INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

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

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

UMI Dissertation Information Service
University Microfilms International
A Bell & Howell Information Company
300 N. Zeeb Road, Ann Arbor, Michigan 48106
Liu, Nien-Chen

A METHODOLOGY FOR SPECIFYING AND ANALYZING COMMUNICATION PROTOCOLS AND SERVICES

The Ohio State University

University Microfilms International

Copyright 1986 by Liu, Nien-Chen

All Rights Reserved
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark \( \checkmark \).

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages \( \checkmark \)
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Dissertation contains pages with print at a slant, filmed as received
16. Other

University
Microfilms
International
Copyright by
Nien-Chen Liu
1986
Dedicated To

My Father 劉志遠 (Chih-Yuan Liu)
My Mother 魏鳳翎 (Feng-Ling Wei)
and
My Wife 張瓊慧 (Chiung-Hui Chang)
ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Prof. Ming T. Liu, who guided me with his wisdom and expertise during my entire graduate education. It has been a pleasure to have worked with him and to have learned from him in many respects.

Also, I want to give special thanks to Prof. Neelam Soundararajan. Attending his semantics group seminars was always fruitful and enjoyable, and his continuing encouragement and inspiration have been important in the completion of this dissertation.

Many people at OSU have influenced my graduate years in one way or another. In particular, Prof. Bruce Weide taught me how to present ideas effectively. Prof. William Ogden taught me how to express deep concepts in a concise and yet humorous way. My dearest friends Ian Chiou and Lin Chiu were always there when I needed help, advice, or someone to listen to my complaints. To these people, many thanks.

Above all, I wish to thank my wife Chiung-Hui for her patience, understanding, and cheerful support over the years. My parents have provided for me a pleasant environment in which I could pursue my interests ever since my childhood. To them there are no words to express my gratitude.

Finally, I am greatly indebted to Dr. Charles J. Graff for funding this research under Contract No. DAAB07-83-K-K542 from the U. S. Army Communications-Electronics Command (CECOM).
VITA

April 10, 1957  Born, Taipei, Taiwan, Republic of China

1979  B.S., Department of Engineering Science
      National Cheng Kung University,
      Tainan, Taiwan, Republic of China

1981—1983  Graduate Teaching Associate,
           Dept. of Computer and Information Science
           The Ohio State University,
           Columbus, Ohio

1983  M.S., Dept. of Computer and Information Science
      The Ohio State University,
      Columbus, Ohio

1983—present  Graduate Research Associate,
               Dept. of Computer and Information Science
               The Ohio State University,
               Columbus, Ohio
PUBLICATIONS

• "CSP-Based Specifications for Network Protocols and Services,"
  Co-author: Ming T. Liu

• "Conformity Analysis for Communication Protocols,"
  Co-author: Ming T. Liu

RESEARCH INTERESTS

• Distributed Computing Systems: Concurrent (Distributed) Programming Languages, Network Architectures, Operating Systems, Database Systems.


• Software Engineering: Programming Methodology, Program Correctness, Data Abstractions, Design and Analysis of Algorithms.
# Table of Contents

Dedication ................................................. ii  
Acknowledgment ........................................ iii  
Vita ............................................................ iv  
Table of Contents ........................................ vi  
List of Tables .............................................. ix  
List of Figures ............................................. x

1. Introduction ........................................... 1  
   1.1 Motivations ........................................ 1  
   1.2 Research Objectives and Contributions ........ 4  
   1.3 Organization of the Thesis .................... 7

2. Communication Services and Protocols .......... 10  
   2.1 Introduction ....................................... 10  
   2.2 The Concepts of Service and Service Specification .... 11  
   2.3 The Concepts of Protocol and Protocol Specification .... 13  
   2.4 Layers of Functions versus Levels of Abstractions .... 14  
   2.5 Discussions ....................................... 17

3. Formal Specification and Verification Techniques ... 20  
   3.1 Introduction ....................................... 20  
   3.2 State-Oriented Models ......................... 21  
   3.3 Programming Language Model ................. 24  
   3.4 Sequence-Oriented Models .................... 25  
   3.5 OSI Formal Description Techniques ........... 28

4. The CSP-based Specification Technique .......... 32  
   4.1 Motivations ....................................... 32  
   4.2 Communicating Sequential Processes .......... 34  
       4.2.1 Concurrency Mechanism .................... 36  
       4.2.2 Communication Mechanism .................. 36  
       4.2.3 Nondeterminism Mechanism ................. 38
List of Tables

Table 2.1. Software Engineering Principles ................................ 19
Table 4.1. Primitives of the Simplified Transport Service ............. 57
List of Figures

Figure 2.1. The Service of layer N ........................................... 12
Figure 2.2. The Protocol of layer N ........................................... 14
Figure 2.3. OSI Architecture .................................................. 16
Figure 3.1. A Transition in ESTELLE ................................. 29
Figure 4.1. A T Type Interaction ........................................... 38
Figure 4.2. The Structure of ABP Communication System .... 46
Figure 4.3. The Specification of ABP ..................................... 48
Figure 4.4. Comparisons between EFSM and CSP ............... 53
Figure 4.5. The Conceptual Structure of a Service Provider .... 54
Figure 4.6. Functional Behavior of a Simplified Transport Service .................................................. 60
Figure 4.7. The Structure of the TS Specification ............... 61
Figure 4.8. Specification of a Simplified Transport Service ... 62
Figure 4.9. Viewing a CSP specification as a sequence gener- ator .......................................................... 67
Figure 5.1. The Objective of Conformity Analysis ................ 72
Figure 5.2. Two Communicating Agents ............................... 83
Figure 5.3. The Integrated Communicating Agent ............... 84
Figure 5.4. The Tree Representation of Agents ..................... 86
Figure 5.5. An Overview of Conformity Analysis ................. 90
Figure 7.1. ABP Communication System ......................... 108
Figure 7.2. SERVICE Graph .............................................. 121
Figure 7.3. ABP Graph .................................................. 122
Figure 7.4. X.25 DTE/DCE Interface State Diagrams ........... 124
Figure 7.5. Unspecified Reception Error ............................. 125
Figure 7.6. CSP-based specification of DTE ....................... 126
Figure 7.7. CSP-based specification of DCE ....................... 128
Figure 7.8. Transmission Media ........................................... 130
CHAPTER I
Introduction

1.1 Motivations

Ever since the successful merger of information processing systems and data communication networks in the early 70s, computer communication has evolved into an indispensable tool in almost any data communications and information processing application. In the last decade, the rapid proliferation of computer networks, either as the product of commercial manufacturers or the result of academic research, has greatly propelled the revolution of the information world.

At the same time, the design of well-structured communication protocols has become ever more important. As stated in [34], protocols form the nervous system of data communication networks. In the design of computer networking architecture, they play central roles in ensuring that information is exchanged between distributed information processing systems in a coherent way. In fact, computer networking architecture itself can be viewed as a hierarchical structure of protocols defining the
functional operations and capabilities of various components of the whole system.

An important research area in the design of communication protocols is the support of specification and verification techniques by which protocols can be defined and analyzed. Past experience has shown that informal and ad hoc methods can hardly manage the complexity and size of communication protocols. It has now been widely recognized that application of formal techniques is the only way to cope with the design of sophisticated communication systems.

Formal specification techniques, from the software engineering point of view, should meet various general requirements such as accuracy, readability, and generality, which are considered crucial in the development of software systems. Besides these requirements, communication protocols, being distinct from common software systems, present further challenges to the formal specification techniques as is discussed below.

First, protocols are the rules defining the interactions between distributed system components running in parallel. Consequently, rather than specifying the simple input/output behaviors of common software systems, a technique must be able to describe the communication behaviors presented in the course of the execution of concurrent processes.
Second, protocols usually involve nondeterministic activities, disruption of normal communications, or the use of time-out mechanism, all of which impose further burdens on their specifications. Finally, an ordinary software system is commonly designed to run on a particular machine and is developed by the same group of people throughout the entire development cycle. In contrast, communication protocols are intended to run on diverse types of machines and are implemented by communities in different locations. As a result, protocol specifications act as the only blueprint referenced in a variety of environments. To ensure that different implementations exhibit identical functionality, the specifications should provide precise and comprehensible guidance and, at the same time, leave the machine-dependent details to the implementors.

In recent years, despite the proposal of various approaches, the study of specification techniques still remains in the state-of-the-art stage.

Coupled with the specification techniques are the formal verification techniques. Essentially, protocol verification is a demonstration of the correctness of a protocol design. A protocol is considered to be correct if it satisfies two kinds of properties, viz., syntactic properties (or general properties) and functional properties (or specific properties). The syntactic properties are those desired properties common to all protocols such as freedom from deadlock, completeness and
progress. They form the set of implicit requirements which any protocol should fulfill to ensure that its logical structure is absent from syntactical errors. The absence of syntactical errors, however, does not necessarily imply that the protocol will attain its intended functionality. In this regard, the functional properties of a protocol define the specific objectives of the protocol. They are usually presented in terms of a set of behaviors, called the communication service, as perceived by the protocol users. As mentioned earlier, a protocol can engage in extremely complicated interactions which are beyond human anticipation. A formal analysis is required to ensure that the functional behavior of a protocol conforms to the designer’s intention.

To date, while the syntactic properties of protocols have been extensively studied and relatively well understood, much work remains to be done on the functional analysis.

1.2 Research Objectives and Contributions

In this research, we aim to explore a methodology for specifying and analyzing communication systems. The Open System Interconnection (OSI) Reference Model [10, 19] is targeted as our underlying architectural basis. The major results of this work can be described in two stages:
(1) Formal Specifications for Protocols and Services

In the OSI Reference Model, communication services and communication protocols represent two levels of abstraction. Due to their inherently distinct characteristics, most of the existing formal specification techniques are experiencing a dilemma. As we have found, a particular technique may be likely to handle one level of abstraction quite well but have difficulty (if at all possible) dealing with the other.

In this thesis, a formal specification technique based on Communication Sequential Processes (CSP) [17] is proposed. Modifications are made in order to fit this language into the protocol domain. The primary advantages of this technique are listed below.

- A uniform formalism is used to specify communication protocols as well as their intended services. Therefore, both communication protocols and services can be interpreted and analyzed on a common semantic basis.

- For communication protocols, the specifications can be viewed as State Transition Machines—the most widely used approach to describe protocols.

- For communication services, the specifications can be viewed as Sequence Generators—a natural way to express services.

(2) Conformity Analysis for Protocols against their Services

As mentioned previously, most of the early techniques for protocol
verifications were intended to show the absence of syntactic errors. Among others, much of the work has endeavored to verify a set of ad hoc and probably incomplete functional properties. Few attempts have been made to prove that a protocol can indeed provide the communication service for which it has been designed.

As the second goal of this research, we propose a transformational approach for the conformity analysis of communication protocols — a demonstration showing that the functionality exhibited by a protocol conforms to its intended service. Based on the CSP-based specification technique, a transformation system is developed to extract from a specification the communication sequences that may arise during its execution and to express these sequences in terms of behavior expressions in Milner's CCS (Calculus of Communicating Systems) [28]. By performing algebraic manipulations and the equivalence proof on these expressions, we can show that the external behavior of a protocol conforms to its intended service. The features of this approach are listed below.

• In this approach, rather than performing logic reasoning as in the conventional axiomatic approaches for programming languages, functional analysis is mainly conducted through systematic transformation procedures and algebraic manipulations.

• The transformation system is simple and syntax-directed. In particular, the "invariant properties" can be easily obtained for communicating agents which perform cyclic operations.
Besides analysis purposes, the derived behavior expressions can further aid the understanding of the communication behaviors presented by the specifications.

1.3 Organization of the Thesis

In this chapter, the primary research areas, objectives and approaches have been outlined. The remainder of the dissertation is organized as follows.

In Chapter 2, the concepts of communication services and protocols in the framework of the Open System Interconnection (OSI) Reference Model are introduced. They serve as the architectural foundation on which our specification and analysis techniques are developed.

In Chapter 3, we briefly discuss the major specification and verification techniques proposed in the literature. To pave the way for making comparisons to our approach, the advantages and shortcomings of these techniques are also examined.

Chapter 4 presents a detailed discussion of the CSP-based specification technique. We first introduce the major constructs of CSP and explain their roles in the protocol domain. To present the way using this technique to specify protocols and services, we use the following format:
(i) the essential requirements imposed on any description technique for specifying protocols or services are examined, our basic schemes for using the CSP-based technique to meet these requirements are then explained; (ii) an example is given to demonstrate the applicability of the technique; and (iii) a comparison with other specification techniques is discussed. Throughout this chapter, we emphasize that, following certain disciplined practices, a single formalism can be adapted to meet the different characteristics exhibited by communication protocols and services.

In Chapter 5, the meaning of conformity analysis is defined. We investigate the essential supporting mechanisms required for conformity analysis. Based on these observations, we point out the fundamental difficulties of the conventional axiomatic approach, and introduce the transformational approach. Milner's Calculus of Communicating Systems (CCS), a tool used in the transformational approach, is also introduced.

Chapter 6 details the transformational approach. The highlight of a formal transformation system is first given, and then a set of transformation rules are defined. Several examples are used to show how the transformation procedures are conducted. In Chapter 7, a complete process of conformity analysis is presented through the case study of alternating bit protocol. In addition, the X.25 packet level DTE/DCE interface is used to demonstrate the detection of syntactic errors in the transformational approach.
Chapter 8 concludes the study with a brief review of our research findings. Some closing remarks and directions for future work are also pointed out.
CHAPTER II
Communication Services and Protocols

2.1 Introduction

Facilitating communications between information processing systems in a heterogeneous environment requires a universal framework of computer networking architecture. It was for this purpose that the International Standards Organization (ISO) initiated development of worldwide standards for creation of an "open system environment". When complying with these standards, an information system would be open to communicate with any other system obeying the same standards. After years of efforts by ISO, the result of this standardization work is the well-known seven-layer Open Systems Interconnection (OSI) Reference Model [10]. It provides a common basis to guide the future development of mutually compatible information processing systems which will greatly benefit both computer vendors and users.

The major contributions of the OSI work are not only the creation of a common framework for intersystem communications but also the
defining of a set of terminologies, conventions, and concepts so that research work and literature can be stated in and interpreted through a common glossary. In the area of formal specification and verification, the concepts of service and protocol are crucial.

2.2 The Concepts of Service and Service Specification

In the OSI Reference Model, the overall structure of an information processing system is divided into a series of layers, each with its own functionality. The most significant advantage of this layered structure is that each layer only bears the responsibility of a group of related functions, and it only interacts with its neighboring layers at the boundaries. It is this kind of boundary which conveys the notion of Service.

In [19], for layer N, (N)-service is defined as:

“A capability of the (N)-layer and the layers beneath it, which is provided to (N+1)-entities at the boundary between the (N)-layer and the (N+1)-layer.”

In the framework of the OSI Reference Model, services are defined by means of the Service Specifications. As stated in [20],

“The service of a layer consists of a set of elementary services of this layer. The service specification for layer N is a specification of a module, consisting of the entities of the layer N and the layers below, given in an abstract view showing only the interactions at the (N)-service-access-points.”
Following the common conventions, those parts of a system making use of a service are called the service users, while those parts providing the service are collectively called the service provider.

As illustrated in Figure 2.1, the \((N)\)-service represents the logical interface between the users, which normally are the \((N+1)\)-entities, and the service provider, which consists of the layer \(N\) and all the lower layers.

\[\text{Figure 2.1. The Service of layer } N\]
2.3 The Concepts of Protocol and Protocol Specification

While the services define the server/user relationships between adjacent layers, they provide no information on how entities in the same layer (i.e. peer entities) communicate with each other. In this regard, Protocols are used to define the peer-to-peer interactions taking place within layers.

In [19], the \((N)\)-protocol is defined as:

"A set of rules and formats (semantic and syntactic) which determines the communication behavior of \((N)\)-entities in the performance of \((N)\)-functions."

To describe protocols, Protocol Specifications are required. As stated in [20],

"The protocol specification for layer \(N\) is the set of the specifications of the modules which represent the entities of layer \(N\). This module(s) represents an \((N)\)-layer entity providing service through one (or more) \((N)\)-service-access-points, and accessing the service of the layer below through one (or more) \((N-1)\)-service-access-points."

As illustrated in Figure 2.2, the peer entities of layer \(N\) communicate with each other through the \((N-1)\)-service provider with the objective of providing service to the users in the next higher layer.
2.4 Layers of Functions versus Levels of Abstractions

In the context of the OSI Reference Model, it is important to distinguish between two independent notions: layers of functions and levels of abstractions.

Layering is a structuring technique by which a system can be logically decomposed into smaller subsystems. In the OSI work, the layering approach subdivides the functionality of an open system into
seven layers, each responsible for a specific set of functions. This approach has at least two significant advantages.

- The whole system is subdivided into individual pieces of manageable size which are more comprehensible and subject to independent implementation and maintenance.

- A portion of the system is able to perform its function before the completion of the other parts. This is especially important in establishing standards. As we can see, at the present time, while the lower layers of the OSI model have already been developed and become functioning, the standardization of the upper layers are still in process.

**Abstraction** is an architectural concept applying to all layers of an open system. For each layer N, there are two levels of abstractions — (N)-Service and (N)-Protocol. At the higher level of abstraction, (N)-Service defines the interface between (N)-layer and (N+1)-layer. At the lower level of abstraction, (N)-Protocol defines the behavior of (N)-entities inside (N)-layer.

As illustrated in Figure 2.3, from the viewpoint of (N+1)-layer, (N)-Service represents the capability of the (N)-layer and all the layers below; it is not concerned about how the capability is realized. The (N)-entities, when making use of their underlying (N-1)-Service, constitute a "logical implementation" of the (N)-Service. The use of abstraction has several advantages:
- Each layer, knowing the service provided from its lower layer, can be designed and developed with little knowledge of the internal operations in the lower layers.

- The effect of any future changes of a protocol is localized within a layer provided that the service offered to the higher layer remains the same.

---

**Figure 2.3.** OSI Architecture
2.5 Discussions

The design of any large and sophisticated software system, such as an information processing system, is a difficult task. It is conceivable that certain rules and principles are needed to cope with the complexity and size of such systems.

The seven layers of the OSI model are logically viewed as a vertical sequence. Each layer performs its own set of functions with the objective of enhancing (or "add value" to) the capabilities of the lower layer to its next higher layer. Furthermore, by introducing the concept of service, it is able to distinguish the capabilities provided by a layer from the operations performed inside the layer. In the service model, the service users at a particular layer view all the layers below as an abstract machine — the service provider. All the detailed information about the operations performed inside this abstract machine is hidden from the users.

By using the layering approach and introducing the service model, the OSI architecture can thus be established in a well-structured manner. In fact, layering a system and defining the services are the realizations of the two general principles stated below.
• **Hierarchical Structuring:**
  To decompose the system structure into modules positioned at different levels according to their functionalities.

• **Information Hiding:**
  To hide the system details within modules by exposing only the interfaces to the users of the modules.

In computer science, these two principles have been widely accepted as important software engineering practices. In the design of software systems, the common approach is to subdivide the entire system into separate modules. It has been recognized that with this kind of modular structure [30] the costs of implementation and maintenance can be greatly reduced. In addition, to aid the software design, many modern programming languages have provided various facilities to improve the structure of programs. The most important one is data abstraction [35] — a key mechanism to support the principle of information hiding.

With regard to operating system, the idea of hierarchical design was first raised by Dijkstra [12]. The resulting level structure has laid the foundation of modern operating system design and revealed significant advantages in implementation, modification, and testing. More recently, the concept of information hiding has also been incorporated
into the operating system design and is called the object-oriented approach. With this approach, system resources are viewed as individual objects which are *encapsulated* in the sense that the internal structure of an object is invisible to the object users. Object-oriented systems are considered superior in reliability and security.

In summary, the layered protocol structure and the conceptual service model of the OSI architecture for information processing systems can be viewed as the realizations of hierarchical structuring and information hiding, two principles which have been applied to many different areas in computer science (Table 2.1).

<table>
<thead>
<tr>
<th>Table 2.1. Software Engineering Principles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Hierarchical Structuring</strong></td>
</tr>
<tr>
<td>Information Processing Systems</td>
</tr>
<tr>
<td>Layered Protocol Structure</td>
</tr>
<tr>
<td>Service model</td>
</tr>
<tr>
<td><strong>Information Hiding</strong></td>
</tr>
<tr>
<td>Software Systems</td>
</tr>
<tr>
<td>Modular Structure</td>
</tr>
<tr>
<td>Data Abstraction</td>
</tr>
<tr>
<td>Operating Systems</td>
</tr>
<tr>
<td>Level Structure</td>
</tr>
<tr>
<td>Object model</td>
</tr>
</tbody>
</table>
3.1 Introduction

As we have pointed out in Chapter 1, formal specification and verification are two essential activities in the design of communication services and protocols. In the past, a variety of specification and verification techniques have been proposed in the literature. However, each technique has its advantages as well as its shortcomings. Moreover, by the increased experience in this area and a better understanding of the nature of services and protocols, researchers have attempted either to work on new techniques or to revise the existing ones.

At present, a great amount of work is still ongoing and new techniques are emerging periodically. In fact, the proliferation of new methods as well as a variety of hybrid techniques have resulted in the fact that no simple classification can cover the entire scope of proposed techniques. In this chapter, some representative specification techniques and their related verification techniques are briefly introduced.
of the advantages and drawbacks of these techniques can facilitate com-
prehension of the work presented in the remaining chapters.

3.2 State-Oriented Models

- Finite State Machine:
  
The finite state machine model [6] is the most traditional ap-
proach used to specify protocols. In this model, each protocol entity
is represented by a finite state automaton with each state cor-
responding to a different control stage. Each transition is labeled
with either an input event by which the transition is to be triggered
or an output event which would take place as part of the transition.

The major advantage of this approach is its simplicity and thus,
its ability to facilitate automated validation. The reachability
analysis (or state exploration) [43] is commonly employed to deter-
mine whether there are any violations of the general requirements.
By this method, all the possible system states (i.e. the product of
individual entity states) are systematically generated in order to en-
sure that no undesirable states occur under any circumstances.

While the finite state machine model provides a straightforward
method for specifying and validating protocols, it has two major
drawbacks. First, since all the information pertinent to a protocol entity can only be encoded into the single state component of a finite state automaton, the number of states needed may grow rapidly with the increasing size and complexity of the protocols. Second, although the reachability analysis can be aided by machines, its main function is limited to the detection of the syntactic errors of a protocol via the simple examination of individual system states each time they are generated. This approach cannot easily verify the functional properties because it lacks the ability to analyze the paths of the trees generated from protocol specifications.

• Petri Net Model:

A petri net is a graph containing a set of places (represented by circles) and a set of transitions (represented by bars). Directed arcs are used to connect places to transitions, and transitions to places. A number of tokens distributed in the places represent a marking which is analogous to a state in the finite state machine model. A transition is called enabled (or firable) if its input places contain token(s). The firing of a transition causes a re-distribution of tokens and thus forms a new marking.
In this model, places and transitions of a petri net are used to specify certain conditions and events, respectively; the relations among places and transitions are defined to describe the behavior of a protocol or service [11]. Enumerative analysis, similar to the reachability analysis in the finite state machine model, can be employed to determine the syntactic properties of the modeled system. Structural induction, a proof showing that certain properties hold throughout the topology of a graph, is commonly used to derive functional properties.

Since the states of a protocol are encoded as the different distributions of tokens in a petri net, a specification may have a more compact form than its counterpart in the finite state machine model. However, to understand the behavior of a petri net, one has to monitor the dynamic movement of the tokens, hence making the specifications of complicated systems quite unreadable. In order to cope with this limitation, many variations derived from the basic model have been proposed, such as the numerical petri net [39] and the time petri net [27]. While these enhanced models improve the modeling power, they also render the analyses quite difficult.
3.3 Programming Language Model

The idea of using programming languages to specify communication systems is motivated by the observation that protocols are essentially a set of procedures or algorithms to provide communication services. In this respect, the programming language approach [7, 24, 38, 42] has the advantage of offering adequate guidance for the implementors to build compatible systems. In addition, by using programming languages, the resulting specifications are constructible and executable [26], and thus facilitate system analysis and testing at the design stage. On the other hand, due to the lack of proper mechanisms and language constructs to deal with special characteristics of communication systems, most conventional programming languages are not powerful enough to express communication behavior or are forced to restrict the implementations unnecessarily.

The program correctness proof is the common analysis method associated with the programming language approach. To show that a protocol meets certain properties, logic assertions are established and justified by the Floyd-Hoare axiomatic method [16]. The advantage of this approach is that a wide range of protocol properties, especially the functional requirements, can be formulated and proved. However, a great deal of human ingenuity is required to conduct the proof. Furthermore,
since a protocol is a set of communicating entities running in parallel, the original axiomatic method has to be extended to deal with concurrent systems. Despite considerable progress in this direction, much more experience is still needed. We shall discuss the axiomatic method further in Chapter 5.

3.4 Sequence-Oriented Models

- Transmission Grammar:

  Transmission Grammar [40] is a formal language proposed for specifying communication protocols. Its syntax is derived from the notation of context-free grammar. The nonterminal symbols in a transmission grammar are equivalent to the states of a finite state automaton, whereas the terminal symbols represent the actions which can be executed by a protocol. In this approach, a protocol entity is described in terms of a set of production rules. The behavior of an entity is characterized by all the valid terminal sequences which can be derived via the production rules.

  Clearly, when considering the expressive power, transmission grammar and other formal languages have their natural counterparts in various kinds of automata. However, due to differing notations, the formal language approach is distinct from the automata approach in
certain ways. In particular, a formal language approach can express the patterns of sequences much clearer than can the automata approach — the major strength of sequence-oriented approaches.

The drawback of the grammar approaches is that they have no clean method to combine individual grammars into a composite grammar. Specifically, when each communicating entity is modeled by a separate grammar, an important issue is the integration of these grammars into a new grammar which can characterize the behavior of the entire system. In the case of transmission grammar, the proposed method in protocol analysis is the simple checking of syntactic properties, which is similar to reachability analysis in the finite state machine model; there is no clear solution to derive system behavior from the behaviors of individual entities.

• **Protocol Expression:**

An extended type of regular expression, called *protocol expression* [18], has been proposed for the specification and analysis of protocols.

Similar to other sequence-oriented approaches, a protocol expression is used to represent the set of possible execution sequences of a protocol. Besides the common operators such as union, concatenation...
tion, and iteration in the regular expression, two new operators are introduced: the division and cross operators. The division operator is used to distinguish between input and output actions; messages to be received are placed in the numerator position, whereas messages to be sent are placed in the denominator position. The cross operator is used to capture the interactions between two protocol expressions. By using the cross operator and certain reduction rules, protocol expressions for individual entities can be combined into a new expression, which can be subsequently checked for design errors.

The advantage of this approach is its simplicity. In addition, the cross product of two protocol expressions can be automated in a much more efficient way than can the reachability analysis in the finite state model. However, it can only be applied to a limited set of protocols due to its restricted expressive power. In particular, the lack of variables and the conditional branching mechanism make it very difficult to model the data manipulations in a protocol.
3.5 OSI Formal Description Techniques

Despite the existing approaches discussed above, as we have already mentioned, the study of new techniques is still underway. The recent work of OSI has shown that appropriate specification techniques are a crucial factor in ensuring the success of the standardization efforts. Due to this recognition, the ISO began development of the two formal description techniques described below.

• ESTELLE — an extended state transition model:

In the basic finite state machine model, all the necessary information to describe the dynamic behavior of a protocol has to be represented as explicit states of a finite state machine, thereby, making it impractical for complicated systems. To eliminate this drawback, an extended version of the finite state machine model, called ESTELLE [22], has been developed. ESTELLE extends the basic finite state machine model in the following ways:

(1) State:

Instead of simply having a set of finite states, the state space of an automaton (or module) consists of a set of major states and a set of context variables. The major states, acting as the conventional states of a finite state machine, are used to represent the control status (e.g. connection, disconnection) of a protocol. The context variables, acting as the variables in a programming lan-
guage, are used to store the information on the data status (e.g. sequence numbers, data) of a protocol.

(2) Transition:
Instead of labeling a transition with only a simple input or output action, each transition is associated with an input interaction, an enabling predicate and an action block. An enabling predicate is a boolean function on the context variables. An action block is a segment of operations which may change the values of context variables and/or initiate output interactions.

A transition may take place from one state to another when (i) the input interaction occurs and (ii) the enabling predicate is satisfied. Moreover, the action block is subsequently executed as part of the transition. In ESTELLE, a transition is expressed in a Pascal-like notation. Figure 3.1 illustrates both the formal syntax and a graphic representation of a transition.

![Transition Diagram]

**Figure 3.1.** A Transition in ESTELLE

We can see that ESTELLE is basically a hybrid model combining the features of the finite state machine and programming language. It has shown to be a suitable model for protocol specifications.
LOTOS — a language for temporal ordering specification:

LOTOS [23] is a temporal ordering specification language. It is a sequence-oriented model designed to describe the behavior of a system in terms of the temporal ordering of interactions. In this language, a set of operators are introduced to define various temporal relations, such as sequential ordering or parallel ordering, in which interactions may occur. By using LOTOS, a specification of a system consists of a number of algebraic expressions which can be composed of other smaller expressions through the use of operators. Since LOTOS is quite close to Milner's Calculus of Communicating Systems (CCS), a tool used in our method of conformity analysis, the detailed features of LOTOS can be conveyed through the discussions of CCS in Chapter 5.

Compared with the ESTELLE model, LOTOS is designed to express the behavior of a system at a more abstract level. As a result, it is a more effective language for service specifications.

As stated in [41], ESTELLE and LOTOS are intended to complement each other rather than to compete with each other. As knowledge
of protocol and service specifications continues to grow, it is expected that current techniques can be stepwise revised or replaced by newer techniques.
CHAPTER IV
The CSP-based Specification Technique

4.1 Motivations

Formal specifications of services and protocols bear the responsibility of describing the exact requirements and rules imposed on the information processing systems. In the design phase, they provide the designer with a formal tool to convey the preliminary scheme as well as to discover the undetected errors.

As explained earlier, services represent the logical interfaces between adjacent layers while protocols represent the operations performed inside layers. Accordingly, the service specification and the protocol specification describe the behavior of a system at two different levels of abstraction. A service specification is responsible for defining the valid sequences of interactions visible at the boundary between two adjacent layers, whereas a protocol specification defines the behavior of protocol entities inside a particular layer in terms of the interactions between peer entities.
Because of the fundamental disparity between a service specification and a protocol specification with respect to abstraction, it is not surprising to observe that a technique which can handle services reasonable well becomes awkward when describing protocols and vice versa. Specifically, it can be noticed that:

- A service specification deals primarily with the allowable order of interactions taking place at the service boundary — the kind of properties that can be described naturally by the sequence-oriented approaches.

- A protocol specification deals primarily with the operations performed by protocol entities — the kind of behavior that can be described more effectively by the state-oriented approaches.

However, a uniform technique for specifying services and protocols is in demand. First, both service and protocol specifications can be constructed and interpreted in terms of a single formalism. Second, and most importantly, using a uniform technique would facilitate the functional analysis of protocols. As we shall see in the next chapter, the primary goal of functional analysis is to demonstrate that (N)-entities, when making use of (N-1)-service, can indeed provide (N)-service, i.e., (N)-entities + (N-1)-service provider = (N)-service provider. The difficulty of functional analysis resides in the "+" operator, namely, to compose (N)-layer protocol and (N-1)-service. It is clear that, if
protocols and services can be expressed in a single formalism, composition could be carried out in a much orderly manner.

In this chapter, we present a formal technique based on Communicating Sequential Processes (CSP) [17]. By using this technique, both services and protocols can be specified in a uniform way. In section 4.2, the major constructs of CSP, along with their roles in specifying protocols and services, are explained. In section 4.3, a modification of CSP, called generalized distributed termination, is presented. Sections 4.4 and 4.5 demonstrate the applicability of this technique to communication protocols and communication services, respectively. In each section, the basic specification schemes are first presented, a typical example follows, and a comparison with other techniques is made. Section 4.6 summarizes the chapter.

4.2 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a high level concurrent language designed for systems with multiple processors. A CSP program consists of a number of processes that are mutually disjoint in address spaces, and communication between processes is accomplished through message passing. In addition, guarded commands are used to describe nondeterministic behavior. Overall, these CSP features provide a suitable basis to model a distributed environment.
After CSP was first introduced by Hoare in [17], it has stimulated subsequent research work in several areas. The concepts and notations of CSP have greatly influenced the design of concurrent programming languages [1]. Several proof systems for CSP [2, 25, 36] have been proposed, each exploring an alternative way of reasoning about the behavior of concurrent programs. Moreover, CSP has also been used in the design of VLSI algorithms [29], data base systems [29], and operating systems [13].

From the standpoint of specifying services and protocols of communication systems, a specification language should be equipped with appropriate mechanisms to describe three essential properties exhibited by the behavior of services and protocols, viz., concurrency, communication, and nondeterminism. With respect to these three, the constructs of parallel commands, input/output commands, and the alternative and repetitive commands in CSP carry out their respective duties in an elegant way. In the following, a brief overview of these major CSP constructs is given.
4.2.1 Concurrency Mechanism

Parallel command: \[ P_1 \parallel P_2 \parallel \ldots \parallel P_n \]

A parallel command specifies the concurrent execution of its constituent processes \( P_1, P_2, \ldots, P_n \). Each individual process keeps its own local internal states and thus can be physically distributed.

As a result, in the context of communication services and protocols, we can specify a service user, a protocol entity, a transmission medium, or other system components (e.g. a timer) as individual processes.

4.2.2 Communication Mechanism

Input command: \( P_j \,(variable) \)

Output command: \( P_i \!(expression) \)

The communications between processes are accomplished through message passing. Specifically, an input command \( P_j\,(variable) \) in process \( P_i \) and an output command \( P_i\!(expression) \) in process \( P_j \) form a communication matching pair. The effect of a communication taking place between a matching pair is to assign the value of \( expression \) to \( variable \). Furthermore, communication is conducted synchronously in the sense that, each participating process executes its part of the I/O pair at the same time.
In the specifications of services and protocols, the communication matching pair can be used to model the interactions occurring between two system components which appear as two processes in the specification. In the OSI Reference Model, an interaction can occur only if the two participating components execute the interaction simultaneously. This requirement makes a communication matching pair a natural representation of an OSI interaction since both of them are defined based on the rendezvous principle.

In CSP, besides the simple I/O commands, input and output commands can be associated with a constructor identifier $T$ in the form of $P_j ? T(\text{variable})$ and $P_i ! T(\text{expression})$. Constructor identifiers are used to further control the communication structure of concurrent processes. In order to engage a communication, the input command and output command of a matching pair should not only refer the process name of each other, but also have an identical constructor name, as illustrated in Figure 4.1. Throughout this study, we shall make use of constructor names extensively and interpret them as the Type of interactions.

Finally, following the convention of CSP, $P_j ? T()$ and $P_i ! T()$ represent either the I/O commands (of type $T$) which are used purely for the purpose of synchronization, or the I/O commands in which the actual value passed through the communication is of no interest.
4.2.3 Nondeterminism Mechanism

Alternative command:

\[
\begin{align*}
\text{b}_1; \text{I/O}_1 & \rightarrow \text{command list}_1 \\
\text{b}_2; \text{I/O}_2 & \rightarrow \text{command list}_2 \\
\vdots & \\
\text{b}_n; \text{I/O}_n & \rightarrow \text{command list}_n
\end{align*}
\]

Repetitive command:

\[
\begin{align*}
\ast \text{b}_1; \text{I/O}_1 & \rightarrow \text{command list}_1 \\
\text{b}_2; \text{I/O}_2 & \rightarrow \text{command list}_2 \\
\vdots & \\
\text{b}_n; \text{I/O}_n & \rightarrow \text{command list}_n
\end{align*}
\]

An alternative command consists of a list of guarded commands. A guarded command consists of a boolean expression (or an implicit constant true if this part is empty), an I/O command (or an implicit null command if this part is empty), and a command list. An alter-
native command specifies the execution of exactly one of its guarded commands. Any one of the constituent guarded commands can be chosen if its boolean expression is evaluated to be true and the process named by its I/O command is ready to execute the corresponding I/O command. The effect of executing a guarded command is the execution of its associated I/O command followed by any commands in the command list. It is important to note that no assumptions are made in the order of guard evaluations and, if more than one of the guarded commands are executable, the choice is made nondeterministically.

A repetitive command has the same semantics in the manner of choosing its guarded commands. However, it differs from an alternative command in that it specifies the execution of as many iterations as possible until no executable guarded command is remaining.

Nondeterministic situations occur frequently in the behavior of communication services and protocols. They are a major source of complication in a specification regardless of which description technique is used. In this respect, alternative and repetitive commands provide us with natural constructs to express nondeterministic behavior.
4.3 Generalized Distributed Termination

In CSP, a repetitive command terminates when all of its guards fail. A guard fails if either its boolean expression is false or the process named in its I/O command has terminated. As a result, the execution of a repetitive command may terminate due to the termination of other processes. This semantic property, called distributed termination [14], is considered a powerful feature. While this distributed termination convention is confined to a repetitive command, we shall generalize it to the process level as follows.

- Generalized Distributed Termination:
  For an I/O command,
  
  1) If it is in the outermost scope of a process (i.e., it is neither in the guard of a guarded command nor nested in any alternative or repetitive command).

  and

  2) If the process addressed by this command has already terminated at the time it is to be executed.

Then, the I/O command along with the remaining part of the process are skipped and the whole process terminates properly.
Example 1:

\[
A :: [ B !go() \rightarrow \text{skip} \\
     C !go() \rightarrow \text{skip} ]
\]

\[
B :: A ?go();
\]

\[
C :: A ?go();
\]

Process A consists of only an alternative command in which one of the guarded commands is to be executed. If the first one is selected, Process B will receive a go() signal and continue the execution of the remaining commands. Process C, on the other hand, terminates without executing any command since the input command A ?go() is in the outermost scope and Process A has terminated. A similar statement can be made for the case when the second guarded command in Process A is selected.

By the definition of generalized distributed termination, a process may terminate due to the termination of another process and we shall make use of this feature to model the disruption behavior presented in communication services and protocols, as explained below.

In an OSI service or protocol, "normal operations" can be disrupted if a special event occurs. As a typical example, during the connection establishment phase for two protocol entities, the underlying service provider may break off the negotiation between the entities at any time by initiating a disconnection indication to both of them.
To understand how generalized distributed termination can handle the behavior of disruption, consider the following:

Example 2:

\[
P_1 ::
\begin{align*}
&\text{Con!a()} \rightarrow \text{[(Con?d() \rightarrow \text{skip}]} \\
&\text{Dis?stop1()} \rightarrow \text{skip} \\
&\text{Dis?stop1()} \rightarrow \text{skip}
\end{align*}
\]

\[
P_2 ::
\begin{align*}
&\text{Con?b()} \rightarrow \text{[(Con!c() \rightarrow \text{skip}]} \\
&\text{Dis?stop2()} \rightarrow \text{skip} \\
&\text{Dis?stop2()} \rightarrow \text{skip}
\end{align*}
\]

\[
\text{Con :: }
\begin{align*}
P_1 ?a(); \\
P_2 b(); \\
P_2 c(); \\
P_1 d();
\end{align*}
\]

\[
\text{Dis :: }
\begin{align*}
\text{[P_1!stop1() \rightarrow P_2!stop2()]} \\
\text{P_2!stop2() \rightarrow P_1!stop1()}
\end{align*}
\]

Process \(P_1\) and \(P_2\) would go through a conversation via process Con(nection). The "normal operation" is a sequence of communications in the order of \(a(), b(), c(), d()\) as reflected in process Con. However, process Dis(connection) can disrupt this sequence by sending \(\text{stop1()}\) to \(P_1\) and \(\text{stop2()}\) to \(P_2\) in an arbitrary order. \(P_1\) and \(P_2\) will terminate upon the reception of \(\text{stop1()}\) and \(\text{stop2()}\), respectively.

Now, what is going to happen in process Con when a disruption occurs? By the definition of generalized distributed termination, it should be clear that process Con may finish its execution anywhere, depending on whether \(P_1\) or \(P_2\) has terminated. In the normal case, it should execute all of its communications — representing the completion of a successful conversation between \(P_1\) and \(P_2\). In the other case, it may terminate even without executing any of its commands —
representing the situation in which $P_1$ and $P_2$ are disrupted by $\text{Dis}$ before any attempt of a conversation could have been made. In fact, it can be examined that all the valid communication sequences, when leaving out $\text{stop1()}$ and $\text{stop2()}$, are $[ ]$, $[a()]$, $[a(),b()]$, $[a(),b(),c()]$, and $[a(),b(),c(),d()]$.

The original distributed termination convention in CSP has been the focus of much discussion in the literature. As Hoare stated, distributed termination is an extremely powerful and convenient feature, allowing one to describe many behaviors without introducing explicit "end()" signals [17]. The generalized distributed termination presented in this section is motivated for the same purpose, but we have extended it to the process level. The major disadvantage of distributed termination and the generalized version is that their implementation is complicated. However, here we use CSP language as a specification tool and thus mainly focus on its semantics. The implementation issues have been studied extensively in the literature, but they are of no direct concern in this study. Finally, it should be mentioned that we shall make use of generalized distributed termination to model a simplified version of OSI Transport Service in Section 4.5.
4.4 Protocol Specifications

4.4.1 Specification Schemes

As we have known, a communication protocol is a set of rules governing the exchange of information between protocol entities. A protocol entity can be viewed as an “abstract machine” interacting with its environment. There are two key attributes characterizing the “state” of an abstract machine.

• Control Status:
The control status determines the responses of an entity to the incoming events from the environment. It reflects what has been done so far and what is expected to happen during the execution of its functions. Typical examples of the control status are “connection has established” or “waiting for the next message”.

• Data Status:
The data status contains the current value of any information manipulated by a protocol entity in the course of its execution. Typical examples are the sequence numbers or the message content. In contrast to the control status, of which the domain is usually a small set, a value in the data status can fall within a wide range.

As mentioned in Chapter 3, the Finite State Model can easily describe the control status of a protocol by encoding the control infor-
formation as the state of a finite automaton. However, it has difficulty handling data since it is impractical to record all possible values of data information as different explicit states.

In the programming language model, program variables can be used to store the data status in a natural way. Unfortunately, conventional programming languages have difficulty expressing the control structure of a protocol due to the lack of proper constructs to support concurrency, communication, and nondeterminism.

The major advantages of using CSP to specify protocols should be clear at this point. The data status can be encoded into variables just as in any programming language and the control structure of a protocol can be described by CSP constructs in an elegant way.

4.4.2 Alternating Bit Protocol

In this section, we demonstrate how to use the CSP-based technique to specify protocols by using the Alternating Bit Protocol as an example.

The Alternating Bit Protocol [5] provides a one-way error-free data transmission service over an unreliable transmission medium. As shown in Figure 4.2, two protocol entities, Sender and Receiver, together with
a Transmission medium, form a communication system for User1 and User2.

![The Structure of ABP Communication System](image)

**Figure 4.2.** The Structure of ABP Communication System

The functions performed by each of the system components are described as follows:

- **Sender:**
  Sender is the protocol entity which interacts with User1. Upon the reception of a data block, it attaches a single bit sequence number to the data block to form a frame. The resulting frame will be transmitted to User2 through the Medium one or more times until a correct acknowledgment (i.e. the one with the same sequence number) is received. The sequence number is switched between 0 and 1 each time a new data block is sent to detect duplicated acknowledgment. Also, the Sender may interact with a timer for the purpose of recovery from data or acknowledgment loss. It is as-
sumed that the time-out interval is properly set such that time-out will only occur after a transmission loss has occurred.

• Transmission medium:
  In the context of the OSI Reference Model, it is clear that the transmission medium is actually the underlying service provider to Sender and Receiver. It provides a half-duplex transmission service (i.e. it may transmit in either direction, one direction at a time). The service is unreliable since messages may be lost completely or may be corrupted, in which case Sender or Receiver will be informed that an error has occurred.

• Receiver:
  Receiver is the protocol entity which interacts with User2. It continues to check the sequence number of each in-coming frame. If a frame arrives error-free and has the expected sequence number, the Receiver then delivers the data block of this frame to User2 and sends an acknowledgment with the same sequence number back to the Sender. Otherwise, it replies with an acknowledgment with the opposite sequence number.

  The specification of ABP is shown in Figure 4.3. Each system component is modeled by a process, and the entire communication system is described by all of the processes running in parallel.
**ABP :: [ Sender || Medium || Receiver ]**

(a) The Whole System

Sender ::

```plaintext
frame : record
data : ...;
seq : (0,1,error)
end;
DATA : ...;
Seq : (0,1);
Ack : (0,1,error);
done : boolean;
```

Seq := 1;

```plaintext
* [User1?(DATA) →
  Seq := (Seq+1)mod2;
  frame.data := DATA;
  frame.seq := Seq;
  done := false;
  *[ ~done; Medium!(frame) →
    Timer!reset();
    [ Medium?(Ack) →
      [ Ack=Seq → done:=true
        ]
      Ack=(Seq+1)mod2 → skip
      [ Ack=error → skip
        ]
    ]
    Timer!out() → skip
  ]
  ]
```

(b) The Sender

Figure 4.3. The Specification of ABP
medium ::
frame : /* same as in Sender */;
Ack : (0,1,error);
correct, corrupted, lost : boolean;
correct:=true; corrupted:=true; lost:=true;
[ Sender?(frame) — ► [ correct — ► Receiver!(frame) ]
corrupted — ► frame.seq:=error;
Receiver!(frame)
lost — ► skip; ]
Receiver?(Ack) — + [ correct — ► Sender!(Ack) ]
corrupted — ► Sender!(error)
lost — » skip ]
]
(c) The Medium

Receiver ::
frame : /* same as in Sender */;
bit : (0,1); /* opposite of the expected sequence number */
bit := 1;
[Medium?(frame) — » [ frame.seq=(bit+1)mod2 — » User2!(frame.data); bit := (bit+1)mod2
frame.seq=bit — » skip
frame.seq=error — » skip ];
Medium!(bit) ]
(d) The Receiver

Figure 4.3. The Specification of ABP
(continued)
4.4.3 Comparisons with EFSM Model

In our experience, the most distinct advantage of using CSP is that it provides an effective way to describe both the control and data aspects of protocols.

As discussed in Chapter 3, the Extended Finite State Machine (EFSM) model is a hybrid model combining the expressive power of the finite state machine to specify control status with the expressive power of the programming language model to specify data status.

Now, by carefully examining the ABP specification, we find that the CSP-based approach actually captures many features of the Extended Finite State Machine model. To clarify this observation, the following is a comparison between the Extended Finite State Machine model and the CSP language model, through the study of the ABP case.

1) An interaction between two modules in the EFSM specification corresponds to a communication between two processes in the CSP specification. For example, sending a frame from the sender to the medium is modeled by the matching communication pair:

   Sender :: [ ... Medium!(frame) ... ]
   Medium :: [ ... Sender?(frame) ... ]

2) The major states of a module in the EFSM specification are implicitly reflected by the control points (or locations) just before the input commands of the corresponding process in the CSP
specification. For a repetitive or alternative command with all its guards containing input commands, the control point at the beginning of that command is analogous to a major state from which one of several possible transitions may occur depending on which input interaction takes place. For example, the major states of the Sender are the control point at which it waits for a data block from User1 and the control point at which it may receive either an acknowledgment or a timeout signal.

3) A transition from a major state to another major state in the EFSM specification is analogous to the execution flow (or execution locus) from a control point to another control point in the CSP specification. Moreover, the action associated with a transition in the EFSM specification corresponds to the assignment and output commands executed between two control points in the CSP specification. For example, when the Sender gets a data block from User1, it will prepare a frame with a proper sequence number, send this frame, reset the timer, and wait at the next control point.

4) Associating enabling predicates with transitions in the EFSM specification is analogous to using boolean guards to guide the execution flow in the CSP specification. For example, upon receiving an acknowledgment, the Sender will either retransmit the last frame or wait for another data block from User1, depending on whether the received acknowledgment is the expected one or not.
In fact, when we take a closer look at both models, the basic building blocks of the EFSA model can actually be represented by the CSP constructs as illustrated in Figure 4.4.

So far, the comparisons we have made are from the standpoint of modeling power. It should be noted that the structures of the EFSA-based specification and the CSP-based specification for the same protocol are not necessarily identical or "isomorphic". For instance, auxiliary boolean variables may be introduced in a CSP process to control the execution flow (such as the "end" variable in Sender of ABP). On the other hand, the execution flow of a finite state machine can be determined solely by its transition function. In this regard, it seems that the Finite State Model is nicer than CSP. However, the need for auxiliary boolean variables in CSP-based specifications stems from the fact that there is no "go to" construct in CSP. In contrast, each transition in a finite state machine is in essence a "go to" command. As a result, when dealing with a complicated protocol, the specification in the Finite State Model will be much less structured than the specification in CSP.
Figure 4.4. Comparisons between EFSM and CSP
4.5 Service Specifications

4.5.1 Specification Schemes

As discussed in Chapter 2, a service specification describes the interface between two adjacent layers. A service provider interacts with its upper users through the service access points (SAPs). Typically, a service is accomplished by running through several phases, each of which is responsible for a specific function. Thus, the conceptual structure of a service provider can be illustrated as in Figure 4.5.

![Figure 4.5. The Conceptual Structure of a Service Provider](image)

A service can be characterized by two kinds of constraints as below.
• **Local Constraints:**

Local constraints describe the allowable interaction sequences taking place at each of the service access points. In other words, the local constraints at a service access point constitute the behavior of the service provider perceived by the service user at that access point.

• **Global Constraints:**

In general, an interaction occurring at one access point will have consequences at the other access point. The global constraints are used to describe the overall interaction sequences presented at both of the access points. In other words, they define the behavior of the service provider perceived by an observer who can record the interactions taking place at either of the access points in the order of occurrence.

The basic method of specifying a service is as follows. First, the local constraints at each access point are described as a single process. We make use of these processes to reflect the behavior of individual service users. Then, we describe the global constraints on a service provider by a set of processes; each one represents the behavior of a single phase. As we shall see, one important characteristic of the processes modeling the individual phases is that these processes never communicate with

*When concurrent behaviors are exhibited in a single phase, such as simultaneous data transmissions in both directions, we may decompose the specification of this phase as separate processes.*
each other. This should not be surprising if we realize that each of them represents the behavior of the same service provider during a different phase. As a natural consequence, all of them should communicate only with the service user processes. Furthermore, since a service specification is used to define the interaction sequences, the processes will mainly be involved in communications except that some auxiliary boolean variables may be used to guide the execution flow.

4.5.2 A Simplified OSI Transport Service

In this section, we demonstrate how to use the CSP-based technique to specify services by considering a simplified version of the OSI Transport Service (TS) [21].

The purpose of the Transport Service is to provide transparent transfer of data between TS users. It relieves the TS users of any concern about the supporting communication media by which the data transfer is achieved. In the context of OSI services, a service primitive is a logically instantaneous and indivisible interaction between a service user and the service provider. The Simplified Transport Service considered here is defined by means of eight service primitives which can be invoked in three different phases, as listed in Table 4.1. Among these primitives, Normal Data-request and Normal Data-indication
are associated with TS user-data as their parameters, whereas the others have none.

Table 4.1. Primitives of the Simplified Transport Service

<table>
<thead>
<tr>
<th>Phase</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection Establishment</td>
<td>Connection-request (Creq)</td>
</tr>
<tr>
<td></td>
<td>Connection-indication (Cind)</td>
</tr>
<tr>
<td></td>
<td>Connection-response (Cres)</td>
</tr>
<tr>
<td></td>
<td>Connection-confirmation (Ccon)</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>Normal Data-request (NDreq)</td>
</tr>
<tr>
<td></td>
<td>(TS user-data)</td>
</tr>
<tr>
<td></td>
<td>Normal Data-indication (NDind)</td>
</tr>
<tr>
<td></td>
<td>(TS user-data)</td>
</tr>
<tr>
<td>Connection Release</td>
<td>Disconnect-request (Dreq)</td>
</tr>
<tr>
<td></td>
<td>Disconnect-indication (Dind)</td>
</tr>
</tbody>
</table>

The functional behavior of the Simplified Transport Service is illustrated in Figure 4.6 and explained below. Throughout the discussion, we assume that User1 is the calling TS user — the TS user who initiates a Transport-Connection establishment request; User2 is the called TS user — a TS user with whom a calling TS user wishes to establish a Transport-Connection.

- **Connection Establishment Phase** (Fig.4.6(a)):
  The connection establishment phase begins when User1 issues a
Connection-request primitive to the underlying Service Provider. The Service Provider, in response to this request, issues a Connection-indication primitive to User2*. If User2 is willing to accept this connection, a Connection-response primitive is issued to the Service Provider who will in turn issue a Connection-confirmation to User1.

- **Data Transfer Phase (Fig.4.6(b))**:  
The data transfer phase starts on User2's side as soon as a Connection-response is issued, whereas it starts on User1's side when a Connection-confirmation occurs. Once in this phase, either User1 or User2 can issue a sequence of Normal Data-request primitives such that their associated TS user-data are conceptually entered into a queue in the Service Provider. At the same time, the Service Provider can remove the data elements in each queue by issuing Normal Data-indication primitives to the other side.

- **Connection Release Phase (Fig.4.6(c))**:  
On each side, a connection release phase is entered when either of the following two situations occurs. First, the user issues a Disconnect-request primitive to the service provider. Second, the user is informed by a Disconnect-indication which is either issued by the service provider itself or is the consequence of a Disconnect-request issued by the other user. An important feature of the Transport Service is that a connection release is permitted to

---

*We follow the common convention to postulate that a Connection-request issued by a calling TS user will always result in a Connection-indication to the called TS user.*
disrupt the normal operations of other phases at any time. That is, the user is allowed to issue a Disconnect-request to terminate the connection regardless of the current phase; moreover, it should also be ready to receive a Disconnect-indication from the service provider at any time. Once the release phase is entered, no further primitives may be performed.

Next, we shall present the CSP-based specification of the simplified Transport Service. The specification is divided into eight processes as shown in Figure 4.7. User1 and User2 are constructed to model the behavior of the calling TS user and the called TS user, respectively. The other six processes together constitute the behavior of the Service Provider. As suggested by the names of the processes, Connection is used to model the connection establishment phase; Dataqueue1 and Dataqueue2 are used to model the two independent data flows in the data transfer phase; and the remaining three processes — Disconnect1, Disconnect2, and DisconnectP — are used to model the connection release phase initiated by the calling TS user, the called TS user, and the Service Provider, respectively.

The CSP-based specification of the Simplified Transport Service is given in Figure 4.8. In order to highlight the control structures, all the declarations of variables in the processes are omitted. As they are quite clear from the context, the identifiers named end in processes User1 and
Figure 4.6. Functional Behavior of a Simplified Transport Service
User2 are boolean variables with "false" as their initial values. The parameter data is a variable whose value can be any positive integral number of octets. In addition, Dataqueue1 is associated with an abstract data object Q1, and Dataqueue2 is associated with Q2. They are two first-in-first-out queues with the elements of the same type as data. On these data objects, enqueue(Q,data) is an operation which adds data to the rear end of Q. The dequeue(Q) operation removes a data element from the front end of Q and returns the value of the removed data. Also, empty(Q) is a function which returns true if Q is empty and false otherwise.

A detailed explanation of the specification is presented below.
(a) Primitive Sequences on the Calling TS User’s side

User1::  Connection?Creq();
    [ Connection?Ccon() →
        *[  -end; Dataqueue1!NDreq(data) → skip
            -end; Dataqueue2?NDind(data) → skip
            -end; Disconnect1!Dreq() → end:=true
            -end; Disconnect2?Dind() → end:=true
            -end; DisconnectP?Dind() → end:=true
        ]
        Disconnect!Dreq() → skip
        Disconnect2?Dind() → skip
        DisconnectP?Dind() → skip
    ]

(b) Primitive Sequences on the Called TS User’s side

User2::  Connection?Cind();
    [ Connection?Cres() →
        *[  -end; Dataqueue2!NDreq(data) → skip
            -end; Dataqueue1?NDind(data) → skip
            -end; Disconnect2!Dreq() → end:=true
            -end; Disconnect1?Dind() → end:=true
            -end; DisconnectP?Dind() → end:=true
        ]
        Disconnect2!Dreq() → skip
        Disconnect1?Dind() → skip
        DisconnectP?Dind() → skip
    ]

Figure 4.8. Specification of a Simplified Transport Service
(c) Primitive Sequence in the Connection Establishment Phase

\[
\text{Dataqueue}_1 :: \\
* \text{ User1}\text{!NDreq(data)} \rightarrow \text{enqueue(Q1, data)} \\
\emptyset \\
\neg \text{empty(Q1)}; \text{ User2}\text{!NDind(dequeue(Q1))} \rightarrow \text{skip}
\]

\[
\text{Dataqueue}_2 :: \\
* \text{ User2}\text{!NDreq(data)} \rightarrow \text{enqueue(Q2, data)} \\
\emptyset \\
\neg \text{empty(Q2)}; \text{ User1}\text{!NDind(dequeue(Q2))} \rightarrow \text{skip}
\]

(d) Primitive Sequence in the Data Transfer Phase

\[
\text{Disconnect}_1 :: \\
\text{ User1}\text{?Dreq();} \\
\text{ User2}\text{!Dind()}
\]

\[
\text{Disconnect}_2 :: \\
\text{ User2}\text{?Dreq();} \\
\text{ User1}\text{!Dind()}
\]

\[
\text{DisconnectP} :: \\
* \text{ User1}\text{!Dind()} \rightarrow \text{skip} \\
\emptyset \\
\text{ User2}\text{!Dind()} \rightarrow \text{skip}
\]

(e) Primitive Sequences in the Connection Release Phase

Figure 4.8. Specification of a Simplified Transport Service (continued)
• **The Calling User’s Side (Fig.4.8(a))**:  
  User1 is used to model the behavior at the calling user’s side. It starts by issuing a Creq() signal to the Connection process. Upon receiving a Ccon() signal, it enters into the inner repetitive command in which the data transfer may start. During the data transfer phase, User1 may send data by issuing a NDreq(data) to Dataqueue1 or receive data from Dataqueue2 through NDind(data). Also, it is important to note that, after a Creq() has been issued, the process may terminate at any time when it sends a Dreq() or receives a Dind().

• **The Called User’s Side (Fig.4.8(b))**:  
  User2 is used to model the behavior at the called user’s side. It is similar to User1 except that it first receives a Cind() from Connection and data transfer starts when it responds with a Cres().

• **Connection Establishment Phase (Fig.4.8(c))**:  
  Connection consists of four consecutive I/O commands showing the sequence of primitives required for a successful connection establishment.

  It should be emphasized that Connection will execute all of its commands only if a connection is going to be established "successfully". As we have mentioned, a disruption can occur when either the service provider or any of the users wants to end the service, even in the course of a connection establishment. For instance, after receiving a Cind() signal, User2 may refuse the connection request by sending a Dreq() signal. At that moment, Connection has already finished the first two commands and is...
waiting for the Cres() signal from User2, but User2 has terminated. It should be recalled that, by generalized distributed termination, Connection will ignore the remaining two commands and terminate properly.

- **Data Transfer Phase (Fig. 4.8(d))**:  
  Dataqueue1 and Dataqueue2 represent the behavior of two independent data flows in the data transfer phase. For each process, a succession of data requests from the sending user results in the accumulation of data at one end of the queue and, at the same time, a succession of the data indications to the receiving user results in the removal of data from the other end of the queue.

- **Connection Release Phase (Fig. 4.8(e))**:  
  Disconnect1 and Disconnect2 represent a Disconnect request issued by User1 and User2, respectively, in each case leads to a Disconnect indication sent to the other user. Process DisconnectP, on the other hand, describes the case in which the Service Provider issues a disconnect indication to each of the users.

  Again, the generalized distributed termination will come into effect when disruptions occur, for instance, when User1 issues a Dreq() to Disconnect1 and, at the same time, DisconnectP issues a Dind() to User2. The termination of User2 will result in the proper termination of both Disconnect1 and Disconnect2.

  At this point we have completed the study of the Simplified Transport Service. It can be seen that the specification is presented in a
highly modular structure with each process specifying a specific portion of the entire functionality. In addition, the global constraints are realized by all possible sequences of communications when the processes are running in parallel. Therefore, the specification as a whole can be viewed as a sequence generator in the sense that it generates all the valid sequences of primitives presented by a service.

4.5.3 Comparisons with Sequencing Expressions

To specify a service, a specification language should be able to define the constraints on the ordering structure of the interaction sequences. As shown in the case of Transport Service, we construct a CSP specification as a communication sequence generator. In specifying a service, by virtue of the synchronous communication mechanism, the occurrence of a service primitive can be modeled by a CSP matching I/O command pair. Furthermore, the temporal ordering of the service primitives is defined by the semantic structure of the specification. In a specification, while the overall allowable sequences of primitives are defined by the global communication sequences, individual processes of the specification can also characterize the local constraints on the service users and the separate phases of the service provider. We believe that this modular structure, which may not be obtainable by other approaches, is important in specifying services.
To express sequences of events, the most well-known approach is the regular expression. With this description language, all the valid sequences can be represented in a closed form using the basic operators such as union, concatenation, and iteration. It can be seen that the basic idea of this approach is very similar to the way in which we use CSP to specify services. When specifying communication services, we essentially intend to use the CSP constructs as the ordering operators to express the sequences of interactions of a service. Thus, the resulting specifications usually bear considerable resemblance to regular expressions. A simple example is given in Figure 4.9 to clarify the above statements.

![Figure 4.9. Viewing a CSP specification as a sequence generator](image-url)
 Needless to say, CSP language has much more expressive power than does the regular expression. In [3], Apt showed that we can associate with any CSP process a regular language which stands for the set of all possible communication sequences of that process only when the boolean guards are not interpreted. It can be readily seen that regular expressions are not capable of capturing the "irregular" communication sequences of a CSP process when the boolean guards are used to direct the execution flow.

Several extended forms of regular expressions have been studied [8, 18]. In these regular expression-based techniques, extensions have been made in order to deal with concurrency, communications, and disruptions when they are used to specify communication services. The advantage of the approaches using sequencing expressions is that they have simple semantics and thus the specifications are easy to understand. However, they have difficulty dealing with the data status of communication systems since there are no variables in these sequencing approaches.
4.6 Summary

In this chapter we have used a CSP-based specification technique to specify communication protocols and services. With the CSP approach, protocols and services can be specified using a common formalism.

An important philosophy underlying the design of CSP, as Hoare emphasizes, is that CSP was not intended to be a complete programming language and only a minimum set of essential primitives was included. While CSP was not intended to be a realistic programming language, we think it is exactly this reason which makes CSP a good basis for a specification language. In our approach, the basic constructs of CSP are used as the backbone of our specification language. To accommodate specific communication protocols and services requirements, we can make further extensions or modifications without changing the major semantics of the original language. For instance, abstract data objects and the generalized distributed termination convention have been introduced to facilitate the specification of Transport Services.

From our experience, we believe that what makes CSP a remarkable specification language is its simple but powerful constructs for describing communication systems. As we have seen, the message passing primitives provide an elegant mechanism for modeling the basic in-
interactions within protocols and services. The alternative and repetitive primitives which employ guarded commands provide natural control constructs for modeling the nondeterminism in communication behaviors.

We have also mentioned that the approach based on the EFSM model is considered suitable for protocol specification whereas the sequencing expression approach is considered suitable for service specification. This fact leads to a natural question: what are the relations between the CSP language model and the other two approaches referred to above? In this chapter, we have made a comparison between the CSP language model and the EFSM model. We conclude that, on one hand, the constructs of CSP can be used to accomplish the same expressive power as does the extended finite state machine model. On the other hand, CSP can also be used as sequencing expressions which generate all the valid interaction sequences of the defined communication service.
CHAPTER V
Conformity Analysis

5.1 Introduction

The advantage of using formal description techniques for software systems is twofold. First, by virtue of the unambiguous and rigorous semantics of the specification language, a formal specification can be uniquely interpreted by any implementor. Second, a formal specification language provides a sound foundation on which the designers can analyze a system. Having previously studied the issues with respect to the specifications of protocols and services, we now begin to explore the analysis method associated with the CSP-based technique.

As stated in Chapter 1, the second goal of this research is to investigate a method by which the functional analysis on protocols and services can be performed. Essentially, the functional analysis is to demonstrate that the protocol entities of a certain layer, when making use of the underlying service, can indeed provide the service for which the protocol has been designed — we call this Conformity Analysis.
In the context of the CSP approach, if the (N)-entities and (N-1)-service provider are specified as a set of processes, our task is to show that, after hiding all the internal communications, the set of observable communication sequences of these processes with respect to the users should conform to the set of communication sequences exhibited by the processes of the (N)-service provider (Fig. 5.1).

Figure 5.1. The Objective of Conformity Analysis
5.2 Axiomatic Approach

For CSP, a number of proof systems have been proposed [2, 25, 37]. While each of these provides a different way to reason the distributed programs written in CSP, all are based on Hoare’s axiomatic approach [16]. In this approach, one can make use of a set of axioms and inference rules to prove that the behavior of a program has some desired properties.

The axiomatic approach has been considered a successful tool in the design of sequential programs. Given an initial condition which is satisfied at the beginning of a program, the prover can systematically derive the logic assertions at different control points, ultimately establishing a desired postcondition at the end of the program. The application of the axiomatic approach to distributed programs is, however, far from well understood. Unlike the simple input/output behavior presented by a sequential program, a distributed program usually has a number of interacting processes which are mutually dependent in the course of their executions. Despite the many techniques that have been proposed to tackle the new problems associated with distributed programs, more experience is needed before they can be of practical use.

Besides the fact that the axiomatic approach is still in the experimental stage, there are certain fundamental difficulties which prevent
us from using this approach for the conformity analysis of communication protocols.

- Most of the axiomatic-based systems can only deal with partial correctness of CSP programs. In other words, they are used to prove that certain properties will hold after the execution of a program, provided that it terminates. In contrast, we are interested in the communication sequence patterns presented by systems which usually involve infinite computations.

- In axiomatic-based systems, a program behavior is described by a set of logic formula. In general, it is difficult to guarantee that these formula can completely characterize the properties of the program; they can at best serve as a substantial but incomplete description of the program behavior. However, to establish the equivalence of two program behaviors, the strongest descriptions of the programs are required.

- The axiomatic-based systems aim at proving some desired properties of a CSP program, i.e., a closed set of processes communicating with each other. However, we are concerned with the external behavior of a set of processes which may interact with the environment, i.e., an open system.
5.3 Transformational Approach

Rather than taking the axiomatic approach, we developed a transformational approach for the conformity analysis of communication systems.

The basic idea of our approach is as follows. Instead of performing the logic reasoning on the CSP processes, we intend to transform a CSP process into a set of algebraic expressions. These algebraic expressions should represent the complete description of the communication behavior of the original process. Furthermore, the algebraic system itself should be equipped with the appropriate operators to support the activities of conformity analysis. Once we can achieve this, we are able to perform the analysis of a set of CSP processes by simple algebraic manipulations of their derived expressions.

The immediate advantage of this approach is that, in general, algebraic manipulations can be carried out more systematically and mechanically than the mathematical logic inferences performed in the axiomatic approach. However, in order to obtain this advantage, a major premise is that the transformation from CSP processes to algebraic expressions should be performed in a simple and orderly manner. For this purpose, we developed a transformation system consisting of a set of rules by which the transformation is conducted. Milner’s
Calculus of Communicating Systems (CCS) [28] was chosen as the target language of our transformation system for the following reasons:

- CCS bears many similarities with CSP, thus making the transformation system simple and straightforward. In particular, the concept of “interaction” in both languages is based on synchronous communication.

- Besides being an elegant notation for describing communication behavior, CCS provides a set of operators to manipulate communication behaviors. Especially, the composition operator can be used to derive the integrated behavior of a set of cooperating system components, while the restriction operator can be used to hide internal communications.

- CCS is associated with a sound underlying theory to show the equivalence of two communication behaviors — an essential activity in conformity analysis.

To end this section, we like to point out that the transformational approach [31] has been used for software development. That is, starting with a formal specification, a transformation process is performed for transforming the specification into an implementation. In contrast, we use the transformational approach to derive from the specifications the properties of communication behaviors in terms of algebraic expressions, which are subsequently used for functional analysis.
5.4 Calculus of Communicating Systems

Milner's Calculus of Communicating Systems (CCS) [28] is intended to describe the communication behaviors of concurrent systems in terms of a small set of operators. In addition, an associated mathematical theory is provided for reasoning about the communication behaviors. In this section, we present a brief overview of CCS's major concepts and notations which are pertinent to the transformation system presented in the next chapter.

5.4.1 Actions

In CCS, a communication behavior is modeled by a basic component called an agent. An agent can perform the following three kinds of atomic actions.

- **Input Action:** \( \alpha x \)

  \( \alpha \) is an arbitrary identifier called a *label* which is used to bind a *value variable* \( x \). The effect of this action is that variable \( x \) will be bound to an input value through an \( \alpha \)-action.

- **Output Action:** \( \bar{\alpha} e \)

  \( \bar{\alpha} \) is a label *complementary* to \( \alpha \), and \( e \) is a *value expression*. The effect of this action is to output the value of \( e \) through a \( \bar{\alpha} \)-action.
• Silent Action: \( \tau \)

The special symbol \( \tau \) stands for an *invisible* action performed by an agent. As we shall see, a \( \tau \)-action represents the autonomous behavior of an agent, which is usually the consequence of an internal communication between the sub-components inside the agent.

A communication may occur between two agents when complementary actions are performed simultaneously by these two agents. For instance, if \( \alpha x \) is an input action of agent \( P \) and \( \bar{\alpha} e \) is an output action of \( Q \), a communication (we shall call it an \( \alpha \)-type interaction) may take place. As a result, variable \( x \) will be qualified to the value of \( e \). As a special case, an interaction may occur solely for the purpose of synchronization if the variable and expression are not specified in a pair of complementary actions such as \( \alpha \) and \( \bar{\alpha} \).

5.4.2 Operators

In CCS, an agent is expressed as a behavior expression. For the construction of behavior expressions, a set of operators is defined in CCS as explained below. In the following, we denote \( B \) as an arbitrary behavior expression, and \( A \) as a set of labels.
• **Inaction:** Nil

Nil is a nullary operation. It stands for an agent which has no actions. Usually, it is used as a “terminator” of the behavior expressions which exhibit finite behaviors.

• **Action:** \( \mu.B \) where \( \mu \) is \( \alpha \), \( \bar{\alpha} \), or \( \tau \)

The expression \( \mu.B \) is the concatenation of an action \( \mu \) and a behavior expression \( B \). It stands for an agent which performs \( \mu \) first, and then behaves like \( B \).

• **Summation:** \( B_1 + B_2 + \ldots + B_n \)

This expression is a sum of \( n \) alternatives, with each alternative representing one of the potential behaviors. In other words, the whole expression stands for an agent which may act either as \( B_1 \) or \( B_2 \), ..., or \( B_n \).

• **Composition:** \( B_1 \mid B_2 \)

This expression represents an agent whose behavior is the concurrent execution of \( B_1 \) and \( B_2 \). Communications may occur between \( B_1 \) and \( B_2 \) through the complementary actions.

• **Restriction:** \( B \setminus L \) where \( L \in \Lambda \)

The expression \( B \setminus L \) stands for an agent which behaves like \( B \) ex-
cept that any \( \alpha \)-action or \( \overline{\alpha} \)-action is prohibited if \( \alpha \) lies in the label set \( L \). Intuitively, this operator imposes that certain interactions can only be performed internally in \( B \).

\[ \bullet \text{ Relabeling: } B[S] \text{ where } S: A \rightarrow A \]

This expression stands for an agent which behaves as agent \( B \) with its actions relabeled according to the function \( S \). \( S \) is a mapping from \( A \) to \( A \) which respects complement, i.e., \( S(\alpha) = S(\overline{\alpha}) \). In addition, \( S \) is defined such that \( S(\tau) = \tau \).

\[ \bullet \text{ Parameterized Behavior: } B(x_1, x_2, \ldots, x_n) \]

\( B \) is the identifier of a behavior expression, in which the parameters \( x_1, \ldots, x_n \) appear as \textit{free variables}, i.e., value variables which are not bound by any input actions. For instance, suppose we have \( B(y) = \alpha x.\text{Nil} + \beta y.\text{Nil} \), while variable \( x \) is bound by input action \( \alpha \), \( y \) is a free variable whose value will be determined by a supplied constant value.

\[ \bullet \text{ Recursion: } B(x_1, x_2, \ldots, x_n) \leftarrow P \]

where \( P \) is a behavior expression in which \( B \) appears, and all the free variables in \( P \) are in \( \{x_1, x_2, \ldots, x_n\} \).

The behavior of \( B \) is defined recursively through the expression \( P \). Recursion provides the ability to define infinite behaviors. For instance, suppose \( B(x) \leftarrow \overline{\alpha x}.B(x+1) \); then \( B(1) \) is an agent which outputs an infinite number of positive integers in increasing order.
• **Conditional:** if \( b \) then \( B_1 \) else \( B_2 \)
  
  where \( b \) is a boolean expression

  This expression represents an agent which behaves either like \( B_1 \)
  if the value of \( b \) is true or like \( B_2 \) if \( b \) is false. Conditional ex-
  pressions allow agents to behave according to the values of certain
  variables.

### 5.4.3 Expansion Theorem

In general, a behavior expression \( B \) can be presented in the form
of \( B = \sum_i \mu_i B_i \) (called a *sumform* of \( B \)). For any \( i \), \( \mu_i B_i \) is called a
*summand* of \( B \), where \( \mu_i \) can be an input action, an output action, or a
\( \tau \)-action. In CCS, there is an expansion theorem which can be used
to expand the behavior of \( (B_1 | B_2 | \ldots | B_n) \) into a sumform.
Expansion Theorem:

Let \( P = (B_1 \mid B_2 \mid \ldots \mid B_n) \setminus L \) where \( L \) is a set of label

Then

\[
P = \sum \mu.(B_{ij} \mid \ldots \mid B'_{ij} \mid \ldots \mid B_n) \setminus L
\]

where \( \mu.B_i' \) is a summand of \( B_i \),

the label of \( \mu \not\in L \)

\[
+ \sum \tau.(B_{ij} \mid \ldots \mid B'_{ij} \mid \ldots \mid B_{ij} \mid \ldots \mid B_n) \setminus L
\]

where \( 1 \leq i, j \leq m, i \neq j \)

\( \lambda.B_i' \) is a summand of \( B_i \)

\( \bar{\lambda}.B_j' \) is a summand of \( B_j \)

Intuitively, the expansion theorem reflects the way a set of concurrent agents is going to proceed. In this theorem, the first term means that any agent \( B_i \) in \( P \) can perform an unrestricted action, namely, a \( \tau \)-action, an \( \alpha \)-action, or an \( \bar{\alpha} \)-action, where \( \alpha \not\in L \). The second term means that any two agents in \( P \) can engage in an internal communication by performing a pair of \( \lambda \) and \( \bar{\lambda} \) actions which will result in a \( \tau \) appearing in the expression. The sumform of \( P \) can be obtained by repeating the expansion steps until no further actions are possible. In this theorem, it can be seen that, due to the definition of "\( \setminus \)" operator, any restricted action, i.e., the action whose label is in \( L \), can only be performed through internal communications. To demonstrate the use of the expansion theorem, a simple example is given below.
Example: Two agents $B_1$ and $B_2$, with the actions they intend to perform, can be conceptually pictured as in Fig. 5.2.

![Figure 5.2. Two Communicating Agents](image)

Their actual behaviors are described as the following expressions:

$$B_1 = a \cdot \bar{s} \cdot c \cdot \text{Nil} \quad B_2 = b \cdot s \cdot d \cdot \text{Nil}$$

In other words, $B_1$ will perform an a-type input, an s-type output, a c-type input, and finally stop. $B_2$ will perform a b-type input, an s-type input, a d-type input, and finally stop.

Now, we shall derive the behavior of $(B_1|B_2) \backslash s$ — an agent behaves as the concurrent execution of $B_1$ and $B_2$, with the s-type action restricted as the internal communication.

By the expansion theorem, we can get:

$$\begin{align*}
( B_1 \mid B_2 ) \backslash s &= ( a \cdot \bar{s} \cdot c \cdot \text{Nil} \mid b \cdot s \cdot d \cdot \text{Nil} ) \backslash s \\
&= a \cdot ( \bar{s} \cdot c \cdot \text{Nil} \mid b \cdot s \cdot d \cdot \text{Nil} ) \backslash s \\
&\quad + b \cdot ( a \cdot \bar{s} \cdot c \cdot \text{Nil} \mid s \cdot d \cdot \text{Nil} ) \backslash s
\end{align*}$$
= a. b. ( \overline{s}. c. Nil | s. d. Nil ) \setminus s
+ b. a. ( \overline{s}. c. Nil | s. d. Nil ) \setminus s

= a. b. r. ( c. Nil | d. Nil ) \setminus s
+ b. a. r. ( c. Nil | d. Nil ) \setminus s

= a. b. r. ( c. (Nil | d. Nil) \setminus s + d. (c. Nil | Nil) \setminus s )
+ b. a. r. ( c. (Nil | d. Nil) \setminus s + d. (c. Nil | Nil) \setminus s )

= a. b. r. ( c. d. Nil + d. c. Nil )
+ b. a. r. ( c. d. Nil + d. c. Nil )

It can be seen that, from the external environment, the resulting behavior can be perceived as a single agent (Fig. 5.3) which first performs an a-type and a b-type input in an arbitrary order, and then performs a c-type and a d-type input also in an arbitrary order.

Figure 5.3. The Integrated Communicating Agent
5.4.4 Observation Equivalence

In CCS, the concept of *observation equivalence* is defined in order to compare the *external behaviors* of different agents. As we shall see, the appearance of \( \tau \)-actions in behavior expressions is an important factor in determining the equivalence of two agents — which explains why they should be kept in an expression even though they actually represent *invisible* actions.

Suppose \( \Lambda \) is a label set and \( s \) is an arbitrary sequence of labeled actions, i.e., \( s \in \Lambda^* \). The notation \( P \xrightarrow{s} P' \) means "\( P \) may proceed to \( P' \) when \( s \) has been executed". Moreover, in the course of executing \( s \), an arbitrary number of \( \tau \) actions can be performed at any point in \( s \). Based on this definition, the observation equivalence relation is defined as follows:

**DEFINITION:** Observation Equivalence (\( \approx \))

For two behavior expressions \( P \) and \( Q \),

1. \( P \approx_0 Q \) is always true.
2. \( P \approx_{k+1} Q \) if \( \forall s \in \Lambda^* \)
   - (i) if \( P \xrightarrow{s} P' \), then for some \( Q' \), \( Q \xrightarrow{s} Q' \) and \( P' \approx_k Q' \)
   - (ii) if \( Q \xrightarrow{s} Q' \), then for some \( P' \), \( P \xrightarrow{s} P' \) and \( P' \approx_k Q' \)
3. \( P \approx Q \) if \( \forall k \geq 0, P \approx_k Q \)
Intuitively, this definition shows that two agents are observation equivalent if after performing a common sequence of visible actions by both agents*, any action sequences that are possible for one agent in the future should also be possible for the other. The best way to understand this definition is by investigating some representative examples.

Example: Three agents $P_I$, $P_g$, and $P_s$ are defined by the expressions below and can be pictured in tree presentations as in Fig. 5.4.

$$P_I = \alpha. ( \beta. \text{Nil} + \gamma. \text{Nil} )$$
$$P_g = \alpha. ( \beta. \text{Nil} + \tau. \gamma. \text{Nil} )$$
$$P_s = \alpha. \beta. \text{Nil} + \alpha. \gamma. \text{Nil}$$

Figure 5.4. The Tree Representation of Agents

By the definition of observation equivalence, the relations between these three agents can be found as follows:

*Recall that the invisible $\tau$-actions performed in the course of the execution can be different in the two agents.
According to the definition, any pair of agents is in the relation.

The relation in CCS is analogous to the notion of "two machines accept the same language" in automata theory. It can be readily seen that \( \{ \epsilon, \alpha, \alpha\beta, \alpha\gamma \} \) is the set of possible sequences of actions for all three agents.

After performing an \( \alpha \)-action,

(i) In \( P_1 \), \( \beta \)-action and \( \gamma \)-action are always possible as the next action.

(ii) In \( P_2 \), while \( \gamma \)-action is always possible as the next action, \( \beta \)-action is only possible when the \( \tau \)-action has not performed.

(iii) In \( P_3 \), only one (either \( \beta \) or \( \gamma \)) action is possible.

From (3), we conclude that none of the three agents is observation equivalent to the others.

This example demonstrates that CCS’s observation equivalence relation is very strict in the sense that, for two agents to be equivalent, one agent should be able to "simulate" the other from any point after they have executed any common sequence of visible actions. While this
equivalence relation is far from straightforward, we believe it is crucial when we consider the conformity of communication protocols to their services.

5.5 Summary

The purpose of the conformity analysis is to show that the composite behavior of the (N)-protocol specification and the (N-1)-service specification with respect to the upper users conforms to the (N)-service specification. Consequently, any method for conformity analysis should be able to reason the communication behavior of a given specification, to integrate several specifications into an overall behavior, to hide the internal communications, and to demonstrate the equivalence of two communication behaviors.

To perform the conformity analysis on the CSP-based specifications, we developed a transformation system, which will be presented in the next chapter, to extract from a CSP process the communication sequences which may arise during its execution and to express these sequences in terms of CCS behavior expressions. Based on this system, we are able to transform a set of cooperating CSP processes into a set of CCS expressions, and then derive the integrated behavior with respect to the environment by using the CCS composition and restriction operators.
Also, the conformity of the (N)-protocol to its service can be shown by proving that the CCS expression, representing the integrated behavior of the (N)-entities and (N-1)-service provider with respect to the users at the next layer, is observation equivalent to the CCS expression which represents the behavior of (N)-service provider. The overall steps in conformity analysis are outlined in Fig. 5.5.
Figure 5.5. An Overview of Conformity Analysis
CHAPTER VI
Transformational Approach

6.1 Perspective View

The major activity in the transformational approach, as stated in the previous chapter, is to extract information about the communicating behavior of processes. A transformation system consisting of a set of transformation rules is developed such that, for a given CSP process, the corresponding CCS behavior expression, which characterizes its communication sequences, can be generated. The basic concepts of the transformation system are as follows:

1) For individual processes:

(a) The variables $x_1, x_2, \ldots, x_n$ (denoted as a vector $\mathbf{x}$) of a CSP process are mapped into the parameters of the corresponding CCS expressions. Throughout the discussion, a state will refer to either the values of variables of a CSP process or the values of the parameters of a CCS expression, depending on the context.
DEFINITION 1: for a process with variables $x_1, x_2, \ldots, x_n$,

$$\bar{x} = <x_1, x_2, \ldots, x_n>$$  i.e. $\bar{x}$ is a state variable.

(b) The transformation rule, from a CSP command $C$ (in a process with $\bar{x}$ as its variables) to a CCS expression $E$, is in the form of:

$$\llbracket C \rrbracket(\bar{x}) = E$$

For simple commands such as assignment, input/output, or skip command, the general form of their corresponding CCS expressions is as follows:

GENERAL FORM:  $<\text{action}> \cdot \bar{a} \bar{x} \cdot \text{Nil}$

That is, the effect of a simple statement is to perform the action part, then to output the resultant state, and finally to end its life.

(c) For the sequential composition of two CSP commands $C_1; C_2$, we use a derived CCS operator "before" to link the CCS expressions derived from $C_1$ and $C_2$.

DEFINITION 2: for two behavior expressions $E_1$ and $E_2$,

$$E_1 \textbf{ before } E_2 = ( E_1 | \beta/\alpha | \beta \bar{x}.E_2 ) \setminus \beta$$

Based on this operator, the sequential composition rule can be expressed as follows:
Sequential Composition Rule:

\[ \llbracket C_1 ; C_2 \rrbracket(x) = \llbracket C_1 \rrbracket(\overline{x}) \text{ before } \llbracket C_2 \rrbracket(\overline{x'}) \]

Intuitively speaking, the "before" operator in the sequential composition rule is used for the following purposes:

(i) To create a binding for the state variable \( \overline{x}' \) by \( \beta \overline{x}' \) (recall the definition of before operator). As we will see below, the variable \( \overline{x}' \) is bound to the resultant state after performing the actions of \( \llbracket C_1 \rrbracket(\overline{x}) \).

(ii) To synchronize the actions of \( \llbracket C_1 \rrbracket(\overline{x}) \) and \( \llbracket C_2 \rrbracket(\overline{x'}) \) by first performing the action part of \( \llbracket C_1 \rrbracket(\overline{x}) \). Then the resultant state is delivered to \( \overline{x}' \) via \( \beta \) communication (recall that an \( \overline{a} \overline{x} \) action is expected to be at the end of \( \llbracket C_1 \rrbracket(\overline{x}) \), and its name has been changed to \( \beta \) by the relabeling operation \( [\beta/\alpha] \)). Finally \( \llbracket C_2 \rrbracket(\overline{x'}) \) is performed.

(iii) To hide the state-passing communication by the restriction operation \( \setminus \beta \).

In short, the sequential composition rule states that the behavior of \( C_1;C_2 \) at state \( \overline{x} \) is first to perform the action part of \( \llbracket C_1 \rrbracket(\overline{x}) \), then to pass the resultant state to \( \overline{x}' \) "secretly", and to behave as \( \llbracket C_2 \rrbracket(\overline{x'}) \).
2) For a set of cooperating processes:

The integrated behavior of a set of processes is obtained by applying the parallel composition operation $|$ to a set of CCS expressions derived from individual processes and, at the same time, by hiding all the internal communications between the processes. Consequently, the resultant expression will characterize all the sequences of communications with respect to the environment, i.e., externally observable behavior.

Formally, let $P_1, P_2, \ldots, P_n$ be a set of processes with $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ as their state vectors, respectively. The externally observable behavior can be derived by the following rule:

\[
\text{Parallel Composition Rule:}
\]

\[
[P_1 || P_2 || \ldots || P_n](\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) = \bigl(\bigl( \bigcap_{i=1}^{n} [P_i](\bar{x}_i) \setminus \text{internal} \bigr) \bigr)
\]

where

\[
\text{internal} = \{ t \mid t \text{ is a name in } [P_l] \text{ and } [P_m], \ l, m \in 1, \ldots, n \} \cup \{a\}
\]
6.2 Transformation Rules

In the following, the transformation rules for the CSP commands are presented, each of which is followed by a brief explanation.

**Skip Command:**

\[
\llbracket \text{Skip} \rrbracket (\vec{x}) = \vec{a} \vec{x} . \text{Nil}
\]

The skip command is a null command. Consequently, there is no action part in its corresponding expression (recall the general form of simple commands). All it does is to output the value of the current state and to end its life.

**Assignment Command:**

\[
\llbracket x_i := e \rrbracket (\vec{x}) = \vec{a} \vec{x} \{e/x_i\} . \text{Nil}
\]

The assignment command assigns the value of expression \( e \) to the variable \( x_i \). Here, the effect of state change is achieved by \( \vec{a} \vec{x} \{e/x_i\} \), which denotes the output of the current state with the exception that \( x_i \) is replaced by the value of \( e \).
**Input Command:**

\[ \llbracket P_i?t(x_i) \rrbracket(\bar{x}) = t x_i . \bar{x} . \text{Nil} \]

The command \( P_i?t(x_i) \) (say, in process \( P_m \)) expresses the intention to communicate with \( P_m!t(e) \) in process \( P_i \). The CCS expression for input command denotes that the actions performed are \( t \)-type input, followed by state-passing output.

It should be noted that the variable \( x_i \) in \( \bar{x} \) is bounded by \( t \); thus, its value sent by \( \bar{x} \) is the one received by \( t x_i \). In addition, we impose that the constructor \( t \) can only occur in a fixed pair of processes. This convention will ensure proper communications during the parallel composition step even though we drop the process name in this rule.

**Output Command:**

\[ \llbracket P_i!t(e) \rrbracket(\bar{x}) = \bar{e} . \bar{x} . \text{Nil} \]

For an output command, the action is to output the value of expression \( e \) through a \( t \)-type communication and then to output the unchanged state. Again, since the communication pairs can be determined solely by the type names when performing the parallel composition, we can drop the process name here.
Abort:

\[
[[ \text{abort} ]] (\vec{x}) = \omega \quad \text{where} \quad \omega = \tau . \omega
\]

Abort is not a CSP command but a behavior to model the failure situation of the alternative command. It is specified as a recursive expression \( \omega \) which does nothing but perform meaningless \( \tau \)-actions an infinite number of times.

Alternative Command:

\[
\lfloor \left[ \left[ G_1 \rightarrow C_1 \uplus G_2 \rightarrow C_2 \uplus \ldots \uplus G_n \rightarrow C_n \right] \right] \rfloor (\vec{x}) =
\]

\[
\left( \sum_i \text{if } b_i(\vec{x}) \text{ then } \tau . \lfloor C_i \rfloor (\vec{x}) \text{ else } \text{Nil} \right)
\]

\[
\text{where } G_i = b_i
\]

\[
+ \sum_j \text{if } b_j(\vec{x}) \text{ then } \lfloor P_j.t(x_j);C_j \rfloor (\vec{x}) \text{ else } \text{Nil}
\]

\[
\text{where } G_j = b_j;P_j.t(x_j)
\]

\[
+ \sum_k \text{if } b_k(\vec{x}) \text{ then } \lfloor P_k.t(e);C_k \rfloor (\vec{x}) \text{ else } \text{Nil}
\]

\[
\text{where } G_k = b_k;P_k.t(e)
\]

\[
+ \text{if } \neg(\forall b_i) \text{ then } \lfloor \text{abort} \rfloor (\vec{x}) \text{ else } \text{Nil}
\]

\[
\text{where } \forall b_i = b_1 \lor \ldots \lor b_n
\]
For an alternative command, there is an if-then-else term for each guarded command representing one of the many possible behaviors. By the definition of the if-then-else construct in CCS, the value of the boolean condition determines the actual behavior. That is, if the boolean condition is true, the whole expression can be textually replaced by the then-part, otherwise the else-part.

Thus, the rule for an alternative command can be explained as follows:

1) The boolean parts of all the guarded commands are evaluated first.

(a) For any guarded command with the boolean part being false, its corresponding CCS term is Nil, which means it contributes nothing to the actual behavior (for any CCS expression $\xi$, $\xi + \text{Nil} = \xi$).

(b) For any guarded command with the boolean part being true, and its guard is:

(i) boolean guard:
   Its corresponding CCS term is a $r$-action, which models the potential commitment of executing this guarded command, followed by the expression derived for its command list.

(ii) I/O guard:
   Its corresponding CCS term is the expression derived from the sequential composition of the input/output command in its guard and its command list.
(c) If all the boolean parts are evaluated to be false, the alternative command fails. As shown in the transformation rule, the failure situation is modeled as abort.

2) The CCS expression for the alternative command is the sum of the terms derived in step 1). Each of these terms represents the potential actual behavior of the alternative command.

### Repetitive Command:

\[
\begin{align*}
\omega(\vec{x}) &= \omega(\vec{x}) \\
\omega(\vec{x}) &= (\sum_i \text{if } b_i(\vec{x}) \text{ then } \tau \cdot [C_i]((\vec{x})) \text{ before } \omega(\vec{x}') \text{ else } \text{Nil} \\
&\quad \text{where } G_i = b_i \\
&\quad + \sum_j \text{if } b_j(\vec{x}) \text{ then } [P_j; t(x_j); C_j]((\vec{x})) \text{ before } \omega(\vec{x}') \text{ else } \text{Nil} \\
&\quad \text{where } G_j = b_j; P_j; t(x_j) \\
&\quad + \sum_k \text{if } b_k(\vec{x}) \text{ then } [P_k; t(e); C_k]((\vec{x})) \text{ before } \omega(\vec{x}') \text{ else } \text{Nil} \\
&\quad \text{where } G_k = b_k; P_k; t(e) \\
&\quad + \text{if } \neg(\forall b) \text{ then } [\text{skip}]((\vec{x})) \text{ else } \text{Nil} \\
&\quad \text{where } \forall b_i = b_1 \lor \ldots \lor b_n
\end{align*}
\]

The behavior of a repetitive command is presented as a recursive expression \(\omega(\vec{x})\), which is similar to the expression for an alternative command except that:
1) After one of its terms is performed, the resultant state will be passed to $\tilde{x}'$, and then it behaves like $\omega(\tilde{x}')$.

2) At the time when all guards fail, the behavior is the same as that of a skip command.

In this rule, we exclude the distributed termination convention in the original CSP. In other words, a repetitive command can terminate only when all the boolean parts of its guarded commands become false. A version of the rule which considers the distributed termination is presented in Appendix A.

6.3 Transformation Procedures

In the following, two examples are given to show how a transformation process is conducted.

---

**Example 1:** A binary semaphore, shared by two users User1 and User2, can be specified as the following CSP process:

\[
\text{Sema::} \quad * \mid \text{User1}\?P_1() \rightarrow \text{User1}\?V_1() \\
\quad \mid \text{User2}\?P_2() \rightarrow \text{User2}\?V_2()
\]

To transform it into a CCS behavior expression, we have:

\[
\llbracket \text{Sema} \rrbracket = \text{SEM}A
\]
This expression describes the binary semaphore as a communicating agent which in each cycle engages in either a pair of $P_I, V_I$ interactions or a pair of $P_g, V_g$ interactions. Notice that in this example $\bar{a}$ is solely for the synchronization purpose, since there are no variables in the process.

Example 2: A process $T$ (eacher), which communicates with process $S$ (tudent), is specified below. The label $L$ is used to stand for the inner repetitive command.
In this process, the state is the value of variable "done". The transformation procedure for process $T$ is presented as follows (we skip the intermediate steps when they are obvious):

Let $\llbracket T \rrbracket(done) = T(done)$

$T(done) \leftarrow \llbracket S?ask(); done:=false; L \rrbracket(done) \textbf{before} T(done')$

$\quad = \text{ask. } \tau. \tau. \llbracket L \rrbracket(false) \textbf{before} T(done') \quad (1)$

It should be clear that $T(true)$ is same as $T(false)$, since the behavior of process $T$ does not depend on the initial value of "done" which can be true, false, or (in fact) undefined. Later we will use $T$ to represent the behavior of process $T$ in the final step of the transformation procedure.

Let $\llbracket L \rrbracket(done) = L(done)$

By the transformation rule of the repetitive command, we get:

$L(true) = \llbracket \text{skip} \rrbracket(true)$

$L(false) \leftarrow \overline{\text{answer. } \tau. ( \llbracket S?ok(); done:=true \rrbracket(false)}$

$\quad + \llbracket S?Nok(); skip \rrbracket(false) \textbf{ before} L(done')$
Let $[ L ](false)$ before $T(done') = W$ (2)

$W \leftarrow \text{answer. } r. ( \text{ok. } r. \ r. \ \text{\texttt{skip}}(true)
+ \text{Nok. } r. \ r. \ L(false) ) \text{ before } T(done')$

$= \text{answer. } r. ( \text{ok. } r. \ r. \ \text{\texttt{true}} \text{ before } T(done')
+ \text{Nok. } r. \ r. \ L(false) \text{ before } T(done') )$

$= \text{answer. } r. ( \text{ok. } r. \ r. \ T(true) + \text{Nok. } r. \ r. \ W )$

$\approx \text{answer. } ( \text{ok. } T(true) + \text{Nok. } W )$ (3)

From (1), (2), (3), we can finally get:

$[ T ](done) = T$

$T \approx \text{ask. } W$

$W \leftarrow \text{answer. } ( \text{ok. } T + \text{Nok. } W )$

The expressions above can characterize the communication behavior of process $T$. In words, $T$ first receives an ask signal and responds with an answer signal. It then waits for an input signal. If the signal is
ok, a cycle is completed and T will be ready for another ask signal. On the other hand, if the signal is Nok, T will retransmit the answer signal and wait for another incoming signal.

6.4 Discussions

In this chapter, a transformation system has been presented in detail. Based upon this transformation system, the communication behaviors presented by the service and protocol specifications can be derived and expressed in CCS behavior expressions, on which the functional analysis can be subsequently performed by algebraic manipulations and observation equivalence proof.

It can be argued that CCS could have been adopted as a specification language. Why then do we need to use CSP as the specification language and later transform the specifications into CCS expressions? In our experience, while a CCS-like language can express many communication behaviors elegantly, the complexity of the specifications grows quickly when systems become large. In particular, due to its lack of state component (which is realized by process variables in CSP) and its lack of iteration construct (which is realized by the repetitive command in CSP), the specifications are usually presented as many instantiations of parameterized behaviors which are defined in a mutually recursive
fashion. This may greatly reduce the comprehensibility of the specifications to the implementors. Furthermore, while a CCS-like language can specify services reasonable well, it may be awkward to specify protocols (in contrast to using extended finite state machines or CSP), as some of the information can fit naturally into the state components or variables. The observations above led us to feel that the CCS-like languages should be used more as an analysis tool rather than as a specification language.

In fact, in our approach, we can view the specifications based on CSP as operational specifications and the derived CCS expressions as more abstract specifications showing only the observable behaviors. We believe that they can complement each other in the protocol design phase — operational specifications are easier to construct and to understand, whereas abstract specifications are more amenable to analysis.

The transformation system from CSP to CCS is straightforward and syntax-directed. For a given CSP process, the system allows us to suppress the local computations and deal with its communication behavior only. In particular, for a process which performs cyclic operations, the derived CCS expression can serve as an “invariant” property on its communication behavior. This provides a certain advantage over axiomatic proof systems, since by using these systems, it is the prover
who has to elaborate the invariant properties for cyclic computations — a heavy burden.

In [15], Hennessy, Li, and Plotkin proposed a technique for translating CSP into CCS. Their approach is different from ours in many ways. Most importantly, they aimed at translating a CSP program into a CCS expression with the same States×States relation, i.e., given an initial state, both the CSP program and the CCS expression will reach the same final state when they terminate. In contrast, we are more concerned with the communication behavior exhibited during the execution of CSP processes, rather than the States×States relation. In fact, as shown in this chapter, we aim to study the communication behaviors of systems which may run forever (thus there is no final state) and may communicate with not only other system components but also the environment (thus they are not "programs"). Moreover, in their approach, guards and commands are translated into "CCS contexts" — CCS terms with holes in them. The CCS term for two consecutive commands is obtained by filling the hole in the term of the first command with the term of the second command. We transform each CSP command into a CCS expression; the expression for two consecutive commands is derived by linking two expressions via a special β-communication. Finally, each variable in a CSP process is modeled by an individual CCS expression in their approach; we, however, treat the variables of CSP processes as the parameters of the corresponding CCS expressions.
CHAPTER VII
Case Studies

In this chapter, the Alternating Bit Protocol [5] is used as a case study to illustrate how we can perform the conformity analysis based on the transformation system discussed in the previous chapter. Also, the X.25 packet level DCE/DTE interface is used to show the detection of syntactic errors in the transformational approach.

7.1 Alternating Bit Protocol

As shown in Fig. 7.1, two protocol entities S and R, together with a transmission medium M, which is the underlying service provider, form a communication system for User1 and User2. Due to the occasional malfunction of transmission medium, data may be corrupted or lost during transmission. To cope with message corruption, the protocol entities S and R use a control bit as the error recovery mechanism to guarantee the reliable data transfer from User1 to User2. To cope with message loss, a timer is used to trigger the sending entity S to
retransmit the last message. It is required that the time-out interval is properly set so that time-out can occur only after a transmission loss has happened.

![ABP Communication System Diagram](image)

Figure 7.1. ABP Communication System

In Section 7.1.1, the intended service as well as the protocol entities and transmission medium are formally specified. In order to highlight the control structures, the protocol specification is slightly different from the one in Chapter 4. Specifically, the control information, such as the sequence number and error situation, are encoded as different types of communication. Also, the actual contents of the transmitted messages are ignored. Finally, some alternative commands and repetitive commands are associated with labels which will be used during transformation procedures. In Section 7.1.2, we demonstrate the conformity analysis based on the transformation system.
7.1.1 Service and Protocol Specifications

- **ABP Service:**

  \[
  \text{Service :: } *| \text{User1?GET()} \rightarrow \text{User2!OUT()} |
  \]

  From the users' point of view, ABP provides a reliable one-way data transfer service from User1 to User2. The whole communication system can be described as a process Service. It engages in the communication GET() each time User1 offers a new message, followed by the communication OUT() when the message is delivered to User2. Here, we are only interested in the types of communications; the contents of the messages are irrelevant and have been omitted.

  **Transformation:**

  In process Service, \( \hat{x} = () \) i.e., since there are no variables, the state vector is null.

  \[
  [\text{Service}] \leftarrow \\
  \begin{align*}
  &\text{if true then } [\text{User1?GET(); User2!OUT()}] \text{ before } [\text{Service}] \text{ else Nil} \\
  &\text{if false then } [\text{skip}] \text{ else Nil} \\
  &= [\text{User1?GET(); User2!OUT()}] \text{ before } [\text{Service}] \\
  &= \text{GET. OUT. } [\text{Service}]
  \end{align*}
  \]

  Let \( SERVICE = [\text{Service}] \), we have the final result:
[[ Service ]] = SERVICE

SERVICE ← GET. OUT. SERVICE

• ABP Protocol Entities and Transmission Medium:

1) Protocol Entity S:

S ::= 
  bit := 0;
  L: * User1?GET() →
      done := false;
  K: [ bit = 0 →
      TO: *(!done; M!D0()) →
          [ M?a0() → done:=true;
            bit:=1
            ]
          M?a1() → skip
          ]
      M?ae() → skip
      ]
      Timer?tick() → skip
  ]
  ]
  ]
  bit = 1 →
  T1: *(!done; M!D1()) →
      [ M?a1() → done:=true;
        bit:=0
        ]
      M?a0() → skip
      ]
      M?ae() → skip
      ]
      Timer?tick() → skip
      ]
S is the protocol entity which interacts with User1 through communication GET(). After receiving a new message, S sends it one or more times through the transmission medium until an appropriate acknowledgment is received. Depending on the value of the control bit, a message may be sent through a communication of type D0 or D1, each of which represents that the associated sequence number of the message is 0 or 1, respectively. The input communications a0, a1, and ae represent the reception of an acknowledgment with sequence number 0, an acknowledgment with sequence number 1, and an erroneous acknowledgment, respectively. The input communication tick represents the time-out situation.

**Transformation:**

In process S, $\bar{x} = (\text{bit,done})$.

In the following, we use * to represent “any values” or “don’t care”.

\[
\begin{align*}
\langle S \rangle (\ast, \ast) &= \langle \text{bit}: = 0 \rangle (\ast, \ast) \text{ before } \langle L \rangle (\bar{x}') \\
&= \tau \cdot \langle L \rangle (0, \ast) \\
\langle L \rangle (0, \ast) &\Leftarrow \text{Get. } \langle K \rangle (0, \text{false}). \text{ before } \langle L \rangle (\bar{x}') \\
\langle L \rangle (1, \ast) &\Leftarrow \text{Get. } \langle K \rangle (1, \text{false}). \text{ before } \langle L \rangle (\bar{x}') \quad (1') \\
\langle K \rangle (\ast, \text{false}) &= \text{if } \text{bit}=0 \text{ then } \langle T0 \rangle (0, \text{false}) \text{ else } \text{Nil} \\
&\quad + \text{ if } \text{bit}=1 \text{ then } \langle T1 \rangle (1, \text{false}) \text{ else } \text{Nil} \\
&\quad + \text{ if } \neg(\text{bit}=0 \lor \text{bit}=1) \text{ then } \langle \text{Abort} \rangle \text{ else } \text{Nil} \quad (2)
\end{align*}
\]
\[ [ T0 ](0,\text{false}) \triangleq \overline{D0}. \tau. ( \begin{align*} & a0. [ T0 ](1,\text{true}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & a1. [ T0 ](0,\text{false}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & \text{ae. } [ T0 ](0,\text{false}) \\
& \quad + \end{align*} \]
\[ \text{tick. } [ T0 ](0,\text{false}) \quad \) \quad \) \quad (3) \]
\[ [ T0 ](1,\text{true}) = [ \text{skip} ](1,\text{true}) \quad (4) \]

\[ [ T1 ](1,\text{false}) \triangleq \overline{D1}. \tau. ( \begin{align*} & a1. [ T1 ](0,\text{true}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & a0. [ T1 ](1,\text{false}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & \text{ae. } [ T1 ](1,\text{false}) \\
& \quad + \end{align*} \]
\[ \text{tick. } [ T1 ](1,\text{false}) \quad \) \quad \) \quad (3') \]

\[ [ T1 ](0,\text{true}) = [ \text{skip} ](0,\text{true}) \quad (4') \]

From (1), (2), (3) and (4), we get
\[ [ L ](0,\ast) = \text{Get. } [ K ](0,\text{false}). \text{before } [ L ](\overline{x'}) \]
\[ \approx \text{Get. } [ T0 ](0,\text{false}). \text{before } [ L ](\overline{x'}) \]
\[ \approx \text{Get. } \overline{D0}. ( \begin{align*} & a0. [ T0 ](1,\text{true}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & a1. [ T0 ](0,\text{false}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & \text{ae. } [ T0 ](0,\text{false}) \\
& \quad + \end{align*} \]
\[ \text{tick. } [ T0 ](0,\text{false}) \quad \) \quad \) \quad ) \quad \text{before } [ L ](\overline{x'}) \]
\[ \approx \text{Get. } \overline{D0}. ( \begin{align*} & a0. [ L ](1,\text{true}) \\
& \quad + \end{align*} \]
\[ \begin{align*} & a1. [ T0 ](0,\text{false}). \text{before } [ L ](\overline{x'}) \\
& \quad + \end{align*} \]

\[ + \]
ae. \[[ T0 \]](0,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \\
+ \\
tick. \[[ T0 \]](0,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \\
)

Similarly, from (1'), (2), (3') and (4'), we get
\[[ L \]](1,\ast) \approx \text{Get.} \[[ T1 \]](1,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \\
\approx \text{Get.} \overline{D1}. ( \begin{array}{l} a1. \[[ L \]](0,\text{true}) \\
+ \\
a0. \[[ T1 \]](1,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \\
+ \\
ae. \[[ T1 \]](1,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \\
+ \\
tick. \[[ T1 \]](1,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \\
) \\

Now, let \( S_0 = \[[ L \]](0,\ast) \)
\( S_I = \[[ L \]](1,\ast) \)
\( \tau_0 = \[[ T0 \]](0,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \)
\( \tau_I = \[[ T1 \]](1,\text{false}) \textbf{before} \[[ L \]](\tilde{x'}) \)

The final result is as follows:

\[
\begin{array}{l}
\[[ S \]](\ast,\ast) = \tau. \ S_0 \\
S_0 \approx \text{Get.} \ \tau_0 \\
\tau_0 \leftarrow \overline{D0}. ( \text{a0.} \ S_I + \text{a1.} \ \tau_0 + \text{ae.} \ \tau_0 + \text{tick.} \ \tau_0 ) \\
S_I \approx \text{Get.} \ \tau_I \\
\tau_I \leftarrow \overline{D1}. ( \text{a0.} \ S_0 + \text{a1.} \ \tau_I + \text{ae.} \ \tau_I + \text{tick.} \ \tau_I ) \\
\end{array}
\]
2) Transmission Medium M:

\[
M ::
\]

\[
*[S?D0() \rightarrow \{true \rightarrow R!d0() \} [true \rightarrow R!de() \} [true \rightarrow \text{skip} ]
\]

\[
S?D1() \rightarrow \{true \rightarrow R!d1() \} [true \rightarrow R!de() \} [true \rightarrow \text{skip} ]
\]

\[
R?A0() \rightarrow \{true \rightarrow S!a0() \} [true \rightarrow S!ae() \} [true \rightarrow \text{skip} ]
\]

\[
R?A1() \rightarrow \{true \rightarrow S!a1() \} [true \rightarrow S!ae() \} [true \rightarrow \text{skip} ]
\]

As mentioned earlier, the transmission medium may occasionally corrupt or lose the message under transmission. Accordingly, when M receives a message with sequence number 0 or 1 from S through D0() or D1(), it may deliver the message to R through d0() or d1(); or it may inform R through de() that a transmission error has occurred; or it may do nothing. Similar behavior is presented for the transmission of acknowledgments from R to S.

Transformation:

In process M, \( \bar{x} = () \)

\[
[[ M ]] \leftarrow \begin{array}{c}
D0. (r. \overline{d0}. \overline{a} + r. \overline{de}. \overline{a} + r. \overline{a}) \text{ before } [[ M ]] \\
+ D1. (r. \overline{d1}. \overline{a} + r. \overline{de}. \overline{a} + r. \overline{a}) \text{ before } [[ M ]] \\
+ A1. (r. \overline{a1}. \overline{a} + r. \overline{ae}. \overline{a} + r. \overline{a}) \text{ before } [[ M ]] \\
+ A0. (r. \overline{a0}. \overline{a} + r. \overline{ae}. \overline{a} + r. \overline{a}) \text{ before } [[ M ]] 
\end{array}
\]
= D0. (\( \overline{t} \overline{a0}. [M] + \overline{t} \overline{de}. [M] + \overline{t} \overline{M} \))  \\
+ D1. (\( \overline{t} \overline{d1}. [M] + \overline{t} \overline{de}. [M] + \overline{t} \overline{M} \))  \\
+ A1. (\( \overline{t} \overline{a1}. [M] + \overline{t} \overline{ae}. [M] + \overline{t} \overline{M} \))  \\
+ A0. (\( \overline{t} \overline{a0}. [M] + \overline{t} \overline{ae}. [M] + \overline{t} \overline{M} \))

Let \([M] = M\); the final result is:

\[
[M] = M
\]

\[
M \leftarrow D0. (\( \overline{t} \overline{a0}. M + \overline{t} \overline{de}. M + \overline{t} M \))  \\
+ D1. (\( \overline{t} \overline{d1}. M + \overline{t} \overline{de}. M + \overline{t} M \))  \\
+ A1. (\( \overline{t} \overline{a1}. M + \overline{t} \overline{ae}. M + \overline{t} M \))  \\
+ A0. (\( \overline{t} \overline{a0}. M + \overline{t} \overline{ae}. M + \overline{t} M \))
\]

3) Protocol Entity R:

\[
R ::
\]

\[
\begin{align*}
\text{exp} & := 0; \\
\text{J: } * \{ \text{exp} = 0 \rightarrow \text{U: } [M?do()] & \rightarrow \text{User2!OUT()}; \\
& \rightarrow M!A0(); \\
& \text{exp} := 1 \\
& [M?d1()] & \rightarrow M!A1(); \\
& [M?de()] & \rightarrow M!A1() \\
& \} \\
& \} \\
\text{exp} & := 1 \rightarrow \text{V: } [M?d1()] & \rightarrow \text{User2!OUT()}; \\
& \rightarrow M!A1(); \\
& \text{exp} := 0
\end{align*}
\]
The protocol entity $R$ is responsible for receiving the messages from the transmission medium. For each reception, $R$ delivers a message to User2 if it has the expected sequence number and then replies $S$ with an appropriate acknowledgment. Otherwise, it only sends back an acknowledgment with the opposite value of the expected sequence number.

**Transformation:**

In process $R$, $\bar{x} = (\text{exp})$. Again, we use $*$ to stand for any values.

\[
\begin{align*}
[ R ](* &= [ \text{exp}=1 ](*) \text{ before } [ J ](x') \\
&= \tau. [ J ](0) \\
[ J ](*) &\Leftarrow \begin{cases} 
\text{if exp}=0 \text{ then } [ U ](0) \text{ before } [ J ](\bar{x}') \text{ else } \text{Nil} \\
&+ \begin{cases} 
\text{if exp}=1 \text{ then } [ V ](1) \text{ before } [ J ](\bar{x}') \text{ else } \text{Nil} \\
&+ \begin{cases} 
\text{if } \neg(\text{exp}=0 \lor \text{exp}=1) \text{ then } [ \text{skip} ] \text{ else } \text{Nil} 
\end{cases}
\end{cases}
\end{align*}
\]

\[
[ U ](0) = \begin{cases} 
\text{d0. out. A0. }\bar{a}1. \text{ Nil} \\
&+ \begin{cases} 
\text{dl. A1. }\bar{a}0. \text{ Nil} \\
&+ \begin{cases} 
\text{de. A1. }\bar{a}0. \text{ Nil}
\end{cases}
\end{cases}
\end{cases}
\]

\[
\begin{align*}
&\text{M?d0()} \rightarrow \text{M!A0()} \\
&\text{M?de()} \rightarrow \text{M!A0()}
\end{align*}
\]
From (1) and (2), we have:

\[
\llbracket J \rrbracket(0) \leftarrow \llbracket U \rrbracket(0) \text{ before } \llbracket J \rrbracket(x')
\]

\[
= d_0. \overline{\text{out}}. A_0. \llbracket J \rrbracket(1) \\
+ d_1. A_1. \llbracket J \rrbracket(0) \\
+ \text{de. } A_1. \llbracket J \rrbracket(0)
\]

Similarly,

\[
\llbracket J \rrbracket(1) \leftarrow \llbracket V \rrbracket(0) \text{ before } \llbracket J \rrbracket(x')
\]

\[
= d_1. \overline{\text{out}}. A_1. \llbracket J \rrbracket(0) \\
+ d_0. A_0. \llbracket J \rrbracket(1) \\
+ \text{de. } A_0. \llbracket J \rrbracket(1)
\]

Now, let \( R_0 = \llbracket J \rrbracket(0) \), \( R_1 = \llbracket J \rrbracket(1) \)

We have the final result:

\[
\llbracket R \rrbracket(*) = \tau. R_0 \\
R_0 \leftarrow d_0. \overline{\text{out}}. A_0. R_1 + d_1. A_1. R_0 + \text{de. } A_1. R_0 \\
R_1 \leftarrow d_1. \overline{\text{out}}. A_1. R_0 + d_0. A_0. R_1 + \text{de. } A_0. R_1
\]
7.1.2 Conformity Analysis

In the previous section, the communication behaviors of the service, the protocol entities, and the transmission medium from their specifications have been derived. We list them together as follows:

1) Service:

\[
\text{[Service]} = SERVICE
\]

\[
SERVICE \leftarrow \text{GET. OUT. SERVICE}
\]

2) Protocol Entity S:

\[
\text{[S]}(\text{bit, done}) = S
\]

\[
S = \tau \cdot S_0
\]

\[
S_0 = \text{GET} \cdot T_0
\]

\[
T_0 \leftarrow \text{DO} . ( a_0 \cdot S_I + a_1 \cdot T_0 + \text{ae. } T_0 + \text{tick. } T_0 )
\]

\[
S_I = \text{GET} \cdot T_I
\]

\[
T_I \leftarrow \text{DI} . ( a_1 \cdot S_0 + a_0 \cdot T_I + \text{ae. } T_I + \text{tick. } T_I )
\]

3) Transmission Medium M:

\[
\text{[M]} = M
\]

\[
M \leftarrow \text{DO} . ( \tau \cdot \text{do} . M + \tau \cdot \text{de. } M + \tau \cdot M )
\]

\[
+ \text{DI} . ( \tau \cdot \text{di} . M + \tau \cdot \text{de. } M + \tau \cdot M )
\]

\[
+ \text{AO} . ( \tau \cdot \text{a0} . M + \tau \cdot \text{ae. } M + \tau \cdot M )
\]

\[
+ \text{AI} . ( \tau \cdot \text{ai} . M + \tau \cdot \text{ae. } M + \tau \cdot M )
\]
4) Protocol Entity R:

\[
\begin{align*}
\ll R \rr (\exp) &= R \\
R &= \tau \cdot R_0 \\
R_0 &\leftarrow \text{d0. OUT. A0. R}_I \\
&+ \text{d1. A1. R}_0 \\
&+ \text{de. A1. R}_0 \\
R_I &\leftarrow \text{d1. OUT. A1. R}_0 \\
&+ \text{d0. A0. R}_I \\
&+ \text{de. A0. R}_I 
\end{align*}
\]

To prove that the external behavior of the Alternating Bit Protocol conforms to its intended service, we apply the parallel composition rule on the above CCS expressions derived from S, M, and R. Namely:

\[
\text{ABP} = (S \mid M \mid R) \setminus \{\text{d0, d1, d0, d1, de, A0, A1, a0, a1, ae, tick } \alpha\}
\]

The detailed composition steps are presented in Appendix B. We list the result as follows:

\[
\begin{align*}
\text{ABP} &= \text{GET . W} \\
W &\leftarrow \tau \cdot \text{OUT. (}\tau \cdot \text{ABP + } \tau \cdot \text{TRAP}) + \tau \cdot W \\
\text{TRAP} &\leftarrow \tau \cdot \text{ABP + } \tau \cdot \text{TRAP}
\end{align*}
\]
From the definition of observation equivalence, we can prove by induction (see Appendix C) that:

\[ SERVICE \approx ABP \]

At this point we can conclude that the integrated external behavior, presented by the sending entity S, the transmission medium M, and the receiving entity R, is observation equivalent to its intended service.

It should be noted, however, that the behavior expression \( ABP \) may perform an infinite number of \( r \)-actions due to the terms \( r.\psi \) in (2) and \( r.TRAP \) in (3). By examining the steps of parallel composition in Appendix B, it can be readily found that the presence of \( r.\psi \) is due to the transmission loss or the occurrence of de interaction, i.e., a transmission error has occurred while a message is in transit. Similarly, the presence of \( r.TRAP \) is due to the transmission loss or the occurrence of ae interaction, i.e., a transmission error has occurred while an acknowledgment is in transit. Indeed, when looking at the specification of the transmission medium, one can see that a transmission loss or error can occur nondeterministically, and nothing prevents it from persistently malfunctioning.
In fact, it is easier to see the situation when we draw the transition graphs for both SERVICE (Fig. 7.2) and ABP (Fig. 7.3). Both graphs show that the externally observable behavior is an indefinite repetition of a pair of \texttt{GET;OUT} communications. However, the transition graph of \texttt{ABP} has an $\tau$-loop at $W$, corresponding to the $\tau.W$ term in (2); and an $\tau$-loop at $\texttt{TRAP}$, corresponding to the $\tau.\texttt{TRAP}$ term in (3). The former $\tau$-loop reflects the potential error or loss situation during the transmission from sending entity to receiving entity, while the latter reflects the transmission error or loss in the opposite direction.

As stated in [28], to make the CCS observation equivalent relation sensitive to infinite $\tau$-actions, one has to deal with the issue of fairness. Here, we leave the responsibility of reasoning the infinite $\tau$-actions to the prover. In the case of Alternating Bit Protocol, it should be clear that, under the assumption that the medium cannot permanently corrupt or lose messages, infinite $\tau$-actions can never occur.

\begin{center}
\begin{tikzpicture}
\node[circle, draw] (A) at (0,0) {$\texttt{SERVICE}$};
\node[circle, draw, fill=white, right of=A] (B) at (1,0) {};
\node[circle, draw, below of=A] (C) at (0,-1) {$\texttt{GET}$};
\draw [->] (A) to node [auto,swap] {$\texttt{OUT}$} (B);
\draw [->] (B) to node [auto,swap] {} (C);
\draw [->] (C) to node [auto,swap] {} (A);
\end{tikzpicture}
\end{center}

\textbf{Figure 7.2.} SERVICE Graph
7.2 X.25 Packet Level DTE/DCE Interface

In the previous section, we studied how to perform the conformity analysis using the transformational approach. In this section, we intend to show that, in addition to proving conformity, the transformational approach can also be used to detect syntactic errors of protocols. The X.25 packet level DTE/DCE interface is used as a case study.

The CCITT recommended X.25 is a set of procedures which define the exchange of information between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) in the packet-switched data networks. In X.25, the packet level (level 3) DTE/DCE interface describes the control procedures for the setup of a virtual call between DTE and DCE over an existing physical connection.
In [32], the behaviors of DTE and DCE, defined in the 1976 Geneva version [9], are specified in terms of the state diagrams as shown in Figure 7.4. Starting with the READY state, the DTE or DCE can either send a control packet (arcs labeled with a "-" sign) or receive a control packet (arcs labeled with a "+" sign). The virtual call is established when both DTE and DCE have reached the Data Transfer state. By the perturbation technique (reachability analysis), it has been shown in [32] that an "unspecified reception" error can occur in this version, as illustrated in Figure 7.5. Specifically, when DTE sends a call request packet to the DCE at the same time that the DCE sends a clear indication packet to the DTE, the action taken by the DCE on receipt of the call request packet is not specified in Figure 7.4.
Figure 7.4. X.25 DTE/DCE Interface State Diagrams (1976 version)
Figure 7.5. Unspecified Reception Error
DTE ::
*[-est; M1!CALL-REQUEST() →
  [ M2?call-connected() → est:=true
   
   M2?incoming-call() →
   [ M2?call-connected() → est:=true
   
   M2?clear-indication() → confirm:=false
   *[~confirm; M2?clear-indication() → skip
   
   ~confirm; M1!CLEAR-CONFIRM() → confirm:=true
   ]
   ]
]
]

M1!CLEAR-REQUEST() → confirm:=false
*[~confirm; M1!CLEAR-REQUEST() → skip
 
  ~confirm; M2?call-connected() → skip
  
  ~confirm; M2?clear-indication() → confirm:=true
  
  ~confirm; M2?clear-confirm() → confirm:=true
  ]
]
]

M2?clear-indication() → confirm:=false
*[~confirm; M2?clear-indication() → skip
 
  ~confirm; M1!CLEAR-CONFIRM() → confirm:=true
  ]
]
]

M1!CLEAR-REQUEST() → confirm:=false
*[~confirm; M1!CLEAR-REQUEST() → skip
 
  ~confirm; M2?call-connected() → skip
  
  ~confirm; M2?clear-indication() → confirm:=true
  
  ~confirm; M2?clear-confirm() → confirm:=true
  ]
]
]

Figure 7.6. CSP-based specification of DTE
$\text{est; } \text{M2?incoming-call()} \rightarrow$

$[ \text{M1!CALL-ACCEPTED()} \rightarrow \text{est}:=\text{true}$

$\text{M2?clear-indication()} \rightarrow \text{confirm}:=\text{false}$

$\star \text{[~confirm; M2?clear-indication()} \rightarrow \text{skip}$

$\text{~confirm; M1!CLEAR-CONFIRM()} \rightarrow \text{confirm}:=\text{true}$

$\star \text{M1!CLEAR-REQUEST()} \rightarrow \text{confirm}:=\text{false}$

$\star \text{[~confirm; M1!CLEAR-REQUEST()} \rightarrow \text{skip}$

$\text{~confirm; M2?call-connected()} \rightarrow \text{skip}$

$\text{~confirm; M2?clear-indication()} \rightarrow \text{confirm}:=\text{true}$

$\text{~confirm; M2?clear-confirm()} \rightarrow \text{confirm}:=\text{true}$

$\star \text{~confirm; M2?clear-indication()} \rightarrow \text{skip}$

$\text{~confirm; M1!CLEAR-CONFIRM()} \rightarrow \text{confirm}:=\text{true}$

$\text{~est; M1!CLEAR-REQUEST()} \rightarrow \text{confirm}:=\text{false}$

$\star \text{[~confirm; M1!CLEAR-REQUEST()} \rightarrow \text{skip}$

$\text{~confirm; M2?call-connected()} \rightarrow \text{skip}$

$\text{~confirm; M2?clear-indication()} \rightarrow \text{confirm}:=\text{true}$

$\text{~confirm; M2?clear-confirm()} \rightarrow \text{confirm}:=\text{true}$

$\star \text{~confirm; M2?clear-indication()} \rightarrow \text{skip}$

$\text{~confirm; M1!CLEAR-CONFIRM()} \rightarrow \text{confirm}:=\text{true}$

$\text{~est; M2?clear-indication()} \rightarrow \text{confirm}:=\text{false}$

$\star \text{[~confirm; M2?clear-indication()} \rightarrow \text{skip}$

$\text{~confirm; M1!CLEAR-CONFIRM()} \rightarrow \text{confirm}:=\text{true}$

$\star \text{~confirm; M2?clear-indication()} \rightarrow \text{skip}$

$\text{~confirm; M1!CLEAR-CONFIRM()} \rightarrow \text{confirm}:=\text{true}$

$\text{Figure 7.6. CSP-based specification of DTE (continued)}$
DCE ::
*[-est; M1?call-request() →
   [ M2!CALL-CONNECTED() → est:=true
     [ M1?clear-request() → confirm:=false
       *[-confirm; M1?clear-request() → skip
         [ confirm; M2!CLEAR-CONFIRM() → confirm:=true
       ]
     ]
   ]
M2!CLEAR-INDICATION() → confirm:=false
   *[-confirm; M2!CLEAR-INDICATION() → skip
     [ confirm; M1?call-accepted() → skip
     ]
   [ confirm; M1?clear-request() → confirm:=true
     [ confirm; M1?clear-confirm() → confirm:=true
   ]
   [ confirm; M2!CLEAR-INDICATION() → confirm:=false
     *[-confirm; M2!CLEAR-INDICATION() → skip
       [ confirm; M1?call-accepted() → skip
     ]
     [ confirm; M1?clear-request() → confirm:=true
     ]
     [ confirm; M1?clear-confirm() → confirm:=true
   ]
   [-est; M2!INCOMING-CALL() →
     [ M1?call-accepted() → est:=true
       [ confirm; M1?call-request() →
         [ M2!CALL-CONNECTED() → est:=true
           [ M1?clear-request() → confirm:=false
             *[-confirm; M1?clear-request() → skip
               [ confirm; M2!CLEAR-CONFIRM() → confirm:=true
             ]
           ]
         ]
         [ confirm; M2!CLEAR-INDICATION() → confirm:=false
           *[-confirm; M2!CLEAR-INDICATION() → skip
             [ confirm; M1?call-accepted() → skip
           ]
           [ confirm; M1?clear-request() → confirm:=true
             [ confirm; M1?clear-confirm() → confirm:=true
           ]
         ]
       ]
     ]

Figure 7.7. CSP-based specification of DCE
Figure 7.7. CSP-based specification of DCE (continued)
The CSP-based specification for the same version of the X.25 Packet level DTE/DCE Interface is shown in Figures 7.6 and 7.7. The underlying transmission media between DTE and DCE in each direction are specified in Figure 7.8. By the transformation rules, we can derive the CCS behavior expressions for DTE, DCE and the transmission media as follows:

(a) From DTE to DCE

\[ \begin{align*}
M1 &:: \ast [ \text{DTE}\text{-}\text{CALL-REQUEST}() \rightarrow \text{DCE}\text{-}\text{call-request}() \\
&\quad \land \text{DTE}\text{-}\text{CALL-ACCEPTED}() \rightarrow \text{DCE}\text{-}\text{call-accepted}() \\
&\quad \land \text{DTE}\text{-}\text{CLEAR-REQUEST}() \rightarrow \text{DCE}\text{-}\text{clear-request}() \\
&\quad \land \text{DTE}\text{-}\text{CLEAR-CONFIRM}() \rightarrow \text{DCE}\text{-}\text{clear-confirm}()] \\
\end{align*} \]

(b) From DCE to DTE

\[ \begin{align*}
M2 &:: \ast [ \text{DCE}\text{-}\text{CALL-CONNECTED}() \rightarrow \text{DTE}\text{-}\text{call-connected}() \\
&\quad \land \text{DCE}\text{-}\text{INCOMING-CALL}() \rightarrow \text{DTE}\text{-}\text{incoming-call}() \\
&\quad \land \text{DCE}\text{-}\text{CLEAR-INDICATION}() \rightarrow \text{DTE}\text{-}\text{clear-indication}() \\
&\quad \land \text{DCE}\text{-}\text{CLEAR-CONFIRM}() \rightarrow \text{DTE}\text{-}\text{clear-confirm}()] \\
\end{align*} \]

Figure 7.8. Transmission Media
• Behavior expression of DTE

\[
DTE \leftarrow \text{CALL-REQUEST.} \{ \text{call-connected.Nil} \\
+ \text{incoming-call.} \{ \text{call-connected.Nil} \\
+ \text{clear-indication.} DCE-CLEAR-IND \\
+ \text{CLEAR-REQUEST.} DTE-CLEAR-REQ \\
+ \text{clear-indication.} DCE-CLEAR-IND \\
+ \text{CLEAR-REQUEST.} DTE-CLEAR-REQ \\
+ \text{CLEAR-REQUEST.} DTE-CLEAR-REQ \\
+ \text{clear-indication.} DCE-CLEAR-IND
\]

where

\[
DCE-CLEAR-IND \leftarrow \text{clear-indication.} DCE-CLEAR-IND + \text{CLEAR-CONFIRM.} DTE
\]

\[
DTE-CLEAR-REQ \leftarrow \text{CLEAR-REQUEST.} DTE-CLEAR-REQ + \text{clear-indication.} DTE
+ \text{call-connected.} DTE-CLEAR-REQ + \text{clear-confirm.} DTE
\]
• Behavior expression of DCE

\[ DCE \leftarrow \text{call-request.} \left( \text{CALL-CONNECTED.Nil} \right. \]
\[ + \]
\[ \text{clear-indication. DCE-CLEAR-IND} \]
\[ + \]
\[ \text{CLEAR-REQUEST. DTE-CLEAR-REQ} \right) \]
\[ + \]
\[ \text{INCOMING-CALL.} \left( \text{call-accepted.Nil} \right. \]
\[ + \]
\[ \text{call-request.} \left( \text{CALL-CONNECTED.Nil} \right. \]
\[ + \]
\[ \text{clear-request. DTE-CLEAR-REQ} \]
\[ + \]
\[ \text{CLEAR-REQUEST. DTE-CLEAR-REQ} \]
\[ + \]
\[ \text{CLEAR-INDICATION. DCE-CLEAR-IND} \right) \]
\[ + \]
\[ \text{clear-request. DTE-CLEAR-REQ} \]
\[ + \]
\[ \text{CLEAR-REQUEST. DTE-CLEAR-REQ} \]
\[ + \]
\[ \text{CLEAR-INDICATION. DCE-CLEAR-IND} \]

where

\[ \text{DTE-CLEAR-REQ} \leftarrow \text{clear-request. DTE-CLEAR-REQ} + \text{CLEAR-CONFIRM. DCE} \]

\[ \text{DCE-CLEAR-IND} \leftarrow \text{CLEAR-INDICATION. DCE-CLEAR-IND} + \text{clear-request. DCE} \]
\[ + \text{call-accepted. DCE-CLEAR-IND} + \text{clear-confirm. DCE} \]
• Behavior expressions of M1 and M2

\[
M_1 \Leftarrow \text{CALL-REQUEST. call-request. } M_1 \\
+ \text{CALL-ACCEPTED. call-accepted. } M_1 \\
+ \text{CLEAR-REQUEST. clear-request. } M_1 \\
+ \text{CLEAR-CONFIRM. clear-confirm. } M_1
\]

\[
M_2 \Leftarrow \text{CALL-CONNECTED. call-connected. } M_2 \\
+ \text{INCOMING-CALL. incoming-call. } M_2 \\
+ \text{CLEAR-INDICATION. clear-indication. } M_2 \\
+ \text{CLEAR-CONFIRM. clear-confirm. } M_2
\]

The system behavior can be examined by performing the parallel composition of DTE, DCE, M1, and M2. In the following, we show the portion of the composition steps which lead to the unspecified reception error. Here, "internal" stands for all the communications between DTE, DCE, M1, and M2.

\[
(DTE \mid M_1 \mid M_2 \mid DCE) \text{ \textbackslash internal}
\]

\[
= (\text{CALL-REQUEST... + incoming-call... + CLEAR-REQUEST... + clear-indication}...
\]

\[
| \text{CALL-REQUEST call-request. } M_1 + ...
\]

\[
| \text{CLEAR-INDICATION. clear-indication. } M_2 + ...
\]

\[
| \text{call-request ... + INCOMING-CALL ... + clear-request ... + CLEAR-INDICATION} ...
\]

\text{\textbackslash internal}
The first two steps represent a call request packet sent by DTE and a clear indication packet sent by DCE, respectively. The third step represents the reception of the clear indication packet by DTE. At this point, while the call request packet is on the way from DTE to DCE, the DCE is only expecting either to send a clear indication packet or to receive a clear-request, call-accepted, or clear confirm packet — an unspecified reception error occurs.
By this example, we show that, while the transformational approach is used primarily for functional analysis, it can also detect syntactic errors presented in communication protocols.
CHAPTER VIII

Conclusions

8.1 Summary

The goal of this research was to explore two essential activities regarding the design of communication systems. First, a formal specification technique was studied for modeling both communication protocols and communication services. Second, a technique for functional analysis was proposed to demonstrate that communication protocols can indeed achieve the communication services for which the protocols have been designed.

In Chapter 2, we introduced the concepts of communication protocols and communication services in the context of the Open System Interconnection Model. To clarify their roles in communication systems, we compared them with other computing systems and concluded that they are the realization of two software engineering principles — hierarchical structuring and information hiding. Then, in Chapter 3, we presented a brief overview of the most representative specification and
verification techniques proposed in the past.

In Chapter 4, a CSP-based technique was presented for specifying both communication protocols and services. It has been emphasized that, following the disciplined practices, this technique can be employed as a compromise of the state-oriented approach and the sequence-oriented approach. When specifying protocols, system components are modeled as a set of interacting processes which communicate with each other through message passing. The control status of a protocol entity is reflected by the current control point in the course of execution, whereas the data status is represented as process variables. When specifying services, the service provider is modeled by a set of concurrent processes which can merely interact with the service user processes. Thus, the global constraints and local constraints can be decoupled in an orderly manner. In addition, we extended the CSP by introducing the convention of generalized distributed termination in order to model the disruption behavior of communication systems.

Chapter 5 presented the concept of conformity analysis. Apart from showing the absence of syntactic errors (usually referred as protocol validation) or the satisfaction of certain ad hoc properties (usually referred as protocol verification), the purpose of conformity analysis is to
demonstrate that the functional behavior of protocol entities according to the protocol specification, when working together with the underlying service, can match the requirements stated in the service specification. To perform the conformity analysis based on the CSP specification technique, we first elaborated the reasons by which the conventional axiomatic approach has been ruled out. Then the motivations leading to the proposed transformational approach were explained.

The transformational approach was presented formally in Chapter 6. By a set of transformation rules, the communication behaviors of CSP-based protocol and service specifications can be extracted and expressed in terms of CCS behavior expressions. This approach has two major advantages. First, for any cyclic computation, the derived expression captures its entire communication behavior; thus, the strongest invariant property can be obtained in a systematic way. Second, by virtue of the laws of CCS, the composition of the behaviors of protocol entities and the service provider as well as the hiding of their internal communications can be performed by simple algebraic manipulations. Several examples were used to demonstrate the transformation procedures.

In Chapter 7, we presented a complete conformity analysis via the case study of Alternating Bit Protocol. In addition, the X.25 packet level
DTE/DCE interface was used to illustrate the detection of syntactic errors in the transformational approach.

8.2 Concluding Remarks and Future Work

Considering the rapid evolution of communication networks and the increasing demands on distributed data processing systems, there is no doubt that a solid engineering practice for protocol design will become a dominant factor in the design of future information systems. Due to this recognition, the term Protocol Engineering has been recently used to identify all the aspects involved in the design of communication protocols as a technical domain in its own right.

Over the past several years, while much progress has been made in the improvement of protocol design, protocol engineering is still in its infancy. As we have seen, in the area of software engineering, it took almost two decades to stabilize the fundamental principles of software design, and modern technology is still being developed to meet numerous new challenges. Likewise, in order to further the maturation of protocol engineering, more research and experience are definitely needed. As we stated above, however, the increasing demand for distributed information processing today allows very little time for protocol researchers as well as practitioners to learn. We believe that the most important issue, at
present, is to investigate the protocol design techniques which not only have a sound theoretical basis but are also practical enough to expedite the transition from research laboratories into practices.

To date, many protocol description and analysis techniques have been proposed in the literature. While some of these techniques are very appealing from the theoretical standpoint, the lack of engineering practice has prevented them from being of practical use. On the other hand, it is not unusual to come across many specifications that are still written in natural languages in the real world. It is clear that a great deal of effort has to be made to bridge the gap between theoretical work and common practice — which is the goal we have attempted to pursue in this thesis. We believe the CSP-based specification technique is practical and helpful for both protocol designers and implementors. Its underlying semantics is rigorous and simple. Its representation is concise, understandable, and easy to construct. With regard to protocol analysis, the transformational approach is systematic and intuitively straightforward. The transformation procedures are conducted mechanically and in a syntax-directed manner.

While we have laid a methodology and theoretical basis for the design of communication systems in this study, we feel that further efforts are worthwhile to extend this work in the directions suggested below.
• **Automated tools:**

Automated tools can be developed to facilitate the analysis process. Specifically, a large portion of the transformation from a CSP-based specification to the CCS behavior expressions is the direct application of the transformation rules and thus can be performed automatically. In addition, during the conformity analysis, mechanical steps such as the derivations of the integrated behaviors from separate behavior expressions can be aided by appropriate automated tools.

• **Liveness properties:**

In general, liveness properties assert that certain behaviors will eventually happen. As we have shown in the ABP example, the present solution to ensure the liveness properties lies in the designer's understanding of the protocols. In general, a system's behavior is not live because of two situations — deadlock and infinite computation. In the transformational approach, deadlock situations will be detected when performing behavior integrations. However, since the CCS observation equivalence relation is not sensitive to infinite $\tau$-actions, the designer is responsible for detecting any occurrences of infinite internal computations during the system integration. Further studies can be made in this regard.

• **Performance Prediction and Automated Implementation:**

As we have pointed out, the CSP-based specifications are operational and executable specifications. Therefore, many activities other than the functional analysis can also be performed via machines. Performance prediction [4, 33] is a technique by which the performance of a protocol can be calculated directly by
the execution of its formal specification. In this approach, *time specification*, usually presented in terms of the probabilities and the time which elapses during the execution of certain segments of a protocol, is integrated into the formal specification. By executing a specification, the designer is able to estimate protocol performance prior to actual implementation. Another area which can be aided by machines is automated implementation. In this approach, instead of coding by implementors, a large portion of the implementation can be generated by the direct compilation of a protocol specification, thus saving a great deal of development time and human errors. Both performance prediction and automated implementation are promising research directions in the framework of protocol engineering and we believe it is worthwhile to explore these two areas based on the CSP-based specification technique.
APPENDIX A
Transformation Rules for Distributed Termination

In this appendix, we extend the transformation rules for repetitive command and parallel composition in order to take into account the distributed termination convention. It shall be seen that although the rules are slightly complicated in respect to notations, the underlying idea is still simple and intuitively straightforward. A number of definitions are first given below.

**Definition 1:**

**Pattern** = powerset of \{1, \ldots, n\} - \{\emptyset, \{1\}, \ldots, \{n\}\}

**Meaning:** An element in **Pattern** is any arbitrary subset of \{1, \ldots, n\} with at least two elements. We shall interpret \{1, \ldots, n\} as the indices of processes \(P_1, \ldots, P_n\).

**Definition 2:**

\[\text{Term}_i \leftarrow \overline{\sigma}_{\{i\}} \cdot \text{Term}_i\]

**Meaning:** Term\(_i\) is a behavior expression which keeps sending the \(\sigma_{\{i\}}\) signal. We shall interpret the \(\sigma_{\{i\}}\) signal with the meaning that the process \(P_i\) has already *terminated*. 

143
Definition 3:

\[ \text{Termset}_{patt} = \sigma_{\{i_1\}} \cdots \sigma_{\{i_m\}} \omega_{patt} \]

\[ \omega_{patt} = \sigma_{patt} \cdot \omega_{patt} \]

where \( patt = \{i_1, \ldots, i_m\} \in \text{Pattern} \)

**Meaning:** Termset\(_{patt}\) is a behavior expression which acts as follows:

(i) it first waits for receiving signals \( \sigma_{\{i_1\}} \cdots \sigma_{\{i_m\}} \); (ii) once it has received all these signals, it is ready to send an indefinite number of \( \sigma_{patt} \) signal. Here, the signal \( \sigma_{patt} \) represents the termination of all processes \( P_{i_1}, \ldots, P_{i_m} \).

Definition 4:

For each repetitive command,

\[ [* \{ G_1 \rightarrow C_1 \} \quad G_2 \rightarrow C_2 \quad \cdots \quad G_n \rightarrow C_n ] \]

a). \( \text{PB} = \{ i \mid G_i = b_i \} \)

b). \( \text{IO} = \{ j \mid G_j = b_j; P_j^i ? t(x_i) \text{ or } b_j; P_j^i ! t(e) \} \)

c). \( \text{Addr} : \text{IO} \rightarrow \{1, \ldots, n\} \)

\( \text{Addr}(j) = l \quad \text{if} \quad G_j = b_j; P_j^i ? t(x_i) \text{ or } b_j; P_j^i ! t(e) \)

**Meaning:** For a repetitive command, \( \text{PB} \) contains the indices of all the pure boolean guards; \( \text{IO} \) contains the indices of all the I/O guards. The function \( \text{Addr} \) is a mapping from the index of an I/O guard to the index of a process which is addressed by this I/O guard.
Repetitive Command:

\[ ([ G_1 \rightarrow C_1 \[ G_2 \rightarrow C_2 \[ \cdots \[ G_n \rightarrow C_n ] \] ] ) (\vec{x}) = \omega(\vec{x}) \]

\[ \omega(\vec{x}) = (\sum_i \text{if } b_i(\vec{x}) \text{ then } \tau \cdot [ C_i ] (\vec{x}) \text{ before } \omega(\vec{x}') \text{ else } \text{Nil} \]

where \( G_i = b_i \)

\[ + \sum_j \text{if } b_j(\vec{x}) \text{ then } [ P_i; t(x_j); C_j ] (\vec{x}) \text{ before } \omega(\vec{x}') \text{ else } \text{Nil} \]

where \( G_j = b_j; P_i; t(x_j) \)

\[ + \sum_k \text{if } b_k(\vec{x}) \text{ then } [ P_i; t(e); C_k ] (\vec{x}) \text{ before } \omega(\vec{x}') \text{ else } \text{Nil} \]

where \( G_k = b_k; P_i; t(e) \)

\[ + \text{if } \neg(\forall b_{p_{k_i}}) \text{ then } \]

\[ \text{case } \]

\[ \neg b_{i_{o_1}} \land \neg b_{i_{o_2}} \land \cdots \land \neg b_{i_{o_m}} : [\text{skip}] (\vec{x}) \]

\[ b_{i_{o_1}} \land \neg b_{i_{o_2}} \land \cdots \land \neg b_{i_{o_m}} : \sigma_{\{\text{addr}(i_{o_1})\}} \cdot [\text{skip}] (\vec{x}) \]

\[ \vdots \]

\[ b_{i_{o_1}} \land b_{i_{o_2}} \land \cdots \land b_{i_{o_m}} : \sigma_{\{\text{addr}(i_{o_1}), \ldots, \text{addr}(i_{o_m})\}} \cdot [\text{skip}] (\vec{x}) \]

\[ \text{endcase} \]

\[ \text{else } \text{Nil} \]

where \( p_{k_i} \in \text{PB} \)

\[ i_{o_j} \in \text{IO} \]

The only difference between this rule and the one in Chapter 6 is the last term of \( \omega(\vec{x}) \), which is explained below:
1) If any of the pure boolean guards is true, this term is Nil, i.e., to affect nothing.

2) If all the pure boolean guards are false, and all the I/O guards have their boolean parts being false, this term is \([\text{skip}]\) which means that the execution of this repetitive command has finished.

3) If all the pure boolean guards are false, and some of the I/O guards have their boolean parts being true, then this term is in the form as \(\sigma_{\text{patt}}[\text{skip}]\), where \(\text{patt}\) is the set of the indices of all the processes addressed by those I/O commands.

Intuitively, this rule simply reflects the semantics of distributed termination by requiring that, at any time during the execution a repetitive command, if some of the I/O guards are waiting for communications but all their communication partners have terminated, an “ending” signal \(\sigma\) will be received and the execution can then terminate.

Now, in order to offer the “ending” signals, the parallel composition rule is modified as below:

**Parallel Composition Rule:**

\[
\| P_1 \|\| P_2 \|\| \ldots \|\| P_n \|((x_1, x_2, \ldots, x_n) =
\|
\|
\|
\]
This rule extends the parallel composition rule presented in Chapter 6 by the following modifications:

1) \( \text{Term}_i \) is attached to the end of \( \llbracket P_i \rrbracket(\bar{x}_i) \) such that, after the execution of \( \llbracket P_i \rrbracket(\bar{x}_i) \), the signal \( \sigma_i \) will be constantly available to inform the fact that process \( P_i \) has terminated.

2) \( \text{Termset}_{\text{patt}} \) expressions are included in this rule such that, once all the processes \( P_i \)'s \( (i \in \text{patt}) \) have terminated, \( \text{Termset}_{\text{patt}} \) will send the signal \( \sigma_{\text{patt}} \) to inform this situation.

It can be seen that, if a system exhibits a finite behavior, then the expressions in this rule will finally progress to a situation in which all of them are attempting to send their respective \( \sigma \)-signals. By the CCS definition, \( (\bar{\sigma}_1 | ... | \bar{\sigma}_n | \bar{\sigma}_{\text{patt}}) \{ \sigma_{i} \} \cup \{ \sigma_{\text{patt}} \} = \text{Nil} \), which means that the entire system will terminate properly.

Finally, we should point out that all the extensions in this appendix are developed to deal with the distributed termination of the original CSP. No modifications are necessary for the Generalized Distributed Termination discussed in Chapter 4. The reason is simple; just as the case in which all the ending \( \sigma \)-signals will converge into \text{Nil}, all the "hanging" input or output interactions in the individual expressions will eventually be "truncated" because of the \( \backslash \) (restriction) operation. It should be noted, however, that if there exist any hanging interactions which are not derived from the I/O commands at the outermost level of a CSP process, then a deadlock situation has been detected.
APPENDIX B
Parallel Composition for ABP

This appendix shows the composition of the sending entity, the receiving entity, and the underlying transmission medium in the Alternating Bit Protocol discussed in Chapter 7. As we shall see, the composition steps are conducted in a very mechanical way.

In the ABP, as noted earlier, it is required that the time-out interval is properly set such that the sending entity will be triggered by a timeout signal only after the transmission loss has really occurred. This assumption is pertinent to the outcome of the system behavior and should be reflected in the composition steps. Here, we formalize this assumption by adding the tick output action in the expression $M$ for transmission medium shown in the next page. When considering $S$ and $M$ together, we can see that $S$ will receive a tick signal only when $M$ executes the term $\tau.t\text{ick}.M$, namely, when the loss situation occurs.

- Protocol Entity $S$:
  
  $S = \tau . S_0$
  
  $S_0 = \text{GET} . \tau_0$
  
  $\tau_0 \leftarrow \text{DO} . ( a0. S_1 + a1. \tau_0 + \text{ae. } \tau_0 + \text{tick. } \tau_0 )$
  
  $S_1 = \text{GET} . \tau_1$
  
  $\tau_1 \leftarrow \text{DI} . ( a1. S_0 + a0. \tau_1 + \text{ae. } \tau_1 + \text{tick. } \tau_1)$
• Transmission Medium M:

\[ M \leftarrow \text{DO.} ( r. \overline{\text{do}}. M + r. \overline{\text{de}}. M + r. \overline{\text{tick}}. M ) \]
\[ + \text{Dl.} ( r. \overline{\text{dl}}. M + r. \overline{\text{de}}. M + r. \overline{\text{tick}}. M ) \]
\[ + \text{A0.} ( r. \overline{\text{a0}}. M + r. \overline{\text{ae}}. M + r. \overline{\text{tick}}. M ) \]
\[ + \text{A1.} ( r. \overline{\text{al}}. M + r. \overline{\text{ae}}. M + r. \overline{\text{tick}}. M ) \]

• Protocol Entity R:

\[ R = r \cdot R_0 \]
\[ R_0 \leftarrow \text{d0. OUT. AO. } R_1 \]
\[ + \text{d1. A1. } R_0 \]
\[ + \text{de. A1. } R_0 \]
\[ R_1 \leftarrow \text{d1. OUT. } A1. \ R_0 \]
\[ + \text{d0. AO. } R_1 \]
\[ + \text{de. AO. } R_1 \]

Now, if we define \( ABP \) as the system behavior, we have:

\[ ABP = ( S \mid M \mid R ) \ \{ \text{DO, Dl, d0, d1, de, A0, A1, a0, a1, ae, tick, } \alpha \} \]

Let \( \{ I \} = \{ \text{DO, Dl, d0, d1, de, A0, A1, a0, a1, ae, tick, } \alpha \} \)

Then, by the expansion theorem, the first few steps are:

\[ ABP = ( S \mid M \mid R ) \ \{ I \} = (S_0 \mid M \mid R_0 \ \{ I \} = \text{GET.} (T_0 \mid M \mid R_0 \ \{ I \} \]

In the following, we shall derive the expressions:

\[ (T_i \mid M \mid R_j) \ \{ I \} \quad \text{where } i, j \in \{0,1\} \]

Intuitively, these expressions describe the behaviors when the sending entity is going to Transmit a data block with sequence number \( i \) and the receiving entity is expecting to Receive a data block with sequence number \( j \).
\[(T_0 \mid M \mid R_0)\{1\}
= \tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid \tau.d0.M + \tau.de.M + \tau.tick.M \mid R_0)\{1\}\) / * correct D0 */
+ \tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid \tau.de.M \mid R_0)\{1\}\) / * erroneous D0 */
+ \tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid \text{tick}.M \mid R_0)\{1\}\) / * losing D0 */
\]

\[\tau.\tau.\tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid M \mid \text{OUT}.A0.R_j)\{1\}\}
+ \tau.\tau.\tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid M \mid \text{AL}.R_0)\{1\}\}
+ \tau.\tau.\tau.\{(T_0 \mid M \mid R_0)\{1\}\})
\]

\[\tau.\tau.\tau.\text{OUT} \tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid \tau.a0.M + \tau.ae.M + \tau.tick.M \mid R_j)\{1\}\}
+ \tau.\tau.\tau.\tau.\{(a0.S_I+a1.T_0+ae.T_0+\text{tick}.T_0 \mid \tau.aL.M + \tau.ae.M + \tau.tick.M \mid R_0)\{1\}\}
+ \tau.\tau.\tau.\tau.\{(T_0 \mid M \mid R_0)\{1\}\})
\]

\[\tau.\tau.\tau.\text{OUT} \tau.\{(\tau.GE,T_I \mid M \mid R_j)\{1\}\}) / * correct A0 */
+ \tau.\tau.\tau.\{(T_0 \mid M \mid R_j)\{1\}\}) / * erroneous A0 */
+ \tau.\tau.\tau.\{(T_0 \mid M \mid R_I)\{1\}\}) / * losing A0 */
\]

\[\tau.\tau.\tau.\tau.\tau.\{(T_0 \mid M \mid R_0)\{1\}\}) / * correct A1 */
+ \tau.\tau.\tau.\tau.\{(T_0 \mid M \mid R_0)\{1\}\}) / * erroneous A1 */
+ \tau.\tau.\tau.\tau.\{(T_0 \mid M \mid R_0)\{1\}\}) / * losing A1 */
\]

\[\tau.\tau.\tau.\tau.\tau.\tau.\{(T_0 \mid M \mid R_0)\{1\}\})
\]
\[(T_I | M | R_J)\{I\}\]
\[= \tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | \tau.d1.M+\tau.de.M+\tau.tick.M | R_J)\{I\}\]
\[= \tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | \overline{\text{d1}}.M | R_J)\{I\} /*\text{correct D1}*/ + \tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | \overline{\text{de}}.M | R_J)\{I\} /*\text{erroneous D1}*/ + \tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | \overline{\text{tick}}.M | R_J)\{I\} /*\text{losing D1}*/\]
\[= \tau.\tau.\tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | M | \overline{\text{OUT.AI}}.R_J)\{I\} + \tau.\tau.\tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | M | \overline{\text{AO}}.R_J)\{I\} + \tau.\tau.\tau.(T_I | M | R_J)\{I\}\]
\[= \tau.\tau.\tau.\overline{\text{OUT}}.\tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | \tau.\overline{\text{al}}.M+\tau.\overline{\text{ae}}.M+\tau.\overline{\text{tick}}.M | R_J)\{I\} + \tau.\tau.\tau.\tau.(al.S_0+a0.T_I+ae.T_I+tick.T_I | \tau.\overline{\text{ao}}.M+\tau.\overline{\text{ae}}.M+\tau.\overline{\text{tick}}.M | R_J)\{I\} + \tau.\tau.\tau.\tau.(T_I | M | R_J)\{I\}\]
\[= \tau.\tau.\tau.\overline{\text{OUT}}.\tau.\tau.\tau.(\tau.\text{GET.}(T_0 | M | R_J)\{I\}) /*\text{correct A1}*/ + \tau.\tau.\tau.(T_I | M | R_J)\{I\} /*\text{erroneous A1}*/ + \tau.\tau.\tau.(T_I | M | R_J)\{I\} /*\text{losing A1}*/\]
\[+ \tau.\tau.\tau.\tau.(\tau.\tau.\tau.(T_I | M | R_J)\{I\}) /*\text{correct A0}*/ + \tau.\tau.\tau.\tau.(T_I | M | R_J)\{I\} /*\text{erroneous A0}*/ + \tau.\tau.\tau.\tau.(T_I | M | R_J)\{I\} /*\text{losing A0}*/ + \tau.\tau.\tau.\tau.(T_I | M | R_J)\{I\}\]
\((T_0 \mid M \mid R_1)\) \{I\}

\[
= \tau.(a_0.S_i+al.T_0+ae.T_0+\text{tick}.T_0 \mid \tau.\overline{d0}.M+r.\overline{de}.M+r.\overline{\text{tick}}.M \mid R_1)\{I\}
\]

\[
= \tau.(a_0.S_i+al.T_0+ae.T_0+\text{tick}.T_0 \mid \overline{d0}.M \mid R_1)\{I\} \quad \text{/* correct D0 */}
+ \tau.(a_0.S_i+al.T_0+ae.T_0+\text{tick}.T_0 \mid \overline{de}.M \mid R_1)\{I\} \quad \text{/* erroneous D0 */}
+ \tau.\tau.(a_0.S_i+al.T_0+ae.T_0+\text{tick}.T_0 \mid \overline{\text{tick}}.M \mid R_1)\{I\} \quad \text{/* losing D0 */}
\]

\[
= \tau.\tau.\tau.(a_0.S_i+al.T_0+ae.T_0+\text{tick}.T_0 \mid M \mid \overline{A0}.R_1)\{I\}
+ \tau.\tau.\tau.(T_0 \mid M \mid R_1)\{I\}
+ \tau.\tau.\tau.(T_0 \mid M \mid R_1)\{I\}
\]

\[
= \tau.\tau.\tau.(a_0.S_i+al.T_0+ae.T_0+\text{tick}.T_0 \mid \tau.\overline{a0}.M+r.\overline{ae}.M+r.\overline{\text{tick}}.M \mid R_1)\{I\}
+ \tau.\tau.\tau.(T_0 \mid M \mid R_1)\{I\}
\]

\[
= \tau.\tau.\tau.\tau.(T_i \mid M \mid R_1)\{I\} \quad \text{/* correct A0 */}
+ \tau.(T_0 \mid M \mid R_1)\{I\} \quad \text{/* erroneous A0 */}
+ \tau.(T_0 \mid M \mid R_1)\{I\} \quad \text{/* losing A0 */}
+ \tau.(T_0 \mid M \mid R_1)\{I\}
\]
\[(\mathcal{T}_I \mid M \mid R_0) \setminus \{I\}\]

\[= \tau. (a_1.S_0 + a_0.T_I + a.\mathcal{T}_I + tick.T_I \mid \tau.d\overline{1}.M + \tau.\overline{de}.M + \tau.tick.M \mid R_0) \setminus \{I\}\]  

\[= \tau. (\tau. (a_1.S_0 + a_0.T_I + a.\mathcal{T}_I + tick.T_I \mid \overline{d1}.M \mid R_0) \setminus \{I\} \quad \text{/* correct D1 */}\]

\[+ \tau. (a_1.S_0 + a_0.T_I + a.\mathcal{T}_I + tick.T_I \mid \overline{de}.M \mid R_0) \setminus \{I\} \quad \text{/* erroneous D1 */}\]

\[+ \tau. (a_1.S_0 + a_0.T_I + a.\mathcal{T}_I + tick.T_I \mid \overline{tick}.M \mid R_0) \setminus \{I\} \quad \text{/* losing D1 */}\]

\[= \tau.\tau.\tau. (a_1.S_0 + a_0.T_I + a.\mathcal{T}_I + tick.T_I \mid M \mid \overline{Al}.R_0) \setminus \{I\}\]

\[+ \tau.\tau.\tau. (a_1.S_0 + a_0.T_I + a.\mathcal{T}_I + tick.T_I \mid M \mid \overline{Al}.R_0) \setminus \{I\}\]

\[+ \tau.\tau.\tau. [\mathcal{T}_I \mid M \mid R_0] \setminus \{I\}\]

\[= \tau.\tau.\tau.\tau. (\tau. (S_0 \mid M \mid R_0) \setminus \{I\} \quad \text{/* correct A1 */}\]

\[+ \tau. (\mathcal{T}_I \mid M \mid R_0) \setminus \{I\} \quad \text{/* erroneous A1 */}\]

\[+ \tau. (\mathcal{T}_I \mid M \mid R_0) \setminus \{I\} \quad \text{/* losing A1 */}\]

\[+ \tau.\tau.\tau. [\mathcal{T}_I \mid M \mid R_0] \setminus \{I\}\]
To simplify the expressions by applying the laws of CCS, we can summarize the results as follows:

\[
\begin{align*}
(T_0 \mid M \mid R_0) \{ I \} &= \tau. \text{OUT.} \ (\tau. \text{GET.} \ (T_1 \mid M \mid R_1) \{ I \}) \\
+ \tau. \ (T_0 \mid M \mid R_1) \{ I \} \\
+ \tau. \ (T_0 \mid M \mid R_0) \{ I \}
\end{align*}
\]

\[
\begin{align*}
(T_1 \mid M \mid R_1) \{ I \} &= \tau. \text{OUT.} \ (\tau. \text{GET.} \ (T_0 \mid M \mid R_0) \{ I \}) \\
+ \tau. \ (T_1 \mid M \mid R_0) \{ I \} \\
+ \tau. \ (T_1 \mid M \mid R_1) \{ I \}
\end{align*}
\]

\[
\begin{align*}
(T_0 \mid M \mid R_1) \{ I \} &= \tau. \ (S_1 \mid M \mid R_1) \{ I \} \\
+ \tau. \ (T_0 \mid M \mid R_1) \{ I \}
\end{align*}
\]

\[
\begin{align*}
(T_1 \mid M \mid R_0) \{ I \} &= \tau. \ (S_0 \mid M \mid R_0) \{ I \} \\
+ \tau. \ (T_1 \mid M \mid R_0) \{ I \}
\end{align*}
\]

The first two expressions above exhibit an identical behavior, and the last two expressions exhibit an identical behavior.

Let \( W = (T_0 \mid M \mid R_0) \{ I \} = (T_1 \mid M \mid R_1) \{ I \} \)

\[ TRAP = (T_0 \mid M \mid R_1) \{ I \} = (T_1 \mid M \mid R_0) \{ I \} \]
and since $ABP = (S_0 \mid M \mid R_0) \setminus \{1\} = \text{GET.} (T_0 \mid M \mid R_0) \setminus \{1\} = \text{GET.} (T_I \mid M \mid R_I) \setminus \{1\} = (S_I \mid M \mid R_I) \setminus \{1\}$

We have the final result:

\[
ABP = \text{GET.} \ W
\]

\[
W \leftarrow \tau. \text{OUT.} \ (\tau. ABP + \tau. \text{TRAP}) + \tau. \ W
\]

\[
\text{TRAP} \leftarrow \tau. ABP + \tau. \text{TRAP}
\]
This appendix presents the equivalence proof for the Alternating Bit Protocol, namely, SERVICE $\approx$ ABP, where SERVICE represents the behavior of the intended service for which ABP has been designed, while ABP represents the externally observable behavior actually exhibited by the ABP communication system. First, we have a lemma:

**Lemma 1:** Suppose $A$ and $B$ are identifiers of behavior expressions; if $A \preceq^r B + r. A$ then $A \approx B$

**Proof:**
We prove by induction on $k$, where $k$ is the depth of the equivalence relation given in the definition of observation equivalence (see page 85).

First, it is trivial that $A \approx_0 B$.

Next, we perform the induction step: $A \approx_k B$ implies that $A \approx_{k+1} B$
Recall that, by the definition of observation equivalence,

\[ A \approx_{k+1} B \iff \forall s \in \Lambda^* \]

(i) if \( A \xrightarrow{S} A' \), then for some \( B' \), \( B \xrightarrow{S} B' \) and \( A' \approx_k B' \)

(ii) if \( B \xrightarrow{S} B' \), then for some \( A' \), \( A \xrightarrow{S} A' \) and \( A' \approx_k B' \)

Case (i): Let \( A \xrightarrow{S} A' \)

if \( s = \epsilon \) and

(1) \( A' = A \), then \( B \xrightarrow{\xi} B \) and \( A \approx_k B \), as required

(2) \( A' = B \), then \( B \xrightarrow{\xi} B \) and \( B \approx_k B \), as required

(3) \( A' = \tau.B + \tau.A \),

Since \( A \leftarrow \tau.B + \tau.A \), and by the \text{Idem} Law of CCS (page 76 in [28]), we have:

\[ A \equiv \tau.B + \tau.A \], which implies \( A \approx \tau.B + \tau.A \).

Also, since \( A \approx_k B \), we have

\( \tau.B + \tau.A \approx_k B \) as required.

if \( s \neq \epsilon \), then this can only be due to

\[ A \xrightarrow{\tau} B \xrightarrow{S} A' \], but then

\( B \xrightarrow{S} A' \), and \( A' \approx_k A' \) as required.

Case (ii): Let \( B \xrightarrow{S} B' \) then we have \( A \xrightarrow{\tau} B \xrightarrow{S} A' \), for any \( s \).

This completes the induction proof, yielding \( A \approx B \).
Now, we are ready to prove:

\[ \text{SERVICE} \approx \text{ABP} \]

where

\[ \text{SERVICE} \leftarrow \text{GET. OUT. SERVICE} \]

and

\[ \text{ABP} = \text{GET. } W \]

\[ W \leftarrow \tau. \text{OUT.} (\tau. \text{ABP} + \tau. \text{TRAP}) + \tau. W \]

\[ \text{TRAP} \leftarrow \tau. \text{ABP} + \tau. \text{TRAP} \]

**Proof:**

By \[ W \leftarrow \tau. \text{OUT.} (\tau. \text{ABP} + \tau. \text{TRAP}) + \tau. W \] and Lemma 1, we have \[ W \approx \text{OUT.} (\tau. \text{ABP} + \tau. \text{TRAP}) \]

By \[ \text{TRAP} \leftarrow \tau. \text{ABP} + \tau. \text{TRAP} \] and Lemma 1, we have \[ \text{TRAP} \approx \text{ABP} \]

Therefore,

\[ \text{ABP} = \text{GET. } W \]

\[ \equiv \text{GET. OUT.} (\tau. \text{ABP} + \tau. \text{TRAP}) \]

\[ \equiv \text{GET. OUT.} (\tau. \text{ABP}) \]

\[ \equiv \text{GET. OUT.} \text{ABP} \]

Also, by the \text{Idem} Law and \[ \text{SERVICE} \leftarrow \text{GET. OUT. SERVICE} \], we have \[ \text{SERVICE} \equiv \text{GET. OUT. SERVICE} \]

This completes the proof, yielding \[ \text{SERVICE} \approx \text{ABP} \]
REFERENCES

"Concepts and Notations for Concurrent Programming".  

"A Proof System for Communicating Sequential Processes".  

"A Static Analysis of CSP Programs".  

"Protocol Performance Prediction".  

"A Unified Model for the Specification and Verification of Protocols".  

[6] Bochmann, G.V.  
"Finite State Description of Communication Protocols".  

[7] Bochmann, G. V.  
"Logical Verification and Implementation of Protocols".  
Proceedings of the Fourth Data Communications Symposium, pp. 8.5-8.20, Quebec, Canada, 1975.
[8] Bochmann, G. V.
"A General Transition Model for Protocols and Communication Services".


"The OSI Reference Model".

"Modeling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models".

[12] Dijkstra, E. W.
"The Structure of the THE-Multiprogramming System".

"Communicating Sequential Processes for Centralized and Distributed Operating System Design".

[14] Francez, N.
"Distributed Termination".

"A First Attempt at Translating CSP into CCS".

[16] Hoare, C.A.R.
"An Axiomatic Basis for Computer Programming".

"Communicating Sequential Processes".
*Communications ACM* 21(8):666-677, August 1978.
[18] Holzmann, G. J.
"A Theory for Protocol Validation".

[19] ISO.
"Information Processing Systems - Open Systems Interconnection -
Basic Reference Model".

"Concepts for Describing the OSI Architecture".

"Information Processing Systems - Open Systems Interconnection -
Transport Service Definition".
ISO/DIS 8072, 1983.

[22] ISO.
Information Processing Systems - Open Systems Interconnection -
ESTELLE - A Formal Description Technique based on an

[23] ISO.
Information Processing Systems - Open Systems Interconnection -
LOTOS - A Formal Description Technique based on the Temporal

"Verification of a Class of Link-Level Protocols".

[25] Levin, G. M., and Gries, D.
"A Proof Technique for Communicating Sequential Processes".

[26] Logrippo, L.
"Constructive and Executable Specifications of Protocol Services
by using Abstract Data Types and Finite State Transducers".
[27] Menache, M. and Berthomeiu B.
"Time Petri Nets for Analyzing and Verifying Time Dependent
Communication Protocols".

[28] Milner, R.
*A Calculus of Communicating Systems.*
Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[29] Ossenfort, M.
"Correctness Proofs of Communicating Processes: Three Illustrative
Examples from the Literature".

"The Modular Structure of Complex Systems".

[31] Partsch H. and Steinbruggen R.
"Program Transformation Systems".

[32] Rudin, H.
"Automated Protocol Validation: Some Practical Examples".

[33] Rudin, H.
"From Formal Protocol Specification Towards Automated Perform­
ance Prediction".

[34] Rudin, H.
"An Informal Overview of Formal Protocol Specification".

[35] Shaw, M.
"Abstraction Techniques in Modern Programming Languages".

[36] Soundararajan, N.
"Correctness Proofs of CSP Programs".
[37] Soundararajan, N.
"Axiomatic Semantics of Communicating Sequential Processes".

[38] Stenning, N. V.
"A Data Transfer Protocol".

[39] Symons, F.J.W.
"Introduction to Numerical Petri Nets - A General Graphical Model of Concurrent Processing Systems".

[40] Teng, A. Y. and M. T. Liu.
"A Formal Approach to the Design and Implementation of Network Communication Protocol".

[41] Vissers, C. A., Tenney, R. L., and Bochmann, G. V.
"Formal Description Techniques".
*Proceedings of the IEEE* 71(12):1356-1364, December 1983.

[42] Watson, R. W.
"An Experience in Transport Protocol Specification using an Algorithmic Programming Language".

"General Technique for Communications Protocol Validation".
1978
*IBM J. Res. Develop* 22(4), 393-404.