grammar Models

Another group of commonly used state transition models is based on the theory of formal grammars [32, 34, 35, 81, 82, 84]. To specify a communication protocol in a formal grammar model is to define a set of grammar rules which make up the actions of the protocols. The motive of using formal grammars to describe cooperation between peer processes is, according to [35], as follows: In the case when the protocol is rather complex and the finite state description becomes difficult due to the large number of states required, the use of formal grammar description is more convenient.

A special form of formal grammar model proposed by Albert Y. Teng [81, 82, 83, 84] for specifying communication protocols, called the Transmission Grammar (TG), is introduced here. Terminal
symbols in the TG grammar rules represent protocol events, and the non-terminals are equivalent to the states of a finite state automaton. A grammar rule, also called an action rule, is written as \(<u>::=x\), where \(<u>\) is a non-terminal and \(x\) is a string of terminals and a non-terminal. The meaning of a grammar rule is that the process in the state specified by the left hand non-terminal may take the action specified by the terminals and enter the state specified by the right-hand non-terminal.

The specification of the aforementioned alternating bit protocol (ABP) using TG is given in Figure 5. The relationship between a TG specification and the FSA specification should be clear by comparing Figure 5, which gives the TG specification, with the FSA specification as given in Figure 3. However, in addition to the specifications of the transmitter and receiver, specification of the actions of the clock is also given in Figure 5 to make the specification more complete.

In order to validate the protocol using reachability analysis, Teng suggested a model of the global state, called Validation Automata (VA), that consists of the communication medium and peer processes. The VA validation technique is very similar to the reachability analysis technique used by the FSA models; therefore, it will not be discussed here.

In Teng's proposal, TG model allows the protocol designer to
Transmitter:

<ready-0> ::= New_data <send-0>

<send-0> ::= Send_data_0 Send_start_timer <wait-0>
   | Send_error Send_start_timer <wait-0>

<wait-0> ::= Receive_ack_0 Send_stop_timer <ready-1>
   | Receive_error Send_stop_timer <send-0>
   | Receive_ack_1 Send_stop_timer <send-0>
   | Receive_timeout <send-0>

<ready-1> ::= New_data <send-1>

<send-1> ::= Send_data_1 Send_start_timer <wait-1>
   | Send_error Send_start_timer <wait-1>

<wait-1> ::= Receive_ack_1 Send_stop_timer <ready-0>
   | Receive_error Send_stop_timer <send-1>
   | Receive_ack_0 Send_stop_timer <send-1>
   | Receive_timeout <send-1>

Receiver:

<wait-0> ::= Receive_data_0 <received-0>
   | Receive_error <ack-1>
   | Receive_data_1 <ack-1>

<received-0> ::= Use_data <ack-0>

<ack-0> ::= Send_ack_0 <wait-1>
   | Send_error <wait-1>

Figure 5: Specification of alternating bit protocol using Transmission Grammar model
Figure 5: (continued)

\[
\begin{align*}
\text{<wait-1>} & \::= \text{ Receive_data_1 <received-1>} \\
& \quad | \text{ Receive_error <ack-0>} \\
& \quad | \text{ Receive_data_1 <ack-0>}
\end{align*}
\]

\[
\begin{align*}
\text{<received-1>} & \::= \text{ Use_data <ack-1>}
\end{align*}
\]

\[
\begin{align*}
\text{<ack-1>} & \::= \text{ Send_ack_1 <wait-0>} \\
& \quad | \text{ Send_error <wait-0>}
\end{align*}
\]

Clock:

\[
\begin{align*}
\text{<sleep>} & \::= \text{ Receive_start_timer <awake>}
\end{align*}
\]

\[
\begin{align*}
\text{<awake>} & \::= \text{ Receive_stop_timer <sleep>} \\
& \quad | \text{ Send_timeout <sleep>}
\end{align*}
\]
use context-free grammars to specify communication protocols. According to the well-known Chomsky hierarchy [13, 15] (after the famous linguist Noam Chomsky, who defined four classes of formal grammars as potential models of natural languages), context-free grammars are more powerful in expressiveness than the FSA. However, it remains unclear if context-free grammars are necessary for protocol specifications. In fact, it has been discovered that all examples given in [34, 35, 81, 82, 83, 84] are expressible in regular grammars, which are known to be equivalent to FSA models [14].

While the specification techniques based on formal grammars and those based on FSA are essentially equivalent, their different representations have an important effect on the comprehensibility of the specifications. In the author's opinion, specification based on FSA explicitly describes the concept of a state and the actions associated with it, while this is implicit in formal grammars based specifications. Because the concept of state is important in describing the behavior of an interactive system, FSA models are preferable to formal grammar models in this regard.
2.3.3. Petri Net Models

The Petri net models (PN) also belong to the state transition type of specification techniques [17, 51, 52, 53]. In PN models, conditions are represented by nodes and events are represented by transition bars. The holding of a condition is modeled by placing a token on the corresponding node. An event (transition bar) can fire (be initiated) when all nodes input to that event hold tokens. When an event fires, it removes one token from each input node and places one token on each output node.

Once again the alternating bit protocol is used as an example for protocol specifications. The specification of ABP in PN model taken from [52] is given in Figure 6. In this figure, the condition in which the transmitter is ready to transmit a data frame with parity 0 is represented by placing a token at node AO. When event (transition bar) 11 fires, the token is removed from AO, and another token is placed in node MO and WO to represent the data frame being in transit. Then the transmitter enters a state of waiting for an acknowledgement frame with parity 0. If the receiver is ready to receive a data frame with parity 0, represented by placing a token in node BO, event 12 fires when a token is placed in MO. Upon this action, node CO gains a token and, therefore, event 13, which represents the receiver transmitting an acknowledgement frame with parity 0 to the transmitter, can fire. The rest of the
specification can be interpreted similarly.

The validation techniques used by the Petri net models are also similar to those used by FSA or formal grammar models. A global state of PN models is defined as the names of nodes holding tokens. The global state space showing the possible transitions from one global state to another is called the token machine (TM). The TM for the ABP, which is also taken from [52], is given in Figure 7. Heavy lines represent the conditions in which no failures occur, and thin lines model the paths that represent failures.

In classification, formal protocol models using finite state automata, formal grammars and Petri nets all belong to state transition techniques. Therefore, they all have the same advantages and disadvantages as protocol design tools, as discussed in Subsection 2.3.1.

2.3.4. Programming Language Models

In a programming language model, a protocol is specified as a concurrent program [3, 30, 31, 33, 41, 77]. Depending on the abstraction level of the language, such a concurrent program may serve as a specification or an implementation of a protocol. Because a protocol specified in a programming language model can be similar to an actual implementation, unessential or implementation dependent features are often combined with the essential properties
Figure 6: Specification of alternating bit protocol using the Petri net model [52]
Figure 7: Global state space of the alternating bit protocol generated by reachability analysis in Petri net model [52]
of the protocol.

In spite of the aforementioned shortcoming, programming language models are extremely important long-term research projects. Their ability to handle variables and parameters, allowing them to take on widely varying values, is one of the major advantages of such an approach as compared with the state transition approach.

Another important advantage of programming language models is their ability to deal with the full range of protocol properties to be verified, rather than only general correctness properties. The program verification techniques, which involve the formulation of assertions of the desired properties, are readily applicable. Ideally, any desired properties that can be formulated by appropriate assertions can be verified by the Floyd/Hoare style of proof technique [23, 36].

As stated in [30], logical assertions attached to the program abstract information from the representation of the state and allow reasoning about classes of states. This avoids the combinatorial state space explosion, and the length of the verification need not grow unmanageably as the protocol complexity increases. This is considered as the third advantage of programming language models.

Even with the aforementioned advantages, only modest progress has been made to date in programming language models, and they are at least 10 years away from being of any major significance [25].
The difficulties in applying program proof techniques to verifying communication protocols include:

1. As noted above, any properties to be verified must be formulated by appropriate assertions. The services provided by the protocol are among the most interested properties. However, services in most protocols have not been rigorously defined, thereby discouraging the use of program verification techniques. In addition, assertions formulated to reflect the correctness properties are usually quite complex and are not amenable to verification.

2. Program verification is very often a long and arduous task with the effort to prove the program far exceeding that required to develop it. The steps in reasoning about the properties of the program place too much dependence on human ingenuity and intuition, thereby making the automation of verification almost impossible.

As an illustration of the use of programming language model, the specification of the alternating bit protocol using an Algol-like language taken from [18] is shown in Figure 8.

Programming language models will be more thoroughly discussed in Chapter 4. In that chapter a new program verification technique that is applicable to verify a wide range of protocol correctness properties with reduced verification complexity will be presented.
RP = Receiver parity
TP = Transmitter parity
FP = Frame parity
AP = Acknowledgement parity
TS = Transmitter state
Initial conditions: RP \# TP, TS = READY

TRANSMITTER

Transmit new frame
IF TS=READY THEN BEGIN
  FP:=TP;
  TRANSMIT FRAME;
  QUEUE FRAME;
  START TIMER;
  TS:=NOT READY;
END

Receiving Acknoledgement
IF NO ERROR AND AP=TP THEN BEGIN
  DEQUEUE FRAME;
  STOP TIMER;
  TP:=TP+1 MODULO 2;
  TS:=READY;
END
ELSE RETRANSMIT;

Time-out
RETRANSMIT;

Retransmit
FP:=TP;
TRANSMIT QUEUED FRAME;
RESTART TIMER;

RECEIVER

Frame received
IF NO error AND FP\#RP THEN BEGIN
  DELIVER DATA;
  RP:=FP;
END
AP:=RP;
SEND ACKNOWLEDGEMENT

Figure 8: Specification of alternating bit protocol using the programming language model [18]
2.4. Summary

In this chapter, basic concepts that are related to the design of communication protocols have been presented. Layered network architecture has been presented, especially the International Standards Organization's Reference Model of Open Systems Interconnection in which protocols can be defined precisely and formally. Several major formal techniques for designing communication protocols have been reviewed. A simple data transfer protocol, the alternating bit protocol, has been used to illustrate the use of these techniques.

State transition models have, as discussed in Section 2.3, the advantage of ease in verifying general correctness properties such as completeness and freedom from deadlock in protocol specifications. However, their power has been limited to rather simple protocols because of the problem of state space explosion.

In the next chapter, a new protocol design method, PDCPD, will be presented. Protocol specifications in PDCPD are based on an enhanced state transition model. This enhanced state transition model exploits the coupling relationship between communicating processes to define which interactions of processes are desirable and which are not. A synchronization mechanism will be proposed to enable the communicating processes to detect undesirable interactions and convert them into desirable ones. As a result, the
complexity necessary for protocol validation is drastically reduced; therefore, the problem of state space explosion is largely alleviated.

Programming language models are superior to state transition models in the aspects of handling variables (or parameters) and verifying the full range of protocol properties, rather than only general correctness properties. However, they are, in practice, quite difficult to use because the amount of effort needed to verify a protocol specified in a high-level language far exceeds that required to develop the protocol.

In applying the programming language models to verify a communication protocol, the most difficult part lies in investigating the effect of communications. In Chapter 4, a new verification system for verifying a broad range of protocol properties will be presented. This new system makes use of the patterns in communications to reduce the consideration in dealing with communication commands, and exploits history variables to examine the interference-freeness between the verifications of individual processes, thereby lowering the complexity involved in protocol verification.
3. PRIORITY DRIVEN COMMUNICATION PROTOCOL DESIGN

In Chapter 2 a number of state transition models for formal protocol specification and validation were described. As discussed previously, state transition models are effective in representing control structure of communication protocols and are well-suited to checking general correctness properties such as completeness and deadlock-freeness. However, those models suffer from serious limitations; in particular, the state space explosion problem involved in protocol validation makes them applicable only to protocols with modest complexity. Even with the aids of automated tools, the exponential growth of validation complexity makes it impossible to generate and check all reachable states in reasonable time and storage.

In this chapter a new protocol design method, which is called the Priority Driven Communication Protocol Design [46, 47, 48], will be introduced. This new method possesses the aforementioned advantages of state transition models as a protocol design tool. However, PDCPD exploits a coupling relationship among communicating processes to produce a state diagram which records only those "desirable" interactions among the processes, and uses a
synchronization mechanism to detect and resolve the undesirable ones. As a result, the task of protocol specification becomes much simpler while the complexity in protocol validation is drastically reduced, making PDCPD superior to the conventional state transition techniques for protocol design.

This chapter is as follows: In Section 3.1 a generalized protocol validation model, called state perturbation technique developed by Collin H. West [89], is presented. This validation technique represents a typical reachability analysis technique used by conventional state transition models to examine the general correctness properties in protocol design. In Section 3.2 a state transition model for protocol specification upon which PDCPD is developed is discussed. A series of figures illustrating an example of designing a connection establishment protocol are presented. The state perturbation technique is applied to this example in order to discover some potential errors often embedded in protocol design, giving the reader an intuitive grasp of the problem. In Section 3.3 the model given in Section 3.2 is enhanced by adding an explicit specification, expressed in form of intention functions, of the coupling relationship among communicating processes. The consistency between the state transition model and intention functions is also discussed in this section. In Section 3.4 a new model of global system state and a reachability analysis algorithm
for checking the general correctness properties are presented. The discussion emphasizes those aspects which differ from conventional reachability analysis techniques. The reachability graph generated by this new algorithm is called desirable interaction graph (DIG). A comparison between the validation complexity of this new algorithm to those required by the conventional state transition techniques is made, and the relationship between the DIG and the individual processes is discussed. However, during their operation processes do not always interact in a desirable fashion. In Section 3.5 the conditions which violate the DIG are pointed out. Such conditions are named collisions. In Section 3.6 a synchronization mechanism which enables the communicating processes to detect and resolve collisions dynamically during the protocol execution is discussed. In Section 3.7 PDGP is applied to a real protocol, the CCITT X.21 Recommendation [10], to demonstrate the usefulness and advantage of this new protocol design method. Finally, a summary of this chapter is given in Section 3.8.

3.1. A Generalized Validation Technique

In Section 2.3 a simplified protocol reachability analysis technique that uses the empty medium abstraction was presented. Using this validation technique, a global state of the communication system consists only of the states of communicating processes. The
states of communication channels are ignored; therefore, such a validation technique checks only the protocol interactions where no messages are in transit. As discussed in that section, some errors may go undiscovered.

This section presents a more useful protocol validation technique which was first proposed by Collin H. West [89]. One assumes that the communication system is composed of \( n \) peer processes \( P_1, \ldots, P_n \), each being specified by a finite state automaton. The current state of process \( P_i \) is denoted at all times by \( P_s(i) \). Message exchange between a pair of processes \( P_i \) and \( P_j \) is assumed to take place via a couple of communication channels \( C_{i,j} \) and \( C_{j,i} \) which transport messages from \( P_i \) to \( P_j \) and from \( P_j \) to \( P_i \), respectively. In order to model in-transit messages, the state of channel \( C_{i,j} \) at all times is represented as \( C_s(i,j) \). In West's technique, a global state of the communication system is represented by an \( n \times n \) matrix

\[
\begin{bmatrix}
P_s(1) & C_s(1,2) & \cdots & C_s(1,n) \\
C_s(2,1) & P_s(2) & \cdots & \cdots \\
\vdots & \vdots & \ddots & \vdots \\
C_s(n,1) & \cdots & \cdots & P_s(n)
\end{bmatrix}
\]

A perturbation of a given global state is defined as another global state which can be reached by executing a single transition.
in one process $P_i$ from its current state $Ps(i)$. The rules governing perturbation generation, as described in [89], include:

1. When a message arrives at a process, it must be received immediately, and consequently it is removed from the channel. Because there is no specified time for a channel to transport messages from one process to another, it follows that the current state of a process must be able to absorb the first incoming message in all channels incident to the process at all times.

2. At any time any transition from the current state of a process which does not correspond to the reception of any message must be executed.

3. No transition may take place which results in an overflow of a channel's storage capacity.

The reachability analysis used by this validation technique begins with defining the initial global state. The initial global state is one in which every $Ps(i)$ contains the initial state of process $P_i$ and every $Cs(i,j)$ represents an empty channel. Next, all of the perturbations emanating from the initial global state followed by the perturbing procedure are generated. The perturbing procedure is applied to each newly generated global state. This process continues until no new global state can be generated.

The general correctness properties are readily examined by the
generated graph of reachable global states. For example, if in a
global state an incoming channel \( C_{j,i} \) to process \( P_i \) contains a
message but there is no departing transition from the current state
\( P_s(i) \) by which the first incoming message can be received, an error
of incompleteness is detected. A deadlock is represented by a
global state from which no perturbation can be generated. The
reader is referred to [89] for the details of this validation
technique.

West's perturbation technique will be applied to validate some
protocols in the following section.

3.2. FSA Protocol Specification Model

The FSA model for protocol specification has been briefly
described in Section 2.3. Because it is the one upon which the
PDCPD is built, it is worth discussing further.

In a FSA model a protocol is represented as a number of
communicating processes, as described in Chapter 2. Each process
is modeled by a finite state automaton (FSA) which consists of a
finite set of states, representing the possible conditions of the
process, and a finite set of transitions, representing actions taken
by the process. At each time the process can be at one of the
states. One state where the process begins its operation is
designated as the initial state. Some states (possibly none) where
the process stops operation are designated as final states. Each transition is associated with a transition type. Transitions departing from a certain state are called output transitions of that state.

Each FSA can be represented by a directed graph in the following way. The vertices correspond to the states of the FSA. A transition of type $t$ directing from state $s_1$ to state $s_2$ is represented by an arc labeled $t$ from the vertex corresponding to state $s_1$ to the vertex corresponding to state $s_2$ in the graph. A (directed) path is a finite alternating sequence of states and transitions of the directed graph, beginning with an arbitrary state and ending with an arbitrary state, such that each transition is oriented from the state proceeding it to the state following it.

In regard to protocol specification, one assumes that a FSA specifying a protocol process satisfies the following additional properties:

1. Output transitions of the same state have distinct types; i.e., the FSA must be deterministic.
2. Each state, unless it is a final state, has at least one output transition. A final state has no output transition.
3. Each state is reachable from the initial state via a path.

One assumes that transitions are associated with the meaning of signal (message) exchange for the purpose of communication. A
process must be either active or passive, but not both, on each transition. If process $P$ is active on a transition of type $t$ (abbreviated as transition $t$ hereafter) which directs from state $s_1$ to state $s_2$, then $P$ may initiate $t$ when $P$ is at $s_1$. Upon initiating the aforementioned transition, $P$ moves from $s_1$ to $s_2$ and transmits a signal (or message) to its peer processes indicating the initiation of this transition. Conversely, if a process $Q$ is passive on transition $t$ which directs from $s_3$ to $s_4$, then $Q$ will follow $t$ when it is at state $s_3$ and receives a signal which indicates the initiation of $t$. Upon following this transition, $Q$ moves from $s_3$ to $s_4$. A process is said to take a transition if it either actively initiates or passively follows this transition.

For simplicity, one assumes that each pair of communicating processes are connected by a full-duplex, noise-free and first-in-first-out (FIFO) channel. Under the current network techniques, these assumptions can be accomplished by using the mechanisms such as sequence numbering, positive acknowledgement, message retransmission upon timeout, etc.

A simplified connection establishment protocol taken from [90] is shown in Figure 9 to illustrate the aforementioned FSA protocol specification model. Process $P_1$ is specified by FSA $g_1$ and process $P_2$ by FSA $g_2$. Each transition is preceded by a positive/negative sign. If a process is active (passive) on a transition, then the
transition is preceded by a negative (positive) sign.

Connection request is represented by transition $t_2$. For example, whenever $P_1$ tries to establish a connection with $P_2$, $P_1$ initiates $t_2$ at state $s_0$ (in $g_1$) by transmitting a signal indicating $t_2$ to $P_2$. When $P_2$ receives this signal at state $s_0$ (in $g_2$), $P_2$ follows $t_2$ and a connection between these two processes is established. Transition $t_3$ can be initiated only by process $P_2$ to close an established connection.

Figure 10 depicts the validation of the protocol in Figure 9 using the state perturbation method as described in Section 3.1. From Figure 10 it is discovered that an error will arise when both processes initiate transition $t_2$ simultaneously. In that situation, both processes transmit a signal indicating $t_2$ at their respective state $s_0$ and then move from state $s_0$ to state $s_1$. However, at $s_1$ there is no provision for reception of a signal of $t_2$. According to the discussion given in Chapter 2, this error is an example of incompleteness.

An improved protocol, also taken from [90], is shown in Figure 11. This new protocol includes transitions which acknowledge the connection request transitions. The reader may verify that the aforementioned error of incompleteness still exists if the transitions of acknowledgement are chosen differently from the transitions of connection request. Therefore, the transition type
Figure 9: A simplified connection establishment protocol [90]
Figure 10: Validation of the protocol in Figure 9 based on state perturbation technique
of acknowledgement is chosen to be identical to that of the connection request, as shown in Figure 11. Although the error of incompleteness of the previous protocol has been eliminated, another error has been introduced. As depicted in Figure 12, two different sequences of state perturbations lead to the same resulting global state. Although process $P_1$ has taken the same sequence of transitions $-t_2 o + t_2$ in both cases ($o$ represents the concatenation operator), it never knows whether process $P_2$ has taken a sequence of transitions $+t_2 o - t_2$ (Figure 12, top) or another sequence, that of $-t_2 o + t_2$ (Figure 12, bottom). This phenomenon is called ambiguity in [90] because a process is not aware of the transition sequence taken by another process.

It is important to realize that in practice ambiguities are not necessarily a form of design error. In certain cases one process does not care which of several alternatives its peer process chooses. In spite of those possible exceptions, ambiguities are considered erroneous in this dissertation.

Zafiropulo proposed a correction, shown in Figure 13, for the aforementioned connection establishment protocol. The ambiguities are eliminated by choosing transitions of acknowledgement (type $t_4$) different from the transitions of connection request (type $t_2$). In addition, when a collision of simultaneous initiations of $t_2$ occurs, reception of $t_2$ takes process $P_1$ from state $s_1$ to state $s_4$ and takes
Figure 11: An improved connection establishment protocol [90]
Figure 12: Ambiguity in Figure 11
P₂ from s₁ to s₂. Hence, the collision is resolved in favor of P₁ in initiating the connection.

The above examples demonstrate that even very simple protocols can exhibit quite complicated behavior. Experience has demonstrated that erroneous behavior is usually difficult to foresee during the protocol design phase. For more complex protocols, errors can occur in extremely intricate ways (such an example can be seen in [74]) so that conventional protocol validation techniques may become impractical or even obsolete. This convinces us to develop a new methodology in order to alleviate this problem.

3.3. Enhanced FSA Protocol Specification Model

In this section, the FSA model described in the last section is enhanced with the specification of coupling relationship between message transmission and message reception.

Before presenting the enhanced FSA protocol specification model, the following notation will be defined. Assume a given protocol @ consisting of n communicating processes P₁,...,Pₙ is specified by n FSA (as defined in the previous section) g₁,...,gₙ, with gᵢ (1≤i≤n) specifying Pᵢ. STATEᵢ and TRANSITIONᵢ are used to denote the set of states and the set of transitions in gᵢ, respectively. For each state s in gᵢ, Tᵢ(s) and Rᵢ(s) are used to denote the sets of output transitions upon which Pᵢ is active and
Figure 13: A corrected connection establishment protocol [90]
3.3.1. Intention Functions

We now momentarily return to the simple connection establishment protocol given in Figure 9. When process $P_1$ initiates transition $t_2$ at state $s0$ (in $g_1$) to request a connection, one expects, if the processes are in synchronization, that process $P_2$ will respond to $P_1$'s request by following $t_2$ at state $s0$ (in $g_2$). Similarly, when $P_2$ initiates transition $t_2$ at state $s0$ (in $g_2$), and one also expects that $P_1$ will follow $t_2$ at $s0$ (in $g_1$). The same principle applies to transition $t_3$. When $P_2$ closes an established connection, it initiates transition $t_3$ at state $s1$ (in $g_2$), and one also expects $P_1$ to follow $t_3$ at $s1$ (in $g_1$). In order to clearly express the aforementioned relationship between transitions in different processes, three dash-lines are drawn to specify the coupled transitions of message transmission and reception between communicating processes $P_1$ and $P_2$.

In PDCPD it is required that the protocol designer specify such a coupling relationship by intention functions. Given a protocol $@$, for each pair of communicating processes $P_i$ and $P_j$ of $@$, a intention function $I_{@,i,j}$ is defined as a partial function mapping from $STATE_{@,i} \times TRANSITION_{@,i}$ to $2^{STATE_{@,j}}$. For $s \in STATE_{@,i}$ and $t \in TRANSITION_{@,i}$, let $I_{@,i,j}(s,t)$ be defined if and only if process
$P_i$ is active on output transition $t$ at state $s$. The meaning of $I_{@,i,j}(s,t)$ is as follows: If transition $t$ is initiated by process $P_i$ at state $s$, then one expects that process $P_j$ will respond to $P_i$'s initiation by following transition $t$ at one of the states specified by $I_{@,i,j}(s,t)$. To illustrate the use of the above notations, Figures 14 and 15 show the elements of $T^\prime$s, $R^\prime$s and $I^\prime$s for the protocols in Figures 9 and 11, respectively.

$$T_{@,1}(s_0) = \{t_2\}$$
$$T_{@,2}(s_0) = \{t_2\}$$
$$T_{@,1}(s_1) = \emptyset$$
$$T_{@,2}(s_1) = \{t_3\}$$
$$R_{@,1}(s_0) = \{t_2\}$$
$$R_{@,2}(s_0) = \{t_2\}$$
$$R_{@,1}(s_0) = \{t_3\}$$
$$R_{@,2}(s_0) = \emptyset$$
$$I_{@,1,2}(s_0,t_2) = \{s_0\}$$
$$I_{@,1,2}(s_0,t_2) = \{s_0\}$$
$$I_{@,1,2}(s_1,t_3) = \{s_1\}$$

Figure 14: Elements of $T_{@,1}$, $T_{@,2}$, $R_{@,1}$, $R_{@,2}$, $I_{@,1,2}$ and $I_{@,2,1}$, where $@$ is the protocol given in Figure 9

The term Direct coupling has been used by Bochmann [4, 5, 6] to define a somewhat similar concept as intention functions. Both express the idea of linking transitions in physically separated processes, and the linked transitions are expected to be executed in parallel. Nevertheless, there are significant differences between
these two terms: Direct coupling describes that for a given process, certain transitions are directly coupled with certain transitions in other processes. However, it is unclear from Bochmann's papers which transitions should be coupled and which should not. More importantly, Bochmann provided no means for preserving parallel execution of coupled transitions. Conversely, in PDCPD the intention functions specify the linkage between initiation and reception of each transition in such a way that no transition is exempt from such a specification. In addition, the coupling relationship between communicating processes is preserved by the synchronization mechanism (to be discussed in Section 3.6) of PDCPD so that this concept becomes useful under all possible
conditions that could arise during the course of communication.

3.3.2. Consistency

Therefore, specifying a communication protocol in PDCPD consists of two parts: (1) a set of FSA describing the behavior of individual processes; and (2) the intention functions linking transitions between different processes. Because FSA and intention functions are specified independently, they may contradict each other. A contradiction may arise under the following condition: One assumes $P_i$ and $P_j$ are two communicating processes in protocol $\Theta$, and the FSA specifications of $P_i$ and $P_j$ satisfy the following three properties:

(A1) States $s_1$ and $s_2$ are in processes $P_i$ and $P_j$, respectively.
(A2) $P_i$ is active on a transition $t$ at state $s_1$.
(A3) $P_j$ is NOT passive on transition $t$ at state $s_2$.

In contrast, one assumes that the intention function $I_{\Theta,i,j}$ satisfies the property

$$s_2 \in I_{\Theta,i,j}(s_1,t);$$

i.e., if process $P_i$ initiates transition $t$ at state $s_1$, it is expected that process $P_j$ will respond to $P_i$ by following $t$ at state $s_2$. However, this contradicts the above assumption (A3) because
(A3) states that $P_j$ cannot follow transition $t$ at state $s_2$.

Due to the above consideration, the specification of a communication protocol is required to satisfy the following assertion:

**Consistency Requirement:** For each state $s_1$ and transition $t$ in $P_i$, and for each state $s_2$ in $P_j$,

$$s_2 \in \mathcal{I}_{@,i,j}(s_1, t) \Rightarrow t \in \mathcal{R}_{@,j}(s_2)$$

If the specification of a protocol satisfies the above consistency requirement, protocol is regarded as **consistent**.

Let $@$ be a protocol which comprises $N$ processes $P_1, \ldots, P_n$. Then $@$ is regarded **symmetric** if it satisfies the following properties:

1. Between each pair of processes $P_i$ and $P_j$, there is a bijective (one-to-one and onto) correspondence function $C_{@,i,j}$ mapping from $\text{STATE}_{@,i}$ to $\text{STATE}_{@,j}$. If $C_{@,i,j}(s) = s'$, $s$ and $s'$ are regarded corresponding. In addition, all the correspondence functions in $@$ satisfy the following properties: (1) For each pair of processes $P_i$ and $P_j$, $C_{@,i,j}$ and $C_{@,j,i}$ are inverse to each other; i.e.,

   $$C_{@,j,i}(C_{@,i,j}(s)) = s \quad \text{and} \quad C_{@,i,j}(C_{@,j,i}(s')) = s'$$

   (2) For every three processes $P_i$, $P_j$ and $P_k$,
\[ C_{@,j,k}(C_{@,i,j}(s)) = s' \Rightarrow C_{@,i,k}(s) = s'. \]

2. For each pair of processes \( P_i \) and \( P_j \),
\[ t \in T_{@,i}(s) \Rightarrow t \in R_{@,j}(C_{@,i,j}(s)). \]

3. For each pair of processes \( P_i \) and \( P_j \),
\[ t \in T_{@,i}(s) \Rightarrow I_{@,i,j}(s,t) = \{ C_{@,i,j}(s) \}. \]

In other words, in a symmetric protocol all the processes have the same "shape". However, if one process is active on an output transition \( t \) at state \( s \), then all the other processes are passive on transition \( t \) at their respective corresponding states.

It is easy to verify that a symmetric protocol, such as the protocol shown in Figure 9, is always consistent.

### 3.4. Reachability Analysis

As noted in Section 2.3, one disadvantage of conventional state transition models is their limited ability in representing the communication channels. A generalized protocol validation technique presented in Section 3.1 includes the channels states into its representation of global state; however, the validation is still restricted to those situations in which a limited number of in-transit messages are allowed at any given time. Consequently, some errors which appear only when more in-transit messages are allowed will go undiscovered in such a validation technique.

A striking difference between PDCPD and conventional state
transition based validation techniques is that in PDCPD channel states are not important to validation. The effects of in-transit messages upon communication are eliminated by the synchronization mechanism which will be introduced in Section 3.6; therefore, the communication channels can be considered empty at all times. Thus, in PDCPD, a global state of a given protocol @ is defined to be the Cartesian product of the states of communicating processes which are involved in @. Assuming that protocol @ comprises n processes \( P_1, \ldots, P_n \), then a global state of @ can be modeled as a vector

\[
< \text{Ps}(1), \ldots, \text{Ps}(n) > ,
\]

where \( \text{Ps}(i) \) denotes the current state of process \( P_i \).

In this section, a simplified protocol validation algorithm based on the enhanced FSA model introduced in Section 3.3 is first presented. The validation algorithm is also based on reachability analysis. From an initial global state, new global states are generated by perturbing each global state reached. This procedure is applied to newly generated states repeatedly until no new states can be generated. The resulting graph is called the desirable interaction graph (DIG). Next, validating the general correctness properties from the DIG is explained. Then, a comparison of the validation complexity of this new validation algorithm with that of West's state perturbation technique is made. Finally, the
relationship between the DIG and the individual processes is discussed.

3.4.1. Reachability Analysis Algorithm

The following is the author's reachability analysis algorithm for validating the general correctness properties of communication protocols. The form of presentation of this algorithm is taken from [28].

**Input**: A consistent protocol @ consisting of n processes $P_1, \ldots, P_n$.

**Output**: Desirable interaction graph of @, denoted by DIG@. Initially DIG@ is empty.

**Steps**: (1) Add a global state $<P_s(1)^0, \ldots, P_s(n)^0>$ to DIG@, where $P_s(i)^0 (1 \leq i \leq n)$ is the initial state of process $P_i$. 

(2) For each global state $<P_s(1), \ldots, P_s(n)>$ in DIG@,
   - if a transition $t \in T_{@, i}(P_s(i))$ directing from state $P_s(i)$ to state $P_s(i)'$ is in $P_i$, and
   - for each process $P_j$ ($P_j \neq P_i$), $P_s(j) \in E_{@, i, j}(P_s(i), t)$ (and, therefore, by consistency, $t \in R_{@, j}(P_s(j))$),
   - then add a global state $<P_s(1)', \ldots, P_s(n)'>$ to DIG@ if no such a state is already in it, and
   - add a global transition labeled $[P_i, t]$ from global state $<P_s(1), \ldots, P_s(n)>$ to global state $<P_s(1)', \ldots, P_s(n)'>$ to DIG@ if no such a transition is already in it.

(3) a. if either a new global state or a new global transition has been added to DIG@ in step (2),
   - then repeat step (2).

   c. if neither a new global state nor a new global transition can be added to DIG@ in step (2),
   - then stop and DIG@ is completed.
If a global transition \([P_i, t]\) is added to a global state \(<P_s(1),...,P_s(n)>\) in \(D\Gamma_G\) by the reachability analysis algorithm, then the meaning of this transition is as follows: According to the meaning of intention function \(I_{\Gamma, i, j}(P_s(i), t)\), if transition \(t\) is initiated by process \(P_i\) at its current state \(P_s(i)\), it is expected that process \(P_j\) will follow \(t\) at its current state \(P_s(j)\). Therefore, global transition \([P_i, t]\) represents a desirable interaction among processes \(P_1,...,P_n\). (This is the reason why the graph generated by the above reachability analysis algorithm is called a desirable interaction graph.) Output global transition \([P_i, t]\) is executed only when \(t\) is initiated by \(P_i\) and followed by the other processes at their current respective states \(P_s(1),...,P_s(n)\).

It must be emphasized, however, that the above condition is an ideal one, which only occurs when the communicating processes are in perfect synchronization. In reality, processes often compete with each other by initiating various transitions, resulting in interactions that are not specified by \(D\Gamma_G\). This point will be discussed further in Section 3.5.

The \(D\Gamma\)'s for the protocols in Figures 9 and 11 are shown in Figures 16 and 17, respectively. Due to the symmetry between the individual processes of these protocols, the resulting \(D\Gamma\)'s, as
shown in Figures 16 and 17, have the same shapes as their individual processes. The DIG for the protocol in Figure 13 is given in Figure 18. The individual processes of this protocol are not symmetric. Interestingly, the resulting DIG of this protocol is even simpler than each of the individual processes. This point will be discussed again in Subsection 3.4.4.

3.4.2. Validating General Correctness Properties

The general correctness properties of protocol @ can be checked from its DIG as follows.

**COMPLETENESS:**

If every global state \( <Ps(1),...,Ps(n)> \) in \( \text{DIG}_@ \) satisfies the following completeness assertion, then protocol @ is complete.

**Completeness Assertion:**

\[
\forall i,j \in \{1,...,n\} \quad \text{if } i \neq j \quad \forall t \in T_{\text{@}}, i(Ps(i)) \cdot (Ps(j) \notin I_{\text{@}}, i,j(Ps(i),t))
\]

The completeness assertion is used to test whether the initiations of some transitions will result in signal reception problems. Assume a global state \( <Ps(1),...,Ps(n)> \) in \( \text{DIG}_@ \) does not satisfy the completeness assertion. That means,

\[
\exists i,j,t \quad \text{if } i \neq j \quad (t \in T_{\text{@}}, i(Ps(i)) \cdot Ps(j) \notin I_{\text{@}}, i,j(Ps(i),t))
\]
Figure 16: DIG for the protocol in Figure 9
Figure 17: DIG for the protocol in Figure 11
Figure 18: DIG for the protocol in Figure 13
Now assume that process $P_i$ initiates transition $t$ at its current state $Ps(i)$ and transmits a signal to process $P_j$ (and every other process) to indicate this initiation. However, the protocol designer does not expect $t$ to be followed by $P_j$ at its current state $Ps(j)$, as expressed by the assertion $\neg I_{i,j}(Ps(i),t)$. Therefore, when process $P_j$ receives the aforementioned signal from $P_i$, $P_j$ may either reject this signal (if $t \not\in R_{i,j}(Ps(j))$) or follow $t$ (if $t \in R_{i,j}(Ps(j))$) violating the intention of the designer. In other words, protocol $@$ is not complete because when the execution of $@$ reaches a reachable global state $<Ps(1),...,Ps(n)>$, the initiation of $t$ by $P_i$ will cause reception difficulty in $P_j$.

The reader may verify that in a symmetric protocol $@$, each global state $<s,C_1,1_1(s),...,C_i,n_i(s)>$ satisfies the completeness assertion.

**DEADLOCK OR TERMINATION:**

A reachable global state $<Ps(1),...,Ps(n)>$ which satisfies the following deadlock assertion represents either a deadlock or a desired terminating state.

**Deadlock Assertion:**

$$\forall i \in \{1,...,n\} \forall t \cdot (t \not\in T_{i,j}(Ps(i)))$$

If a reachable global state satisfies the deadlock assertion
states, then no transition can be initiated by any process (as expressed by the assertion $t_{\mathcal{G},i}(P_s(i))$). In such a case, a deadlock or a terminating condition in $\mathcal{G}$ is detected.

**OTHER CORRECTNESS PROPERTIES:**

Other general correctness properties can also be checked from the DIG. A cyclic sequence of global transitions with no progressive meaning represents the existence of a livelock. Instability is manifested when a sequence of global transitions takes the communicating processes to an asynchronous condition and prevents them from returning to "normal" operation.

3.4.3. Comparison of Validation Complexity

It is apparent that the above reachability analysis algorithm has many similarities as the state perturbation technique described in Section 3.1. Since the channel states are excluded from the new algorithm, however, its complexity is substantially lower than that of the conventional technique. The following example clearly illustrates this point.

Suppose a given protocol $\mathcal{G}$ consists of $n$ communicating processes and $q$ different types of transition. One assumes that each participating process is modeled by a FSA which has $p$ states and each communication channel may contain up to $M$ signals (or
messages) at any time. Then, at each given time, a communication channel can be at any of the \( q^M \) possible states.

Remember that each global state in West's state perturbation technique consists of the states of \( N \) individual processes and \( C(n,2) \) channels. \( C(m,n) \) denotes the \( n \)-combination of an \( m \)-element set. It is assumed that the representation of each individual state occupies one unit of storage space. Then the space complexity of this technique is, therefore, \( O(n^2 * (p^n * q^M * C(n,2)) \) or \( O(n^2 * (p^n * q^M * n^2)) \). During the validation process, assume that each global state will not be generated repeatedly more than a constant number of times. Because each newly generated global state must be compared to the existing global states for identity, the time complexity of West's state perturbation technique, which is a conventional reachability analysis technique, is \( O(n^2 * (p^n * q^M * n^2)^2) \).

In contrast, since the new protocol reachability analysis algorithm does not consider the states of communication channels, the space and time complexities of validation are reduced to \( O(n \times p^n) \) and \( O(n \times p^{2n}) \), respectively, which are significantly lower than those of West's state perturbation technique. The exclusion of channel states from the global state is reasonable in PDCPD because, as noted before, the effects of in-transit messages are eliminated by the synchronization mechanism.
3.4.4. Relationship between Desirable Interaction Graph and the Individual Processes

Before exploring the relationship between the desirable interaction graph (DIG) and individual processes of a given protocol, the following notations must be defined.

Assume that $L$ is a path (see the definition of path in Section 3.2). $|L|$ is used to denote the length of $L$, with each transition having one unit of length. Let $tr(L,k)$ and $init(L,k)$ denote the $k$th transition of $L$ and the initial subpath of $L$ with $|init(L,k)|=k$, respectively, if $k\leq |L|$. Given a protocol $@$ and an individual process $P_i$ in $@$, $state_{@,i}(L)$ is used to represent the resulting state of path $L$. A more complete discussion on path and its related functions can be found in Chapter 5.

For example, $-t_2o+t_3o+t_2$ ($o$ denotes concatenation operation) is a path for process $P_i$ in the protocol given in Figure 9. Let $@$ denote this protocol. Then

\[
\begin{align*}
|t_2o+t_3o+t_2| &= 3 \\
tr(-t_2o+t_3o+t_2,2) &= +t_3 \\
init(-t_2o+t_3o+t_2,2) &= -t_2o+t_3 \\
state_{@,i}(-t_2o+t_3o+t2,2) &= s1
\end{align*}
\]

Let the term of path and the aforementioned functions ($tr$, $init$ and $state$) be defined similarly on the protocol's DIG. Let $L_1$ be a path in the DIG of $@$ and $L_2$ be a path of a process $P_i$ in $@$. Then it can be said that $L_1$ and $L_2$ correspond to each other if they satisfy
either of the following conditions:

(1) Both \( L_1 \) and \( L_2 \) are empty.

(2) \(|L_1| = |L_2|\), and for each \( k \) such that \( 1 \leq k \leq |L_1|\),

\[
((\text{tr}(L_1,k)=[P_j,t] \& i=j) \Rightarrow \text{tr}(L_2,k)=-t)
\]

\&

\[
((\text{tr}(L_1,k)=[P_j,t] \& i\neq j) \Rightarrow \text{tr}(L_2,k)=+t)
\]

The following two theorems establish the relationship between the DIG and the individual processes.

**Theorem 3.1** Assume process \( P_i \) is involved in a given protocol \( \mathcal{G} \). Then, for each path in the DIG of \( \mathcal{G} \), there is a unique corresponding path in \( P_i \).

**Proof:**

Let \( L_1 \) be a path in the DIG of \( \mathcal{G} \). This theorem is proven by induction on the length of \( L_1 \).

The basis, \(|L_1|=0\), is trivial, since an empty path in \( P_i \) corresponds to \( L_1 \).

For induction step, let \( N \geq 0 \) and assume that for each path \( L \) in the DIG of \( \mathcal{G} \) with \(|L|\leq N\), there is a unique corresponding path in \( P_i \). Let \( L_1=L^\prime o[P_j,t] \) be a path in the DIG of \( \mathcal{G} \) with \(|L^\prime|=N\). By using inductive hypothesis, there is a unique path \( L'' \) in \( P_i \) which corresponds to \( L^\prime \). Now that \([P_j,t]\) is an output transition at the
resulting state of $L'$, the step (2) in the reachability analysis algorithm shows that there is an output transition of type $t$ ($-t$ if $P_i = P_j$ and $+t$ otherwise) at state $q_i(\overline{L''})$, and the determinism of $P_i$ implies that there is only one such transition. Then either $\overline{L''}o-t$ or $\overline{L''}o+t$ is the unique path in $P_i$ which corresponds to $L'o[P_j,t]$ in the DIG of $\emptyset$.

**Q.E.D.**

**Theorem 3.2** If process $P_i$ is involved in a given protocol $\emptyset$, then a path in $P_i$ does not necessarily have a corresponding path in the DIG of $\emptyset$.

**Proof:**

This theorem can be proved by an example. The DIG of the protocol given in Figure 13 is shown in Figure 18. From these figures it can be seen that path $-t2o+t2o+t4$ of process $P_1$ has no corresponding path in the DIG.

**Q.E.D.**

In Subsection 3.4.3 an analysis was made demonstrating that if a protocol involves $n$ communicating processes, each consisting of $p$ states, then potentially up to $p^n$ global states may be generated by the reachability analysis algorithm. This means that, even though
the validation complexity of our algorithm is significantly lower than that of conventional reachability analysis techniques, the resulting DIG can still be quite complex. However, Theorem 3.1 suggests that, in reality, the DIG is at most as complex as an individual process. This implies that the actual validation complexity of the new reachability analysis algorithm is so low that the problem of state space explosion is by and large eliminated in PDCPD. This point can be verified from Figures 16, 17 and 18.

Remember that each global transition in the desirable interaction graph represents a cooperation among the communicating processes. Viewing from this aspect, Theorem 3.1 is not surprising because the "image" of cooperation is already captured in each individual process.

Theorem 3.2 suggests that in some protocol certain paths of the individual processes may have no corresponding paths in the protocol's DIG. Since a path in the DIG represents a sequence of intended interactions among communicating processes, a path in a process without a corresponding path is regarded redundant in the design. Redundancy in protocol design is generally used to resolve erroneous conditions. For example, in Figure 13 path \(-t_2o+t_2o+t_4\) in process \(P_1\) has no corresponding path in the DIG given in Figure 18. This path is used to resolved the possible collision of simultaneously initiations of connection requests by both processes.
P₁ and P₂. Redundant paths in the individual processes are not implemented in PDCPD because collisions will be automatically resolved by the synchronization mechanism.

Traditionally, collisions have been resolved by adding extra states and extra transitions into the protocol specification (which introduce redundant paths), as shown by the examples described in Section 3.2. There are, however, at least three disadvantages involved in such an approach:

1. The protocol designer must examine all possible collisions during the design phase, thereby increasing the complexity of his task and the possibility of introducing design errors.

2. The validation techniques of this approach suffer from the problem of state space explosion, thereby limiting their applicability only to those protocols of modest complexity.

3. There is no general rule which tells how the extra states and transitions should be added once collisions are discovered.

Because of the use of synchronization mechanism (see Section 3.6), the above problems are avoided in PDCPD.
3.5. Collisions

As discussed in Sections 3.2 and 3.4, a collision is intuitively a condition in which two or more transitions are simultaneously initiated by different processes. However, in a network environment where processes communicate by exchanging messages, the concept of "simultaneity" becomes vague because each process has its own local time and each message transmitted from one process to another will be delayed for an unspecified amount of time by the communication channels. (A discussion on time and the related concepts in a distributed system can be found in [42].) In this section a simple formal definition of collisions will be given.

For ease of explanation, given a protocol, a tree, which is called execution tree, is used to record all paths in the protocol's desirable interaction graph (DIG). The execution tree of the protocol in Figure 11 is shown Figure 19.

In the following the basic notions of trees are used. The root of a tree is considered as the father of its subtrees; these vertices are, in turn, the sons of the root. The level of a vertex p in a tree T is defined recursively as follows: The level of p is 0 if p is the root of T; otherwise, the level of p is one plus the level of the father of p in T.

More formally, if @ is a protocol which consists of n processes P_1,...,P_n, then the execution tree of @ is a tree which satisfies
Figure 19: Execution tree for the protocol in Figure 11
the following properties:

1. Every vertex in the tree represents a global state of \( @ \).
2. The root of the tree represents the initial global state of \( @ \).
3. Every edge in the tree represents a global transition in the DIG of \( @ \).
4. If a transition \([P_i, t]\) directing from state \( s_1 \) to state \( s_2 \) is in the DIG of \( @ \), then each vertex representing \( s_1 \) has a son representing \( s_2 \) and the edge between these two vertices is labeled \([P_i, t]\).

Terms such as initiation, transition, path, etc. will be used and will mean the same on execution trees as on DIG's.

Having introduced the notion of execution trees, a collision is formally defined as a condition in which two or more transitions are initiated by different processes at the same vertex in an execution tree. Notice that this definition does not necessarily imply that a collision occurs if more than one transition is initiated at the same global state by different processes. The reason is that the same global state may appear at more than one place on a path (e.g., there are loops in the protocol's DIG). If different processes initiate transitions at different vertices along a path, it does not represent a collision even though these vertices represent the same global state.
Given a protocol, since the sequence of transitions taken by a process forms a directed path in the protocol DIG, this sequence is called the path of the process. An interesting observation about collisions is that the paths of two communicating processes diverge if they collide, and the vertex of divergence indicates how they collide. We now use the protocol given in Figure 11 as an example. Assume that the sequences of transitions taken by processes $P_1$ and $P_2$ are $[P_1, t_2]o[P_2, t_2]o[P_3, t_3]$ and $[P_2, t_2]o[P_1, t_2]o[P_3, t_3]$, respectively. As shown in Figure 19, the paths of processes $P_1$ and $P_2$ diverge at the root vertex because they collide at the very beginning.

We have seen errors of incompleteness and ambiguity due to collision in the simple establishment protocols (see Figures 9 and 11). Collisions may also cause errors of deadlock, livelock, etc. The reader is referred to [46] for more examples and other types of errors.

It is important to realize that collisions may not always be a form of error in some protocols. However, from experience with protocol design, it seems every collision will result in some error. Therefore, for the sake of discussion it is assumed that every collision is an error in protocol communication.
3.6. Synchronization Mechanism

In the preceding sections a procedure for designing communication protocols was presented. This procedure starts by specifying the individual processes as a set of finite state automata. Next, the intention functions are specified to define the coupling relationship among communicating processes. Finally, the individual processes are combined and transformed into a desirable interaction graph, which describes the intended communication between participating processes.

As seen from Theorem 3.1, the desirable interaction graph of a protocol is no more complex than any individual FSA. Therefore, it is advantageous to describe the synchronization mechanism using the protocol desirable interaction graph.

It has been mentioned that the function of the synchronization mechanism is to provide the communicating processes with the abilities to detect and resolve collisions automatically. The synchronization mechanism consists of two parts: collision detection and collision resolution.
3.6.1. Collision Detection

As noted in Section 3.5, the paths of two communicating processes diverge upon collision, and the divergent point indicates where they collide. Therefore, the idea of collision detection is to enable each individual process to keep track of its own path as well as the paths of the other processes.

The information for one process to keep track of the path of another process comes from the signals received from the latter. As mentioned in Section 3.2, a process transmits a signal to each of its peer processes upon the initiation of a transition, and this signal must identify the transition that has been initiated. However, in this section it will be shown that recording the path of another process using such a message exchange scenario can be misleading. In order to solve this problem, we find that two items of information, logical time and confirmation, must be added into interprocess communication for a process to record correctly the paths of the other processes.

First, the logical time of a vertex in the protocol's execution tree is defined to be the level of that vertex in the tree (see Section 3.5). (Logical time can also be defined in terms of the paths of the protocol's DIG. The logical time of a path is defined to be the length of that path, with each transition having one unit of length. The reader may verify that these two definitions are
The necessity of logical time can be seen from Figure 20. As illustrated in Figure 20(a), processes A and B collide at vertex NO by initiating transitions [A,e1] and [B,e2], respectively. Let $S_a$ and $S_b$ denote the signals transmitted by A and B to indicate the initiations of these two transitions, respectively. Assume that process A receives signal $S_b$ at vertex N1. Because transition [B,e2], which is indicated by $S_b$, is also an output transition of N1, A will overlook the collision at NO. Similarly, assume that process B receives signal $S_a$ at vertex N2. B will not detect the collision either because transition [A,e1], which is indicated by $S_a$, is also an output transition of N2. Note that further signal exchanges may not reveal this collision. Figure 20(b) demonstrates such a case. Suppose vertices NO, N1 and N2 in Figure 20(a) correspond to global states $s_0$, $s_1$ and $s_2$, respectively, in the protocol's DIG which is depicted in Figure 20(b). Then, both vertices N3 and N4 in Figure 20(a) correspond to the same global state $s_3$ in Figure 20(b) because both the output transitions [B,e2] of state $s_1$ and [A,e1] of state $s_2$ have the same destination state $s_3$. Both processes will reach state $s_3$ without detecting the collision at vertex NO. In such a case, even though further communication can continue, its meaning is questionable, since the processes have failed to detect the collision and collisions are regarded erroneous. However, such an oversight can be avoided if
the logical time of vertex NO is attached to signals S_a and S_b. Therefore, in PDCPD when a process initiates a transition at a vertex, we require that this process appends the logical time of the vertex to the signal which indicates the initiated transition in order to tell the other processes "when" this transition is initiated.

Another item of information necessary for path recording is called confirmation. Confirmation is used by a process to report to the other processes the transition it takes at each (global) state which has more than one output transition. The necessity of confirmation can be seen from Figure 21. Let [A,e1] and [B,e2] be the only output transitions of vertex NO. Assume processes A and B have collided by initiating transitions [A,e1] and [B,e2], respectively, at vertex NO, and [C,e3] is an output transition at both vertices N1 and N2. Now that process C cannot initiate any transition at NO, in the original message exchange scenario it does not transmit any signal to the other processes. Assume that B has now detected the collision at NO. Then, when process B receives a signal indicating the initiation of transition [C,e3] with logical time t+1 at vertex N2, it represents two possibilities:

2. Process C had followed [B,e2] at NO before it initiated
Figure 20: The use of logical time in collision detection
However, the aforementioned ambiguity can be eliminated if process C transmits a signal to indicate the transition it has followed at vertex N0 to the other processes even though C itself does not initiate any transition. Therefore, in PDCPD a process is required to transmit a special signal, which is called confirmation signal, to the other processes to indicate the transition that it has passively followed at each vertex which has more than one output transition.

In Chapter 5 we will prove that the combined use of the logical time and confirmation helps processes correctly record the paths of the others, thereby making collision detection possible.

3.6.2. Collision Resolution

To recover from collisions, a priority assignment and a backtracking scheme are used.

Priority assignment is described as follows: For a given protocol @, the protocol designer is required to give a total ordering among the output transitions of each reachable global state in @. If t is an output transition of global state s, Pty@*(s,t) is used to denote the priority level assigned to transition t of state s. The priority level of each transition should be determined by its importance or urgency. Suppose [A,e1] and [B,e2] are two distinct
Figure 21: The use of confirmation signals in collision detection
output transitions of state s in $\emptyset$. Then $Pty@_g(s,[A,e1])<Pty@_g(s,[B,e2])$ represents that handling $[B,e2]$ is more important or more urgent than handling $[A,e1]$.

Having detected a collision, the processes use a backtracking scheme in addition to the priority assignment to resolve the collision as follows. Assume two transitions $[A,e1]$ and $[B,e2]$ have been initiated by processes A and B at the same vertex N, thereby resulting in a collision. Let the logical time of vertex N be $t$, and let $N$ correspond to global state $s$, where $Pty@_g(s,[A,e1])<Pty@_g(s,[B,e2])$ (i.e., the priority of $[A,e1]$ is lower than that of $[B,e2]$ at state $s$). Then, when a process detects the collision, it takes one of the following actions:

1. If this process has taken $[A,e1]$, it backtracks to the vertex where the collision occurs (i.e., N) by discarding its path beyond that vertex, and then takes $[B,e2]$. In order to allow the other processes to know its backtracking step, it also transmits a confirmation signal to indicate taking transition $[B,e2]$ with logical time $t$.

2. If this process has taken $[B,e2]$, it does not backtrack because transition $[B,e2]$ has precedence over transition $[A,e1]$ at vertex N.

In the example of Figure 20, assume that the priority of transition $[B,e2]$ is higher than that of transition $[A,e1]$ at vertex
NO. Let $S_a$ and $S_b$ denote the signals transmitted by processes A and B when they collide at NO by initiating $[A, e_1]$ and $[B, e_2]$, respectively. Then, when process A receives signal $S_b$, which indicates transition $[B, e_2]$ with logical time $t$, it detects the collision at NO. According to the collision resolution scheme it backtracks to NO, takes the transition $[B, e_2]$ at NO, and transmits a confirmation signal indicating $[B, e_2]$ with logical time $t$. The same backtracking action is taken by every process which has taken transition $[A, e_1]$ at NO when they detect the collision. Thus this collision is resolved. Conversely, when process B receives signal $S_a$, which indicates transition $[A, e_1]$ with logical time $t$, it also detects the collision at NO. However, it does not backtrack because the priority of $[B, e_2]$ is higher than that $[A, e_1]$ at NO. Similarly, any process which has followed transition $[B, e_2]$ at vertex NO is not responsible for resolving the collision.

In summary, the synchronization mechanism of PDCPD can help communicating processes detect and resolve collisions during the protocol execution. Collisions can be detected by path recording. Two items of information, logical time and confirmation, are necessary for correct path recording. Once a collision is detected, a priority assignment and a backtracking scheme are used to resolve it.

Two simple examples (illustrated in Figures 20 and 21) have
been used to demonstrate how communicating processes detect and resolve collisions. One may question, however, the applicability of the synchronization mechanism to other more complex examples. One way to test the usefulness of this mechanism is to apply it to real protocols, as will be demonstrated in CCITT X.21 Recommendation in the next section. Nevertheless, the ultimate question is: Is PDCPD workable on arbitrary protocols. This question will be answered in Chapter 5.

3.7. Application of PDCPD to X.21 Recommendation

The preceding sections have described a method, PDCPD, for designing reliable communication protocols. In this section this method is tested on a real communication protocol: the CCITT's X.21 Recommendation [10].

The CCITT X.21 Recommendation is chosen because of its current interest and also because of its popularity as a test case for formal protocol validation techniques. In this section we first briefly describe X.21 Recommendation and its FSA specification. The validation result of X.21 from several other sources are then presented. Next, the enhancement of X.21 specification is demonstrated, and the desirable interaction graph is derived for X.21 using the reachability analysis algorithm given in Section 3.4. Finally, it is shown how the synchronization mechanism described in
the last section can be used to detect and resolve those errors discovered in X.21.

3.7.1. The Specification of X.21 Recommendation

X.21 Recommendation was proposed by the International Telegraph and Telephone Consultative Committee (CCITT), which is an international group sponsored by the United Nations to provide compatibility in telephone and data communication systems, as a standard means of connecting Data Terminal Equipment (DTE) to Data Circuit-Terminating Equipment (DCE) in a public data network. The specification of X.21 is provided by a textual description, a state diagram, a timing diagram and time limit tables. We are only interested in the logical structure of the call establishment procedure specified by the state diagram.

Four interchange circuits collectively define the state of the system. The DTE uses circuits T and C to transmit data and control information, respectively, to the DCE. The DCE, on the other hand, uses circuits R and I to transmit data and control, respectively, to the DTE. The details of these interchange circuits are not important to this discussion.

West and Zafiropulo derived individual finite state automata from the aforementioned state diagram for the DTE and DCE to define the separate communicating processes for their validation system.
These finite state automata are reproduced in Figures 22 and 23. However, integers are used here instead of the values of interchange circuits to denote transitions in these figures.

Both DTE and DEC can start a connection establishment procedure. If the DTE takes the initiative, it must first initiate a CALL REQUEST (transition of type 1). Upon the acknowledgement of this request (transition of type 2) by the DCE, the DTE indicates the call destination and facilities required (by initiating a transition of type 4). Further exchanges of signals will finally lead the system to the data transfer phase (state 13). If the DCE takes the initiative of connection establishment, it starts by initiating an INCOMING CALL (transition of type 11). The DTE acknowledges this action by initiating a transition of type 4. The following procedure is generally the same as the one initiated by the DTE.

It is important to mention that in X.21 a CALL COLLISION state (state 15 in both processes) is created to handle the possible call collision between the DTE and DCE. In such a collision, the CALL REQUEST by DTE is given precedence over the INCOMING CALL by DCE. This state and the transitions associated with it will be omitted in the following discussion because all collisions are automatically resolved by the synchronization mechanism described in Section 3.6.

Either process can, at any time, initiate a CLEAR REQUEST
Figure 22: Specification of the DTE
Figure 23: Specification of the DCE
transition (type 13) to trigger a connection clear sequence which leads back to the READY state (state 1 in both processes).

Several states are designed for the quiescent phase of X.21. DCE NOT READY (state 18 in DCE) indicates that no service is available. DCE READY states (state 1 and state 21 in DCE) indicate that the network is ready to enter the operational phase. DTE UNCONTROLLED NOT READY (state 21 in DTE) is entered when the DTE is out of operation due to an abnormal condition. DTE CONTROLLED NOT READY (state 14 in DTE) indicates that the DTE is temporarily unable to operate. DTE READY (states 1 and 18 in DTE) represent readiness of the DTE to enter the operational phase.

The reader is referred to [10, 88] for further information about X.21 call establishment procedure.

3.7.2. The Validation Result

Several research groups have worked on the validation of X.21. The validation result was first reported by IBM Zurich Laboratory [88]. That validation was based on a technique called duologue-matrix analysis due to Zafiropulo [90], and later it was repeated by the state perturbation technique described in Section 3.1. Inspired by the work of IBM, a group of researchers at UCLA used a Petri-net like model, which was called SARA, to perform the same task [65, 66]. Recently, Umbaugh and Liu at the Ohio State
University used the Transmission Grammar, which was described in Chapter 2, to define the control structure of X.21 and verified this protocol with an automated validation system based on the state perturbation technique [85].

In general, the results by the above three groups agree with each other. Approximately there were 30 errors discovered in X.21. According to [88], these errors can be divided into three categories.

The first category consists of eight errors which are the results of collisions that arise when either the DTE or DCE is initiating a connection establishment procedure while the other is initiating a transition to one of \texttt{NOT READY} states.

The second category consists of 10 errors which result from collisions that arise when one process is initiating a connection clear procedure while the other is initiating another transition.

The third category consists of 11 miscellaneous errors which are also results of collisions between the DTE and DCE.

3.7.3. Desirable Interaction Graph of X.21 Recommendation

Using the result of West and Zafiropulo's specification of the DTE and DCE (see Figures 22 and 23), the specification can be enhanced by finding the elements of sets T's and R's for both the DTE and DCE and by specifying the intention functions between these
two processes.

The elements of sets $T_{X21,DTE}$, $T_{X21,DCE}$, $R_{X21,DTE}$ and $R_{X21,DCE}$ can be directly derived from the finite state automaton representations of the DTE and DCE. In order to define the intention functions, one must first understand the purpose of each transition. The results are listed in Figure 24.

The desirable interaction graph of X.21 is depicted in Figure 25. Due to the symmetry (see Section 3.3.2) between the DTE and DCE, this desirable interaction graph has the same shape as each of the individual processes.

The priority assignment of the conflicting transitions is shown in Figure 26. As discussed in Subsection 3.7.1, output transitions $[DCE,13]$, $[DTE,13]$ and $[DTE,14]$ of global state $<1,1>$ represent some urgent conditions in the network; therefore, they are assigned higher priorities over the other two output transitions $[DTE,1]$ and $[DCE,11]$ of $<1,1>$. Among those three privileged transitions, $[DCE,13]$ indicates that network fault conditions or network test loops are activated, forcing the whole network out of service. It is believed that handling these conditions takes higher priority than handling the conditions indicated by $[DTE,13]$ and $[DCE,14]$; hence, the highest priority is assigned to $[DCE,13]$ at state $<1,1>$. The priority order between $[DTE,13]$ and $[DTE,14]$ is unimportant because the DTE cannot initiates both transitions at the same time;
**DTE**

<table>
<thead>
<tr>
<th>DTE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{x21,DTE}(1) = {1, 13, 14}$</td>
<td>$R_{x21,DTE}(1) = {11, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(2) = {13}$</td>
<td>$R_{x21,DTE}(2) = {2, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(3) = {3, 13}$</td>
<td>$R_{x21,DTE}(3) = {13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(4) = {4, 13}$</td>
<td>$R_{x21,DTE}(4) = {13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(5) = {13}$</td>
<td>$R_{x21,DTE}(5) = {5, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(6A) = {13}$</td>
<td>$R_{x21,DTE}(6A) = {6, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(6B) = {13}$</td>
<td>$R_{x21,DTE}(6B) = {6, 7, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(6C) = {13}$</td>
<td>$R_{x21,DTE}(6C) = {8, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(7) = {13}$</td>
<td>$R_{x21,DTE}(7) = {5, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(8) = {4, 13}$</td>
<td>$R_{x21,DTE}(8) = {13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(9) = {13}$</td>
<td>$R_{x21,DTE}(9) = {12, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(10) = {13}$</td>
<td>$R_{x21,DTE}(10) = {5, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(10BIS) = {13}$</td>
<td>$R_{x21,DTE}(10BIS) = {5, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(11) = {13}$</td>
<td>$R_{x21,DTE}(11) = {9, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(12) = {10, 13}$</td>
<td>$R_{x21,DTE}(12) = {10, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(13) = {13}$</td>
<td>$R_{x21,DTE}(13) = {13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(14) = {8}$</td>
<td>$R_{x21,DTE}(14) = \emptyset$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(16) = \emptyset$</td>
<td>$R_{x21,DTE}(16) = {13}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(17) = \emptyset$</td>
<td>$R_{x21,DTE}(17) = {8}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(18) = \emptyset$</td>
<td>$R_{x21,DTE}(18) = {8}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(19) = {13}$</td>
<td>$R_{x21,DTE}(19) = \emptyset$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(20) = \emptyset$</td>
<td>$R_{x21,DTE}(20) = {8}$</td>
</tr>
<tr>
<td>$T_{x21,DTE}(21) = {8}$</td>
<td>$R_{x21,DTE}(21) = \emptyset$</td>
</tr>
</tbody>
</table>

**DCE**

<table>
<thead>
<tr>
<th>DCE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{x21,DCE}(1) = {11, 13}$</td>
<td>$R_{x21,DCE}(1) = {1, 13, 14}$</td>
</tr>
<tr>
<td>$T_{x21,DCE}(2) = {2, 13}$</td>
<td>$R_{x21,DCE}(2) = {13}$</td>
</tr>
<tr>
<td>$T_{x21,DCE}(3) = {13}$</td>
<td>$R_{x21,DCE}(3) = {3, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DCE}(4) = {13}$</td>
<td>$R_{x21,DCE}(4) = {4, 13}$</td>
</tr>
<tr>
<td>$T_{x21,DCE}(5) = {5, 13}$</td>
<td>$R_{x21,DCE}(5) = {13}$</td>
</tr>
</tbody>
</table>

Figure 24: Specification of X.21. (a) Elements of T's and R's.  
(b) Definition of intention function
<table>
<thead>
<tr>
<th>T_{X21,DCE}(6A)</th>
<th>RX_{X21,DCE}(6A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{6,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(6B)</td>
<td>RX_{X21,DCE}(6B)</td>
</tr>
<tr>
<td>{6,7,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(6C)</td>
<td>RX_{X21,DCE}(6C)</td>
</tr>
<tr>
<td>{8,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(7)</td>
<td>RX_{X21,DCE}(7)</td>
</tr>
<tr>
<td>{5,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(8)</td>
<td>RX_{X21,DCE}(8)</td>
</tr>
<tr>
<td>{13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(9)</td>
<td>RX_{X21,DCE}(9)</td>
</tr>
<tr>
<td>{12,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(10)</td>
<td>RX_{X21,DCE}(10)</td>
</tr>
<tr>
<td>{5,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(10BIS)</td>
<td>RX_{X21,DCE}(10BIS)</td>
</tr>
<tr>
<td>{5,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(11)</td>
<td>RX_{X21,DCE}(11)</td>
</tr>
<tr>
<td>{9,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(12)</td>
<td>RX_{X21,DCE}(12)</td>
</tr>
<tr>
<td>{10,13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(13)</td>
<td>RX_{X21,DCE}(13)</td>
</tr>
<tr>
<td>{13}</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(14)</td>
<td>RX_{X21,DCE}(14)</td>
</tr>
<tr>
<td>\emptyset</td>
<td>{8}</td>
</tr>
<tr>
<td>T_{X21,DCE}(16)</td>
<td>RX_{X21,DCE}(16)</td>
</tr>
<tr>
<td>{13}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T_{X21,DCE}(17)</td>
<td>RX_{X21,DCE}(17)</td>
</tr>
<tr>
<td>{8}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T_{X21,DCE}(18)</td>
<td>RX_{X21,DCE}(18)</td>
</tr>
<tr>
<td>{8}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T_{X21,DCE}(19)</td>
<td>RX_{X21,DCE}(19)</td>
</tr>
<tr>
<td>\emptyset</td>
<td>{13}</td>
</tr>
<tr>
<td>T_{X21,DCE}(20)</td>
<td>RX_{X21,DCE}(20)</td>
</tr>
<tr>
<td>{8}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T_{X21,DCE}(21)</td>
<td>RX_{X21,DCE}(21)</td>
</tr>
<tr>
<td>\emptyset</td>
<td>{8}</td>
</tr>
</tbody>
</table>

(a)

**DTE**

\[ \forall e,s \cdot (e \in T_{X21,DTE}(s) \Rightarrow I_{X21,DTE,DCE(s,e)} = \{s\}) \]

**DCE**

\[ \forall e,s \cdot (e \in T_{X21,DCE}(s) \Rightarrow I_{X21,DCE,DTE(s,e)} = \{s\}) \]

(b)

Figure 24 (continued)
therefore, one arbitrarily assigns \( \text{Pty}_{X_{21}}(<1,1>,[\text{DTE},13]) < \text{Pty}_{X_{21}}(<1,1>,[\text{DTE},14]) \). (\( \text{Pty}_{X_{21}}(s,t) \) denotes the priority level assigned to transition \( t \) of state \( s \).) We preserve the original proposal for resolving the collision between transitions CALL REQUEST and INCOMING CALL by assigning \([\text{DTE},1]\) higher priority over \([\text{DCE},11]\) at global state \(<1,1>\). The rationale of the other priority assignments are omitted here in order to shorten this discussion.

3.7.4. Collision Resolution in X.21 Recommendation

As discussed in Subsection 3.7.2, all the errors discovered in X.21 Recommendation are caused by collisions. Except the aforementioned CALL REQUEST and INCOMING CALL collision, no solution for handling other collisions was provided in [10].

One such error, as particularly mentioned in [88], occurs under the following condition. At the global state \(<1,1>\), a collision arises as the DTE initiates transition 14 and moves to state 8 (in Figure 22) while the DCE initiates transition 11 and moves to state 14 (in Figure 23). However, there is no provision for receiving a signal indicating transition 11 at state 14 in Figure 23, nor is there any provision for the reception of a signal indicating transition 14 at state 8 in Figure 22. Obviously this is an error of incompleteness.
Figure 25: Desirable interaction graph of X.21 Recommendation
<table>
<thead>
<tr>
<th>Global State</th>
<th>Priority Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>[DCE,13]&gt;[DTE,13]&gt;[DTE,14]&gt;[DTE,1]&gt;[DCE,11]</td>
</tr>
<tr>
<td>(i.e., ( \text{Pty}_{X21}(1,1,\text{[DCE,13]}) ))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{Pty}_{X21}(1,1,\text{[DTE,13]}) ))</td>
</tr>
<tr>
<td></td>
<td>( \ldots ))</td>
</tr>
<tr>
<td>2,2</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,2]</td>
</tr>
<tr>
<td>3,3</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,3]</td>
</tr>
<tr>
<td>4,4</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,4]</td>
</tr>
<tr>
<td>5,5</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,5]</td>
</tr>
<tr>
<td>6A,6A</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,6]</td>
</tr>
<tr>
<td>6B,6B</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,6]&gt;[DCE,7]</td>
</tr>
<tr>
<td>6C,6C</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,8]</td>
</tr>
<tr>
<td>7,7</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,5]</td>
</tr>
<tr>
<td>8,8</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DTE,4]</td>
</tr>
<tr>
<td>9,9</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,12]</td>
</tr>
<tr>
<td>10,10</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,5]</td>
</tr>
<tr>
<td>10BIS,10BIS</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,5]</td>
</tr>
<tr>
<td>11,11</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DCE,9]</td>
</tr>
<tr>
<td>12,12</td>
<td>[DTE,13]&gt;[DCE,13]&gt;[DTE,10]&gt;[DCE,10]</td>
</tr>
<tr>
<td>13,13</td>
<td>[DTE,13]&gt;[DCE,13]</td>
</tr>
</tbody>
</table>

Figure 26: Priority assignment of X.21
The synchronization mechanism of PDCPD provides an easy solution to this collision (see Figure 27) as well as the others. The above collision will be detected and resolved by the communicating processes in the following way: Transition [DTE,14] has been assigned with a higher priority over transition [DCE,11] at the global state <1,1> because the former transition represents some urgent condition in the network. Assume the logical time associated with this collision is \( t \), which is appended to the control signals transmitted by the processes at the state <1,1>. When the DTE receives the signal from the DCE indicating the initiation of [DCE,11] with logical time \( t \), the DTE detects the collision at node N1 (in Figure 27). This signal is ignored because the priority of transition [DTE,14] is higher than that of transition [DCE,11] at N1, which corresponds to global state <1,1>, and the DTE expects the DCE to backtrack to N1 and follow [DTE,14]. On the other hand, the DCE also detects the collision at node N1 when it receives the signal from the DTE indicating the initiation of [DTE,14] with logical time \( t \). The DCE then backtracks to N1 and follows [DTE,14]. The DCE also reports its backtracking step by transmitting a confirmation signal for [DTE,14] and \( t \) to the DTE. Hence, the collision is resolved.

The approach used by [66, 88, 85] as well as other conventional state transition techniques to resolve collisions is to introduce
Figure 27: An example of collision resolution in X.21
extra states and extra transitions into the protocol. As mentioned in Section 3.4, a serious drawback of such an approach is that there is no rule to tell how extra states and transitions should be introduced once collisions are discovered through protocol validation. In addition, the modified protocol must be validation again, and this process must be repeated until all collisions have been detected and resolved. Another disadvantage of such an approach is that the protocol becomes more and more complicated; thereby increasing the possibility of introducing more errors.

The above problems associated with conventional state transition techniques can be avoided in PDCPD because, with the use of synchronization mechanism, collisions are automatically resolved during the protocol execution.

3.8. Summary

In this chapter the problem of designing communication protocols has been considered. The method, Priority Driven Communication Protocol Design (PDCPD), proposed in this chapter possesses the advantage of conventional state transition techniques, but validating the general correctness properties of a protocol in PDCPD is much less complex. PDCPD exploits the coupling relationship between communicating processes, specified as the intention functions, to define the desirable interactions among
Collisions, the conditions that violate these intended interactions, are detected and resolved by the synchronization mechanism. It has been found that, in order to correctly detect collisions, two items of information, logical time and confirmation, must be added into communication signals. The priority scheme and the backtracking scheme have been shown to be capable of resolving detected collisions. The success in applying PDCPD to CCITT X.21 Recommendation demonstrates the usefulness and advantages of this new protocol design method.
4. PROTOCOL SEMANTICS AND VERIFICATION OF CONCURRENT PROGRAMS

In the previous chapter a new method for designing communication protocols was presented. We are concerned with protocols that possess some properties such as completeness to handle all conditions that may arise, freedom from deadlock, proper termination, etc. These properties are essential to almost every protocol; therefore, they are referred to as the "general correctness properties" or "syntax" of communication protocols. Since the method aims only at achieving these properties, it is applicable to any protocol without regard to the services provided by the protocol.

In order to fully verify the correctness of a protocol, it is necessary to develop a method of examining the special properties possessed by the protocol or the special functions performed by the protocol. This aspect of correctness properties is generally regarded as the "semantics" of the protocol. In this chapter an axiomatic basis for verifying the semantics of communication protocols will be proposed. Because verifying semantics of a software system is generally a complicated task, the primary goal is to reduce the complexity in verification.
This chapter is organized as follows: Section 4.1 will be devoted to discussing the rationale of using axiomatic semantics. Hoare's *Communicating Sequential Processes* (CSP) will be used [37] as the high-level language for describing protocols. The notation of CSP will be presented in Section 4.2. In Sections 4.3 several existing axiomatic verification systems for CSP will be described with a comparison of their strengths and weaknesses. In Section 4.4 the new axiomatic verification system for CSP will be presented. Discussion will be made as to how the difficulties inherent in the other systems can be overcome. In Section 4.5 several case studies of the new verification system will be demonstrated. Finally a summary of this chapter will be given in Section 4.6.

### 4.1. Programming Language Models for Protocol Semantics: Principle and Practice

As noted in Chapter 2, traditionally programming language models have been used to specify and verify the semantic aspect of communication protocols. Programming language models are motivated by the observation that protocols are simply one type of algorithm, and that high-level programming languages provide a clear and relatively concise way of describing algorithms [7, 78]. An advantage of these models over state transition models lies in their ease in handling variables and parameters which may take on a large number of values. An even greater advantage of programming language
models, in the author's opinion, is that one can assign "meaning" to protocols by augmenting the programs that implement protocols with some assertions. Floyd/Hoare style program verification techniques provide a rigorous, systematic way to handle a wide range of assertions that express the desired protocol properties.

However, the current program verification techniques for reasoning about protocols (or more generally, programs) rely too much dependence on human ingenuity and intuition. The worth of program verification has raised controversy among computer scientists. One extreme consists of those who claim that program verification systems are useless because these systems could have logical defects, that "proofs" (proof is synonymous with verification in the following discussion) of large programs are so complicated that they invariably contain errors, that most systems and consequently their proofs continually change, and that for many systems it is difficult to write formal specifications amenable to verification [20, 57]. The other extreme consists of those who claim that only by using rigorous, mathematically based methods can one expect to understand complex software systems and prove the systems correct [22].

The author's attitude on this controversy is close to that taken by Gregory R. Andrews [1]. That is, even though the end goal of verification, i.e., formal verifications of complex systems, may
at this time be far-away, the methods required to even approach the end goal are in and of themselves valuable. By applying the concepts used by program verification techniques, one may gain much insight into not only program properties, but the relationship between programming methodologies and programs, which may then lead to development of better and more reliable software systems.

The orientation taken in this research has been formed partially from existing applications of verification techniques to real software systems and partially from the author's personal experience with program verifications. One successful application of verification techniques was UCLA Unix Security Kernel Project [87]. The object of that project was to develop a version of Unix that could both provide the same user services as the original Unix and be demonstratively secure in the sense that users have access only to those objects to which they are given explicit authorization. Interestingly, the goal of verification of this project greatly influenced the design of the kernel. Specifically, the design began with assertions about desired security. Then the kernel was implemented to preserve these assertions. A valuable lesson learned from this project was that, even though the current program verification techniques are still difficult to use, the goal of verification led to a well-structured system which is more secure than Unix by itself.
Our experience with program verification was primarily obtained from the proof of Priority Driven Communication Protocol Design (PDCPD), which was discussed in the previous chapter. The proof of PDCPD presented a challenge of reducing the complexity in verification. The complexity of interactions among communicating processes made it extremely difficult, if not impossible, to use the existing program verification techniques. Our efforts led to the development of a new verification system that allows the proof of an individual process to effectively utilize the properties of the other processes.

However, the thesis of this section is that many benefits have been obtained from the research of program verifications. These benefits are sometimes beyond the narrow goal of verifying certain properties of a given software systems. From applications of verification techniques, one not only increases his confidence in the correctness of a system, but also learns many valuable aspects of developing a "good" system. Undoubtedly, current techniques for program verification need continuous employment and improvement before they can be truly useful. The efforts reported in this chapter represent attempts to achieve this goal.
4.2. The Notation of CSP

In a programming language model a communication protocol is specified as a concurrent program, with each component sequential program describing one of the communicating processes. Verifications traditionally follow the inductive assertion approach developed by Floyd [23] and Hoare [37] and have been extended to concurrent programs by various researchers [2, 30, 39, 44, 54, 58].

Various high-level programming languages and concepts have been employed for the purpose of protocol specifications, viz., "free style" PASCAL [3], Algol-like language [41], data abstractions [72, 73], Concurrent PASCAL [77], and numerous pseudo languages. Along with these languages, a number of verification techniques have been used to verify protocol semantics. In this dissertation there is no attempt to evaluate these languages or verification techniques, but rather to present a notation used in regard to protocol specification and verification.

The language used for protocol specification is Communicating Sequential Processes (CSP) which was introduced by C. A. R. Hoare [37]. CSP is a language constructed for describing interprocess communication through message exchange instead of shared memory. This feature matches the environment in which communication protocols operate. Also, by using CSP's nondeterministic alternative commands, repetitive commands and I/O commands, the
interactions between communicating processes can be specified in a concise way. Therefore, CSP is adopted as the language for specifying communication protocols in this dissertation.

In this section the notation of CSP is informally presented so as to get an intuitive grasp of this language. The reader is referred to [37] for further details.

The behavior of a communicating systems is specified by CSP commands. A CSP command can be either a simple command or a structured command. In the following two subsections these commands will be presented in a form similar to that given in [45].

4.2.1. Simple Commands

There are four types of simple commands in CSP. Their syntax is listed in Table 2.

<table>
<thead>
<tr>
<th>Type of command</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>null command:</td>
<td>skip</td>
</tr>
<tr>
<td>assignment command:</td>
<td>$x_1, ..., x_n := e_1, ..., e_n$ or $\overline{x} := \overline{e}$</td>
</tr>
<tr>
<td>input command:</td>
<td>$P?\overline{e}$</td>
</tr>
<tr>
<td>output command:</td>
<td>$P!\overline{x}$</td>
</tr>
</tbody>
</table>

Table 2: Simple commands of CSP.
A null command has no effect on programs. It corresponds to the null statement in Algo60 [56].

An assignment command is executed as follows: The values of expressions $e_1, \ldots, e_n$ are first computed. Then, the results are stored in the variables $x_1, \ldots, x_n$ correspondingly (by position). The value of $e_i$ ($1 \leq i \leq n$) must match in type the target variable $x_i$, and $\vec{x}$ and $\vec{e}$ will be used to represent the vectors of $x_1, \ldots, x_n$ and $e_1, \ldots, e_n$, respectively.

Input and output commands specify the communication between two concurrently operating sequential processes. Communication occurs between two processes $A$ and $B$ whenever (1) execution of $A$ reaches an output command $B!\vec{e}$; (2) execution of $B$ reaches an input command $A?\vec{x}$; and (3) $\vec{x} := \vec{e}$ constitutes a legal assignment command. Under these conditions, the input command and the output command are regarded to be semantically corresponding, and the effect of executing these commands is to assign the value of $\vec{e}$ of the output command to the target variable $\vec{x}$ of the input command.

This should be contrasted to a weaker concept of syntactically corresponding communication commands. Each output command $B!\vec{e}$ in process $A$ and each input command $A?\vec{x}$ in process $B$ form a pair of syntactically corresponding pair regardless of whether or not actual communication will occur between them. An example of a syntactically corresponding pair but not semantically corresponding
can be found in Subsection 4.3.1.

It is worth mentioning that in [37] semantically corresponding communication commands must be executed simultaneously; i.e., a transmitting process is delayed until some process is ready to receive the signal. This assumption implies that an implementation of CSP must have extra information exchanges from which two communicating processes may know that they have reached a pair of semantically corresponding communication commands. In network environments, however, this assumption is rather unrealistic, because it will cause much communication overhead. Therefore, this assumption is not taken in this dissertation, and modification of the execution of communication commands is as follows: If process B is executing an input command A?x, then B is blocked until a signal (message) is received from process A. On the other hand, if A is executing an output command B!e, then A simply transmits a signal (message) which carries the value of e to process B without regard to whether B has reached a semantically corresponding input command A?x.
4.2.2. Structured Commands

Structured commands are formed from simple commands. There are four types of structured commands in CSP. Their syntax is listed in Table 3.

<table>
<thead>
<tr>
<th>Type of command</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>command list:</td>
<td>( S_1; \ldots; S_n )</td>
</tr>
</tbody>
</table>
| alternative command:     | \[[ G_1 \rightarrow CL_1 \]
|                          | \[ \ldots \]
|                          | \[ G_n \rightarrow CL_n ]\]               |
| repetitive command:      | \*\[[ G_1 \rightarrow CL_1 \]
|                          | \[ \ldots \]
|                          | \[ G_n \rightarrow CL_n ]\]               |
| parallel command:        | \([P_1::CL_1||\ldots||P_n::CL_n]\)          |

Table 3: Structured commands of CSP

A command list is the most familiar form of structured commands. In order to execute a command list \( S_1; \ldots; S_n \), we first execute \( S_1 \), then \( S_2, \ldots \), and finally \( S_n \).

Alternative and repetitive commands are formed by guarded commands [21]. A guarded command \( G \rightarrow CL \) consists of a guard \( G \) and a command list \( CL \). A guard, in turn, consists of a boolean expression and a communication part. If the boolean expression is omitted, it is defaulted to be true. The communication can be
either an input command or an output command. If the communication part is omitted, it is defaulted to be a null command. A guard is aborted if its boolean expression is evaluated false. It is blocked if its boolean expression is evaluated true but the communication part is blocked. It is ready if its boolean expression is evaluated true and the communication part is not blocked. A guarded command G \rightarrow CL can be executed only if the guard G is ready.

An alternative command is executed as follows: If all of its component guarded commands are aborted, then the alternative command is aborted. If some guards are blocked but no guards are ready, the alternative command is blocked. If any component guarded command is executable (i.e., some guard is ready), then one of the executable guarded commands will be selected arbitrarily for execution.

Executing a repetitive command is the same as executing its component alternative command except that this alternative command is executed repetitively until this command is aborted. If the component alternative command is aborted, the repetitive command is aborted too.

Executing a parallel command is to execute the component command lists concurrently. Each component command list CL_i represents a participating process and is labeled by a process name P_i. A parallel command terminates if all of its component processes terminate.
4.2.3. Differences from Hoare's CSP

Our CSP differs from Hoare's [37] in four respects.

First, Hoare's CSP assumes simultaneous communications, as described in Subsection 4.2.1. This assumption could result in a cleaner program in some cases. However, as mentioned before, such an assumption will incur undesirable communication overhead. Therefore, this assumption is not taken in our CSP.

Secondly, Hoare allows only an input command in the communication part of a guard. This restriction is eliminated by allowing output commands to appear in the communication part of guards because that will simplify many CSP programs.

Thirdly, Hoare assumes that an alternative command fails (i.e., the execution of program stops) if all the component guarded commands are aborted. Our CSP assumes that the alternative command is aborted under such a condition, as described in Subsection 4.2.2. Such a modification will simplify the semantics.

Finally, Hoare's CSP assumes a repetitive command is aborted automatically if, first, all its component guarded commands are blocked due to input guards and, second, all processes that the repetitive command is waiting for have terminated. We do not include this feature because it not only complicates the semantics but introduces operational overhead. Instead, we assume that a repetitive command is blocked indefinitely if the aforementioned
condition occurs.

4.3. Current Verification Systems for CSP

Numerous methods for verifying computer programs have been introduced during the last decade. The most widely used method is based on the inductive assertion approach developed by Floyd and Hoare [23, 36]. Other popular approaches include Dijkstra's predicate transformer logic [22], program testing [26] and symbolic execution [9, 40]. We will adapt the first method, which is usually referred to as the axiomatic method, for verifying CSP programs.

In the axiomatic method a proof of a program with command list S consists of an "annotated" program {p} S {q} derived in a deductive system comprising axioms and rules of inference that define the semantics of commands used in S. Intuitively, {p} S {q} means that the execution of S begun in a state satisfying assertion p will terminate in a state satisfying assertion q, provided that S terminates. We call p the precondition of S because it states the assumptions under which S begins to execute, and q the postcondition of S because it results from the execution of S, given p. Such a proof system is often regarded as a system dealing with partial correctness of programs because it does not consider the possibility of non-termination (e.g., deadlock, infinite loop). A proof system that is concerned with total correctness must include a scheme for
proving partial correctness and termination. In this dissertation, only proof systems for partial correctness will considered.

Several axiomatic systems for CSP have been proposed during the last few years. In this section their strengths and weaknesses will be discussed. Three typical proof systems for CSP considered here are Levin and Gries' system [45], Apt et al.'s system [2] and Soundararajan's system [76].

All these proof systems reason about programs in a similar manner. First, they verify individual processes in separation. Then, they deduce the desired properties of the complete concurrent program from the proofs of individual processes. One of the essential problems of such a reasoning technique is that the semantics of processes viewed in isolation is inherently incomplete when compared with their semantics in the context of a complete program. The procedure of verifying processes in separation is relatively well-known and standardized to some extent. Therefore, the main challenge is how to tie separate proofs into a meaningful whole. The following discussion will emphasize this challenging aspect.
4.3.1. Axiomatic System of Levin and Gries

A proof of partial correctness of a CSP program in Levin and Gries' system [45] consists of three parts: proofs of individual processes, satisfaction proofs, and non-interference proofs.

Assumptions must be made about the effect produced by executing communication commands while proving individual processes. Satisfaction proofs demonstrate that these assumptions are valid.

Each process of a parallel command must be separated from every other process of the command, in the sense that each variable must be local to a process. Auxiliary variables, however, are allowed to appear in more than one process for ease of verification. (A discussion on the use of auxiliary variables can be found in [58].) Non-interference proofs demonstrate that the use of auxiliary variables does not interfere with the validity of assertions.

The axioms for communication commands are given below:

output command: \( \{p\} A!e \{q\} \),

input command: \( \{p\} A?e \{q\} \),

where \( p \) and \( q \) are arbitrary assertions. These axioms may appear strange because they allow one to deduce any postcondition of the communication command given whatever precondition. For example, the communication axioms may allow wild proof like
which implies that after execution, false is true. However, any postcondition thus introduced will later be checked against some postcondition regarding corresponding communication commands (see the definition of corresponding communication commands in Section 4.2) in satisfaction proof. An arbitrary postcondition will generally fail to pass that proof.

The most serious difficulty of Levin and Gries' system is that every pair of syntactically corresponding communication commands must be checked by a satisfaction proof. Two problems inherent in such a proof scheme are (1) the number of syntactically corresponding communication command pairs between processes grows quadratically with respect to the number of communication commands in each process; and (2) syntactically correspondence of a pair of communication commands does not necessarily imply that communication will ever take place between them. This problem can be seen from the following program taken from [2]. Let $[P_1|P_2]$ be a CSP program where

\[
\begin{align*}
P_1 &::= [ P_2?x \rightarrow \text{skip} \\
& \quad \Box P_210 \rightarrow P_2?x; x:=x+1] \\
P_2 &::= [ P_112 \rightarrow \text{skip} \\
& \quad \Box P_1?z \rightarrow P_111]
\end{align*}
\]
Clearly, $P_2^?x$ of the first guarded command in $P_1$ will never communicate with $P_1^!1$ in $P_2$, nor will $P_2^?x$ of the second guarded command in $P_1$ with $P_1^!2$ in $P_2$. However, Levin and Gries' system requires the aforementioned syntactically corresponding pairs to be examined by satisfaction proofs. Such a problem forces the verifier to use very weak assertions in verifying individual processes, thereby greatly increasing the difficulty of verification. Although the introduction of auxiliary variables may somehow alleviate the difficulty, it is unclear from [45] how this problem should be coped with.

4.3.2. Axiomatic System of Apt, Francez and DeRoever

A proof of partial correctness of a CSP program in Apt, Francez and deRoever's axiomatic system [2] consists of proofs of individual processes and cooperative proofs. The sequential proof for each process follows the axioms and inductive rules introduced for purely sequential constructs of CSP. Then a rule for parallel composition is used to establish joint cooperation between isolated proofs.

This system has many similarities to that of Levin and Gries'. The axioms for communication commands in these two systems are almost the same. Arbitrary postconditions are eliminated by cooperative proof in Apt's system. This new system provides, however, a more specific means to determine semantically
corresponding communication commands. Apt et al. introduce a proof scheme that includes using global invariants. Global invariants are established on the basis of counting the number of communications that actually occur in each process in order to determine whether communication will actually take place between a pair of syntactically corresponding communication commands.

An impression about Apt et al.'s system is that of its complexity. Several extra axioms and rules of inference must be introduced in order to deal with the problems inherent in the use of global invariants. The proofs are generally very complex and difficult to be envisaged for the program verifier.

4.3.3. Axiomatic System of Soundararajan

Proving partial correctness of a concurrent program in Soundararajan's system [76] also begins with proving the properties of the individual processes using the axioms and rules of inference which are applicable to the commands in the individual processes. Then, a rule for parallel composition is used to prove the properties of the entire program which consists of the communicating processes.

A characteristic particular to his system is that while proving a property of an individual process, one must consider it in absolute isolation. Thus, no knowledge of the "expected" behaviors
of the remaining processes in the program may be used when dealing with the proof of an individual process. This characteristic can be seen from its input axiom which is given and explained below.

$$\{\forall u \cdot (P_{\bar{u}}, h_i \circ (j, i, \bar{u}))\} P_j \not\models \{p\}$$

Several notations must be defined to understand what the axiom means. With each process $$P_i$$, one associates a communication history $$h_i$$, the sequence of all communications that $$P_i$$ has so far participated in. We represent $$h_i$$ as a sequence of elements as $$(i, j, \bar{u})$$ (corresponding to a number $$\bar{u}$$ transmitted from $$P_i$$ to $$P_j$$) and elements as $$(j, i, \bar{u})$$ (corresponding to a number $$\bar{u}$$ received by $$P_i$$ from $$P_j$$). We use $$\circ$$ to denote concatenation of an element to the end of a sequence, and use $$p_{\bar{u}}$$ to represent an assertion formed by replacing every free occurrence of $$\bar{x}$$ in $$p$$ by $$\bar{u}$$. Therefore, the aforementioned input axiom can be interpreted as follows. In response to an input command $$P_j \not\models \{p\}$$ in $$P_i$$, every free occurrence of $$\bar{x}$$ in assertion $$p$$ is replaced by the input value $$\bar{u}$$ and the communication history $$h_i$$ is concatenated by the last communication event $$(j, i, \bar{u})$$. The universal quantifier over $$\bar{u}$$ reflects the characteristic of Soundararajan's system; i.e., no idea regarding the actual number that $$P_j$$ may have been transmitted.

An important aspect of Soundararajan's system is its use of history variables. As noted above, the communication history of an
individual process is recorded by the string of values of variable \( h_i \). In this system the interprocess communication is computed by a compatibility function \( \text{compat} \) which has the history variables \( h_1, \ldots, h_n \) as its parameters. The advantage of such an approach is that one can avoid determining the actual interactions that take place among communication processes by checking the cooperation between each pair of syntactically corresponding communication commands.

However, Soundararajan's system has two drawbacks:

1. Because the proofs of individual processes are carried out in absolute isolation, every value must be considered in dealing with an input command. Consequently, the proof of an individual process may be unnecessarily complicated because even impossible input values must be considered.

2. Because no assumption can be made about the input values, the contents of history variables \( h_1, \ldots, h_n \) can be very complex in order to take every possibility into consideration. As a consequence, the evaluation of function \( \text{compat} \) may become very complicated and tedious.
4.4. Proving CSP Programs Using Input and Output Assertions

Having seen several existing proof techniques for CSP, we now present a new axiomatic system for CSP. The goal is to reduce the complexity involved in verifying CSP programs. As discussed in the previous section, existing verification techniques for CSP suffer from the problem of complexity. For Levin and Gries' system and the system of Apt et al., the complexity comes from the satisfaction or cooperation proof in combining the proofs of individual processes. For Soundararajan's system, the complexity is inherent in the proofs of individual processes and the evaluation of function $\text{compat}$. In the new system, we try to incorporate the advantages of the current systems while avoiding their drawbacks.

Verifying a set of communicating processes in the new system also consists of two parts:

1. Developing proofs for each participating process. The proof of an individual process takes care of the process' effect on its local variables and the communication channels incident to and from the process.

2. Combining the proofs of individual processes to deduce the desired properties of the entire CSP program.

The proofs of the individual processes basically follow the standard Floyd/Hoare style techniques for proving sequential programs. Assumptions can be made regarding the effects of
communication commands.

While combining the individual proofs, our axiomatic system provides a means for identifying the assumptions made by each individual proof and provides a rule of inference to examine whether these assumptions support one another.

The rest of this section relates details of the new axiomatic system. An arbitrary concurrent CSP program \( [P_1||...||P_n] \) which consists of \( n \) communicating processes will be considered. Without losing generality one may assume that each process is strictly sequential. The principle which is used by the new system to reduce verification complexity will be discussed in Subsection 4.4.1. In Subsection 4.4.2 the axioms and rules of inference applicable to the proofs of individual processes will be considered. In Subsection 4.4.3 the rule of inference for parallel composition (i.e., combining the individual proofs to deduce the properties of the entire concurrent program) will be dealt with. Finally, several general inference rules which are applicable to both individual processes and the entire concurrent program will be listed in Subsection 4.4.4.
4.4.1. Reducing Complexity in Proving CSP Programs

There are two special problems involved in verification of concurrent programs.

First, within an individual process, the effect produced by executing an input command is uncertain because the value that will be received is generally unknown to the process. Determining the change of the local process upon executing an input command is essentially the watershed between Soundararajan's axiomatic system and the other two systems. On one hand, Soundarajan's system allows no assumptions in reasoning about input commands, requiring the program verifier to consider all values. On the other hand, the systems of Levin and Gries and of Apt, et al. provide flexibility by allowing the verifier to choose the postcondition for each input command.

Second, an important lesson learned from verifying concurrent programs, as observed by Owicki and Gries [59], is that in order to understand concurrent execution, one must focus on the potential interference between the proofs of the processes rather than the interaction of their statements. Because a formal proof of a process is a precise characterization of the hypothesis and conclusion for each statement, the assumptions made by an individual process must be identified and it must be shown that no assumption is invalidated by the actions of any other process.
At the individual process level, the axiomatic system of Levin and Gries and the system of Apt, et al. are superior to the system of Soundararajan in handling communication commands. As noted above, the first two systems allow the program verifier to choose postconditions in reasoning about input commands, thus helping the verifier eliminate the consideration of impossible input values. On the other hand, the latter system forces the verifier to take all values into consideration when dealing with input commands. Consequently, the proofs of individual processes in the first two systems are generally less complex than those in the latter system.

Because no assumption has been made in dealing with input commands, Soundararajan's system, at the system level, avoids checking the mutual support of assumptions made between corresponding communication commands. In order to eliminate those impossible interactions among processes, as discussed in Subsection 4.3.3, the semantics of the entire concurrent program must be derived from the proofs of individual processes and from the compatibility function. However, the advantage of his system is that compatibility is computed on a process basis, rather than on a command basis, as required by the other two systems. The following example explains this point: Suppose a concurrent program is composed of n communicating sequential processes. For each process, assume there are m input commands and m output commands with every
other process. Then the satisfaction proof in Levin and Gries' system or the cooperative proof in Apt, Francez and deRoever's system requires checking \(2^m m^2 \times C(n,2)\) pairs of syntactically corresponding communication commands. On the other hand, in Soundararajan's system the function \(\text{compat}\) takes \(n\) history variables \(h_1, \ldots, h_n\). If patterns exist in these history variables, the evaluation of \(\text{compat}\) can be significantly simpler than checking the satisfaction or cooperation between every pair of corresponding commands.

The unique characteristic of our axiomatic system is the standardized use of assumptions constructed from history variables. This verification system uses history variables which record the sequences of signals that are input and output of the communicating processes. Figure 28 depicts the scheme. Suppose \(P_i\) and \(P_j\) are two communicating processes. Then history variable \(\text{OUT}_{i,j}\) denotes the sequence of output signals which process \(P_i\) has transmitted to process \(P_j\) whereas variable \(\text{IN}_{j,i}\) denotes the sequence of input signals which \(P_j\) has actually received from \(P_i\). Because communication channels are assumed to be FIFO and error-free, the existence of signal propagation delay implies that \(\text{IN}_{j,i}\) must always be an initial subsequence of \(\text{OUT}_{i,j}\). (Suppose \(A\) and \(B\) are two sequences and the length of \(A\) and \(B\) are denoted by \(|A|\) and \(|B|\). \(A\) is said to be an initial subsequence of \(B\), denoted by \(A \preceq B\), if
In dealing with an input command, the new axiomatic system allows the program verifier to use an input assertion to specify the possible values of input. Input assertion can be viewed as assumed patterns of the input signal sequences. Given a pair of communicating processes, say $P_i$ and $P_j$, $is_{i,j}$ is used to denote the input assertion about the input signal sequence which $P_i$ has received from $P_j$ (i.e., $IN_{i,j}$). Note that $is_{i,j}$ does not tell the exact input value which will be received upon executing an input command, but rather gives a range of values which the receiving process may expect. Note that by allowing the program verifier to specify input assertions for input commands, we have incorporated the advantages of Levin and Gries' system (as well as the system of Apt, et al.) into the new axiomatic system for CSP. With the help of input assertions, consideration of impossible input values may be excluded, thereby reducing the complexity in dealing with input commands.

Because input assertions are used as assumptions in proving individual processes, they must be verified when the individual proofs are combined in order to verify the properties of the entire program. The following scheme is adapted for verifying input assertions: First, the program verifier is required to prove some output assertions as a part of the postcondition of each process.
Figure 28: History variables
Given a pair of processes $P_i$ and $P_j$, $os_{j,i}$ is used to denote the output assertion which specifies the pattern of the output signal sequence which $P_j$ transmits to $P_i$ (i.e. $OUT_{j,i}$). Second, while combining the proofs of processes $P_i$ and $P_j$ together, the rule of inference requires the verifier to prove that $os_{j,i}$ implies $is_{i,j}$ to ensure the validity of $is_{i,j}$.

Since the verification of consistency between input and output assertions are carried out on the process basis, as will be shown in Subsection 4.4.3, another important advantage is gained. Earlier in this subsection an example was given to evaluate the complexity in proving that the proofs of individual processes are interference-free. In that example it was assumed that a concurrent program consists of $n$ communicating processes, with each process consisting of $m$ input commands and $m$ output commands with every other process. Then it was concluded that it takes $2*m^2*C(n,2)$ satisfaction proofs or cooperation proofs if the proofs are carried out at the command level. In contrast, because the new axiomatic system takes only one consistency proof between the input and output assertions between each pair of processes, this system needs only $2*C(n,2)$ proofs to ensure that the individual proofs are interference-free.
4.4.2. Semantics of the Individual Processes

The axioms and rules of inference stated in this subsection are applicable to an individual process $P_i$.

A1. Null command

\{p\} \textbf{skip} \{p\}

A2. Assignment command

\{P_{x \leftarrow v}\} \ x := v \{p\}

The axioms for null command and assignment are well-known and will not be discussed here.

A3. Input command

\{\forall u. (\mathrm{is}_{i,j} \Rightarrow p)_{u, \mathrm{IN}_{i,j}}\} \ P_j \ ? \ x \ {p}\}

The assertion is$_{i,j} \Rightarrow p$ is included in the precondition of input command $P_j \ ? \ x$ because the input assertion is$_{i,j}$ is taken as an assumption on the input values. The effect of executing this command is to receive some value $u$ from $P_j$ and assign $u$ to $x$. Therefore, every free occurrence of $x$ in $p$ should be replaced by $u$. At the same time, the history variable $\mathrm{IN}_{i,j}$ is extended by an element $u$; therefore, every free occurrence of $\mathrm{IN}_{i,j}$ should also be replaced by $\mathrm{IN}_{i,j}\omega$ (where $\omega$ represents concatenation operation).
The universal quantifier over $\bar{u}$ is needed because, in process $P_i$ in which the input command appears, the actual value to be received from process $P_j$ is not known. However, in contrast with the input axiom of Soundararajan's axiomatic system, the axiom for input command of the new system is more amenable to verification because the input assertion $i_{i,j}$ reduces the number of input values needed to be considered.

R1. Output command

\[
p \Rightarrow o_{i,j}
\]

\[
\text{OUT}_{i,j} \hspace{1cm} \{P_{OUT_{i,j}} \} \hspace{1cm} P_j \{p\}
\]

The only effect of executing output command $P_j \{\bar{e}\}$ is to extend the history variable $\text{OUT}_{i,j}$ by an element $\bar{e}$. Therefore, every free occurrence of $\text{OUT}_{i,j}$ is replaced by $\text{OUT}_{i,j} \{\bar{e}\}$. However, in the new axiomatic system one must verify that executing this output command will preserve the output assertion $o_{i,j}$; therefore, a proof of $p \Rightarrow o_{i,j}$ is required.

R2. Command list

\[
\{p\} \hspace{1cm} S_1 \{q\}, \hspace{1cm} \{q\} \hspace{1cm} S_2 \{r\}
\]

\[
\{p\} \hspace{1cm} S_1;S_2 \{r\}
\]

The inference rule for the command list (sequence) is a
conventional one and, therefore, will not be discussed here.

R3. Alternative command

\[
\begin{align*}
\{ p & \land B(G_k) \} \land C(G_k) ; S_k \{ q \}, \quad k=1,\ldots,m \\
(p & \land \neg B(G_1) \land \cdots \land \neg B(G_m) ) \Rightarrow \{ q \}
\end{align*}
\]

\[
\{ p \} [ [ k=1,\ldots,m G_k \rightarrow S_k ] \{ q \},
\]

where \([ k=1,\ldots,m G_k \rightarrow S_k ]\) is an abbreviation of \([ G_1 \rightarrow S_1 \land \cdots \land G_m \rightarrow S_m ]\).

Given a guard \( G_k (k=1,\ldots,m) \), \( B(G_k) \) and \( C(G_k) \) are used to represent the boolean expression and the communication part of \( G_k \).

When dealing with an alternative command \([ k=1,\ldots,m G_k \rightarrow S_k ]\), one must consider the effect of every component guarded command. This consideration is taken into account by the proof of \(\{ p \land B(G_k) \} \land C(G_k) ; S_k \{ q \} (k=1,\ldots,m)\). One must also take into consideration the effect of aborting the above alternative command when all of its component guarded commands are aborted. This consideration is done by the proof of \((p \land \neg B(G_1) \land \cdots \land \neg B(G_m)) \Rightarrow \{ q \} \).

R4. Repetitive command

\[
\begin{align*}
\{ p & \land B(G_k) \} \land C(G_k) ; S_k \{ p \}, \quad k=1,\ldots,m \\
(p & \land \neg B(G_1) \land \cdots \land \neg B(G_m) ) \Rightarrow \{ q \}
\end{align*}
\]

\[
\{ p \} *[ [ k=1,\ldots,m G_k \rightarrow S_k ] \{ q \}
\]
where \( p \) in the above rule of inference is the loop-invariant of the repetitive command \([\Box_{k=1,\ldots,m} G_k \rightarrow S_k]\). (A loop-invariant of a repetitive command is an assertion whose truth is preserved by each iteration of the command.) Proving that \( p \) is the loop-invariant of the above repetitive command is entailed by the proof of \( \{p \land B(G_k)\} \land C(G_k); S_k \{p\} \) \((k=1,\ldots,m)\). Aborting a repetitive command is taken care of in the proof of \( \{p \land \neg B(G_k \land \ldots \land \neg B(G_m))\} \rightarrow \{q\} \).

It is worth mentioning here that the rule of inference for repetitive command is simpler than those of the other axiomatic systems because one has excluded the assumption of automatic abortion when all component guarded commands of the repetitive command are blocked due to input guards and all processes that the repetitive command is waiting have terminated.

4.4.3. Rule for Parallel Composition

The following is the rule of inference for combining the proofs of individual processes and proving the properties of the entire program \([P_1||\ldots||P_n]\). Let \( p_i \) and \( q_i \) denote the precondition and postcondition of process \( P_i \), respectively. The rule for parallel composition has the following form:
R5. Parallel Composition

\[ \forall i, j \in \{1, \ldots, N\} \land j \neq i [\{p_i \land i_{IN_i, j}^i = \text{OUT}_i, j = \Lambda\} \land P_i \{q_i\}] \]

\[ \forall i, j \in \{1, \ldots, N\} \land j \neq i [(i_{IN_i, j} = \text{OUT}_i, j = \Lambda) \implies o_{s_i, j}] \]

\[ \forall i, j \in \{1, \ldots, N\} \land j \neq i [(o_{s_i, j} \land i_{IN_i, j} = \text{OUT}_j, i) \implies i_{s_i, j}] \]

\{p_1 \land \ldots \land p_n\} [P_1 | \ldots | P_n] \{ q_1 \land \ldots \land q_n \}
\& \forall i, j \in \{1, \ldots, n\} \land i \neq j [i_{IN_i, j} = \text{OUT}_j, i] \}

where \(\Lambda\) represents an empty sequence.

This rule of inference states that, in order to prove the assertion \(q_1 \land \ldots \land q_n\) is the postcondition of the entire program \([P_1 \| \ldots \| P_n]\), given the precondition \(p_1 \land \ldots \land p_n\), one must first prove the following three properties:

1. \(p_i\) and \(q_i\) are indeed the precondition and postcondition of process \(P_i\), as entailed by the proofs of \(\forall i, j \in \{1, \ldots, N\} \land j \neq i [\{p_i \land i_{IN_i, j} = \text{OUT}_i, j = \Lambda\} \land P_i \{q_i\}]\).

2. Each output assertion must hold before the execution of the entire program begins. This consideration is taken care of in the proofs of \(\forall i, j \in \{1, \ldots, N\} \land j \neq i [(i_{IN_i, j} = \text{OUT}_i, j = \Lambda) \implies o_{s_i, j}]\). The necessity of these proofs will be explained later.

3. Output assertion \(o_{s_j, i}\) must imply input assertion \(i_{s_i, j}\) when \(\text{OUT}_{j, i} = i_{IN_i, j}\), entailed by the proof of \((o_{s_j, i} \land \text{OUT}_{j, i} = i_{IN_i, j}) \implies i_{s_i, j}\). Recall that \(\text{OUT}_{j, i}\) and \(i_{IN_i, j}\) denote the sequence of signals which process \(P_j\) has
transmitted to process \( P_i \) and the sequence of signals which 
\( P_i \) has received from \( P_j \), respectively. When \( \text{OUT}_{j,i} = \text{IN}_{i,j} \), 
they denote the same sequence of signals. Since both \( \text{os}_{j,i} \) 
and \( \text{is}_{i,j} \) specify the pattern of this sequence, they must be 
consistent. This proof is used to test the consistency 
between \( \text{os}_{j,i} \) and \( \text{is}_{i,j} \), and therefore, is called 
consistency proof.

The consistency proof deserves special attention. As mentioned 
above, \( \text{os}_{j,i} \) and \( \text{is}_{i,j} \) specify the pattern of signals transmitted 
from process \( P_j \) to process \( P_i \), but from different prospects. 
Therefore, the variables used in \( \text{os}_{j,i} \) and \( \text{is}_{i,j} \) must be carefully 
selected. For example, assume a local variable \( x \) of \( P_i \) appears in 
\( \text{is}_{i,j} \). According to the syntax of CSP, \( P_j \) has no access to 
\( x \). Because \( \text{os}_{j,i} \) is a verifiable property in \( P_j \), variable \( x \) cannot 
appear in \( \text{os}_{j,i} \). Hence, proving that 
\( (\text{os}_{j,i} \& \text{OUT}_{j,i} = \text{IN}_{i,j}) \rightarrow \text{is}_{i,j} \) is not possible since \( x \) appears in 
the consequent but not in the antecedent of the above implication.

The above discussion suggests that only those variables which 
are common to both processes \( P_i \) and \( P_j \) should be allowed to appear 
in the input assertion \( \text{is}_{i,j} \). However, as mentioned, there is no 
global variables in CSP. Therefore, one must allow auxiliary 
variables to appear in input and output assertions in order to 
relate the disjoint processes. Auxiliary variables must be
restricted in a way that will affect neither the flow of control of any process nor the value of any non-auxiliary variable. They are used only for the proof, but not the computation. These conditions are ensured, as listed by Levin [45], if auxiliary variables appear only: 1) in assertion; 2) in assignments, where expressions that reference auxiliary variables are assigned only to auxiliary variables; and 3) as parameters in input commands and in expressions as parameters of output commands, where the assignment that corresponds to these corresponding communication commands satisfies condition 2).

The most frequently used auxiliary variables in the new axiomatic system are the history variables (e.g., \( \text{OUT}_{j,i}, \text{IN}_{i,j} \), etc.). When consistency between input and output assertions, i.e., 
\[
(\text{os}_{j,i} \land \text{OUT}_{j,i} = \text{IN}_{i,j}) \Rightarrow \text{is}_{i,j},
\]

is proven, the assertion \( \text{OUT}_{j,i} = \text{IN}_{i,j} \) in the antecedent provides a necessary link between \( \text{os}_{j,i} \) and \( \text{is}_{i,j} \).

Another problem associated with consistency proofs is that they make the new axiomatic system appear insufficient. A simple concurrent program \( [P_1 || P_2] \) is used to illustrate this problem. As mentioned early in this section, input assertions \( \text{is}_{1,2} \) and \( \text{is}_{2,1} \) are assumed properties of in the proofs of individual processes \( P_1 \) and \( P_2 \), while output assertions \( \text{os}_{2,1} \) and \( \text{os}_{1,2} \) must be provable properties in \( P_2 \) and \( P_1 \). Therefore, the proofs of \( \text{os}_{2,1} \) and \( \text{os}_{1,2} \)
will make use of the assumptions $i_{2,1}$ and $i_{1,2}$, respectively. On the other hand, in order to prove the consistency between input and output assertions, $i_{1,2}$ and $i_{2,1}$ must be proved to be implied by $o_{2,1}$ and $o_{1,2}$, respectively. When these proof dependencies are put together, they seem to form a cycle, which is illustrated in Figure 29. The arrow directing from $i_{1,2}$ to $o_{1,2}$ indicates that proving $o_{1,2}$ will make use of $i_{1,2}$, etc. From a rigorous point of view, any axiomatic system which involves cyclic proofs may be unsound; i.e., incorrect results may be "proved".

In the following we will give an informal argument to deduce that the aforementioned cyclic dependency, in fact, does not exist. The proposal here is to give the reader an intuitive understanding of the problem, rather then to develop a logical formalism.

Again the aforementioned two-process program $[P_1||P_2]$ is used for illustration. Let $S(1)$ and $R(1)$ denote the length of history variables $OUT_{1,2}$ and $IN_{1,2}$, respectively. Similarly, let $S(2)$ and $R(2)$ denote the length of $OUT_{2,1}$ and $IN_{2,1}$, respectively.

The argument is based on induction over $S(1)$, $R(1)$, $S(2)$ and $R(2)$.

**Basis.** Let $S(1)=R(1)=S(2)=R(2)=0$, i.e.,

$OUT_{1,2}=IN_{1,2}=OUT_{2,1}=IN_{2,1}$. Recall that in the rule for parallel composition the verifier is required to prove that $\forall i,j \in \{1,\ldots,N\} \& j \neq i \quad [(IN_{i,j}=OUT_{i,j} \Rightarrow A) \Rightarrow o_{i,j}]$. Therefore,
Figure 29: A seemingly cyclic proof dependency among $is_{1,2}$, $os_{1,2}$, $is_{2,1}$ and $os_{2,1}$
\((\text{IN}_{1,2} = \text{OUT}_{1,2} = \lambda) \Rightarrow \text{os}_{1,2}\) and 
\((\text{IN}_{2,1} = \text{OUT}_{2,1} = \lambda) \Rightarrow \text{os}_{2,1}\).

Hence, one knows that \(\text{os}_{1,2}\) and \(\text{os}_{2,1}\) are true in this case \((S(1) = R(1) = S(2) = R(2) = 0)\). By the consistency proofs, i.e.,

\((\text{os}_{2,1} \land \text{OUT}_{2,1} = \text{IN}_{1,2}) \Rightarrow i_{1,2}\) and 
\((\text{os}_{1,2} \land \text{OUT}_{1,2} = \text{IN}_{2,1}) \Rightarrow i_{2,1}\),

one knows that \(i_{1,2}\) and \(i_{2,1}\) are also true.

**Induction** Assume that \(\text{os}_{1,2}, i_{1,2}, \text{os}_{2,1}\) and \(i_{2,1}\) are true for all cases in which \(S(1) \leq N_1, R(1) \leq M_1, S(2) \leq N_2\) and \(R(2) \leq M_2\). Consider the following conditions:

- **INPUT:** Assume executing an input command in process \(P_1\) will result in a case in which \(S(1) = N_1, R(1) = M_1 + 1, S(2) \leq N_2\) and \(R(2) \leq M_2\). Because \(\text{IN}_{1,2} \leq \text{OUT}_{2,1}\) (i.e., \(M_1 + 1 = N_2\)), by the consistency proofs one knows that \(\text{os}_{1,2}, i_{1,2}, \text{os}_{2,1}\) and \(i_{2,1}\) are also true in such a case. Similarly, it can be proven that \(\text{os}_{1,2}, i_{1,2}, \text{os}_{2,1}\) and \(i_{2,1}\) are true in the cases in which \(S(1) = N_1, R(1) = M_1, S(2) = N_2\) and \(R(2) = M_2 + 1\).

- **OUTPUT:** Assume executing an output command in process \(P_1\) will result in a case in which \(S(1) = N_1 + 1, R(1) = M_1, S(2) \leq N_2\) and \(R(2) \leq M_2\). Because \(\text{os}_{1,2}\) (in cases in which \(S(1) = N_1 + 1,\)
R(1)\leq M_1) is a provable property in P_1, given is_{1,2} and os_{2,1} as true for S(1)\leq N_1 and R(1)\leq M_1, one knows that os_{1,2}, is_{1,2}, os_{2,1} and is_{2,1} are also true in the cases in which S(1)=N_1+1, R(1)=M_1, S(2)\leq N_2 and R(2)\leq M_2. Similarly, it can be proven that is_{1,2}, os_{2,1} and is_{2,1} are true in the cases in which S(1)\leq N_1, R(1)\leq M_1, S(2)=N_2+1 and R(2)\leq M_2.

Hence, one is assured of the validity of our rule of inference for parallel composition.

From the above discussion, it is now clear that the proofs of \( \forall i, j \in \{1, \ldots, N\} \& j \neq i \left[(IN_i, j=OUT_i, j=\Lambda) \Rightarrow os_{i, j}\right] \) are necessary in order to ensure that the verification of a concurrent program is sound.

4.4.4. Other Useful Rules of Inference

The specification of the axioms and rules of inference for the commands that may appear in the individual processes and the rule for parallel composition have now been completed. Several additional logical rules of inference are also included in the new axiomatic system as follows:

R6. Consequence

\[ p \Rightarrow p', \quad \{p'\} S \{q'\}, \quad q' \Rightarrow q \]

-----------------------------------
{p} S \{q\}

**R7. Conjunction**

\[
\begin{align*}
{p} S \{q\}, & \quad {p} S \{q'\} \\
\hline
{p} S \{q \land q'\}
\end{align*}
\]

**R8. Disjunction**

\[
\begin{align*}
{p} S \{q\}, & \quad {p'} S \{q\} \\
\hline
{p \lor p'} S \{q\}
\end{align*}
\]

Rules R6, R7 and R8 apply to commands in the individual processes as well as to the entire program \([P_1||...||P_n]\).

### 4.5. Case Studies

In order to demonstrate the usefulness of our axiomatic system, we now apply it to prove four examples. All these examples are taken from [2], but two of them will be modified in order to accommodate the changes that have been made to CSP.

**Example 1.** Prove that \{true\} \([P_1||P_2||P_3]\) \{x=u\}, where

\[
\begin{align*}
P_1:: & \quad P_2!x, \\
P_2:: & \quad P_1?y;P_3!y, \text{ and} \\
P_3:: & \quad P_2?u.
\end{align*}
\]
Because there is no obvious communication pattern in this simple example, one may "turn off" the input and output assertions by letting

\[ i, j = \text{true, } i, j = 1, 2, 3 \& i \neq j. \]

The proofs of

\[ (\text{IN}_{i,j} = \text{OUT}_{i,j} = \text{true}, \ j = 1, 2, 3 \& j \neq i) \Rightarrow \text{os}_{i,j}, \ i = 1, 2, 3 \]

and

\[ (\text{os}_{j,i} \& \text{OUT}_{j,i} = \text{IN}_{i,j}) \Rightarrow i, j = 1, 2, 3 \& i \neq j \]

are trivial and therefore are omitted here. We now turn to the proofs of individual processes \( P_1, P_2 \) and \( P_3 \).

Because the assertion \( \text{OUT}_{1,2} = x \Rightarrow \text{true} \) is obviously true, by the rule of inference for output command (R1), it can easily be proven that

\[ \{\text{true} \& \text{IN}_{1,2} = \text{true}, \text{OUT}_{1,2} = \text{true}, \text{OUT}_{1,3} = \text{true}\} \Rightarrow P_1!x \{\text{OUT}_{1,2} = x\} \]

Also by the axiom for input command (A3) and rule R3, we can prove that

\[ \{\text{true} \& \text{IN}_{2,1} = \text{true}, \text{OUT}_{2,1} = \text{true}, \text{OUT}_{2,3} = \text{true}\} \Rightarrow P_1?y; P_3!y \{\text{IN}_{2,1} = \text{OUT}_{2,3} = y\} \]

and
\{true \land \text{IN}_{3,1} = \text{IN}_{3,2} = \text{OUT}_{3,1} = \text{OUT}_{3,2} = \text{A}\} \text{ P}_2 ? u \{\text{IN}_{3,2} = u\}

Hence, the proofs of \text{P}_1, \text{P}_2 \text{ and } \text{P}_3 \text{ have been completed.}

Next the proofs of individual processes are combined to prove the desired property of the entire program \([\text{P}_1 || \text{P}_2 || \text{P}_3]\). According to the rule for parallel composition, one may conclude that

\{true\} [\text{P}_1 || \text{P}_2 || \text{P}_3] \{\text{OUT}_{1,2} = x \\
& \text{\land } \text{IN}_{2,1} = \text{OUT}_{2,3} = y \\
& \text{\land } \text{IN}_{3,2} = u \\
& \text{\land } \text{IN}_{i,j} = \text{OUT}_{j,i}, i, j = 1, 2, 3 \land i \neq j\}

The above postcondition implies that

\[x = \text{OUT}_{1,2} = \text{IN}_{2,1} = \text{OUT}_{2,3} = \text{IN}_{3,2} = u\]

Finally, by using the rule of consequence (R6), one gets the desired proof of

\{true\} [\text{P}_1 || \text{P}_2 || \text{P}_3] \{x = u\}

Q.E.D.

It is important to mention that the use of history variables (e.g., \text{IN}_{1,2}, \text{OUT}_{3,2}, etc.) can not only express what a concurrent program does, but also show how it does. The above example clearly
illustrates this point. It has been proven that \( x = \text{OUT}_{1,2} = \text{IN}_{2,1} = \text{OUT}_{2,3} = \text{IN}_{3,2} = u \) is a postcondition of program \([P_1 || P_2 || P_3]\). This postcondition not only implies \( x = u \) when the program terminates, but also gives the detailed information about how \( x = u \) has been achieved:

1. Assertion \( x = \text{OUT}_{1,2} = \text{IN}_{2,1} \) states that the value of variable \( x \) has been transmitted from process \( P_1 \) to process \( P_2 \).
2. Assertion \( \text{IN}_{2,1} = \text{OUT}_{2,3} \) states that \( P_2 \) has transmitted to \( P_3 \) what it has received from \( P_1 \).
3. Assertion \( \text{IN}_{3,2} = u \) states that \( P_3 \) has assigned the value it has received from \( P_2 \) to variable \( u \).

If the same property is to be proven for the other axiomatic systems, one must extensively modify the assertions used in our proof, thereby increasing the complexity of his task.

**Example 2.** Prove that \{\text{true}\} \([P_1 || P_2] \{x = 2\}\), where

\[
P_1:: [ P_2?x \rightarrow \text{skip} \\
\quad \Box P_2!0 \rightarrow P_2?x; x = x + 1 ]
\]

\[
P_2:: [ P_1!2 \rightarrow \text{skip} \\
\quad \Box P_1?z \rightarrow P_1!1 ]
\]

This example was used in Section 4.3 to explain why communication may not take place between two syntactically
corresponding communication commands. For example, communication will never occur between $P_2?x$ of the first guarded command in $P_1$ and $P_1?l$ in $P_2$. However, such a pair of communication commands must be checked by the satisfaction proof in Levin and Gries' axiomatic system. Inevitably the proof of the above program will be more complicated than is necessary. However, this problem can be overcome by our axiomatic system, as shown below.

Once again the input and output assertions are turned off because there is no obvious pattern on the communication between $P_1$ and $P_2$.

Using the axioms and rules of inference for the commands which appear in $P_1$ and $P_2$ one can prove

$$\{\text{true} \land \text{IN}_{1,2} = \text{OUT}_{1,2} = \lambda\} P_1 \{ (\text{OUT}_{1,2} = \lambda \Rightarrow x = \text{IN}_{1,2})$$
$$\& (\text{OUT}_{1,2} \not= \lambda \Rightarrow x = \text{IN}_{1,2} + 1)\}$$

and

$$\{\text{true} \land \text{IN}_{2,1} = \text{OUT}_{2,1} = \lambda\} P_2 \{ (\text{IN}_{2,1} = \lambda \Rightarrow \text{OUT}_{2,1} = 2)$$
$$\& (\text{IN}_{2,1} \not= \lambda \Rightarrow \text{OUT}_{2,1} = 1)\}$$

Applying the rule for parallel composition results in

$$\{\text{true}\} [P_1 || P_2] \{ (\text{OUT}_{1,2} = \lambda \Rightarrow x = \text{IN}_{1,2})$$
$$\& (\text{OUT}_{1,2} \not= \lambda \Rightarrow x = \text{IN}_{1,2} + 1)$$
$$\& (\text{IN}_{2,1} = \lambda \Rightarrow \text{OUT}_{2,1} = 2)$$
$$\& (\text{IN}_{2,1} \not= \lambda \Rightarrow \text{OUT}_{2,1} = 1)$$
$$\& \text{OUT}_{1,2} = \text{IN}_{2,1} \land \text{OUT}_{2,1} = \text{IN}_{1,2}\}$$

Due to the properties of $\text{OUT}_{1,2} = \text{IN}_{2,1} \land \text{OUT}_{2,1} = \text{IN}_{1,2}$, the following assertions can be proven:
\[
\text{OUT}_{1,2} = \text{IN}_{2,1} \land \Rightarrow x = \text{IN}_{1,2} = \text{OUT}_{2,1} = 2
\]

and

\[
\text{OUT}_{1,2} = \text{IN}_{2,1} \neq \Rightarrow x = \text{IN}_{1,2} + 1 = \text{OUT}_{2,1} + 1 + 1 = 2
\]

Hence, we have proved that \{\text{true}\} [P_1 || P_2] \{x = 2\}.

Q.E.D.

**Example 3.** The following is a program for distributed partitioning of sets. Given two non-empty disjoint sets of integer \(S_0\) and \(T_0\), the program partitions \(S_0 \cup T_0\) into two subsets so that \(|S| = |S_0|, |T| = |T_0|\) (\(|S|\) represents the cardinality of \(S\), etc.) and every element of \(S\) is smaller than any element of \(T\). The program presented here is slightly different from that in [2] because of the changes which have been made to CSP.

The program consists of two communicating processes \(P_1\) and \(P_2\), where

\[
P_1:: \quad \text{mx} := \text{max}(S);
P_2!\text{mx};
P_2?x;
\text{*[mx} > x \quad \rightarrow \quad S := S - \{\text{mx}\} \cup \{x\};
x := \text{max}(S);
P_2!\text{mx};
P_2?x]
\]

and

\[
P_2:: \quad \text{mn} := \text{min}(T);
\]
Variables $S$ and $T$ are local to processes $P_1$ and $P_2$, respectively. Assume $S$ and $T$ have the initial values of $S_0$ and $T_0$, respectively. Intuitively, $P_1$ and $P_2$ exchange the current maximum of $S$, $\max(S)$, with the current minimum of $T$, $\min(T)$, until $\max(S) < \min(T)$.

In order to present the proof of this program, the following notations and functions on sequences will be used:

- $|S|$ \text{ the number of elements in sequence } S.
- $S >$ \text{ sequence } S \text{ is strictly increasing; i.e., if } S = e_1 \ldots e_i e_{i+1} \ldots, \text{ then } e_i < e_{i+1}.$
- $S <$ \text{ sequence } S \text{ is strictly decreasing; i.e., if } S = e_1 \ldots e_i e_{i+1} \ldots, \text{ then } e_i > e_{i+1}.$
- $\text{Last}(S)$ \text{ the last element of sequence } S \text{ (from the lefthand of } S).
- $\text{Lr}(S)$ \text{ the sequence obtained from } S \text{ by removing } \text{Last}(S).
- $\text{Br}(S)$ \text{ the set obtained from } S \text{ by "breaking" } S; \text{ e.g., if } S = e_1 \ldots e_n, \text{ then } \text{Br}(S) = \{e_1, \ldots, e_n\}.$

Compared to the two previous programs, this program is more complicated and more interesting. First the input and output assertions are assigned as follows:
\(is_{1,2} = IN_{1,2} \neq \text{Br}(\text{Lr}(IN_{1,2})) \cup T_0 \ \& \ \text{Last}(IN_{1,2}) \in S_0 \cup T_0\)

\(os_{1,2} = OUT_{1,2} \neq \text{Br}(\text{Lr}(OUT_{1,2})) \cup S_0 \ \& \ \text{Last}(OUT_{1,2}) \in S_0 \cup T_0\)

\(is_{2,1} = IN_{2,1} \neq \text{Br}(\text{Lr}(IN_{2,1})) \cup S_0 \ \& \ \text{Last}(IN_{2,1}) \in S_0 \cup T_0\)

\(os_{2,1} = OUT_{2,1} \neq \text{Br}(\text{Lr}(OUT_{2,1})) \cup T_0 \ \& \ \text{Last}(OUT_{1,2}) \in S_0 \cup T_0\)

The proofs of

\((IN_{i,j}=OUT_i,j=\Lambda, j=1,\ldots,N \ \& \ j \neq i) \Rightarrow os_{i,j}, \ i=1,\ldots,n\)

and

\((os_{j,i} \ \& \ \text{OUT}_{j,i}=IN_{i,j}) \Rightarrow is_{i,j}, \ i,j=1,\ldots,n \ \& \ i \neq j\)

are trivial because of the symmetry between \(is_{i,j}\) and \(os_{j,i}\).

By applying the axioms and rules of inference of the new axiomatic system to the individual processes \(P_1\) and \(P_2\), one gets

\(\{S=S_0 \ \& \ S_0 \cap T_0=\emptyset \ \& \ IN_{1,2}=OUT_{1,2}=\Lambda\} \ P_1 \ \{p_1 \ \& \ os_{1,2} \ \& \ \text{mx} < x\}\)

and

\(\{T=T_0 \ \& \ S_0 \cap T_0=\emptyset \ \& \ IN_{2,1}=OUT_{2,1}=\Lambda\} \ P_2 \ \{p_2 \ \& \ os_{2,1} \ \& \ \text{mn} > y\}\)

where \(os_{1,2}\) and \(os_{2,1}\) have already been specified, and

\(p_1 = [ \ S = S_0 - \text{Br}(\text{Lr}(OUT_{1,2})) \cup \text{Br}(\text{Lr}(IN_{1,2})) \)

\ \& \ \text{mx} = \max(S) = \text{Last}(OUT_{1,2}) \)

\ \& \ x = \text{Last}(IN_{1,2}) \)

\ \& \ |OUT_{1,2}| = |IN_{1,2}|\)

and

\(p_2 = [ \ T = T_0 - \text{Br}(\text{Lr}(OUT_{2,1})) \cup \text{Br}(\text{Lr}(IN_{2,1})) \)

\ & \ |OUT_{2,1}| = |IN_{2,1}|\)
& mn = \text{min}(T) = \text{Last}(\text{OUT}_{2,1})
& y = \text{Last}(\text{IN}_{2,1})
& |\text{OUT}_{2,1}| = |\text{IN}_{2,1}|$

Using the rule for parallel composition, the following conclusion is deducible:

$$\{S=S_0 \& T=T_0 \& S_0 \cap T_0 = \emptyset\} \begin{array}{c} P_1 \mid P_2 \end{array} \{P_1 \& P_2 \ & S_0 \cup T_0 = S \cup T \& S \cap T = S \cap T, 2 \ & \text{os}_{1,2} \& \text{os}_{2,1} \& \text{IN}_{1,2}=\text{OUT}_{2,1} \& \text{OUT}_{2,1}=\text{OUT}_{1,2}\}$$

The properties to be proven include:

(P1) $S \cup T = S_0 \cup T_0$
(P2) $S \cap T = \emptyset$
(P3) $\|S\| = \|S_0\| \& \|T\| = \|T_0\|$
(P4) $\forall a, b \cdot (a \in S \& b \in T \Rightarrow a < b)$

Property (P1) can be proved as follows. Assume $a \in S_0$ but $a \not\in S$.

Then

$$a \in S_0 \& a \not\in S \Rightarrow a \in \text{Br}(\text{Lr}(\text{OUT}_{1,2})) \quad (\text{by } P_1)$$
$$\Rightarrow a \in \text{Br}(\text{Lr}(\text{IN}_{2,1})) \quad (\text{by } \text{OUT}_{1,2}=\text{IN}_{2,1})$$
$$\Rightarrow a \in T \quad (\text{by } P_2)$$

Using the same argument, it can also be proven that $a \in T_0 \& a \not\in T \Rightarrow a \in S$. Therefore, $S_0 \cup T_0 \subseteq S \cup T$. On the other hand, from
is_{i,j} \text{, } os_{i,j} \text{ (} i,j=1,2 \text{ and } i\neq j\text{), } p_1 \text{ and } p_2, \text{ one can prove that } 
S \subseteq S_0 \cup T_0 \text{ and } T \subseteq S_0 \cup T_0. \text{ Hence, the conclusion is that } S \cup T = S_0 \cup T_0.

Property (P2) can be proved as follows. First one prove that 
\( a \in S \Rightarrow a \notin T \). Two cases must be considered:

**Case 1: \( a \in S_0 \)**

By \( p_1 \) and \( IN_{i,j} = OUT_{j,i} \),

\[
\begin{align*}
  a \in S_0 \& a \in S & \Rightarrow a \notin Br(Lr(OUT_{1,2})) \lor a \in Br(Lr(IN_{1,2})) \\
  & \Rightarrow a \notin Br(Lr(IN_{2,1})) \lor a \in Br(Lr(OUT_{2,1}))
\end{align*}
\]

Since \( a \in S_0 \) and \( S_0 \cap T_0 = \emptyset \), \( a \notin T_0 \). By \( os_{2,1} \), one knows that

\[
a \notin Br(Lr(OUT_{2,1}))
\] (2)

Assume \( a \in T \), then by \( p_2 \),

\[
a \notin T_0 \& a \in T \Rightarrow a \in Br(Lr(IN_{2,1}))
\] (3)

Because (3) contradicts with (1) and (2), one must conclude that \( a \notin T \).

**Case 2: \( a \notin S_0 \)**

By \( p_1, p_2 \) and \( IN_{i,j} = OUT_{j,i} \),

\[
\begin{align*}
  a \notin S_0 \& a \in S & \Rightarrow a \notin Br(Lr(OUT_{1,2})) \& a \in Br(Lr(IN_{1,2})) \\
  & \Rightarrow a \notin Br(Lr(IN_{2,1})) \& a \in Br(Lr(OUT_{2,1}))
\end{align*}
\]
Both case 1 and case 2 indicate that \( a \in S \Rightarrow a \notin T \). By the same argument, one can prove that \( a \in T \Rightarrow a \notin S \). Therefore, \( S \cap T = \phi \).

Property (P3) can be easily proven by the use of \( p_1, p_2, o_{s1,2} \) and \( o_{s2,1} \), and the details of this proof are omitted here.

Finally, property (P4) can be proven as follows. Assume \( a \in S \), then

\[
a \leq \max(S) = mx \leq x = \text{Last(IN}_{1,2} = \text{Last(OUT}_{2,1} = \min(T)
\]

Using the same argument, it can also be proven that if \( b \in T \), then \( b \geq \max(S) \). Combining the proofs of \( a \leq \min(T) \), \( b \geq \max(S) \) and property (P2), the conclusion is that \( \forall a, b \ (a \in S \& b \in T \Rightarrow a < b) \).

Q.E.D.

**Example 4.** In this example, consider a program that computes the greatest common divisor of \( n \) numbers \( N_1, \ldots, N_n \), which is denoted by
gcd(N_1, ..., N_n). Because of the changes made to CSP, the program presented here is a variant of that in [2]. The following program consists of n communicating processes. Let each process P_i (i=1, ..., n) contain a local variable x_i, which has N_i as its initial value. This program has the property that when it is blocked in a deadlock state, all x's have the same value gcd(N_1, ..., N_n). As discussed in [2], the interest in such a program arises because

1. It may be easier to write such a program than the corresponding program that will terminate.
2. There exists an automatic transformation transforming every such blocked program into an equivalent terminating program [24].

The communicating processes P_1, ..., P_n are arranged in a ring configuration, with each process P_i communicating with its two immediate neighbors P_{i-1} and P_{i+1} (+ and - are interpreted cyclically in \{1, ..., n\}). In addition to x_i, P_i also contains two boolean variables rsl_i (ready to send left) and rsr_i (ready to send right) to avoid sending x_i to neighbors before x_i is modified again.

The idea of the program is to exchange x_i (i=1, ..., n) among processes. When such a number is received by P_i from a neighbor, this number is compared to x_i. Then x_i is modified according to Euclid's rule. The following is the program [P_1]||...||P_n], where
\[ P_i ::= \text{rsr}_i, rsl_i := \text{true}, \text{true}; \]
\[ \begin{array}{l}
\forall [ rsl_i, rsr_i := \text{true}, \text{true}; rsl_i = \text{false} \\
\quad \text{rsr}_i := \text{false} \\
\quad \text{Pi} = \text{Pi} + 1 \] x_i \rightarrow \text{rsr}_i = \text{false} \\
\quad y_i = x_i \rightarrow \text{skip} \\
\quad y_i < x_i \rightarrow \{ y_i = x_i \rightarrow x_i := y_i \\
\quad y_i = x_i \rightarrow x_i := x_i \mod y_i \} \\
\quad rsl_i, rsr_i := \text{true}, \text{true} \\
\end{array} \]
\[ \begin{array}{l}
\] x_i \rightarrow \text{skip} \\
\quad z_i = x_i \rightarrow \{ z_i = x_i \rightarrow x_i := z_i \\
\quad z_i = x_i \rightarrow x_i := x_i \mod z_i \} \\
\quad rsl_i, rsr_i := \text{true}, \text{true} \\
\] 

The complete proof of this program is quite complicated. Instead of proving the desired property, only the proof of a weaker property will be shown here. This property states that, when the program is blocked in a deadlock state,

\[ \exists m \cdot (x_1 = \ldots = x_n = m \cdot \gcd(N_1, \ldots, N_n)) \]

\( \gcd(N_1, \ldots, N_n) \) will be abbreviated by \( g \) in the rest of discussion. The input and output assertions are assigned as follows:

\[ \begin{array}{l}
is_{i, i-1} = \forall e \cdot (e \in \text{Br}(IN_{i, i-1}) \Rightarrow \exists k \cdot (e = k \cdot g)) \\
is_{i, i+1} = \forall e \cdot (e \in \text{Br}(IN_{i, i+1}) \Rightarrow \exists k \cdot (e = k \cdot g)) \\
os_{i, i-1} = \forall e \cdot (e \in \text{Br}(OUT_{i, i-1}) \Rightarrow \exists k \cdot (e = k \cdot g)) \\
is_{i, i+1} = \forall e \cdot (e \in \text{Br}(OUT_{i, i+1}) \Rightarrow \exists k \cdot (e = k \cdot g)) \\
\end{array} \]
\[(\text{IN}_i,j=\text{OUT}_i,j=\Lambda, j=1,\ldots,N \& j\neq i) \Rightarrow os_{i,j}, \quad i=1,\ldots,n\]

and

\[(os_{j,i} \& \text{OUT}_{j,i} = \text{IN}_{i,j}) \Rightarrow is_{i,j}, \quad i,j=1,\ldots,n \& i\neq j\]

are trivial because of the symmetry between \(is_{i,j}\) and \(os_{j,i}\).

It can be proven that the assertion \(p_i \& os_{i,i-1} \& os_{i,i+1}\) is a loop invariant for the repetitive command in \(P_i\), where

\[
P_i = \exists m_i \ (x_i = m_i \times g) \]

\& \(\text{Last(\text{IN}_{i,i-1})} = y_i\)

\& \(\text{Last(\text{OUT}_{i,i+1})} = z_i\)

\& \(\text{IN}_{i,i-1} \neq \Lambda \Rightarrow x_i \leq y_i\)

\& \(\text{IN}_{i,i+1} \neq \Lambda \Rightarrow x_i \leq z_i\)

\& \(\text{rs}_{i} = \text{false} \Rightarrow \text{Last(\text{OUT}_{i,i-1})} = x_i\)

\& \(\text{rs}_{i} = \text{false} \Rightarrow \text{Last(\text{OUT}_{i,i+1})} = x_i\)

According to the semantics of CSP, an individual process can be blocked only on an input command. Because input commands appear only in the guards, it is concluded that when the program is blocked, for each \(i=1,\ldots,n\),

1. \(\text{rs}_{i} = \text{rs}_{i} = \text{false}\)

2. \(p_i = \text{true}\)

3. \(\text{OUT}_{i,i-1} = \text{IN}_{i-1,i} \& \text{OUT}_{i,i+1} = \text{IN}_{i+1,i}\)

Hence, when the program is blocked,
\( rsl_i = \text{false} \Rightarrow x_i = \text{Last}(\text{OUT}_i,i-1) \)

\( \Rightarrow \text{IN}_{i-1,i} \notin \Lambda \)

\( \Rightarrow x_{i-1} \leq \text{Last} (\text{IN}_{i-1,i}) \)

\( \Rightarrow x_{i-1} \leq \text{Last} (\text{OUT}_{i,i-1}) \)

\( \Rightarrow x_{i-1} \leq x_i \)

Using the same argument, it can also be proven that when the program is blocked, \( x_i \leq x_{i-1} \). By \( p_i \), the conclusion is that

\[ \exists m \ (x_1 = \ldots = x_n = m \cdot \text{gcd}(N_1, \ldots, N_n)) \]

Q.E.D.

4.6. Summary

In this chapter we have considered the semantic aspect of communication protocols. The approach used is based on axiomatic proof of concurrent programs. Axiomatic systems, although difficulties remain, are valuable in understanding the nature of programming. The feature of communication through message exchange and the elegance in its design lead to the adoption of Hoare's Communicating Sequential Processes (CSP) as the language for specifying communication protocols. Each existing axiomatic system for CSP has its own strengths and weaknesses. The new axiomatic system, through the use of history variables and input/output
assertions, has retained the advantages of the other systems. The characteristics of this new system can be seen in its handling communication commands and in verifying the interference-freeness between the proofs of individual processes. The advantage arising from these characteristics is the reduction of complexity in verifying concurrent programs as well as communication protocols.
5. FORMAL SPECIFICATION AND VERIFICATION OF PDCPD

In Chapter 3 a new method for designing communication protocols, Priority Driven Communication Protocol Design (PDCPD), was introduced. PDCPD was demonstrated to be a useful tool for the protocol designer in attaining some general correctness properties (e.g., completeness and freedom from deadlock) in his design. Specifically, a synchronization mechanism was used to solve the problem caused by collisions between communicating processes. Several simple examples were used to show how the processes use the information of logical time and confirmation as well as priority assignment and backtracking scheme to detect and resolve collisions. Nevertheless, the applicability of this synchronization mechanism to the general cases remained questionable. In this chapter the axiomatic system introduced in Chapter 4 for verifying the semantics of concurrent programs and protocols will be used to prove the applicability of this synchronization mechanism to arbitrary protocols.

This chapter is organized as follows. Section 5.1 defines the notations that will be used in the formal specification and verification of PDCPD. The specification and verification of PDCPD
on an arbitrary protocol are given in Sections 5.2 and 5.3, respectively. Finally, Section 5.4 summarizes the work of this chapter.

5.1. Notations

The reader is reminded that the synchronization mechanism of PDCPD was developed based on the model of protocol desirable interaction graphs (DIG), rather than on the states and transitions of individual processes. Therefore, in this chapter the discussion also will be based on DIG.

Given an arbitrary protocol @, the symbols STATE@, TYPE@ and PROCESS@ are used to denote the sets of states, transition types and processes involved in @, respectively. FINAL@ is also used to denote the set of final states in @. For example, if @ denotes the protocol depicted in Figure 30, then

\[
\begin{align*}
\text{STATE}@ & = \{S_1, \ldots, S_{10}\} \\
\text{TYPE}@ & = \{e_1, \ldots, e_7\} \\
\text{PROCESS}@ & = \{P_1, P_2, P_3, P_4\} \\
\text{FINAL}@ & = \{S_{10}\}
\end{align*}
\]

For each state s in @, OT@\{s\} denotes the set of output transitions of s, and OT@ denotes the set of all transitions in @ (i.e., \( OT@ = \bigcup_{s \in \text{STATE}@} OT@\{s\} \)). Tran@ is used to denote the transition function in @; specifically, Tran@ maps
Figure 30: Desirable interaction graph of a hypothetical protocol
STATE@ x [PROCESS@ x TYPE@] into STATE@ in such a way that if \([P,e]\) is a transition in @ which directs from state S1 to state S2, then

\[ \text{Tran}_@ (S1, [P,e]) = S2. \]

According to PDCPD, each signal transmitted between a pair of processes must include at least two items of information. One indicates the transition being initiated or confirmed while the other indicates the logical time at which the transition is initiated or confirmed. A transition, in turn, consists of the name of the process which initiates the transition and the type of the transition. More specifically, each signal can be represented by the form of \(<[P,e],t>\), where \(P \in \text{PROCESS@}\), \(e \in \text{TYPE@}\) and \(t \in \mathbb{N}\) (\(\mathbb{N}\) denotes the set of non-negative integers). In the following discussion, \(\text{SIGNAL@}\) is used to denote the set of signals used in @.

As discussed in Chapter 3, the synchronization mechanism of PDCPD uses the concept of paths, which are sequences of transitions, in order to assist the communicating processes in detecting and resolving collisions. Also, as discussed in Chapter 4, the axiomatic verification system for CSP uses assertions over sequences of input and output signals to help reduce complexity involved in verification. Therefore, it is worth defining some notation for describing sequences before beginning the formal specification and verification of PDCPD.
Suppose $S$ denotes an arbitrary set. Let $S^*$ stand for the set of all finite sequences of elements from $S$. Each element of $S^*$ is called a sequence of $S$. The elements of $S^*$ can be recognized recursively as follow:

1. $\lambda$ (empty sequence) is a sequence of $S$.
2. Every element of $S$ is a sequence of $S$.
3. If $A$ and $B$ are two sequences of $S$, then $A$ concatenated with $B$ (the sequence $A$ followed by the sequence $B$) is also a sequence of $S$. $A \circ B$ will be used to denote $A$ concatenated with $B$.

If $A$ is a sequence of $S$, $|A|$ stands for the length of $A$, with each element of $S$ having one unit in length. Let $A$ and $B$ be two sequences of $S$. $A$ is an initial subsequence of $A \circ B$ in $S$, and $A$ is a proper initial subsequence of $A \circ B$ in $S$ if $B \neq \lambda$. Assuming that $A$ and $B$ are two sequences of $S$, then $A < B$ denotes that $A$ is an initial subsequence of $B$ while $A \prec B$ denotes $A$ is a proper initial subsequence of $B$. It is apparent that if $A$ and $B$ are two sequences of $S$, then

$$|A \circ B| = |A| + |B|$$

$|A| \leq |B|$ if $A$ is an initial subsequence of $B$

$|A| < |B|$ if $A$ is a proper initial subsequence of $B$

In addition, given $S$, the following functions over sequences of $S$ are defined:
Elem(A,k) \triangleq \text{the kth element of A (from the lefthand of A).}

IniSubSeq(A,k) \triangleq \text{the initial subsequence of A with } |\text{IniSubSeq}(A,k)|=k. \text{ (IniSubSeq}(A,k) \text{ is assumed undefined if } |A|<k.)

Common(A,B) \triangleq \text{the longest common initial subsequence of A and B; i.e., A and B are identical up to their first } |\text{Common}(A,B)| \text{ elements.}

Last(A) \triangleq \text{the last element of A.}

PATH@ is used to denote the set of paths in @. Since not every sequence of transitions is a path in @, PATH@ is a subset of OS@.

For example, if @ is the protocol given in Figure 30, then 
\([P_2,e_2]o[P_4,e_5]o[P_1,e_1]\) is a path in @ but 
\([P_1,e_1]o[P_3,e_6]\) is not.

If both p_1 and p_2 are paths in @ (i.e., p_1 \in PATH@ and p_2 \in PATH@) and 
p_1 \triangleright p_2, \text{ then } p_1 \text{ is called a subpath of } p_2 \text{ in @.} \text{ If } p_1 \triangleright p_2, \text{ then } p_1 \text{ is a proper subpath of } p_2 \text{ in @.}

State@ is used to represent a function mapping PATH@ into STATE@ such that if p \in PATH@, then State_@(p) is the resulting state of p. For example,

State_@([P_2,e_2]o[P_4,e_5]o[P_1,e_1]) = S9

if @ is the protocol in Figure 30. This function can be defined recursively as follows:

State_@(A) \triangleq S0,
\[ \text{State}_\mathcal{G}(p_0[p,e]) = \text{Tran}_\mathcal{G}(\text{State}_\mathcal{G}(p),[p,e]), \]

where \( p \in \text{PATH}_\mathcal{G}, [p,e] \in \text{OT}_\mathcal{G} \) and \( p_0 \) represents the initial state in \( \mathcal{G} \). Notice that \( \text{State}_\mathcal{G}(p) \) is defined if and only if \( p \in \text{PATH}_\mathcal{G} \).

As described in Chapter 3, PDCPD uses priorities to resolve collisions. Here \( \text{Pty}_\mathcal{G}(s,[p,e]) \) is used to denote the priority level of output transition \([p,e]\) at state \( s \). A relation \( \prec \) is defined over \( \text{PATH}_\mathcal{G} \) in a way that, assuming that \( p_1 \) and \( p_2 \) are two paths in \( \mathcal{G} \), then \( p_1 \prec p_2 \) if and only if

1. \( p_1 \prec p_2 \); or
2. \( \text{Pty}_\mathcal{G}(\text{State}_\mathcal{G}(\text{Common}(p_1,p_2)),\text{Elem}(p_1,|\text{Common}(p_1,p_2)|+1)) < \text{Pty}_\mathcal{G}(\text{State}_\mathcal{G}(\text{Common}(p_1,p_2)),\text{Elem}(p_2,|\text{Common}(p_1,p_2)|+1)) \).

\( p_2 \) is said to have the precedence of \( p_1 \) if \( p_1 \prec p_2 \).

Figure 31 illustrates the conditions in which \( p_1 \prec p_2 \). The condition illustrated in Figure 31(a) is trivial. Figure 31(b) shows that if \( p_1 \) and \( p_2 \) diverge and at the node of divergence the priority of the transition on \( p_2 \) is higher than that on \( p_1 \), then \( p_1 \prec p_2 \).

One may prove \( \prec \) a is total; i.e.,

1. If \( p_1 \) and \( p_2 \) are two different paths in \( \mathcal{G} \), then either \( p_1 \prec p_2 \) or \( p_2 \prec p_1 \), but not both, must be true.
2. For any path \( p \) in \( \mathcal{G} \), \( p \prec p \).
3. If \( p_1 \prec p_2 \) and \( p_2 \prec p_3 \), then \( p_1 \prec p_3 \).
Figure 31: Conditions for $p_1 < p_2$. 

\[ S = \text{State}_0(\text{Common}(p_1, p_2)) \]

\[ \text{Pty}_0(S, t_1) < \text{Pty}_0(S, t_2) \]
We use $P_1 <_@ P_2$ to denote that either $P_1 < P_2$ or $P_1 = P_2$.

Recall that in PDCPD, communicating processes detect and resolve collisions by recording the path of each other. One process can record the path of another process only by receiving signals from the latter. Suppose $n$ processes $P_1, \ldots, P_n$ are involved in protocol $@$. In an individual process $P_i$ ($1 \leq i \leq n$), function $\text{Path}_{@,k}$ is used to compute the path of another process $P_k$. Therefore, $\text{Path}_{@,k}$ is a function over $\text{PATH}_{@,x}\text{SIGNAL}_{@}$ such that if $p$ is the current path of $P_k$ recorded in $P_i$ and $[P_j, e]$ is a signal newly received from $P_k$, then $\text{Path}_{@,k}(p, [P_j, e])$ is the updated path of $P_k$ recorded in $P_i$. $\text{Path}_{@,k}$ can be defined recursively as follows:

$$\text{Path}_{@,k}(p, [P_j, e], t)$$

$$\begin{align*}
\text{Path}_{@,k}(p, [P_j, e], t) &= \begin{cases} 
\text{po}[P_j, e] & \text{if (1)} \\
\text{IniSubSeq}(p, t)\circ[P_j, e] & \text{if (2)} \\
\text{Path}_{@,k}(\text{po}[P_i(1), e_i(1)], [P_j, e]) & \text{if (3)} \\
\uparrow \text{(undefined)} & \text{if (4)}
\end{cases}
\end{align*}$$

where

1. $t = |p| \& [P_j, e] \in \text{OT}_{@}(\text{State}_{@}(p))$

2. $t < |p| \& [P_i, e] \in \text{OT}_{@}(\text{State}_{@}(\text{IniSubSeq}(p, t)))$
   \& $\text{Pty}_{@}(\text{State}_{@}(\text{IniSubSeq}(p, t)), \text{Elem}(p, t))$
   \< $\text{Pty}_{@}(\text{State}_{@}(\text{IniSubSeq}(p, t)), [P_j, e])$

3. $t > |p| \& P_k \neq P_{i(1)} \& \text{OT}_{@}(\text{State}_{@}(p)) = \{ [P_{i(1)}, e_{i(1)}] \}$
   (see Figure 32(c))

4. Otherwise
Figure 32 illustrates the above conditions (1), (2) and (3).

In Figure 32(a) signal \([P_j,e],t\) indicates that \(P_k\) has taken transition \([P_j,e]\) at the end of its original path, which is indicated by \(p\); therefore, the path of \(P_k\) recorded in \(P_i\) should be updated by concatenating \([P_j,e]\) to \(p\). In Figure 32(b) signal \([P_j,e]\) indicates that \(P_k\) has backtracked to node \(N\) and has taken transition \([P_j,e]\) at \(N\); hence, the path of \(P_k\) recorded in \(P_i\) should be updated by concatenating \([P_j,e]\) to \(\text{IniSubSeq}(p,t)\). The assertion

\[
\text{Ptyg}(\text{State}(\text{IniSubSeq}(p,t)),\text{Elem}(p,t)) < \text{Ptyg}(\text{State}(\text{IniSubSeq}(p,t)),[P_j,e])
\]

emphasizes that at node \(N\) the priority of transition \([P_j,e]\) must be higher than that of the transition on \(p\). (Otherwise, there is no reason for backtracking.) Figure 32(c) depicts a more complicated condition: After reaching the end of \(p\), process \(P_k\) has taken a sequence of transitions \([P_u(1),e_u(1)],...,[P_u(N),e_u(N)]\) \((N=|p|)\) before it takes transition \([P_j,e]\) and transmits signal \([P_j,e],t\) to \(P_i\). Note that for each \(i\) \((1<=i<=N)\),

1. \(P_k\neq P_u(i^-)\). Suppose \(P_k=P_u(i^-)\). Then, because \(P_k\) is the initiator of transition \([P_u(i^-),e_u(i^-)]\), \(P_k\) should have transmitted a signal \([P_u(i^-),e_u(i^-)],|p|+i^-1\) to indicate the initiation of \([P_u(i^-),e_u(i^-)]\) before it transmits signal \([P_j,e],t\).
2. \( \text{State}_@([P_u(1), e_u(1)] \ldots [[P_u(i-1), e_u(i-1)]] \) has only one output transition (if \( i' = 1 \), the aforementioned expression means \( \text{State}_@([P_u]) \)). If it has more than one output transition, according to the synchronization mechanism of PDCPD, \( P_k \) should have transmitted a signal \( <[P_u(i'), e_u(i')], [P_j, e], t> \) to confirm \( [P_u(i'), e_u(i')] \) before it transmits signal \( <[P_j, e], t> \).

One can easily prove that if \( p \) is a path in \( @ \) and \( \text{Path}_{@,k}(p, [P_j, e]) \) is defined, then

\[
\text{Path}_@ (p) \subseteq \text{Path}_{@,k}(p, [P_j, e])
\]

The definition of \( \text{Path}_{@,k} \) is extended to assist a process in computing the path of another process from receiving a sequence of signals, rather than just one signal. Function \( \text{Path}^*_@,k \) is defined over \( \text{PATH}_@ \times \text{SIGNAL}^*_@ \) as follows:

\[
\text{Path}^*_@,k(p, \Lambda) \triangleq p,
\]

\[
\text{Path}^*_@,k(p, S <[P_j, e], t>) \triangleq \text{Path}_@ (\text{Path}^*_@,k(p, S), <[P_j, e], t>)
\]

where \( p \in \text{PATH}_@ \), \( S \in \text{SIGNAL}^*_@ \) and \( <[P_j, e], t> \in \text{SIGNAL}_@ \). From the definition of \( \text{Path}_@,k \), one knows that if the current path of \( P_k \) recorded in \( P_i \) is \( p \), then after a sequence \( S \) of signals is received from \( P_k \), the updated path of \( P_k \) will become \( \text{Path}^*_@,k(p, S) \). Function \( \text{Path}^*_@,k \) plays an important role in the formal specification and
Figure 32: Conditions in the definition of $\text{Path}_{\theta,k}$.
verification of PDCPD. If $IN_{i,k}$ is the sequence of signals which
process $P_i$ has received from process $P_k$, then the path of $P_k$
recorded in $P_i$ will be $Path_{i,k}(\Lambda, IN_{i,k})$.

5.2. Formal Specification of PDCPD

The formal language used to specify PDCPD is Communicating
Sequential Processes (CSP), which has been described in Section 4.2.
The reason for specifying PDCPD in CSP is that in CSP interprocess
communication is carried out by signal (or message) exchanges instead
of shared memory. This feature suitably describes the environment
in which PDCPD operates. Moreover, by using CSP's nondeterministic
alternative commands, repetitive commands and I/O commands, one can
specify the interactions among communicating processes in a concise
and elegant way. However, one is reminded that the rendezvous
assumption (i.e., a transmitting process is delayed until some
process is ready to receive the signal) has been excluded from the
notation of CSP given in Section 4.1.

Figure 33 lists the specification of PDCPD for an arbitrary
protocol @. It is assumed that $n$ communicating processes $P_1, \ldots, P_n$
are involved in @. The entire protocol system is specified as a
concurrent CSP program $[P_1||\ldots||P_n]$. Special information of @ has
been hidden by functions $STATE_@$, $TYPE_@$, etc.; therefore, this
specification of PDCPD is valid for arbitrary protocols. Without
losing generality one may assume that there is at least one final state in @ (i.e., FINAL@ ≠ ∅).

It is important to mention that the specification in Figure 33 represents only one of many alternative ways to specify PDCPD. We must say that this specification is concerned more with ease in verification than with efficiency in execution.

As discussed in Chapter 3, PDCPD provides the communicating processes with a synchronization mechanism which allows the processes to detect and resolve collisions dynamically during the execution of the protocol. Detection and resolution of collisions are made possible by assisting processes in keeping track of the path of each other. As shown in Figure 33, process P_i uses a variable path_i,j to record the path of another process P_j. The path of process P_i is recorded by itself in the variable path_i,i.

An additional variable h-path is also used in P_i, which is used to record the path with the highest precedence among variables path_i,1,...,path_i,n; i.e.,

\[ \exists P_j \in \text{PROCESS} @ ( h\text{-path} = path_i,j ) \]
\[ \& \forall P_j \in \text{PROCESS} @ ( path_i,j \preceq @ h\text{-path} ) \]

Variable h-path is used to resolve the condition in which multiple collisions are detected. Figure 34 illustrates such a condition: In Figure 34, one assumes that three communicating processes P_1, P_2
A Specification of PDCPD for a Given Protocol $\pi$:

$$[P_1||...||P_n]$$

where $P_1, ..., P_n$ are $n$ communicating processes involved in $\pi$.

Specification of an Individual Process $P_i$:

/ variable list:
  path$^i$: the path of process $P_i$;
  path$^{-i}$: the path of process $P_j$ recorded by process $P_i$ $(i \neq j)$;
  h-path : the path with the highest precedence among path$^i$, ..., path$^{-n}$;
  $|S|$ : the cardinality of set $S$.

abbreviation: $\Pi_{i=1,...,n} S_i$ abbreviates $S_1;...;S_n$.

/ All the paths are empty before the program begins. /

path$^i$, ..., path$^{-i}$ := $\Lambda$, ..., $\Lambda$;

h-path := $\Lambda$;

*[ / $P_i$ receives a signal $<[P,e],t>$ from $P_j$. /
  $\Box_{P_j\in\text{PROCESS}} & P_j \neq P_i \Rightarrow [P,e],t$ ---$
  \rightarrow$
    path$^-j$ := Path$^-j$(path$^-j$, $<[P,e],t>$);
    [ h-path$@path^-j$ ---$
  \rightarrow$ h-path := path$^-j$ ]

/ $P_i$ initiates a transition of type $e$ at the end of path$^i$. /
  $\Box e \in \text{TYPE} \Rightarrow (\exists [P_i,e] \in \text{OT} \cap \text{State}(\text{path}^i,i)) & h-path \rightarrow \text{path}^i$ ---$
  \rightarrow$
    $\Pi_{P_j\in\text{PROCESS}} & P_j \neq P_i \Rightarrow [P_i,e],|\text{path}^i,i|$;
    path$^i,i, h$-path := path$^i,i$ $\circ [P_i,e],\text{path}^i,i$ $\circ [P_i,e]$]

Figure 33: Specification of PDCPD
/ P_i "catches up" h-path by taking transition [P_k, e] which is on h-path. /
\[ P_k \in \text{PROCESS}_\varnothing \land e \in \text{TYPE}_\varnothing \]
\[
\{ \text{path}_{i,i} \in \text{h-path} \land [P_k, e] = \text{Elem}(\text{h-path}, |\text{path}_{i,i}|) \\
\land \text{path}_{i,i} \sqsubseteq [P_k, e] \sqsubseteq \text{path}_{i,k} \} \rightarrow \\
[ \not\text{OT}_\varnothing(\text{State}_\varnothing(\text{path}_{i,i})) \| > 1 \rightarrow \\
/ P_i \text{ transmits a confirmation about transition [P_k, e].} / \\
\prod_{P_j \in \text{PROCESS}_\varnothing} \land_{P_j \neq P_i} P_j ! [P_k, e], |\text{path}_{i,i}| > 1; \\
\text{path}_{i,i} := \text{path}_{i,i} \circ [P_k, e];
\]

/ P_i backtracts to Common(h-path, path_{i,i}) and transmits a confirmation about transition [P_k, e]. /
\[ P_k \in \text{PROCESS}_\varnothing \land e \in \text{TYPE}_\varnothing \]
\[
\{ \text{path}_{i,i} \in \text{h-path} \land [P_k, e] = \text{Elem}(\text{h-path}, |\text{Common}(\text{h-path}, \text{path}_{i,i})|) \\
\land \text{Common}(\text{h-path}, \text{path}_{i,i}) \sqsubseteq [P_k, e] \sqsubseteq \text{path}_{i,k} \} \rightarrow \\
\prod_{P_j \in \text{PROCESS}_\varnothing} \land_{P_j \neq P_i} P_j ! [P_k, e], |\text{Common}(\text{h-path}, \text{path}_{i,i})|; \\
\text{path}_{i,i} := \text{Common}(\text{h-path}, \text{path}_{i,i}) \circ [P_k, e]
\]

Figure 33 (continued)
and $P_3$ are involved in protocol @. $P_1$ and $P_2$ first collide at node $N_1$ which corresponds to state $S_1$, and

$$P_{ty}(S_1, [P_1, e_1]) < P_{ty}(S_1, [P_2, e_2]).$$

Later $P_1$ and $P_3$ collide at node $N_2$ which corresponds to state $S_2$ (assume the collision at $N_1$ has not been detected yet), and

$$P_{ty}(S_2, [P_1, e_3]) < P_{ty}(S_2, [P_3, e_4]).$$

Assume that both collisions have been detected, but neither has been resolved. Then, process $P_1$ should backtrack to $N_1$ in order to resolve the collision at $N_1$ rather than to $N_2$ to resolve the collision at $N_2$. (If $P_1$ first backtracks to $N_2$, $P_1$ must backtrack later to $N_1$ because the collision at $N_1$ still remains. In contrast, if $P_1$ directly backtracks to $N_1$, then the collision at $N_2$ becomes unimportant because process $P_3$ must backtrack to $N_1$, too.)

According to the definition of $<@$, one can prove that before these collisions are resolved,

$$path_{1,1} <@ path_{1,3} <@ path_{1,2}.$$  

At this time $h\text{-}path=\text{path}_{1,2}$. If the aforementioned multiple collisions occur, process $P_1$ will backtrack to $N_1$ instead of $N_2$ because $\text{path}_{1,1}$ diverges from $h\text{-}path$ at node $N_1$. The use of $h\text{-}path$ will become clearer after the specification of PDCPD is explained in
As shown in Figure 33, the specification of an individual process \( P_i \) includes a guarded repetitive command

\[ ![P_i \text{ receives a signal }<[P,e],t>] \text{ from } P_j. \]

This guarded repetitive command, in turn, contains four types of guarded commands. The functions of these commands are explained in the following paragraphs.

The guarded command of the first type is used by process \( P_i \) in order to receive a signal from a remote peer process. As specified by Figure 33, such a command has the code

\[
P_j?<[P,e],t> \rightarrow \\
\text{path}_{i,j} := \text{Path}_{i,j}([P,e],t); \\
[ h\text{-path}<@\text{path}_{i,j} \rightarrow h\text{-path}:=\text{path}_{i,j}],
\]

where \( P_j \) is the process from which signal \(<[P,e],t>\) is received. Having received the signal \(<[P,e],t>\), \( P_i \) updated its internal record of the path of \( P_j \), which is indicated by \( \text{path}_{i,j} \), using the function \( \text{Path}_{i,j} \). Variable \( h\text{-path} \) must also be updated accordingly.

Since the above guarded command has a pure input guard, this command will be iterated indefinitely until process \( P_j \) stops transmitting signals to \( P_i \).
Figure 34: The use of variable h-path
A guarded command of the second type is used by process $P_i$ to initiate a transition. As specified by Figure 33, such a command has the code

$$
\{(P_i, e) \in \text{OT}_\text{G}(\text{State}_G(\text{path}_{i,i})) \& \text{h-path} @\text{path}_{i,i}[P_i,e] \} \rightarrow \bigwedge_{P_j \in \text{PROCESS}_G} \& P_j \neq P_i \; \; P_j!\{[P_i,e],[\text{path}_{i,i}]\}; \\
\text{path}_{i,i},\text{h-path} := \text{path}_{i,i} \circ [P_i,e], \text{path}_{i,i} \circ [P_i,e].
$$

The above command is used by $P_i$ to initiate a transition $[P_i,e]$ if it is an output transition of state $\text{State}_G(\text{path}_{i,i})$ (i.e., the resulting state of $\text{path}_{i,i}$). The command has a pure boolean guard whose necessity can be seen in Figure 35. In this figure one assumes $[P_i,e]$ is an output transition of $\text{State}_G(\text{path}_{i,i})$ and $[P_i,e]$ is not on $\text{h-path}$. Assume the assertion $\text{h-path} @\text{path}_{i,i}[P_i,e]$ is not satisfied, then the initiation of $[P_i,e]$ should not be allowed because at node $N$ (which corresponds to $\text{State}_G(\text{path}_{i,i})$) the priority of $[P_i,e]$ is lower than that of the transition which is on $\text{h-path}$.

The guarded command of the third type is used by process $P_i$ to "catch up" $\text{h-path}$. The command has the code

$$
\{(\text{path}_{i,i}, \text{h-path} \& [P_k,e]=\text{Elem}(\text{h-path},[\text{path}_{i,i}]) \\
\& \text{path}_{i,i} \circ [P_k,e] \leq \text{path}_{i,i}\} \rightarrow \\
\{ \|\text{OT}_G(\text{State}_G(\text{path}_{i,i}))\| > 1 \} \rightarrow \\
\bigwedge_{P_j \in \text{PROCESS}_G} \& P_j \neq P_i \; \; P_j!\{[P_k,e],[\text{path}_{i,i}]\}; \\
\text{path}_{i,i} := \text{path}_{i,i} \circ [P_k,e].
$$
Figure 35: The use of boolean guard in the second type of guarded command
The condition in which the above command is invoked to operate is illustrated in Figure 36. The assertion

\[ \text{path}_{i,j} \triangleleft \text{h-path} \& [P_k,e] = \text{Elem}(\text{h-path}, |\text{path}_{i,j}|) \]

in the boolean guard of the above guarded command indicates that process \( P_k \) has initiated transition \([P_k,e]\) at node \( N \) (which is assumed to correspond to state \( \text{State}_{\ominus}(\text{path}_{i,j}) \)). Hence, process \( P_i \) may take this transition after \( P_i \) has reached \( N \). Upon taking \([P_k,e]\) at node \( N \), according to the synchronization mechanism of PDCPD, \( P_i \) must transmit a confirmation signal for \([P_k,e]\) at \( t \) (the logical time of \( N \)) to the other processes if \([P_k,e]\) is not the only output transition of node \( N \).

It should be noted that an additional assertion

\[ \text{path}_{i,k} \triangleleft [P_k,e] \leq \text{path}_{i,k} \]

has been included in the boolean guard of the guarded command. The reason for this assertion is also illustrated in Figure 36. In Figure 36 one assumes the transition \([P_k,e]\) is not the only output transition at node \( N \). If signal propagation time between processes \( P_k \) and \( P_i \) is very long, then it could cause the following condition: \( P_k \) first initiates \([P_k,e]\) at \( N \) and transmits a signal \(<[P_k,e],t>\) to the other processes to indicate its initiation. Another process,
for example, process P\textsubscript{j} receives the signal at N and therefore transmits a signal \(<[P\textsubscript{k},e],t>\) to confirm its action of following transition \([P\textsubscript{k},e]\) at N. Due to the delay between P\textsubscript{k} and P\textsubscript{i}, process P\textsubscript{i} receives the confirmation signal \(<[P\textsubscript{k},e],t>\) from process P\textsubscript{j} before it receives the signal \(<[P\textsubscript{k},e],t>\) from process P\textsubscript{k}, and therefore detects the initiation of \([P\textsubscript{k},e]\) even before the signal which indicates this initiation is received.

Therefore, in a multi-party protocol in which more than two processes are involved, it is possible that a process could detect the initiation of a transition even before the signal indicates the initiation has been received from the initiator. In this case, even though there is no apparent reason to prohibit P\textsubscript{i} from following \([P\textsubscript{k},e]\) before the signal is received from P\textsubscript{k}, a painful experience shows that allowing process P\textsubscript{i} to follow \([P\textsubscript{k},e]\) before the signal \(<[P\textsubscript{k},e],t>\) is received from P\textsubscript{k} will greatly increase the complexity in proving the specification of PDCPD. Therefore, the assertion

\[
\text{path\textsubscript{i},i} \preceq [P\textsubscript{k},e] \preceq \text{path\textsubscript{i},k}
\]

is included in the guard to prevent P\textsubscript{i} from following \([P\textsubscript{k},e]\) before it receives \(<[P\textsubscript{k},e],t>\) from P\textsubscript{k}.

The guarded command of the fourth type is used by process P\textsubscript{i} to resolve a detected collision. This command has the code

\[
\{\text{path\textsubscript{i},i} \ll \text{h-path} & [P\textsubscript{k},e] = \text{Elem}(h\text{-path}, \text{Common}(h\text{-path}, \text{path\textsubscript{i},i}))\}
\]
Figure 36: The use of boolean guard in the third type of guarded command
The condition under which the above command is invoked to operate is illustrated in Figure 37. In Figure 37 one assumes that $h\text{-path}=path_{i,k}$. The assertion $path_{i,i} \notin h\text{-path}$ indicates that processes $P_i$ and $P_k$ collide at node $N$ (which is assumed to correspond to state State(@Common(h-path,path_{i,i}))). Since $h\text{-path}$ has the precedence of $path_{i,i}$, one concludes that the priority of transition $[P_k,e]$, which is on $h\text{-path}$, must be higher than that taken by process $P_i$ at node $N$. Therefore, $P_i$ should backtrack to $N$ and take $[P_k,e]$. Obviously $[P_k,e]$ is not the only output transition of $N$; hence, $P_i$ should also transmit a signal $<[P_k,e],t>$ ($t$ is the logical time of $N$) to confirm its backtracking to $N$ and taking $[P_k,e]$. The additional assertion

$$\text{Common}(h\text{-path},path_{i,i})o[P_k,e] \leqslant path_{i,k}$$

is included in the boolean guard of this command to prevent $P_i$ from taking $[P_k,e]$ before it has received signal which indicates the initiation of this transition from $P_k$.

As mentioned previously, the first type of guarded command has a pure input guard; therefore, such a command will be iterated indefinitely until the source process stops transmitting signals.
Figure 37: The use of boolean guard in the fourth type of guarded command
Also, according to the notation of CSP given in Section 4.2, the repetitive command in an individual process will be blocked when all the processes stop transmitting signals. When such a condition occurs, the entire concurrent program \([P_1||...||P_n]\) is blocked in a deadlock state. As discussed in Chapter 4, such a concurrent program can be transformed into an equivalent terminating program. Figure 38 gives such an equivalent terminating specification of PDCPD. Although this modified specification of PDCPD is of little difference from that given in Figure 33, the verification of this modified specification is more complex.

5.3. Formal Verification of PDCPD

The previous section gave the formal specification of PDCPD for an arbitrary protocol \(\mathcal{A}\). In this section the new axiomatic verification system for CSP, as presented in Chapter 4, will be used to verify the correctness of PDCPD; i.e., the assertion will be proven that collisions will be correctly detected and resolved by the synchronization mechanism of PDCPD. Recall that communication channels are assumed to be noise-free and FIFO; therefore, the possibility of signal damage, loss or re-ordering is not a consideration here.

As discussed in Chapter 4, developing a proof for a concurrent program in the new axiomatic system consists of two steps: First,
A Terminating Specification of PDCPD for a Given Protocol \( \Theta \):

\[ [P_1||...||P_n], \]

where \( P_1, ..., P_n \) are \( n \) communicating processes involved in \( \Theta \).

Specification of an Individual Process \( P_i \):

/ variable list:
  \( \text{path}_{i,j} \) : the path of process \( P_i \);
  \( \text{path}_{i,j}^j \) : the path of process \( P_j \) recorded by process \( P_i \) if \( j \);
  \( \text{h-path} \) : the path with the highest precedence among \( \text{path}_{i,1}, ..., \text{path}_{i,n} \);
  \( \|S\| \) : the cardinality of set \( S \).

abbreviation: \( \prod_{i=1}^n S_i \) abbreviates \( S_1; ...; S_n \).

/ All the paths are empty before the program begins. /

\( \text{path}_{i,1}, ..., \text{path}_{i,n} := \Lambda, ..., \Lambda; \)

\( \text{h-path} := \Lambda; \)

*{ / \( P_i \) receives a signal \( <[P,e],t> \) from \( P_j \). /}

\( \Box_{P_j \in \text{PROCESS}_{\Theta} \land P_j \neq P_i}
  \{ \text{path}_{i,j} \neq \text{path}_{i,j} \land \sim \text{FINAL}_{\Theta}(\text{State}_{\Theta}(\text{path}_{i,j})) \} \rightarrow
  \text{path}_{i,j} := \text{Path}_{\Theta,j}(\text{path}_{i,j},<[P,e],t>);\)

\( \{ \text{h-path}_{\Theta,j} \rightarrow \text{h-path} := \text{path}_{i,j} \} \)

/ \( P_i \) initiates a transition of type \( e \) at the end of \( \text{path}_{i,i} \). /

\( \Box_{e \in \text{TYPE}_\Theta} \{ [P_i,e] \in \text{OT}_\Theta(\text{State}_\Theta(\text{path}_{i,i})) \land \text{h-path}_{\Theta,j}(\text{path}_{i,i} \circ [P_i,e]) \rightarrow
  \prod_{P_j \in \text{PROCESS}_{\Theta} \land P_j \neq P_i} P_j [[P_i,e], ] \text{path}_{i,i} ];\)

\( \text{path}_{i,i}, \text{h-path} := \text{path}_{i,i} \circ [P_i,e], \text{path}_{i,i} \circ [P_i,e] \)

Figure 38: An equivalent terminating specification of PDCPD
/ \( P_i \) "catches up" h-path by taking transition \([P_k, e]\) which is on h-path. /

\[
P_k \in \text{PROCESS} \& e \in \text{TYPE} \\
\{ \text{path}_i, i < \text{h-path} \& [P_k, e] = \text{Elem}(\text{h-path}, |\text{path}_i, i|) \\
\& \text{path}_i, i \circ [P_k, e] \subset \text{path}_i, k \} \rightarrow \\
\{ \| \text{OT}_e(\text{State}(\text{path}_i, i)) \| > 1 \lor \text{FINAL}_e(\text{State}(\text{path}_i, i [P_k, e])) \} \rightarrow \\
/ \( P_i \) transmits a confirmation about transition \([P_k, e]\). /

\[
\prod_{P_j \in \text{PROCESS} \& P_j \neq P_i} P_j! [P_k, e], |\text{path}_i, i| > ] \\
\text{path}_i, i := \text{path}_i, i \circ [P_k, e]
\]

/ \( P_i \) backtracts to \( \text{Common}(\text{h-path}, \text{path}_i, i) \) and transmits a confirmation about transition \([P_k, e]\). /

\[
P_k \in \text{PROCESS} \& e \in \text{TYPE} \\
\{ \text{path}_i, i < \text{h-path} \& [P_k, e] = \text{Elem}(\text{h-path}, |\text{Common}(\text{h-path}, \text{path}_i, i)|) \\
\& \text{Common}(\text{h-path}, \text{path}_i, i) \circ [P_k, e] \subset \text{path}_i, k \} \rightarrow \\
\prod_{P_j \in \text{PROCESS} \& P_j \neq P_i} P_j! [P_k, e], |\text{Common}(\text{h-path}, \text{path}_i, i)| > ; \\
\text{path}_i, i := \text{Common}(\text{h-path}, \text{path}_i, i) \circ [P_k, e]
\]

Figure 38 (continued)
the proofs for individual processes are developed to account for the processes' effect on their local variables and the communication channels which provide transportation services. Then, the proofs of the individual processes are combined to deduce the desired properties of the entire concurrent program. The rest of this section describes the details of the necessary proofs in verifying PDCPD.

5.3.1. Proof of an Individual Process in PDCPD

As discussed in Chapter 4, input and output assertions are used to reduce the complexity of proving the individual processes. An input assertion is the assumed pattern of input signals which one process receives from another process. When reasoning about the effect of an input command in an individual process, one considers only the input values that satisfy the associated input assertion. In contrast, an output assertion is a verifiable pattern of output signals which one process transmits to another, given certain input assertions as assumptions.

Given the formal specification of PDCPD for an arbitrary protocol @, which has been shown in Figure 33, the following lists several properties that are provable in an individual process. Let $P_i$ be an individual process in @. The input and output assertions associated with $P_i$ are specified as follows:
is\textsubscript{i,j} = \text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle \downarrow \\
& \forall e, S_1, S_2 \cdot (\text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle = S_1 \circ [P_i, e] \circ S_2 \\
& \Rightarrow \exists \text{OUT}' \cdot (\text{OUT}' \leq \text{OUT}_{i,j} \\
& & \& \text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{OUT}'\rangle = S_1 \circ [P_i, e]))},

os\textsubscript{i,j} = \text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{OUT}_{i,j}\rangle \downarrow \\
& \forall k, e, S_1, S_2 \cdot (\text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{OUT}_{i,j}\rangle = S_1 \circ [P_k, e] \circ S_2 \& P_k \neq P_i \\
& \Rightarrow \exists \text{IN}' \cdot (\text{IN}' \leq \text{IN}_{i,k} \\
& & \& \text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}'\rangle = S_1 \circ [P_k, e]))},

where f(a) \downarrow means that function f is defined over value a.

The assertion Path\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle \downarrow in is\textsubscript{i,j} is necessary because Path\textsubscript{\textcircled{i},j} is a partial function. As discussed in Section 5.1, if function Path\textsubscript{\textcircled{i},j} is defined over \langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle, then the value of Path\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle is the path of process P_j recorded in process P_i.

The other assertion appearing in is\textsubscript{i,j} is

\forall e, S_1, S_2 \cdot (\text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle = S_1 \circ [P_i, e] \circ S_2 \\
& \Rightarrow \exists \text{OUT}' \cdot (\text{OUT}' \leq \text{OUT}_{i,j} \& \text{Path}\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{OUT}'\rangle = S_1 \circ [P_i, e]))}.

This assertion states the following property: Assuming that transition [P_i, e] is a part of the recorded path of P_j in P_i (i.e., Path\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle), and S_1 and S_2 are two parts in Path\textsubscript{\textcircled{i},j}\langle\text{\Lambda},\text{IN}\textsubscript{i,j}\rangle which precedes and follows [P_i, e], respectively, then [P_i, e] must
have been initiated by process $P_i$ at $Stateg(S_i)$.

The output assertion $os_{i,j}$ also consists of two individual assertions. Their meanings can be interpreted similarly.

From Figure 33, one can verify that the main body of process $P_i$ is a guarded repetitive command. Following the axioms and rules of inference given in Section 4.4, one can verify that the following $LI_i$ is a loop-invariant for this repetitive command: (It is reminded that a loop-invariant is an assertion whose truth is preserved by each iteration of a repetitive command.)

$$LI_i = AS_i(1) \land AS_i(2) \land AS_i(3) \land AS_i(4),$$

where

$$AS_i(1) = \forall P_j \in PROCESS@ \ (path_{i,j} \subseteq g-h-path)$$
$$\land \exists P_j \in PROCESS@ \ (path_{i,j} = h-path)$$

$$AS_i(2) = \forall P_j \in PROCESS@ \land P_j \neq P_i \cdot (os_{i,j})$$

$$AS_i(3) = \forall k,e \in S_1,S_2 \cdot (path_{i,j} = S_1o[P_k,e]oS_2 \land P_k \neq P_i)$$
$$\Rightarrow \exists IN'. (IN' \leq IN_i,k, Path^{\star}_{g,k}(\Lambda,IN'^{\star}) = S_1o[P_k,e])$$

$$AS_i(4) = \forall P_j \in PROCESS@ \land P_j \neq P_i \cdot (AS_i(4,1) \land AS_i(4,2) \land AS_i(4,3))$$

$AS_i(4,1), AS_i(4,2)$ and $AS_i(4,3)$ are defined below:

$$AS_i(4,1) = \text{path}_{i,j} = \text{Path}^{\star}_{g,j}(\Lambda,IN_{i,j})$$

$$AS_i(4,2) = \text{Path}^{\star}_{g,j}(\Lambda,OUT_{i,j}) \land \text{path}_{i,j}$$

$$AS_i(4,3) = \forall L > 0 \cdot (\forall u(1),\ldots,u(L),e(1),\ldots,e(L))$$

$$\cdot (\text{path}_{i,j} = \text{Path}^{\star}_{g,j}(\Lambda,OUT_{i,j})$$
$$\circ [P_{u(1)},e(1)] \circ \ldots \circ [P_{u(L)},e(L)]$$
\[
\forall k \in \{1, \ldots, L\} \cdot (P_u(k) \neq P_i \\
\& \text{OUT}(\text{State}(\text{Path}^\ast_{i,j}(\Lambda, \text{OUT}_{i,j}))) \\
\o[P_u(1), e(1)] \circ \ldots \circ [P_u(k-1), e(k-1)]) \\
= \{(P_u(k), e(k))\})
\]

AS\(_i\)(1) states that variable h-path records the path with the highest precedence among path\(_i,1, \ldots, path\(_i,n, as discussed in Section 5.2.

AS\(_i\)(2) states that output assertion os\(_{i,j} \ (1 \leq j \leq n \text{ and } j \neq i)\) is satisfied before each iteration of the guarded repetitive command in P\(_i\). AS\(_i\)(2) is included in LI\(_i\) because, according to the new axiomatic verification system, output assertions must be a verifiable property of the individual processes.

AS\(_i\)(3) states that, if transition [P\(_k\), e] appears as a part of path\(_i,i\) and if S\(_1\) and S\(_2\) are the parts of path\(_i,i\) that precedes and follows [P\(_k\), e], respectively, then [P\(_k\), e] must have been initiated by process P\(_k\) at State(S\(_1\)).

AS\(_i\)(4,1) states that the recorded path of process P\(_j\) in process P\(_i\) (i.e., path\(_i,j\)) is given by the value of Path\(_{i,j}^\ast((\Lambda, \text{IN}_{i,j}))\), as discussed in Section 5.1.

AS\(_i\)(4,2) states that Path\(_{i,j}^\ast((\Lambda, \text{OUT}_{i,j}))\) is always a subpath of path\(_i,i\), the path of process P\(_i\). It also states that in case Path\(_{i,j}^\ast((\Lambda, \text{OUT}_{i,j}))\) is a proper subpath of path\(_i,i\), as illustrated in
Figure 39, then each transition taken by $P_i$ after $\text{Path}^*_{\epsilon, i}(\lambda, \text{OUT}_{i,j})$ cannot have been initiated by $P_i$ and must be the only output transition at the state where the transition has been initiated. This assertion can be explained by the following example. Suppose that $\text{Path}^*_{\epsilon, i}(\lambda, \text{OUT}_{i,j})$ is a proper subpath of $\text{path}_{i, i}$ and

$$\text{path}_{i, i} = \text{Path}^*_{\epsilon, i}(\lambda, \text{OUT}_{i,j})$$

$$\circ [P_u(1), e(1)] \circ \ldots \circ [P_u(k), e(k)] \circ \ldots \circ [P_u(L), e(L)]$$  \((A1)\)  

where $L > 0$. As shown in Figure 39, if $[P_u(k), e(k)]$ had been initiated by $P_i$ (i.e., $P_i = P_u(k)$), then process $P_i$ would have transmitted a signal $<[P_i, e(k)], t+k-1>$ to indicate its initiation of $[P_i, e(k)]$ at node $N(k-1)$. Conversely, if $P_i \neq P_u(k)$ but transition $[P_u(k), e(k)]$ were not the only output transition of node $N(k-1)$, then process $P_i$ would have transmitted a signal $<[P_u(k), e(k)], t+k-1>$ to confirm its following of $[P_i, e(k)]$ at node $N(k-1)$. In either case, from the definition of $\text{Path}^*_{\epsilon, i}$, one would conclude that

$$\text{path}_{i, i} = \text{Path}^*_{\epsilon, i}(\lambda, \text{OUT}_{i,j})$$

$$\circ [P_u(k+1), e(k+1)] \circ \ldots \circ [P_u(L), e(L)],$$

which contradicts the assumption (A1).

Proving that $\text{LI}_i$ is indeed a loop-invariant of the guarded repetitive command in process $P_i$ is a complicated task; nevertheless, it can be done by mechanically applying the axioms and
Figure 39: Illustration for $AS_4(4,2)$ and $AS_4(4,3)$. 
rules of inference for the commands that appear in $P_i$ (i.e., axioms A1-A3 and rules of inference R1-R4 and R6-R8). These axioms and rules of inference have been given in Section 4.4; therefore, the details of the proof of $LI_i$ are omitted here.

5.3.2. Proof of the Entire Concurrent Program of PDCPD

At this point, the proofs of individual processes can be combined to deduce the desired properties of the entire concurrent program. As stated in Section 5.2, when the entire concurrent program is blocked in a deadlock state, one desires that:

1. All paths of the communicating processes are equivalent or all the collisions that have occurred during the execution of protocol have been detected and resolved.

2. All processes have reached a final state; i.e., they have properly terminated.

The above desired properties can be expressed by the following two assertions:

\[
\begin{align*}
\text{path}_{i,1} = \ldots = \text{path}_{n,n} & \quad \text{(S1)} \\
\forall P_i \in \text{PROCESS} \cdot (\text{State}_G(\text{path}_{i,i}) \in \text{FINAL}_G) & \quad \text{(S2)}
\end{align*}
\]

Their proofs are given as follows.

According to the rule of inference for parallel composition, in proving the properties of the entire concurrent program, in addition
to the proofs of individual processes, one has to prove the following properties:

1. The validity of output assertions before the entire concurrent program starts; i.e.,

\[ \forall P \in \text{PROCESS} \& P \neq P_j \ (\text{IN}_i, j = \text{OUT}_i, j = \Lambda \Rightarrow os_{i, j}) \]  

where \( os_{i, j} \) has been defined in Subsection 5.3.1

2. The consistency between input and output assertions; i.e.,

\[ \forall P \in \text{PROCESS} \ (os_{j, i} \& \text{OUT}_j, i = \text{IN}_i, j \Rightarrow is_{i, j}) \]

\[ \forall P \in \text{PROCESS} \ (os_{j, i} \& \text{OUT}_j, i = \text{IN}_i, j \Rightarrow is_{i, j}) \]  

where \( is_{i, j} \) and \( os_{i, j} \) have been defined in Subsection 5.3.1

The proof of (1) is trivial. The proof of (2) requires some reasoning technique that is not specified in the new axiomatic system. In this dissertation we are not attempting to formalize this technique; however, one should be able to grasp the fundamental principle of this reasoning technique from the following argument:

From \( os_{j, i} \), one knows that \( \text{Path}_j^* \ (\Lambda, \text{OUT}_j, i) \) is defined. Since \( \text{OUT}_j, i = \text{IN}_i, j \) is a given condition in the consistency proof, one concludes that

\[ \text{Path}_j^* \ (\Lambda, \text{IN}_i, j) \downarrow \]

which is a part of \( is_{i, j} \).

In order to complete the consistency proof, one still needs to prove that, given \( os_{i, j} \) and \( \text{OUT}_j, i = \text{IN}_i, j \),
\[ \forall e, S_1, S_2 \cdot (\text{Path}^*_{e, j}(\Lambda, \text{IN}_{i, j}) = S_1 \circ [P_i, e] \circ S_2) \]
\[ \Rightarrow \exists \text{OUT} \leq \text{OUT}_{i, j} \cdot (\text{Path}^*_{e, i}(\Lambda, \text{OUT}^*) = S_1 \circ [P_i, e]) \].

Assuming that \( \text{Path}^*_{e, j}(\Lambda, \text{IN}_{i, j}) = S_1 \circ [P_i, e] \circ S_2 \), then because \( \text{OUT}_{j, i} = \text{IN}_{i, j} \) and \( P_j \neq P_i \), from \( \text{os}_{j, i} \) one knows that

\[ \exists \text{IN} \leq \text{IN}_{j, i} \cdot (\text{Path}^*_{e, i}(\Lambda, \text{IN}^*) = S_1 \circ [P_i, e]) \]

Note that if \( \text{OUT}_{j, i} = \text{IN}_{i, j} \) is true, the time frames referred to by \( \text{os}_{j, i} \) and \( \text{is}_{i, j} \) are different: Given a sequence \( s \) such that \( \text{OUT}_{j, i} = \text{IN}_{i, j} = s \), due to the signal propagation delay, the time frames of \( \text{os}_{j, i} \) precedes that of \( \text{is}_{i, j} \). In other words, suppose that the time frames of \( \text{os}_{j, i} \) and \( \text{is}_{i, j} \) are \( t_1 \) and \( t_2 \), respectively, then \( t_1 < t_2 \).

Let the time frames of \( \text{os}_{j, i} \) and \( \text{is}_{i, j} \) be \( t_1 \) and \( t_2 \), respectively. \( \text{IN}_{j, i} \) (in \( \text{os}_{j, i} \)) denotes the sequence of input signals which process \( P_j \) has received from process \( P_i \) by time \( t_1 \). Let the time at which process \( P_i \) has transmitted the sequence of signals \( \text{IN}_{j, i} \) (in \( \text{os}_{j, i} \)) to \( P_j \) be \( t_0 \). Then, because there is a signal propagation delay, \( t_0 < t_1 \). Since \( t_0 < t_1 < t_2 \) and the time frame of \( \text{OUT}_{i, j} \) (in \( \text{is}_{i, j} \)) is \( t_2 \), one concludes that \( \text{IN}_{j, i} \) (in \( \text{os}_{j, i} \)) is an initial subsequence of \( \text{OUT}_{i, j} \) (in \( \text{is}_{i, j} \)). Therefore, let \( \text{IN}^* \) be an initial subsequence of \( \text{IN}_{j, i} \) (in \( \text{os}_{j, i} \)),

\[ \text{IN}^* \leq \text{IN}_{j, i} \circ \text{(in os}_{j, i} \circ \leq \text{OUT}_{i, j} \circ \text{(in is}_{i, j} \circ \).
\]
Therefore,

\[ \text{Path}^*_{i\in\Lambda,\text{IN}^i} = S_1 \circ [P_i; e], \]

and the proof of consistency between \( os_{j,i} \) and \( is_{i,j} \) is completed.

According to the semantics of CSP given in Section 4.2, an individual process can be blocked only by an input command. As shown in Figure 33, the guarded commands of the first type have a pure input guard, and input commands appear only in input guards. Therefore, when the entire concurrent program of PDCPD is blocked in a deadlock state, the following properties are satisfied

1. Loop-invariant \( LI^*_i \) (1≤i≤n) is true.
2. All the transmitted signals have been received. (Otherwise, the guarded commands of the first type will be invoked to receive signals, contradicting the assumption that the program is blocked.)
3. The boolean guards of the guarded commands of the other three types are false. (Otherwise, these guarded commands will be invoked to execute, contradicting the assumption that the program is blocked.)

The three properties mentioned above are expressed by the following assertions:

\[ LI^*_{i_1} \land \ldots \land LI^*_{i_n} \quad \text{(B1)} \]
\[ \forall i_1, i_2 \in \text{PROCESS} \cdot (\text{OUT}_{i_1, i_2} = \text{IN}_{i_2, i_1}) \quad \text{(B1)} \]
\[ \forall p_i \in \text{PROCESS}_{\leq} \left( \text{path}_{i,j} \preceq h\text{-path} \land \text{path}_{i,j} \not\preceq h\text{-path} \right) \]

& \forall e \in \text{TYPE}_{\leq} \left( \left[ p_i, e \right] \not\in \text{OT}_{\leq} (\text{State}_{\leq} (\text{path}_{i,j})) \right) \] (B3)

Property (S1) can be proved by contradiction as follows:

Since relation \( \preceq \) is linear, one may assume that the path of process \( P_k \) has the highest precedence; i.e., for all processes \( P_i \) in \( \leq \), \( \text{path}_{i,j} \preceq \text{path}_{k,j} \). Assume that there is a process \( P_i \) such that \( \text{path}_{i,j} \preceq \text{path}_{k,j} \). Let \( \text{Last}(\text{path}_{k,j}) = [p_j, e] \).

**Case 1:** \( P_k = P_j \)

From \( \text{AS}_{\leq} (4,3) \) and \( \text{AS}_{\leq} (4,1) \), one knows that

\[
\text{Path}_{\leq}, (\Lambda, \text{OUT}_{i,j}, k) = \text{path}_{k,j} \]

and

\[
\text{Path}_{\leq}, (\Lambda, \text{IN}_{i,j}, k) = \text{path}_{i,j} \cdot
\]

Therefore, by (B2),

\[
\text{path}_{k,j} = \text{Path}_{\leq}, (\Lambda, \text{OUT}_{i,j}, k) = \text{Path}_{\leq}, (\Lambda, \text{IN}_{i,j}, k) = \text{path}_{i,j} \cdot
\]

Also from (B3),

\[
(\text{path}_{i,j} \preceq h\text{-path} \land \text{path}_{i,j} \not\preceq h\text{-path}) \Rightarrow \text{path}_{i,j} = h\text{-path} \cdot
\]

However, combining the above results with \( \text{AS}_{\leq} (1) \), one gets

\[
\text{path}_{k,j} = \text{path}_{i,j} \cdot
\]
which contradicts the assumption that \( \text{path}_{i,i} < \text{path}_{k,k} \).

**Case 2:** \( P_k \neq P_j \)

From \( AS_k(3) \), one knows that there is an initial subsequence \( IN^\prime \) of \( IN_{k,j} \) such that

\[
\text{Path}^*_i,j(\Lambda, IN^\prime) = S_1^0[P_j,e] = \text{path}_{k,k}
\]

Combining the above result with the definition of \( \text{Path}^*_i,j \), (B2) and \( AS_j(1) \), one gets

\[
\text{path}_{k,k} = \text{Path}^*_i,j(\Lambda, IN^\prime) <= @ \text{Path}^*_i,j(\Lambda, IN_{k,j}) = \text{Path}^*_i,j(\Lambda, OUT_{j,k}) <= @ \text{path}_{j,j}.
\]

Since \( \text{Path}_{j,j} \leq @ \text{path}_{k,k} \) is assumed, the above result asserts that \( \text{path}_{j,j} = \text{path}_{k,k} \). Therefore,

\[
\text{path}_{i,i} < @ \text{path}_{j,j} \quad \text{and} \quad \text{Last}(\text{path}_{j,j}) = [P_j,e]
\]

However, the proof of Case 1 now is applicable. Therefore, one must
conclude that $\text{path}_{i,i} = \text{path}_{k,k}$ and (S1) must be true.

From (S1), one knows that all the communicating processes terminate at the same state when the concurrent program is blocked. Let this state be $s$. Then $s = \text{State}_@ (\text{path}_{i,i})$. By (B3), one knows that $s$ has no output transition; therefore, $s$ must be a final state in $@$, and (S2) is proved.

5.4. Summary

In this chapter Hoare's Communicating Sequential Processes (CSP) has been used to specify the Priority Driven Communication Protocol Design (PDCPD) for an arbitrary protocol. The specification has been verified by the new axiomatic system for CSP introduced in Chapter 4 to prove that the communicating processes have resolved all the collisions among them and reached a final state when the execution of the protocol is completed. The verification has been divided into two stages. First the loop invariants for the individual processes have been proven. Then the proofs of individual processes have been combined by the inference rule for parallel composition to derive the desired system properties. Since the verification is independent of any particular property of the protocol, one concludes that PDCPD is applicable to any protocols.
6. SUMMARY AND CONCLUSIONS

This chapter summarizes the main results of the research, indicates some possible directions for future research, and presents some concluding remarks.

6.1. Summary of the Dissertation

The object of this dissertation was to provide a methodology which would be used in designing communication protocols. Chapter 1 pointed out the economic incentive and motivation of using computer networks in today's society. The role of communication protocols in computer networking was described and the importance of correctly functioning protocols was emphasized. Precise representation of protocols has been made possible due to the development of a layered structure of protocols. Two major approaches, state transition models and programming language models, have found extensive use in precise specification of communication protocols. Protocol verification techniques, which are the means to examine the logical correctness of communication protocols, however, have suffered from problems of complexity. The problem of state space explosion appears in the state transition models while the complexity in
proofs of the correctness for concurrent systems in programming language models have limited the application of these techniques to protocols of only modest complexity.

The second chapter described the layered approach to designing complex software systems and its application to protocol design. An international standardized layered architecture of communication protocols, the ISO's Reference Model for Open Systems Interconnections, was presented for illustration. The meaning of specification and verification was clarified in the context of layered protocol architecture. Several major models for protocol specification and verification were compared. Finite state automata, formal grammars and Petri nets were found, even though represented in different forms, to be based on the similar idea of modelling events in protocol processing as transitions between various states. The reachability analysis technique used by these models has the advantage of ease in checking the general correctness properties, viz. freedom from deadlock, completeness, etc., and ease in automating the verification procedure. However, the exponential growth of global system states, as the number or complexity of individual processes increases, limits the applicability of these models only to simple protocols. Programming language models stem from the observation that communication protocols are simply one form of algorithm, and that high-level programming languages provide
a clear and concise means of describing algorithms. These models are superior to state transition models in handling variables and verifying the full range of protocol properties. The Floyd/Hoare style verification techniques have been found, in practice, quite difficult to use because of the amount of human ingenuity and intuition required.

Chapter 3 presented PDCPD, a state transition model based method for designing communication protocols, and discussed its application to an actual protocol, specifically the protocol between the Data Terminal Equipment and Data Circuit-Terminating Equipment. A connection establishment protocol was used to demonstrate that even simple protocols can exhibit very complicated behavior because it must cope with asynchronously operating processes. Conventional state transition models were shown to be inadequate in designing complex protocols. The coupling relationship between transitions in different processes, represented in the form of intention functions, was used to enhance the conventional models. A new reachability analysis algorithm was proposed. The complexity of this algorithm, being a tool for verifying the general correctness properties of protocols, was shown to be significantly lower than that of conventional reachability analysis techniques. The resulting desirable interaction graphs generated by this new algorithm are regarded as the intended interactions among communicating processes
because the intention of each transition has been taken into consideration. The conditions that violate the desirable interactions were called collisions. PDCPD was developed based on the belief that the protocol designer should not be burdened by the work of handling collisions during the design phase. The communicating processes were provided with a synchronization mechanism so that they can detect and resolve collisions dynamically during the execution of the protocol. Two items of information, logical time and confirmation, were found necessary in detecting collisions. A priority scheme and a backtracking scheme were proposed to resolve collisions. The usefulness and advantages of PDCPD were seen in its application to CCITT X.21 Recommendation.

Chapter 4 commented on the controversy of using axiomatic program verification techniques. Although difficulties still remain, such rigorous, mathematically based methods were believed to be a promising approach to understanding complex software systems as well as communication protocols. The notation of CSP, the language adopted as the tool for investigating the semantics of protocols, was described briefly. Several existing axiomatic systems for CSP were discussed and compared. The system of Levin and Gries and the system of Apt et al. were found to provide better means in handling communication commands at the individual process level. On the other hand, the strength of Soundararajan's system lies in proving
that the proofs of individual processes are interference-free. A new axiomatic system for CSP was proposed, which retains the advantages of the other systems. The use of input/output assertions constructed from history variables was discussed, and was determined that it can reduce the complexity in proving individual processes since the consideration of impossible input values is excluded. At the entire concurrent system level, the proof of consistency among the proofs of individual processes is made easier because it is done on the basis of processes, rather than on the basis of individual commands. Several case studies illustrated the advantage of using the new axiomatic system in coping with complex communication protocols.

Finally, Chapter 5 used the axiomatic system described in Chapter 4 to ensure the applicability of PDCPD to arbitrary protocols. PDCPD was specified as a concurrent CSP program. This chapter proved that the synchronization mechanism of PDCPD enables indeed the communicating processes to detect and resolve any collision that may occur, and ensures all the processes will reach a desired final state when the communication is completed.
6.2. Areas for Future Research

As described in Chapter 2, there are two types of specifications for communication protocols. Most formal techniques concentrate on protocol specifications, and little attention has been paid on service specifications. Currently, service specifications are usually done by ad hoc techniques such as using service primitives. We believe that a systematic method to define protocol services will not only provide precise and clear service specifications but influence the development of protocol verification techniques. Research in this area is just beginning.

A concern of PDCPD is the overhead involved in protocol execution. Overhead is generated from two sources: (1) Appending logical time to communication signals means the use of longer signals. (2) The use of confirmation signals increases the network traffic. In our opinion, the overhead generated by PDCPD would not impose any major problem to network performance. The overhead of (1) is negligible since the logical time occupies only a small fraction of the entire signals. The overhead of (2) can be minimized by combining confirmation signals with the "ordinary" signals used to indicate the initiations of transitions. More importantly, operational overhead is also involved in conventional protocol design methods since they resolve collisions by introducing extra states and extra transitions into the implementations of
individual processes. However, a systematic method to evaluate the performance of PDCPD can help verify this point.

A more serious concern of PDCPD is the communication mode used by this method. As discussed in Chapter 3, the synchronization mechanism requires each process to broadcast each transition it has initiated to all other processes. This means that each initiated transition must be made known to all other processes through a signal sent by the initiating process. Full-broadcast communication may not only increase network traffic, but also impose a threat to security and privacy. In some situations, such as battlefield communication, this represents a serious problem. Due to such consideration, research in incorporating other communication modes into PDCPD is necessary. The synchronization mechanism required in a more flexible communication mode is expected to be more complex than that used in the full-broadcast mode.

The new axiomatic system for CSP has been successfully applied to verify that PDCPD does perform its intended function -- to resolve collisions. The suitability of this system in verifying the semantic properties of other real protocols, such as CCITT X.25 [11], X.75 [12] and DoD's TCP [63, 64], remains as an important area for future study. Additional rules of inference, helpful for further reducing the complexity in verifying concurrent programs as well as communication protocols, are expected to be discovered from
this future research effort.

A more fundamental question of much interest is the soundness and completeness of the new axiomatic system. An axiomatic system is sound if and only if its theorems (the assertions which can be proved by the system) are all true in the intended interpretation, and it is complete if and only if any assertions which are true in the intended interpretation are theorems of the system [67]. Soundness is the primary concern since an axiomatic system will not produce correct results unless it is sound. The new axiomatic system introduced in Chapter 4 is intuitively sound, but this point is yet to be proved. Furthermore, because one of the objectives of the research effort is to develop a formal system which can handle a wide range of protocol properties, the completeness of this axiomatic system also remains as an important research topic.

Since most interesting properties of protocol semantics are time-related, the possibility of incorporating temporal logic [43, 61, 62] into the new axiomatic system for CSP should be investigated. The study of combining program verification with temporal logic is just beginning, and we see a need for developing a systematic way to do it.
6.3. Conclusions

Formal techniques for communication protocol design have been drawing much attention since the use of computer networks. Tremendous effort put into the development of these techniques reflects the demand of protocols of increasing variety and complexity. Formal specification techniques provide a clear definition of the processes that cooperate to provide services. Verification techniques assist in demonstrating that processes are indeed adequate to provide their intended services and possess some general correctness properties such as completeness and freedom from deadlock. It is clear that formal techniques can be used to design correct protocols, to resolve the compatibility problem in protocol implementations, and to lower the cost of protocol maintainence.

Although there has been considerable progress in the logical verifications of communication protocols, we believe that more effort should be spent on this area in order to resolve the problem of complexity. As pointed out at the beginning of this dissertation, neither state transition models nor programming language models have provided satisfactory solutions for protocol verifications. The state space explosion problem in the former models and the level of human ingenuity and intuition involved in the latter models have limited the applicability of formal techniques to verifying simple communication protocols.
The use of PDCPD gives the protocol designer a powerful tool to tackle the state space explosion problem. No longer must the designer need to resolve collisions. He can now concentrate on the functional aspects of his task, leaving the resolution of collisions to the synchronization mechanism. The complexity in verifying general correctness properties of protocols has been greatly reduced so that verifications are made possible for complex protocols. That alleviation of verification complexity represents an achievement of this research effort.

The new axiomatic system is an attempt to reduce the complexity in verifying the semantic aspects of communication protocols. This system has retained the advantages of other existing axiomatic systems for verifying concurrent programs. The use of input/output assertions assists the protocol verifier in reasoning the logical effects of communication on local processes and in checking the interference-freeness of the proofs of individual processes. That assistance for the protocol verifier represents another achievement of this research effort.
BIBLIOGRAPHY


44 Lamport, L. "A New Approach to Proving the Correctness of Multiprocess Programs", ACM Transactions on Programming Languages and Systems 1, 1, July 1980, 84-97.


48 Lin, H., M.T. Liu and C.J. Graff "Verification of a Methodology for Designing Reliable Communication Protocols", accepted for presentation in the 8th Data Communication Symposium, Cape Code, Massachusetts, October 1983.


Schindler, S. "Synchronized Data Types and Their Suitability for Protocol Implementations", Proceedings, the 12th Hawaii International Conference on System Science (Honolulu), 1979, 18-27.


90 Zafiroulo, P. "Protocol Validation by Duologue-Matrix Analysis", *IEEE Transactions on Communications* COM-26, 8, August 1978, 1187-1194.