A Methodology for Specification-Based Performance Analysis of Protocols

Nihal Muhammad Nounou

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Science

COLUMBIA UNIVERSITY
1986

To my parents

Aida Bughdady and Muhammad Nounou

with love

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

I would like to express my sincere thanks and gratitude to all the people who contributed to the completion of this work. I thank my advisor, Yechiam Yemini, for his constant guidance and advice. He introduced me to communication protocols and performance analysis, and provided valuable criticism and comments during the course of my research. I also wish to thank the other members of my thesis committee, Gerard Holzmann, Gail Kaiser, Gerald Maguire, Mischa Schwartz and Henryk Wozniakowski, for their interest in this work. I am fortunate to have had such a distinguished committee. Special thanks are due in particular to Professors Leitner, Maguire, and Schwartz for many helpful discussions and suggestions while this work was in progress.

Several members of the Computer Science Department offered invaluable support and friendship that helped greatly during the course of my graduate work; to all of them I express my gratitude. I am especially thankful to Naser Barghouti, Abby Burton, Andrea Danyluk, David Glaser, and John Ment.

I dedicate this work to my parents who always encouraged me to seek the joy of learning and who provided me with constant emotional support. Last, but certainly not least, I owe an immeasurable amount of thanks to my husband, Hussein Ibrahim, whose understanding, tolerance, and encouragement are beyond words. His support, both professional and emotional, contributed greatly to this work. He helped me in revising earlier drafts of this dissertation and in preparing the figures. I probably have spent much time working when we could have been together, and I look forward to correcting this in the future.

Chapter 1

Introduction

The design of communicating distributed processes involves a rather complex set of problems. The processes are allowed to concurrently access shared resources and to proceed asynchronously; they may be executed by heterogeneous processors; their communication channels incur random message delays and are often unreliable; and their behavior is time-dependent. *Protocols*, constituting a set of rules, are thus required to regulate the communication between distributed processes in a computer network.

The first step in designing a protocol is to formally specify the behavior of the various communicating processes involved in it. The concurrent behavior of the protocol can be then computed by following the processes at every state and shuffling the events that can occur at this state, and having a rendezvous event for each corresponding send and receive events. This concurrent behavior then has to be examined to ensure that it satisfies the functional and performance objectives of the protocol. Even for a simple protocol with few communicating processes, the concurrent behavior typically includes too many possibilities to be analyzed manually. Consequently, there has been great demand for automated development tools to aid the protocol designer in verifying the functional requirements and analyzing the performance of such concurrent behaviors.

In this research, we address some of the issues involved in the design of automated protocol development tools. In particular, a methodology for formally specifying and automatically analyzing timing requirements and performance measures of protocols based on their formal specifications is developed. This combination is natural since both problems require a timing model of protocols. The methodology is implemented in ANALYST: an interactive software development environment which is used to automatically analyze the performance of sample protocols.

In the rest of this chapter, we first discuss the nature of protocols and their development. The motivations for this research are described in section 1.2, and the central issues related to performance analysis of protocols are examined in section 1.3. The major contributions of this work are then stated in section 1.4, and in section 1.5 the organization of subsequent chapters is outlined.

1.1. Protocols and Their Development

The International Standards Organization (ISO) has proposed a reference model of protocol architecture [Zimm 80]. The model has seven hierarchical layers as illustrated in Fig. 1-1. Protocols at layers 1 through 4 are referred to as *low-level protocols* whereas those at layers 5 through 7 are referred to as *high-level protocols*.

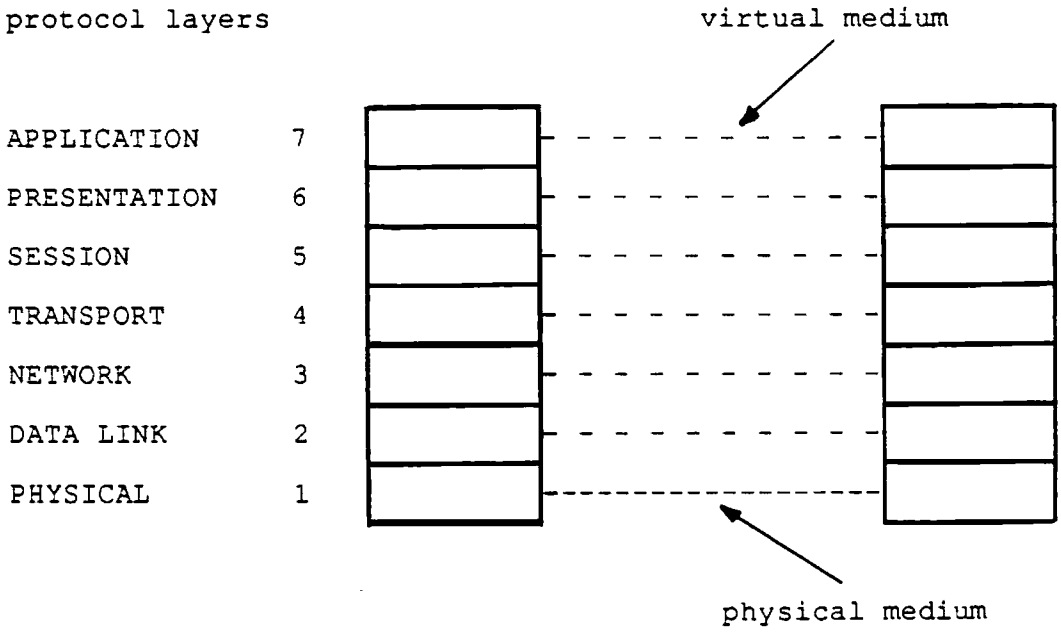


Figure 1-1: Illustration of protocol layers

The purpose of each protocol layer is to provide *services* to the layers above while concealing the details of the layers below. A description of these services including the service interaction

primitives, their possible orders, and their possible parameter values, is referred to as the layer's *service specification.* In addition to the service provided by a protocol, a protocol designer is also concerned with the internal structure and operation of the layer's black box which is illustrated in Fig. 1-2. In this figure each protocol process resides typically at a different site and communicates with other peer processes according to the protocol rules. These rules describe how the processes respond to commands from the upper layer, messages from other peer processes (through the lower layer), and internally initiated actions (e.g., time-outs). They are referred to as a *protocol specification.* Finally, the protocol specification refined into actual code that describes aspects of internal behavior related to inter-process communication and detailed external behavior of each protocol process is referred to as *protocol implementation.* This successive refinement of protocols indicates that they develop in three main phases: *service statement, protocol design,* and *implementation.*

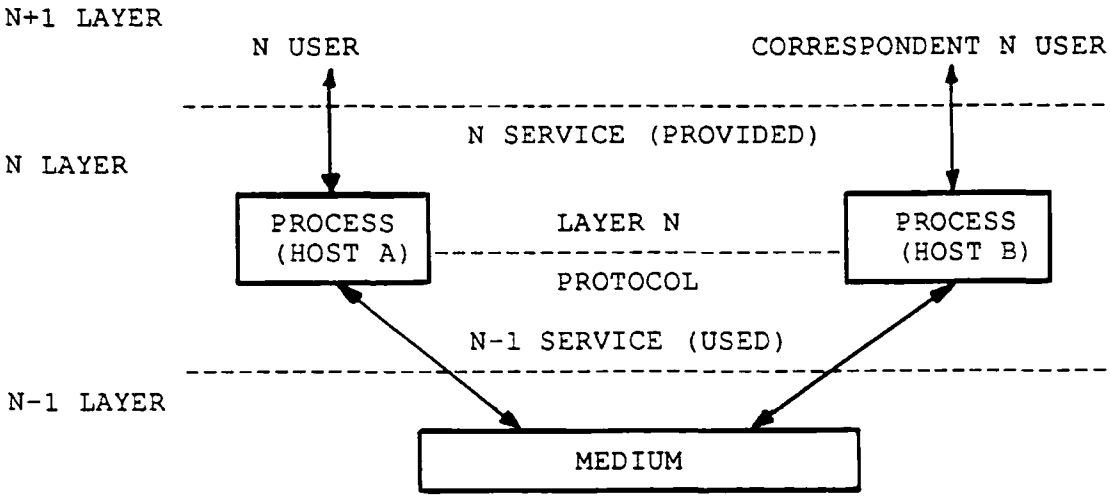


Figure 1-2: A local view of a protocol layer

Development tools are required to support the evolution of protocols from specifications into working systems. Protocol development tools can be classified into *construction tools* for developing and refining protocol specifications, and *validation tools* for assessing how a specification meets its functional (e.g., deadlock freedom) and performance (e.g., maximizing throughput) design objectives [Noun 85]. Functional requirements of a protocol assert that its

behavior is *safe*, i.e., any goal achieved actually satisfies its functional objectives, and *live*, i.e., such goals are guaranteed to be eventually achieved. That is, verifying functional requirements of protocols is concerned with assessing qualitatively their functional behavior.

On the other hand, analyzing the performance of protocols is concerned with assessing quantitatively their timing behavior. More specifically, we view performance analysis of protocols as consisting of 1) analyzing the requirements to be met by its timing behavior (*timing requirements*) to ensure that the protocol performs efficiently, and 2) analyzing key measures that indicate how efficiently the protocol performs (*performance measures*). For example, consider a protocol using time-out to recover from message loss in the transmission medium. Such a protocol performs efficiently if it has a minimal probability of time-out occurring before a loss and a minimal time between a loss and a time-out. (As will be shown later in chapter 6, these two timing requirements are contradictory and some balanced requirement can be defined and analyzed.) One key measure of the efficiency of this protocol is the mean time starting with the sender sending a message until it receives its successful acknowledgment.

1.2. Motivations of This Research

Considerable attention has been given in the past decade to the development of automated tools for verifying functional requirements of protocols [Suns 83]. Works on performance analysis of protocols, however, have not addressed the issue of analyzing their timing requirements and have used manual analyses of their performance measures (see for instance [Tows 79, Bux 82]). The specification and analysis of protocol timing requirements are important to optimally set the protocol parameters, such as the time-out rate in the protocol example discussed above. Moreover, without specifying and analyzing timing requirements, verifying safety properties may be unnecessarily complicated and verifying liveness properties may not be enough. Verifying safety properties may be unnecessarily complicated by considering event sequences in the concurrent behavior that would have a negligible probability if the protocol parameters were optimally set. Verifying liveness properties may not be enough since the eventual goals may take an infinite time to achieve if the protocol parameters are not properly set. This indeed has been shown for such retransmission on time-out protocols [Yemi 82].

Past analyses of protocol performance have been performed *manually* by deducing the timing

behavior of the possible event sequences of the protocol from a human understanding of its operation (see for example [Gele 78, Tows 79, Bux 80, Bux 82]). Such analyses are protocol-dependent, i.e., a completely new analysis is required for every new protocol to be examined, and cannot be integrated with other tools in a protocol development environment. In this research, we address the specification and analysis of timing requirements and performance measures of protocols. Such a combination is in fact natural since both problems require a model of the timing behavior of protocols. It also provides the protocol designer with a complete view of the performance of the protocol under analysis.

Instead of following the *manual approach* we propose a *specification-based approach* where the protocol timing behavior is extracted from its formal specification augmented with timing information. The main advantage of this approach is that it utilizes the formal specification of the protocol, which is required anyhow for verifying its functional requirements, and thus is protocol-independent and can be readily automated.

1.3. Problem Statement

The key problems to be addressed in this research can be divided into three main categories:

1. Develop a methodology for performance analysis of protocol based on their formal specification. This involves the following tasks:
    i. formally specify protocol processes and their communication,
    ii. compute the concurrent behavior of the protocol and detect any progress errors,
    iii. develop a model of the timing behavior of the protocol,
    iv. devise rules for mapping the formal specification of the protocol and probability distributions of its event times into attributes of its timing model, and
    v. demonstrate how timing requirements and performance measures can be formally specified in terms of these attributes, and how the rules can be used to automatically analyze them.
2. Implement this methodology in a protocol development environment.
3. Use the developed software environment in automatically analyzing the performance of several protocols, both low-level and high-level, to demonstrate its wide applicability.

1.4. Contributions of This Research

Contributions of this research fall into three main categories.

1. *The development of a methodology that supports formal specification and automatic analysis of two aspects of protocol performance: timing requirements and performance measures.* Rules that map an algebraic specification of a protocol, and the exponential rates of its events times, to probability and time attributes of its timing behavior are devised. Timing requirements and performance measures of a protocol that can be formally specified in terms of attributes of its timing model are thus automatically analyzed. The analysis of timing requirements yields optimal settings of the protocol's performance parameters, whereas the analysis of its performance measures provides an assessment of the efficiency of its performance.

2. *The design and development of ANALYST: a software environment that supports automated performance analysis of protocols.* Compared to current protocol development environments, see for instance [Holz 84, Chow 85], the design of ANALYST is novel in two main respects. First, it integrates functional and performance specification and analysis of protocols. Since protocol performance is extracted automatically from its functional specification, this integration allows a protocol designer to analyze protocol performance without requiring much expertise in the field. More specifically, a protocol designer is not required to engage in performance modeling of the protocol under analysis, but just to specify performance in terms of timing attributes of the protocol. Second, it facilitates and enhances the design process of protocols. It supports an interactive user interface that allows the protocol designer to readily debug a protocol and iterate through functional and performance specification and analysis thus facilitating experimental protocol design. It also provides the designer with a friendly and uniform user interface to the different modules that perform functional and performance analysis, i.e., the user does not have to explicitly switch from one module to the other to obtain different services.

3. *The automated derivation of performance analysis and optimum timing of a simple connection establishment protocol, the Alternating Bit protocol, and a two phase locking protocol.* In the case of the connection establishment protocol, an upper bound on the rate of terminating connections is computed in order to limit the probability of unsynchronized operation of the connecting parties, and the probability of call collisions is analyzed. A cycle time performance measure for the Alternating Bit protocol that captures a well-known timing error related to the protocol's time-out rate is specified and analyzed. An optimal time-out rate of a simplified version of the protocol is computed, and its maximum throughput and mean delay are analyzed producing results that agreed remarkably well with those obtained manually by other researchers. An automated performance analysis of the two phase locking protocol demonstrates that time-outs may be an alternative to elaborate checks for detecting deadlocks. An optimal setting of the time-out rate is computed, and the protocol's probability of deadlock and mean response time are analyzed.

1.5. The Organization of Subsequent Chapters

The rest of this dissertation is divided into four parts. The first part, chapter 2, is a survey of related works. The survey covers specification tools, verification tools, and performance analysis tools for protocols. The underlying issues and various approaches are examined for each.

The second part, including chapters 3, 4, and 5, presents the methodology of specification-based performance analysis of protocols. The first step of the methodology: functional specification and analysis, is described in chapter 3. A specification algebra is introduced and its use in specifying the functional behavior of protocols is demonstrated. The algebra supports the computation of the concurrent behavior of a protocol which is then analyzed to detect any progress errors. Three functions on protocol behaviors that are used to isolate sub-behaviors of a protocol which a protocol designer is particular interested in for functional and performance analysis, are defined. In chapter 4, the second step of the methodology: performance specification and analysis, is described. A timing model of protocols is presented and some of its key attributes are defined. The use of these attributes to formally specify timing requirements and performance measures of protocols is then demonstrated. In chapter 5, ANALYST: a software development environment that implements the methodology, is described. The functions supported by ANALYST are examined, and a sample scenario for using it for automated performance analysis of protocols is presented. Also, the various elements of its logical architecture are described. Throughout this second part, a simple connection establishment protocol is used to demonstrate the methodology.

The third part, including chapter 6 and 7, describes the application of the proposed methodology to two other protocols. In chapter 6, the Alternating Bit protocol, which provides simple data transfer functions, is considered. In chapter 7, a two phase locking protocol used for concurrency control in data base systems is considered. These protocols together with the simple connection establishment protocol examined in chapters 3, 4, and 5, are chosen to provide a spectrum of different protocol functions. The connection establishment protocol and Alternating Bit protocol are low-level protocols concerned with connection establishment and data transfer functions, respectively. These two functions are common among low-level protocols which are concerned primarily with efficient communications over unreliable transmission channels. The two phase locking protocol is an example of high-level protocols whose functions, on the other hand, are

diverse. ANALYST is used to automatically analyze the performance of these protocols.

Finally, the fourth part, including chapter 8, presents some concluding remarks. A summary of this work is given, and limitations of the proposed methodology and its applicability to performance analysis of protocols are discussed. Directions for future research are then outlined.

Part I

Survey of Related Work

Chapter 2

Development Tools for Communication Protocols

2.1. Introduction

In this chapter we survey related works on protocol development tools that are involved in the methodology proposed in this dissertation. The survey covers specification tools, verification tools, and performance analysis tools for protocols. The underlying issues and various approaches are examined for each.

The protocols considered are those that are implemented as software systems, which belong mostly to layers 2 and above in the ISO protocol hierarchy. Although development tools for general software systems have been studied extensively (see for instance [Lond 80, Ridd 80, Wass 81], their application to protocols is not straightforward. Protocols involve processes that are distributed, concurrent, asynchronous, and whose behavior is often time-dependent. This communication nature of protocols becomes the prime concern underlying the tools. An important objective of protocol verification tools, for example, is to assure robustness of the communication between the protocol processes.

Throughout this chapter, a simple send-and-wait protocolwill be used as an example. The basic function of the protocol is to provide robust message transfer between a source process C and a destination process D over an unreliable transmission channel. There are three distributed processes involved in the protocol: a sender S, a receiver R, and a transmission channel M. The operation of the protocol is as follows. If the sender is idle and receives a new message m from a source C, it sends it to the receiver through the channel which either delivers or loses it. The sender waits for an acknowledgment a to arrive, upon which it again waits for a new message from the source. A new message arriving at the sender that is busy waiting for the acknowledgment of the previous message, is queued. To recover from cases of message loss, if the sender does not receive an acknowledgment after a time-out period T, it retransmits the same

message and then waits again for either an acknowledgment or a time-out.

The receiver process waits for the new message m to arrive from the channel, after which it delivers it to a destination D and then sends an acknowledgment a to the sender through the channel. For the sake of simplicity, it is assumed that the channel does not lose acknowledgments, and that the time-out period is ideally set such that the probability that a time-out occurs only after a message is lost is equal to 1. If the sender and receiver processes are at one protocol layer N, then the source and destination processes would be at the next higher layer N+1 representing the user of the services of the layer N, and the channel process represents the next lower layer N-1.

It should be noted that this is not the most efficient data transfer protocol. For example, in order to make full use of the channel's bandwidth, a more sophisticated protocol would send several messages successively instead of one at a time. In this case it is necessary to assign sequence numbers to messages in order to differentiate between them.

2.2. Specification Tools

Specification tools are construction tools required to describe a protocol at each of its three development phases as a service specification, protocol specification, and protocol implementation. Programming languages are used for describing implementation specifications. These will not be discussed here; throughout the rest of this chapter we limit our discussion to specification tools required for the service statement and protocol design phases.

Experience has shown that protocols specified informally are error-prone even when augmented with some graphical illustrations. For example, 21 errors have been found [West 78a] in the informal specification of the X.21 protocol [X.21 76] (a protocol at layer 2 in Fig. 1-1); they are generally due to the ambiguity and incompleteness of the informal specifications. Formal specifications, on the other hand, are concise, clear, complete, unambiguous, and often used as the basis for other protocol development tools. Protocol development tools are indeed highly dependent on the specification tool used. For example, a different verification tool may be required if the specification tool used in the protocol environment is changed.

In the following section, requirements of specification tools for protocols are outlined. The

various specification tools are surveyed in section 2.2.2, and a taxonomy of the these tools is proposed in section 2.2.3.

2.2.1. Requirements of Specification Tools for Protocols

For a specification tool to adequately describe protocols, it should support the following:

1. Abstract descriptions such that implementation-dependent parts can be left unspecified.
2. Modeling of concurrency.
3. Modeling of nondeterminism, which is a behavior exhibited typically by protocols (e.g., the sender is waiting for either the arrival of an acknowledgment or time-out in the send-and-wait protocol example).
4. Description of the two categories of functions involved in protocols: *control functions*which involves connection initialization and inter-process synchronization, and *data transfer functions* which involves processing of messages texts and related issues such as message sequence numbering.
5. Modular descriptions to facilitate readability and ease of use of specifications.
6. Features to facilitate the application of other protocol development tools. For example, to facilitate performance analysis tools based on formal specifications, a specification method has to be augmented with some timing information.

The extent to which a specification tool exhibits these requirements will be examined next while surveying these tools.

2.2.2. Survey of Specification Tools

2.2.2.1. Finite State Machines

A finite state machine (FSM) consists of the following components: 1) finite set of states, 2) finite set of input commands, 3) transition functions (command$\times$state $\rightarrow$state), and 4) an initial state. A FSM is a natural choice for describing protocol processes whose behavior consist primarily of simple processing in response to commands to or from peer processes in the same layer, and/or the upper and lower protocol layers. A FSM responds to a command according to the input and its current state representing the history of past commands. FSMs have been used in early work on specification of protocols [Bart 69, Suns 75].

Consider using FSMs to describe a protocol specification. Each local process involved in the protocol can then be modeled as a FSM. The behavior resulting from the concurrent execution of

these local processes can be obtained by considering all possible shufflings of the executions of these processes. It is in effect a global description of the operation of the protocol layer. To describe the mode of communication between the distributed processes, three approaches are possible. The simplest assumes that the distributed processes communicate synchronously through *rendezvous interactions* (also referred to as *direct coupling* by Bochmann [Boch 78]). That is, the process issuing a send event should wait for the destination process to issue a corresponding receive event (and vice versa) at which time a rendezvous is said to occur and message exchange takes place. Since messages are not queued in this approach, no modeling of channels between the processes is required. This approach is too restrictive for protocols in which the communicating processes operate asynchronously, or for protocols in which the behavior of the transmission channel is integral to its operation. In the second approach, channels are modeled *implicitly* by specifying their characteristics such as queueing policy (e.g., FIFO) and bound on the number of messages allowed in transit at any one time. Protocols with a number of messages in transit can thus be modeled using this approach. The FSMs specifications in this approach are referred to as *communicating finite state machines* [West 78a, Goud 84a]. In the third approach, channels behavior are specified explicitly as FSMs in which case only channels with a low bound on the number of messages can be feasibly assumed. Even then their FSM specifications are considerably more complex than in the second approach. Their advantage lies in the explicit modeling of channel events such as loss, which can be used in stating functional and timing requirements of protocols.

Following the latter approach, specifications of the three communicating processes in the send-and-wait protocol are shown in Fig. 2-1. In this figure, states are represented by circles, transitions by directed arcs, the initial state is the state labeled 1, and input commands are either events with an over-bar denoting send events or events with an under-bar denoting receive events. Events' subscripts are used such that for event $e_{i,j}$ the flow of data is from process i to process j. Non-deterministic behavior at a state, for example the choice between receiving a time-out or an acknowledgment at state 3 of the sender, is modeled by multiple output arcs from that state. The medium (channel) has a capacity of only one message or acknowledgment. A service specification for the same protocol is shown in Fig. 2-2 in which the service primitive events GET and DELIVER between the protocol system and its users (source and destination processes) and their order, are described.

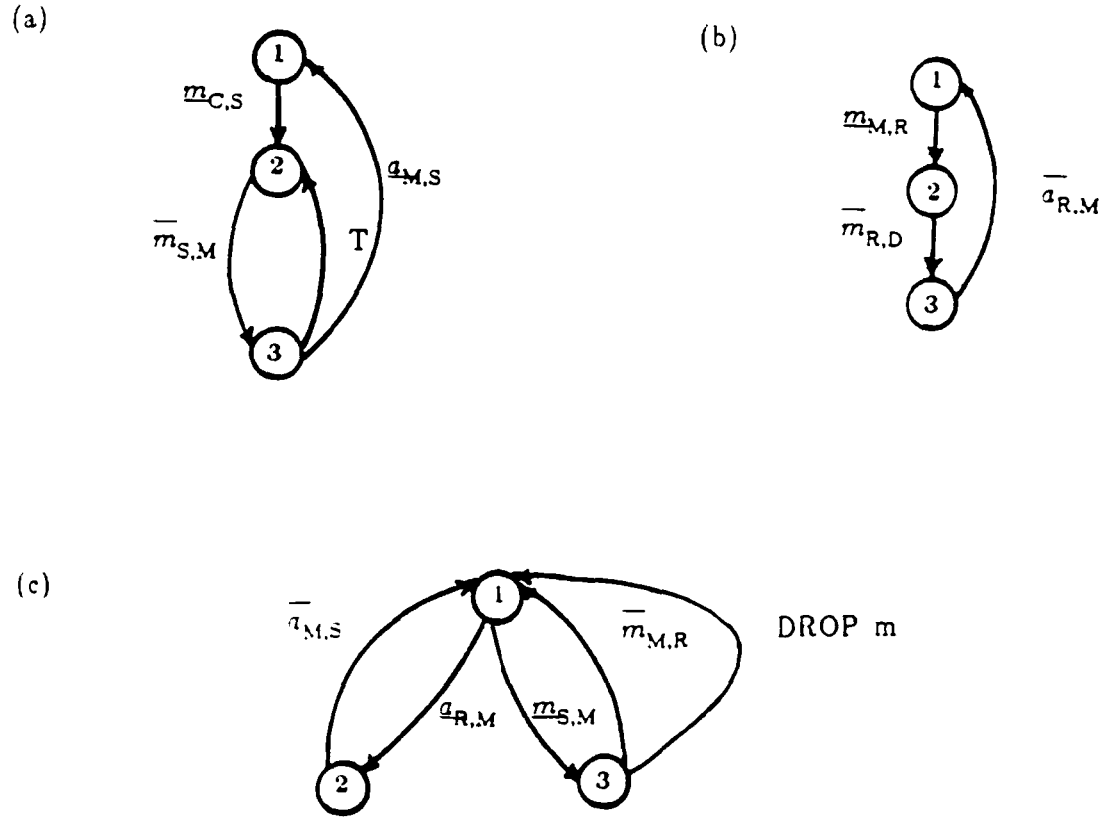


Figure 2-1: A protocol specification of the send-and-wait protocol using FSMs (a) Sender (b) Receiver (c) Medium

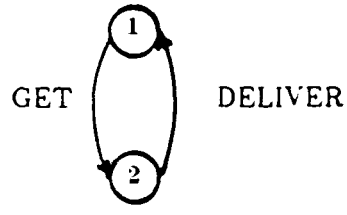


Figure 2-2: A service specification of the send-and-wait protocol using FSMs

In specifying this simple protocol, and control functions of more complex real-life protocols, e.g., the X.21 interface [West 78b], FSM specifications have proven adequate. They are simple, easy to understand and analyze. They fail, however, to describe data transfer functions that include decision (e.g., priority of messages) or timing considerations (e.g., specification of a time-out period). This is because no mechanisms are provided for expressing such features. Moreover, in order to specify messages with sequence numbers using this approach, a state is required for each possible value of a pending message and/or sequence number. This leads to an explosion in the number of states; a phenomena known as the *state explosion problem.* Extensions of the model, as described next, alleviate most of these limitations.

2.2.2.2. State Machine Models

State machines are FSMs augmented with variables and high-level language statements. These statements are associated with transitions and can refer to the variables and input commands. They are either predicates representing conditions for the transition to occur, or actions to be performed upon its occurrence. The state of the machine is represented either by the values of all the variables, or by one of the variables. Consider, for example, extending the send-and-wait protocol with a binary sequence number mechanism for messages so that the receiver can distinguish between messages and their duplicates. A partial state machine specification (whose constructs are adapted from [Boch 83]) of the sender process of this extended protocol, is given in Fig. 2-3. In this specification a variable representing the current message sequence number should be defined at the sender and the receiver. The transition out of a sender's state in which it is waiting for an acknowledgment can have a predicate stating that it should not be corrupted and its sequence number is the one expected; and an action that increments the sequence number of the next message to send.

Bochmann and Gesci [Boch 77a] first used this specification model to specify a simple data transfer protocol and later to specify the HDLC [Boch 77b] and X.25 [Boch 79] protocols. Various other specification systems based on this model have been also developed. They differ essentially in the way they structure the protocol system into sub-processes which are then specified as state machines.

A state machine model proposed by the ISO TC97/SC16/WG1 subgroup B on formal description

```
module Sender

var
  state : (state1, state2, state3);
       (* same states labels as in Fig. 2-1(a) *)


  corrupted : boolean;

  next-msg-to-snd : integer;

  ack-received : integer;
  .
  .

trans    (*  transitions are described in the general
             form of a predicate given by:
             when  <input command> provided
             <boolean expression> from <current state>,
             followed by an action given by:
             to <next state>
             begin <statement> end; *)
   .
   .
   .
   when RECEIVE-A
   provided (not(corrupted)
     and ack-received = next-msg-to-snd)
   from state3
   begin
     next-msg-to-snd := (next-msg-to-snd+1) mod 2;
   end;
   to state1
   .
   .
   .
end module Sender
```

Figure 2-3: A partial state machine specification of the sender process of a modified send-and-wait protocol with binary sequence numbers

techniques (FDT) [ISO 83, Boch 84] employs Pascal-like constructs in extending FSMs. Channels are specified separately from the protocol processes using abstract data types [Gutt 78]. Certain queuing mechanisms can be modeled and time delays before transitions can be specified.

A Specification and Description Language (SDL) [Rock 81] which is primarily represented graphically has been proposed by another standard body, the International Consultative Committee for Telephones and Telegraphs (CCITT). Specifications of channels and timing are not supported. Dickson [Dick 80a], [Dick 80b] has used SDL to specify the packet level of the

X.25 interface [X.25 80].

Examples of other works based on the state machine model for specification have been reported by Schwabe [Schw 81a], Divito [Divi 82] and Shankar and Lam [Shan 84]. These efforts differ in the following. Schwabe differentiates between the specification of the topology describing the connectivity of the processes from the specification of the protocol processes. This feature can be especially desirable in the specification of high level protocols. Divito uses queue histories to record process interactions. This facilitates the specification of certain desirable protocol properties such as the number of messages sent is the same as those received whereas other properties involving order of messages in the histories, for example, are not as naturally expressed. Shankar and Lam allow time variables to be included and time operations to age them. This facilitates the specification of certain protocol real-time requirements such as an upper bound on the time a message can occupy a transmission channel.

Combining the two formalisms of FSMs and high-level languages provides a rich specification tool in which one can express the syntax and the semantics of protocols. On the other hand, such a combination is informal and there is no rule of how much of each to use.

2.2.2.3. Formal Grammars and Sequence Expressions

A formal grammar is defined by a set of *terminal symbols*, a set of *nonterminal symbols*, a *start symbol* and a set of *production rules*. The nonterminal symbols are defined recursively in terms of each other and terminal symbols using the production rules. The start symbol belongs to the set of nonterminal symbols and denotes the language generated by the grammar. In a formal grammar specification of a protocol, nonterminal symbols denote states and terminal symbols denote transitions and operations (e.g., nondeterministic composition). The start symbol denotes protocol behaviors generated by the grammar and production rules define *how* the various protocol behaviors are generated. A formal grammar specification of the sender process of the send-and-wait protocol is given in Fig. 2-4. It is a direct translation of its FSM in Fig. 2-1(a) with terminal symbols (represented by upper-case letters) denoting input commands and non-terminal symbols (represented by lower-case letters) denoting states.

Since regular grammars and FSMs are equivalent, they share the same limitations. The state

```
G = {V,T,S,P},

where the set of nonterminal symbols
V={state1,state2,state3},
the set of terminal symbols
T={GET-M,SEND-M,T,RECEIVE-A},
the start symbol S is state1, and
the set of production rules P is given by

state1 ::= GET-M state2

state2 ::= SEND-M state3

state3 ::= T state2
         ! RECEIVE-A state1

    ``!'' denotes nondeterministic composition.
```

Figure 2-4: A formal grammar specification of the sender process of the send-and-wait protocol

explosion problem is manifested here as an explosion in the number of production rules. To overcome this problem, Harangozo [Hara 77] has used a regular grammar in which indices are added to terminals and nonterminals to allow the representation of sequence numbers. A formal grammar specification of HDLC can be found in [Hara 77]. Teng and Liu [Teng 78] have used a context-free grammar, which provides more expressive power than a regular grammar. They also uses a shuffle operation to integrate grammars defining processes in the same protocol layer by computing all possible shufflings of their behavior A substitution operation is used to integrate grammars defining different protocol layers by substituting terminal symbols in the grammar of the high-level protocol by nonterminal symbols in the grammar of the low-level protocol to form a new integrated grammar.

These two approaches to formal grammar specification for protocols do not support the specification of any predicates or actions associated with protocol behavior. This limitation is overcome by Anderson and Landweber [Ande 84] by using context-free attribute grammars, which are formal grammars in which terminal and nonterminal symbols have attributes associated with them. For example, the terminal symbol SEND-M in the send-and-wait protocol can have the attribute *address* associated with it to determine the address of the addressee. The semantics of protocol operation can then be specified in terms of attribute assignment statements associated

with production rules.

In contrast to formal languages, sequence expressions define directly the valid sequences resulting from protocol execution and not how they are generated. A protocol behavior can be described in one expression where no nonterminal symbols are used. The sender process of the send-and-wait protocol, for example, can be specified as a sequence expression given by

$$\text{SENDER} = \{\text{GET-M} \rightarrow \text{SEND-M} \rightarrow \{\text{T} \rightarrow \text{SEND-M}\}^* \rightarrow \text{RECEIVE-A}\}$$

where operations "*" and "$\rightarrow$" denote the Kleene star and sequential composition operations, respectively.

Sequence expressions have been used by Bochmann for service specification [Boch 80a]. Other examples include work done by Schindler, et al. [Schi 80, Schi 81] to specify the X.25 layer 3 protocol.

2.2.2.4. Petri Net-Based Models

A Petri Net (PN) (see [Pete 77] for a comprehensive survey) graph contains two kinds of nodes: *places* and *transitions*. Directed arcs connect places and transitions. Arcs from places to transitions are called input arcs, and arcs from transitions to places are called output arcs. The execution of the net is controlled by the position and movement of *tokens* which reside in the places. The distribution of tokens in the net at any certain time, known as a *marking*, specifies the state of the net at that time. A PN specification includes a PN graph and an initial marking. A transition in the graph is *enabled* if there are tokens residing in all the input places (i.e., places connected with the transition through input arcs). It can fire any time after it is enabled, upon which tokens are removed from input places and deposited into output places of the transition. PNs are in many ways similar to FSMs, with places in a PN corresponding to states or inputs in a FSM and transitions in a PN corresponding to transitions in a FSM. However unlike FSMs, PNs can directly model interactions between the concurrent processes by merging output arcs from one process to an input arc of another process. Also the concurrent execution of the distributed processes is naturally captured by the presence of more than one token in the net -- a token for each distributed process.

In a protocol modeled as a Petri net, the presence of a token in a place typically indicates that the protocol is waiting for a certain condition to be satisfied, and the firing of a transition represents the occurrence of an event enabled by the condition. Examples of using PNs to model protocols can be found in [Post 76, Azem 78, Dant 80]. A PN specification of the send-and-wait protocol is given in Fig. 2-5. Places are represented as circles, transitions as bars and tokens as filled circles. It should be noted that this PN specification follows the assumption that time-out is ideally set such that a time-out occurs only after a message loss and that acknowledgments are not lost.

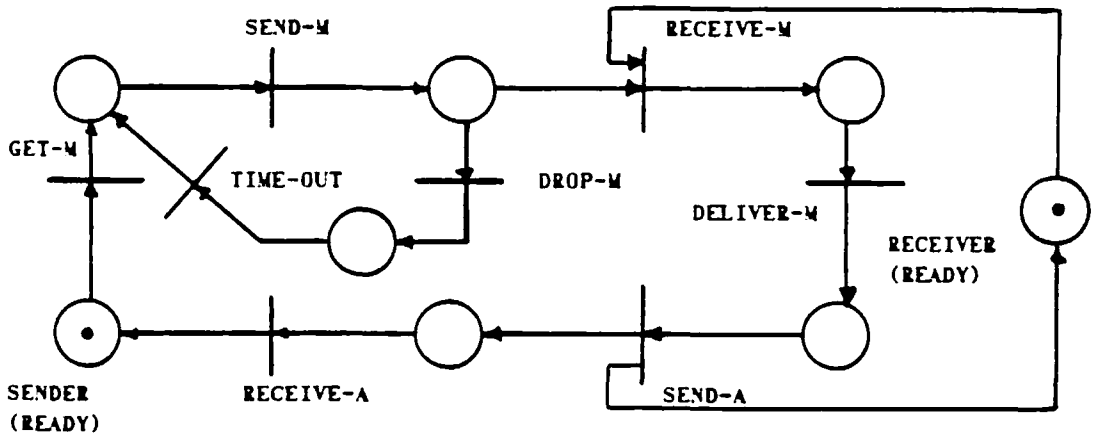


Figure 2-5: A send-and-wait protocol specification using Petri nets

Similar to FSMs, PNs cannot adequately model the complex data transfer of a protocol without suffering from explosion of the net size. Two major extensions to PNs that add to their power in modeling protocols lead to *hybrid PNs* and *timed PNs*. The price for these extensions is more complex validation.

Hybrid Petri Nets

Hybrid Petri nets are extended PNs in which tokens can have identities and transitions can have predicates and actions associated to them. Adding predicates to PNs produces *predicate/transition* nets formalized by Genrich and Lautenbach [Genr 79], where transitions fire only after they are enabled and their associated predicate (i.e., some condition in terms of tokens values) is true. Berthelot and Terrat [Bert 82] have used predicate/transition nets to model the ECMA (European Computer Manufacturer Association) [ECMA 80] transport protocol.

Adding actions to predicate/transition nets produces *predicate/action* nets. Actions are associated with transitions such that when a transition fires, the action is executed and new tokens are put in the output places. For example, data transfer protocols can be modeled as predicate/action nets such that the arrival of a message m with certain parameters is described in a predicate, and the transmission of m is described in the action [Diaz 82].

Keller's model for parallel programs [Kell 76] and numerical PN (NPN) [Symo 80] belong to this category of hybrid PNs. Keller divides systems into a control part and a data part, with places representing control states and transitions representing the changes between states. Variations of this model have been used in modeling protocols [Boch 77a, Azem 78, Baue 82]. NPNs introduced by Symons are similar to Keller's model with the variation of allowing tokens to have any identity not just integer values, and associating read and write memory with the net. Billington has used NPNs to model a Transport service [Bill 82].

Timed Petri Nets

A Timed PN is a PN extended to support some description of time. Timed PNs that have been used for protocols include *time PNs* (TPNs) introduced by Merlin [Merl 76] and *stochastic PNs* (SPNs) introduced by Molloy [Moll 81]. In a TPN, a pair of deterministic time values (t_{min}, t_{max}) is added to each transition of a PN. The pair defines the interval of time in which the transition must fire after it is enabled. This extension allows the modeling of time-out actions of protocols by specifying the t_{min} of the retransmission transition to be equal to the time-out value. Danthine [Dant 80] has used a combination of TPNs and Nutt's evaluation nets [Nutt 72] (a kind of abbreviated PN) to model the transport protocol of the Cyclade network.

SPNs are PNs extended by assigning to each transition a random variable representing the firing delay of that transition. State changes occur in the SPN model with some probability rather than arbitrarily as in a PN. Distributions of the transition delays are restricted to exponential in the continuous case, or geometric in the discrete case. This is because a Markov chain is extracted from the PN graph describing the global protocol behavior; in a Markov chain all transitions should be either exponentially or geometrically distributed. The random representation of time involved in protocol events is used in SPNs to allow for quantitative performance analysis.

2.2.2.5. Algebraic Models

Algebraic specification derives its name from its relationship to universal algebra [Grat 68]. An algebra consists of a nonempty set of *objects* and a set of *operations*. Each operation takes a finite number of objects and produces an object. The meaning of operations is defined in terms of *equational-axioms*. The interpretation of objects and operations when specifying protocols depends on the specific algebraic approach used. We examine next two examples of algebraic systems used for specification of protocols.

In the calculus of communicating systems (CCS) introduced by Milner [Miln 80], objects are expressions describing protocol behaviors; they are generated from a set of send and receive events exchanged between the communicating processes. Operations include "•" denoting sequential composition, "+" denoting nondeterministic composition, "|" denoting concurrent composition, and "NIL" (a nullary operation) denoting deadlock. The concurrent composition of interacting processes produces a new composite process whose behavior includes rendezvous interactions for corresponding send and receive events and shuffling of all other events generated by the interacting processes.

The main characteristic of CCS is that it supports the concurrent composition of processes involved in a protocol as an operation in the algebra to obtain a concurrent behavior of the protocol. This is opposed to the concurrent composition of finite state machines or Petri nets, which is performed by a separate procedure and not part of the specification. The result of the concurrent composition of a set of processes in the algebra is a also process, which allows hierarchical specification of processes. This makes CCS especially suitable for modeling protocols that belong to the ISO hierarchy. For example, a transmission channel process in a protocol at some layer in the ISO architecture can be simply produced or specified as the concurrent composition of the processes providing service from the lower layers.

A CCS specification of the sender process of the send-and-wait protocol is given next. Let τ denote a rendezvous event produced from a previous concurrent composition of the sender with a timer process (for time-out). Also, let m represent a send port for messages and $\bar{a}$ represent a receive port for acknowledgments on the channel between S and M. In addition, let $\bar{d}$ represent a receive port for message incoming from the source. The sender specification S is described

recursively as follows:

$$S = \bar{d} \bullet m \bullet S_1 \qquad\qquad S_1 = \tau \bullet m \bullet S_1 + \bar{a} \bullet S$$

Capabilities for value passing and high-level language statements are also provided. To overcome the imposed synchronous mode of inter-process communication in CCS, one has to explicitly model transmission mediums between any two processes communicating asynchronously.

Many concepts from CCS are employed in the specification language LOTOS (language for temporal ordering specification) proposed by ISO TC97/SC16/WG1 subgroup C [ISO 85, Brin 86]. Holzmann [Holz 82] has also introduced a CCS-variant algebraic model with a division operation used to represent send events and message queues used to allow for asynchronous inter-process communication.

In the AFFIRM system [Muss 80, Suns 82a], the objects of the algebraic model are *abstract data types* [Gutt 78]. The system can be used to specify protocols modeled conceptually as state transition machines as follows: each protocol model is defined as an abstract machine data type, with its variables as *selectors* of the type, and its state transition as *constructors* of the type. A set of axioms defines the effects of each transition on the variables. Abstract data types can also be used in specifying protocol message formats. Desired properties of the protocol are expressed as theorems that refer to the elements of the given specifications. An advantage of this system is its use of abstract data types which provide only abstract description of the systems under consideration. Experience with modeling several protocols in AFFIRM [Suns 82b] has shown the following system limitations: no support for true modeling of concurrency; difficulty in dealing with exception handling, separate specification of local protocol processes, and difficulty in specifying of protocols with more than two processes.

One advantage of algebraic specifications is their rigorous formal base from algebra. Elements of other development tools in a protocol environment can be viewed as an algebra that is homomorphic to the specification algebra [Yemi 82]. One basic limitation of algebraic specifications is the difficulty in dealing with exception handling (for more information on this see [Berg 82]).

2.2.2.6. Temporal Logic Models

Temporal logic [Pnue 77] is an extension of predicate calculus to support the specification of temporal properties of systems (i.e., properties that change during the system execution). Invariant properties that must hold throughout the execution can be stated using predicate calculus. Within the temporal logic framework, the meaning of a computation is considered to be either the sequence of states (state-based approach) or the sequence of events (event-based approach) resulting from the system's execution. The two basic temporal operations in temporal logic besides predicate calculus operations are *henceforth* "□" and *eventually* "◊ ". Let P be any predicate, then $\Box P$ is true at time i (representing the i-th instance of the execution sequence) if and only if P is true at *all* times j , where $j \geq i$, and $\Diamond P$ is true at time i if and only if P is true at *some* time j, where $j \geq i$. A specification in temporal logic consists of a set of axioms that assert properties which must be true of all sequences resulting from a system's execution [Lamp 80, Mann 81].

Temporal logic specifications can be classified into state-based and event-based approaches according to the underlying model of the execution of the protocol. Three different approaches to state-based temporal logic have been pursued by Lamport [Lamp 83], Schwartz and Melliar-Smith [Schw 81b], and Hailpern and Owicki [Hail 80]. The three approaches differ essentially in how close they are to the state machine model with the first being the closest followed by the second and then the third.

Schwartz and Melliar-Smith use a model in which state variables are introduced in the specification only when it is more convenient to express temporal properties in terms of finite history of the past rather than using temporal formulas. The variables used are assumed to be bounded. A specification of the Sender process of the send-and-wait protocol in this approach is given in Fig. 2-6 (adapted from [Schw 82]). Besides the temporal operations eventually and henceforth, the following constructs have also been used in the specification: Until and Latches-Until-After. P Until Q is interpreted as P must remain true until Q becomes true if ever, and P Latches-Until-After Q is interpreted as P when becoming true, remains true until after Q becomes true if ever. Also the predicates at, in, and after, have been used to reason about the currently active control point of each process. The interpretation of at S, in S, or after S is true if control is at the beginning, within, or at the end of the execution of statement S respectively.

A1. $S_o=p$ implies ($S_o=q\neq p$
Latches-Until-After after RECEIVE-A
and $S_o=q\neq p$ Latches-Until-After $S_i=q$)

A2. $\Box\Diamond$ ($S_i=S_o=p$) implies
{$\Diamond$ ~*empty*(InQ) implies $\Diamond$ ($S_o\neq p$ and at SEND-M)}

A3. $S_o=p$ and $\Diamond$ $S_o=q\neq p$ implies
$\Diamond$ ($S_o=q\neq p$ and at SEND-M) Until ($S_i=q\neq p$)

A4. $\Diamond$ at SEND-M Until $\Box$ *empty*(InQ)

Where S_o and S_i are two variables of the underlying state transition model used to record the last message value transmitted by the Sender, and the last acknowledgment value received from the medium, respectively. InQ is a sequence variable representing the queue of message ready at the source. Labels for events are the same as those used in Fig. 2-1(a).

Figure 2-6: A state-based temporal logic specification of the sender process of the send-and-wait protocol

The axioms in Fig. 2-6 have the following interpretations. Axiom A1 states that a message value remains in S_o until both its successful acknowledgment is received and a new message is fetched from the source. Axiom A2 states that whenever the sender gets a message from the source while it is not busy, it eventually sends that message. Axiom A3 states that whenever a new message is placed in S_o, it is infinitely often transmitted until its successful acknowledgment is received. Axiom A4 ensures that message transmission continues until all messages available in InQ are serviced.

The above described approach to temporal logic specifications does not consider the complete set of a system's state space; some of the states are excluded if temporal axioms can be used to reason about them. This sometimes leads to complex specifications requiring several additional constructs (such as Until and Latches-Until-After) and thus rendering specifications complex and difficult to understand. In subsequent work [Schw 83] another approach has been followed in which the protocol required properties are stated on *intervals* of the protocol's execution sequences. It is claimed that this allows higher level temporal logic specifications.

Lamport has considered the complete set of system's variables, and all state transitions are specified in terms of the changes they are allowed to affect the variables. This is done by using an ''allowed changes'' construct in addition to the other basic temporal operations. Although specifications based on this approach are easier to transform into implementations, they are lenghtier than those based on the former approach. Hailpern and Owicki have used unbounded history variables, without employing any states, to record the sequences of messages that are inputs or outputs of the systems. Protocol properties such as the number of messages sent equals number of messages received can be stated quite naturally with this approach, but it would be difficult to state properties that depend on the ordering of a sequence in a history. Moreover, the introduced history variables are actually ''auxiliary'' variables; that is, they are not variables that are required to describe the protocol implementation and thus cannot be used to reason about its correctness.

The state-based temporal logic approach has been used to specify and verify a multidestination protocol [Sabn 82a], and in [Kuro 82] both history variables and internal states have been used in specifying and verifying the three way handshake connection protocol. Shankar and Lam [Shan 84] have used a variant of the eventually operator in stating temporal properties of a bounded length of the global state sequence resulting from a systems' execution.

In the event-based approach, protocol desirable properties are specified using temporal assertions that define constraints on the possible sequences of interaction events. No variables are considered in this approach. Establishing context, meaning a record of the history of previous events, in event-based specifications is much more difficult than in state-based specifications, where states naturally provide the required context. This leads to specifications that are somewhat complicated and lengthy. Vogt [Vogt 82] has used a history variable to represent the sequence of past events and thus establish the required context. In another event-based approach, Wolper [Wolp 82] has introduced extended propositional temporal logic, in which temporal logic is extended with operators corresponding to properties definable by a right linear grammar. This allows the specification of some properties that otherwise cannot be expressed in temporal logic such as stating a proposition that is to hold in every other state in a sequence.

2.2.2.7. Procedural Languages

The unit of specification in a procedural language is a procedure containing type declarations and statements describing detailed computational steps of the system under consideration. Much of the early work done on protocol or service specifications used this approach. Examples of such works can be found in [Sten 76, Haje 78, Krog 78].

The Gypsy programming language [Good 78, Good 82], is a procedural language that includes most of the basic facilities of a Concurrent PASCAL, and has the unique feature of supporting the specification of protocols at any of the three design phases using the same language. Descriptions of service or protocol specifications make use of *queue histories* to record all send and receive operations executed on a system's queue. One limitation of specifications employing queue histories, is the difficulty in modeling unreliable communication mediums [Divi 82] since processes communicate through message queues that do not model loss or corruption of messages. Another limitation is the difficulty of stating properties on a history if the properties depend on the ordering of messages in the queue.

While procedural languages are a natural choice for coding implementation specifications, there has been much controversy regarding their use for service and protocol specifications because of the detailed descriptions of a systems' operation. This makes it rather difficult to specify abstract protocol operation without getting into the details. There is also a biasing effect to implement the protocol in the same language used for specification. The other side of the controversy, though, can argue that such languages, with their rich expressive power, support the specification of both control and data transfer functions of protocols.

2.2.3. A Taxonomy of Specification Tools

Specification tools can be classified along two axes. Along the first axis, they are either *state-based* or *event-based*. The underlying model of a protocol in state-based tools is concerned with the states through which the protocol passes during its operation and with the events that cause changes in its state. States can be either explicitly represented or described by variables. On the other hand, the underlying model in event-based tools is only concerned with the events generated by a protocol without any mention of its state. They include sequence expressions and

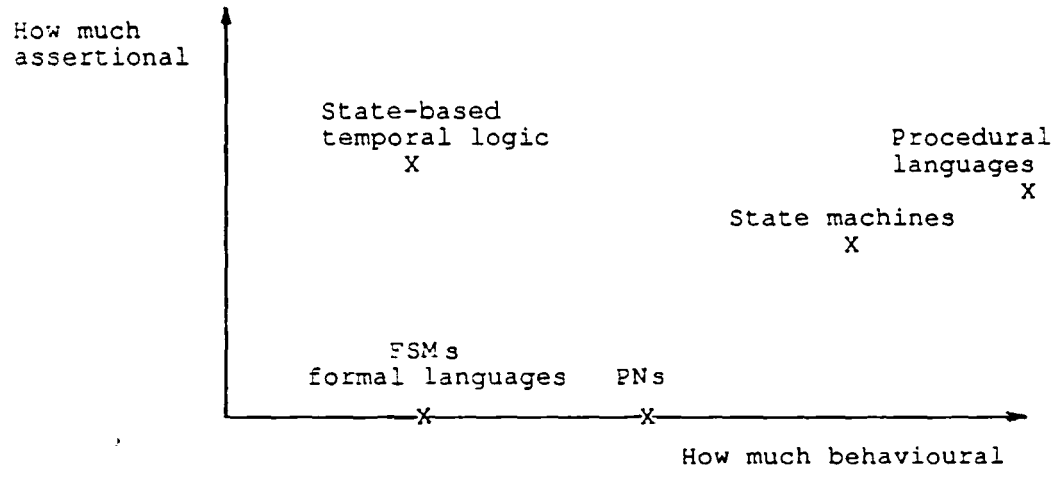
event-based temporal logic specifications whereas the remaining specification tools covered in this section belong to the state-based class. Since state-based specifications describe the actions and responses of protocol operation, they can be directly executable. Event-based tools can at best be first transformed into an executable form. However, they seem to be more abstract than state-based tools since they are not concerned with the internal state of the protocol model.

Alternatively, specification tools can be classified into *behavioral* and *assertional* tools. Specifications belonging to the former class describe the flow of execution of protocols and how it proceeds after each event. They constitute a description of the cause and effect of all modeled protocol events. Assertional specification tools, on the other hand, state the requirements of protocol behavior in terms of desired properties of its possible execution sequences. The more a specification tool is behavioral the more it is executable, and the more a specification tool is assertional the better support it provides for formal verification.

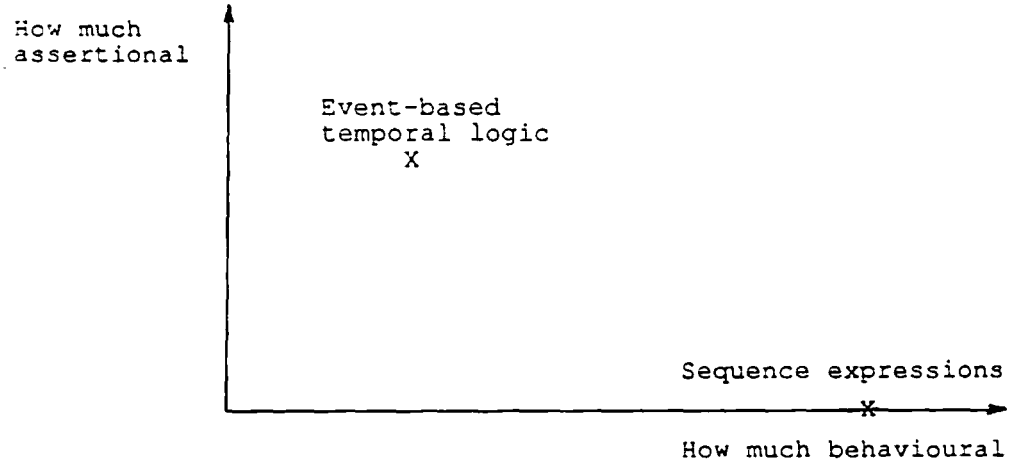
Most specification tools actually exhibit features belonging to both the behavioral and assertional classes. Also, each of these classes constitute a spectrum of specification tools. The extent to which a specification tool is behavioral depends on how much support it provides for the specification of protocol semantics besides its syntax. The extent to which a specification tool is assertional depends on how much support it provides for the statement of functional properties including liveness and safety, and timing properties. Furthermore, specification tools belonging to any of these classes can be either state-based or event-based. Therefore, we illustrate in Fig. 2-7 the relative positions of the various specification tools covered in this section.

2.3. Verification Tools

Protocol verification consists of logical proofs of the *correctness* of each of the specifications of the protocol, and the *mapping* between the service and the protocol specifications and between the protocol and implementation specifications. Proof of correctness of a specification constitutes proving the validity of certain desirable properties that would assure its correct operation under all conditions. Proof of mapping constitutes proving that a specification of a protocol refined at a certain development phase correctly implements the specification input to that phase. Proof of mapping between the service statement phase and the protocol design phase is referred to as *design verification*, and between the design phase and implementation phase is referred to as



(a) State-based specification tools



(b) Event-based specification tools

Figure 2-7: An illustration of the proposed taxonomy of specification tools

implementation verification [Boch 80b].

To prove that a specification is correct, one has to prove that it satisfies protocol *safety* and *liveness* properties [Lamp 77]. Safety properties state the design objectives that a specification must meet if the protocol ever achieves its goals. Liveness properties state that the specification is guaranteed to *eventually* achieve these goals. For example, an informal description of a safety property S and a liveness property L for the send-and-wait protocol specification can be stated as

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

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

Safety and liveness properties such as those listed above are highly dependent on the protocol under consideration. However, there are some general properties that are common to any protocol. They require the given specification of a protocol to be free from general design errors, such as those listed below.

- *Unspecified reception*which indicates that a message reception that can take place is missing in the specification.
- *Nonexecutable interaction* which is a reception or a transmission interaction that is included in the specification but that cannot be exercised under normal operating conditions.
- *Deadlock* which occurs when during the concurrent execution of a protocol, each and every process has no possible transition out of its current state.
- *Tempo-blocking* which indicates that the protocol enters an infinite cycle accomplishing no useful work.
- *Channel overflow* which means that the number of messages in transit in the channel is more than a specified upper bound.

The approach used in proving a mapping between a specification output from a protocol development phase and the specification input to the phase, depends on the specification tool used. Consider the design verification problem. If behavioral specifications are used to describe the protocol service, proof of mapping would be equivalent to proving that the components of the service specification are correctly implemented by those of the protocol specification. On the other hand, if assertional specifications are used, then the service specification constitutes safety and liveness assertions of protocol specification; and design verification coincides with proving

the correctness of protocol specification. That is, since proving the correctness of protocol specification in this case constitutes proving that the protocol specification meets its service assertions, it proves at the same time that the protocol specification is a correct implementation of the service specification.

Since protocol implementations are specified using high-level languages, they can be verified using traditional program verification tools. We will limit our discussion throughout the rest of this section to surveying tools for the verification of service and protocol specifications, and for design verification.

2.3.1. State Exploration

State exploration examines all possible behaviors of a protocol. It is used in verifying specifications belonging to the state-based and behavioral class of Fig 2-7(a). State exploration of the concurrent execution of the processes local to a protocol layer produces a *reachability graph*. In this graph, each node represents the combined states of all the local processes, and each arc represents a local transition. Starting from the initial state of the graph, interactions of the processes are examined by exploring all possible ways in which the initial states and all subsequent states can be reached. Each node the protocol can reach is checked for deadlock and unspecified receptions. The whole graph can be then checked for other general desirable properties of the protocol such as progress, absence of tempo-blocking and channel overflow [Suns 75, West 78a]. In the case of Petri nets specifications, each state in the reachability graph corresponds to a marking of the net [Ayac 81, Diaz 82, Jurg 84].

The reachability graph for the send-and-wait protocol is depicted in Fig. 2-8. All send events in the graph are followed by the corresponding receive event, thus indicating absence of unspecified receptions, and all the transitions in the FSM specification of the communicating processes in Fig. 2-1 have corresponding links in the reachability graph indicating absence from nonexecutable interactions. Also, there is no tempo-blocking because the only cycle in the graph which involves time-out (other than the repetition of the entire protocol behavior) performs useful work each time a message is lost. In addition, since all nodes in the reachability graph have outgoing links, then there is no deadlock in the global behavior of the protocol. To see how a deadlock behavior would be detected by this approach, consider removing the time-out transition from the Sender process in

Fig. 2-1. The system would then deadlock at state 5 in Fig. 2-8 if the channel loses a message. Note that in producing the graph of Fig. 2-8, we followed the idealistic assumption that time-outs only occur after a message loss. However, if one assumes that the time-out period can have any time duration, then one would get another reachability graph that differs from that in Fig. 2-8 in that there would be a time-out transition from each of states 4, and 7 through 12 back to state 2. There would be then a possibility of tempo-blocking due to any of these time-out loops. This illustrates how the behavior of protocols can be time-dependent and the importance of integrating the verification of timing requirements with functional verification.

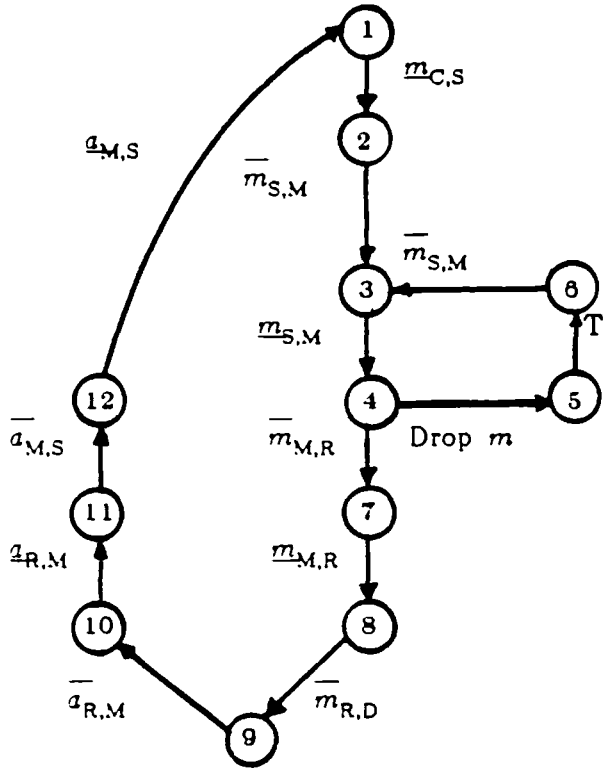


Figure 2-8: A reachability graph of the send-and-wait protocol

Using this verification tool, design verification consists of demonstrating how the protocol's reachability graph can be mapped to its service specification. Such a mapping for the send-and-wait protocol is defined as follows: in Fig. 2-2 states 1 and 2 are implemented by states 1 and 8 in Fig. 2-8 respectively, and events GET and DELIVER in Fig. 2-1 correspond to $m_{C,S}$ and $\overline{m}_{R,D}$ in Fig. 2-8 respectively.

The principal advantage of state exploration is that it can be readily automated. Automated state exploration tools have been used successfully in discovering errors in several protocols; see for example [West 78c, Boch 79]. An automated and interactive verification tool called OGIVE [Prad 79] has been used successfully in proving certain general properties of Petri nets [Jurg 84].

A principal limitation of the state exploration is the explosion in the number of states as the the complexity of the protocol analyzed increases. Note that the number of states in the reachability graph is equal to the product of the number of states in the FSM specifications of each of the communicating processes. In fact, Brand and Zafiropulo proved that the problem of verifying the general properties of communicating FSMs, is generally undecidable [Bran 83] except for a restricted class of communicating FSMs [Bran 83, Goud 84b]. The state explosion problem can be partially overcome by verifying each protocol process separately and then the protocol as a whole [Goud 84b], or limiting the number of messages in the channel [West 82]. Other approaches include assuming direct coupling between corresponding send and receive transitions such that there concurrent composition involves just one rendezvous interaction instead of two possibilities due to the shuffling of the two transitions, or using some equivalence relation to minimize the reachability graph [Rubi 82]. In addition, instead of verifying the complete global behavior of a protocol, considerable simplification can be achieved by verifying projections of that behavior according to the various distinct functions of the protocol (for example separate connection establishment from data transfer functions of data link protocols) [Lam 82]. *Symbolic execution* in which states are grouped into classes that are specified by assertions [Bran 78, Haje 78, Bran 82] is another approach to alleviate the state exploration problem. Various reduction techniques have been also used in verifying Petri net specifications [Diaz 82].

Although state exploration is usually adequate in verifying general properties of protocols, it cannot be used for the verification of specific protocol safety and liveness properties such as properties S and L given above for the send-and-wait protocol. These are addressed by the verification tool discussed next.

2.3.2. Assertion Proof

Assertion proof follows the Floyd/Hoare [Floy 67, Hoar 69] technique for program verification. Safety and liveness properties of a protocol can be expressed as assertions, which are attached to different control points of a specification. To verify an assertion means to demonstrate that it will always be true whenever the control point it is attached to is reached, regardless of the execution path taken to reach that point.

When a protocol specification is decomposed into a number of local process specifications, local invariants are first verified for each process directly from their specifications. Global service invariants can be then verified using the already proven local assertions. Invariants of a specification are special assertions which describe properties that are true at every control point in the specification. To prove assertions of a local process, the introduction of auxiliary variables, which are variables not required in implementing the protocol, is often required. For example, arrays of data sent and received are required in a data transfer protocol employing sequence numbers, in order to make precise statements about the order in which messages are sent and received [Sten 76].

Assertion proof is related to the class of assertional specification tools described in the taxonomy of section 2.2.3. In particular, it is used in verifying assertions associated with specification using procedural languages [Krog 78, Sten 76], state machines [Boch 77a], hybrid Petri nets [Diaz 82], and temporal logic [Hail 80, Schw 82, Sabn 82a, Schw 83]. In the case of procedural languages, inference rules (i.e. rules that define the effect of each statement type on the assertions preceding it) for each type of statement are used in proving local assertions. This also applies to the high-level statements in a state machine specification. In the case of Petri net-based models, net invariants deduced directly from the net structure, are used in proving local assertions. Within the temporal logic framework, temporal axioms, which constitute a temporal logic specification, are used in specifying and verifying safety and liveness assertions. Temporal logic has the unique feature of supporting the specification and verification of liveness properties.

Formulating assertions and proving them require a great deal of user ingenuity. This difficulty can be partially alleviated by using some proof strategy such as induction on the structure of specifications [Suns 81] and by automation as is provided by several verification systems;

examples of verification systems that have been applied to protocols are described in [Good 82, Suns 82a, Divi 82]. It should be noted though that automating assertion proof is considerably more complex than automating state exploration. For a detailed comparison of verification systems used for protocols, the reader is referred to [Suns 82b, Suns 83].

2.4. Performance Analysis Tools

Performance analysis of protocols includes specification and analysis of timing requirements and performance measures. Protocol behavior is typically time-dependent (as shown in section 1.2), and their efficient performance hence depend on certain timing requirements. Performance measures are used as indications of how well a protocol performs. The combination of these two performance analysis problems is natural since both problems are concerned with the timing behavior of protocols. This allows the protocol designer to study the effect of various performance parameters on their timing behavior. We first examine some issues common to the two performance analysis problems and then survey approaches to each of them.

In order to analyze protocol performance, it is necessary to establish a model of the communication medium and the timing behavior of the protocol. The former is provided in the form of data specifying the medium's characteristics. For example, in the case of data link protocols (at layer 2 of Fig. 1-1), the following medium characteristics should be specified: bandwidth, bit error probability, topology, medium configuration (i.e., half or full duplex), and the upper bound on the number of messages in transit at any one time.

A model of the timing behavior of a protocol can be either formulated directly from first principles, or extracted from a formal specification of the protocol. We will refer to the former approach as *direct* and to the latter as *specification-based.* In both approaches, the model should specify the global view of protocol operation. It should also include the specifications of the following features. First, since a protocol's timing behavior is often non-deterministic, the probabilities of all possible protocol events at the various instants of its behavior should be specified. Second, a representation of the times involved in each of the events is also required. Typically, they are represented by their *bounds* or *distributions*. Bounds on an event time specify the minimum and maximum time before its occurrence. This time representation has been used in [Merl 76, Sabn 82b, Krit 84, Shan 84]. Distributions of event times provide more complete

description of their random nature. This time representation is often used especially in evaluating protocol performance measures; see for example [Suns 75, Moll 81, Rudi 84]. Third, some statistics for message lengths should be provided. These are typically considered as constants or represented by their distributions.

2.4.1. Tools for Analyzing Protocol Timing Requirements

Protocol timing requirements are conditions on the protocol's timing behavior to ensure its efficient performance. Consider, for example, a retransmission on time-out protocol such as the send-and-wait protocol. The efficient performance of the protocol depends on the requirement that time-out would occur after a message loss only with a very small probability and that the time between a loss and a time-out is minimized. Another example of a protocol timing requirement is to restrict the lifetime of messages occupying the protocol system [Sloa 83]. A third example of a timing requirement that underlies the behavior of many protocols is that if they do not achieve progress within a specified amount of time, then they either reset or abort. Such a requirement is crucial to prevent protocols from being stalled due to exceptional situations such as when one of the protocol process has crashed, or when the transmission links are heavily loaded.

Specification and analysis of protocol timing requirements can also affect the verification of protocol functional properties. In particular, if timing requirements are ignored, then verification of safety may be unnecessarily expensive and verification liveness may be not enough. Verification of safety properties may be complicated by the consideration of unrealistic protocol behaviors that do not satisfy the given protocol timing requirements. Also, proving that the protocol's goals will be eventually achieved is not enough if these goals are achieved after a very long time. In fact, a timing error has been found in the Alternating Bit protocol [Bart 69], which has been proven safe and live [Yemi 82]. It has been shown that the protocol would never achieve its eventual goal if the time-out rate is not properly set. By specifying and analyzing protocol timing requirements, performance parameters of the protocol (such as the time-out rate in the send-and-wait protocol) can be properly set. The resulting timing behavior would thus be (time-wise) realistic and estimates of its duration can be computed.

Early work on the specification of timing requirements has been done by Merlin [Merl 76] using time PNs (see section 2.2.2.4). A bounds representation of time has been used to describe

minimum and maximum firing times for a time-out transition in the Alternating Bit protocol. Similar time representation has been used by Sabnani [Sabn 82b] but for FSM specifications. Note that in both of these cases, the state exploration of the concurrent behavior of the local processes resulting in a description of the protocol global behavior, should be modified. Consider a state in the global state description where n possible transitions are possible. Let $t_{i,min}$ and $t_{i,max}$ denote the minimum and maximum time for transition i, respectively. The corresponding transition in the global description has the bounds of $(Min[t_{i,min}],Min[t_{i,min}])$, where Min is an n-ary operation to compute the minimum. A transition in one of the local processes with t_{min} greater than the upper bound on the corresponding transition in the global behavior, would be then time-wise unrealizable. The limitation of these two efforts stems from the state explosion problem associated with the specification tools used.

Shankar and Lam [Shan 84] assume a constant time representation and use time variables to refer to the occurrence times of events. By including time variables in the enabling condition of an event e, time constraints of the form "event e can only occur after a given time interval" or "event e will occur within a certain elapsed time interval" are stated as safety properties and verified accordingly.

2.4.2. Tools for Analyzing Protocol Performance Measures

Protocol performance measures are indications of how well the protocol will perform. Examples of such measures include *execution time*, *delay*, and *throughput*. The execution time is the time required by the protocol to reach one of its final states, starting from the initial state. It would be a valuable performance measure for terminating protocols such as a connection establishment protocol where it represents the time required for the distributed processes involved in the protocol to get connected. Throughput is the transmission rate of useful data between processors, excluding any control information or retransmission required by the protocol. It indicates how efficiently the transmission channel is utilized. Delay is the time from starting a message transmission at the sender to the time of successful message arrival at the receiver. It is useful in indicating the degree of service that the protocol provides.

Two tools are typically used in evaluating protocol performance measures: *analytic tools*, and *simulation tools*.

2.4.2.1. Analytic Techniques

Various instances of resource contention and the related queueing delays are often witnessed in the operation of communication protocols. For example, in the send-and-wait protocol a new message arriving at the sender has to be queued if the sender is busy waiting for the successful acknowledgment of a previously sent message. Therefore, queueing theory provides a convenient mathematical framework for formulating and solving protocol performance models [Klei 75, Koba 78, Reis 82]. In such a queueing model, the server denotes the protocol system under consideration which is typically modeled as a stochastic process.

Let us demonstrate how the delay of the send-and-wait protocol can be computed using basic probability laws and the protocol's FSM specification. Assume that the time involved in each transition of the reachability graph in Fig. 2-8 is an exponentially distributed random variable. Also, assume that a negligible delay is involved at both the sender and receiver ends of the medium. Based on these assumptions and considering a single cycle operation of the protocol, a modified reachability graph is shown in Fig. 2-9. The problem can be stated as follows: evaluate the mean value of delay d between state 2 to 8 in Fig. 2-9. The following data is used next: medium bandwidth of 9600 bits/sec. (for terrestrial links), mean message and acknowledgment lengths l of 1024 bits (therefore the mean transmission time t_s is 0.017sec./message), bit error probability p_b of 10^{-5}, mean propagation delay t_d of 0.013 sec./message, and mean time-out t_T of 1 sec./message.

Recall from section 2.1 our assumption that time-out occurs only after the medium has lost a message. This indicates that the probability of time-out is the same as the probability of a lost message. Therefore, the probability of the time-out loop denoted by p is given by

$$p = 1 - (1 - p_b)^l \qquad (2.1)$$

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

The mean delay is given by

$$E[d] = p/(1-p)\,(t_T + t_s) + 3t_s + 2t_d \qquad (2.2)$$
$$= 0.357 \text{ sec./message}$$

and the second moment of d is

$$E[d^2] = p/(1-p)\,(2t_T^2 + 2t_s^2) + 2p^2/(1-p)^2\,(t_T + t_s)^2 + 6t_s^2 + 4t_d^2 = 0.09 \qquad 2.3$$

Assume that messages arrive at state 2 in Fig. 2-9 with rate λ, then the protocol's *mean transfer time T* which is the sum of delay and a waiting time is given by the Pollaczek-Khinchine formula [Klei 75]:

$$T = E[d] + (\lambda\, E[d^2])/(2[1-\lambda\, E[d]]) \qquad (2.4)$$

In Fig. 2-10, we plot T versus λ for various message lengths. As expected, T increases as λ increases and the system becomes saturated when λ approaches $1/E[d]$. Also, as l increases T increases due to the increases in transmission times and p.

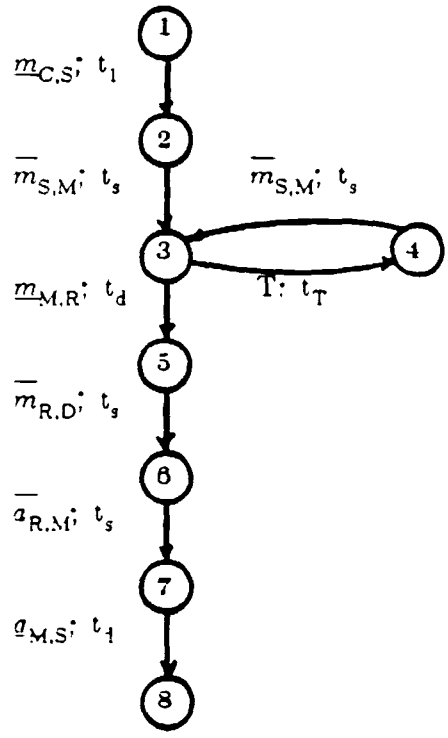


Figure 2-9: A modified reachability graph of the send-and-wait protocol

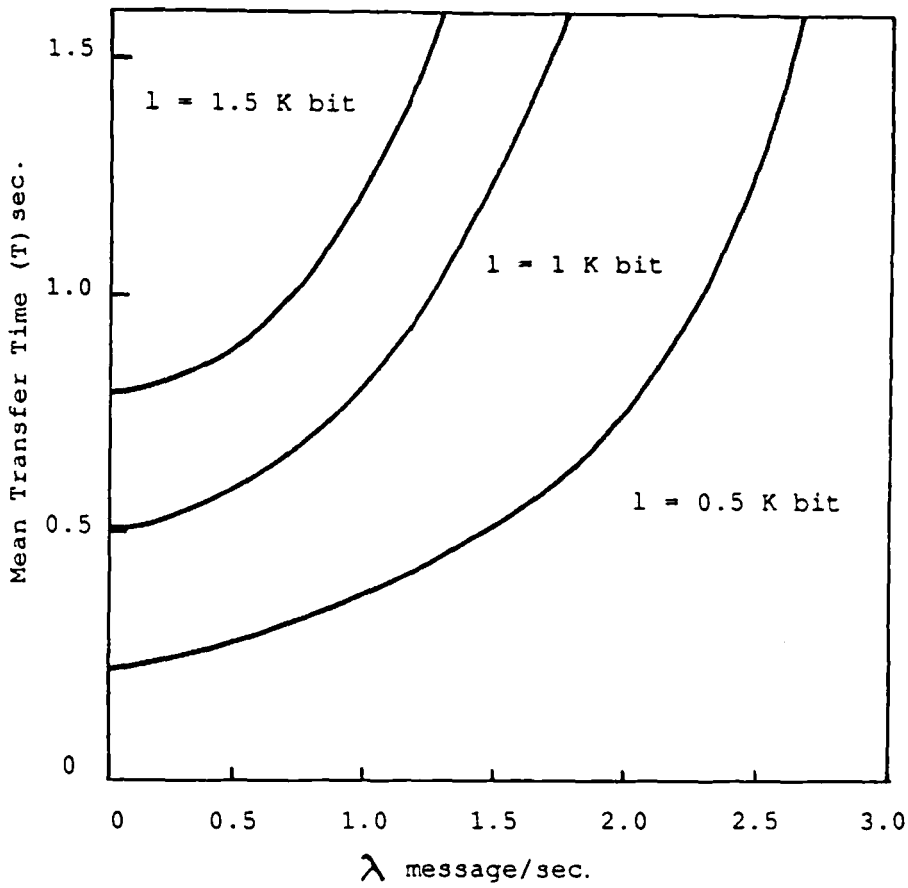


Figure 2-10: Transfer time versus arrival rate of the send-and-wait protocol

One example of specification-based performance evaluation tools is reported by Molloy [Moll 81]. Molloy introduced stochastic Petri nets (SPN) which are Petri nets extended by assigning a random firing delay to each transition in the net. The reachability set of the net is first generated and analyzed for logical correctness, then a Markov chain that is isomorphic to the set, is generated. The steady-state probabilities of the Markov chain are calculated and used in modeling and computing throughput and delay. This approach is limited only to exponentially (in the case of continuous representation of transition firing times) or geometrically (in the discrete case) distributed firing delays. Other specification-based approaches to protocol performance evaluation can be found in [Bolo 84, Krit 84, Razo 84, Rudi 84].

The specification-based approach has the advantage of allowing performance evaluation tools to

be automated. This would also facilitate its integration with other development tools in a protocol development environment. However, the approach largely depends on devising a mapping between protocol specification and the performance model. This mapping may be in some cases too restrictive as is the case, for example, with the Markovian property of the resulting performance model of SPNs.

Examples of works based on the direct approach can be found in [Gele 78, Tows 79, Yu 79, Bux 80]. In this approach, all possible behaviors of the protocol under study has to be extracted directly from a human understanding of its operation.

2.4.2.2. Simulation

Analytic performance models of real-life protocols are usually intractable. In this case, simulation is used in evaluating protocol performance. Even when an approximate model of the system is sought, simulation can be a valuable tool in validating the modeling approximations and assumptions.

In the case of specification-based simulations, the protocol specification used should be executable. Referring to our taxonomy of Fig. 2-7, a specification that is easily executed is one that can also be easily transformed into an implementation. An example on specification-based simulation of protocol can be found in [Regh 82]. Direct protocol simulations, on the other hand, are based on a protocol implementation. For example, a direct simulation of the HDLC procedures has been carried out by Bux, et al. [Bux 82].

The shortcomings of simulation are clearly its high cost in terms of time and effort, and the little understanding of the system gained. The second problem can be alleviated through a large number of simulation runs.

Part II

Methodology

Chapter 3

Protocol Functional Specification and Analysis

3.1. Introduction

The objective of this chapter is twofold. First, to provide an overview of a protocol specification algebra that will serve as the basis of automated performance analysis in the next chapter. The algebra is a variant of Milner's calculus of communicating systems (CCS) [Miln 80] (see section 2.2.2.5). Second, to demonstrate how progress errors (e.g., deadlocks) in the protocol can be easily detected through algebraic calculations. A connection establishment protocol is used as an example throughout the chapter.

The algebra is introduced in section 3.2 and applied to specify protocols in section 3.3. In section 3.4, the algebra is used to detect protocol progress errors which are further classified into deadlock and unspecified reception errors. These errors are the most common protocol design errors [Rudi 85]; verification of freedom of other protocol design errors is not addressed in this work. The use of the algebra in computing the concurrent behavior of a protocol is described in section 3.5. In section 3.6, three functions which are used to isolate sub-behaviors of a protocol that a protocol designer may require for functional and performance analysis, are introduced. Finally, a summary of the issues presented in this chapter, and an outline of how they are used in the first step of the methodology for specification-based performance analysis of protocols are presented in section 3.7.

3.2. A Specification Algebra

3.2.1. Trees Can Provide an Operational Model of Protocols

Consider the connection establishment protocol [Rudi 85] between a terminal process T and a network process N communicating through two half-duplex, FIFO channels R (T to N for call-

request messages) and *I* (*R* to *N* for incoming-call messages). The execution of each of the four processes can be described in terms of a tree as depicted in Fig. 3-1. The nodes of a tree represent a state of execution, and the branches represent occurrence of events. Several branches emanating from a node represent alternative events. The terminal starts in a ready state and becomes connected upon either sending a call-request (!*req*) to the network (via channel *R*), or receiving an incoming-call (!*inc'*) from the network (via channel *I*). When the terminal is in a connected state, its behavior *terminates* ($). The two channels simply receive messages (?*req* or ?*inc*) and deliver them (!*req'* or !*inc'*). The behavior of the network is similar to the terminal (change *req* to *inc*, and conversely).

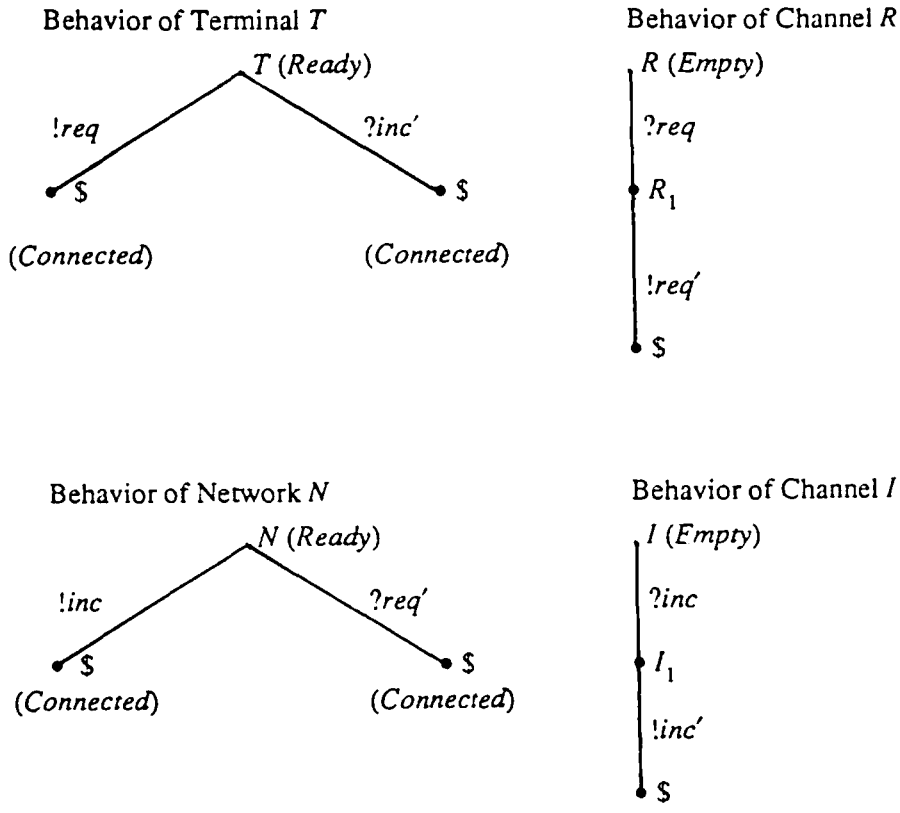


Figure 3-1: Trees describing the execution behavior of the processes in the connection establishment protocol

Generally, an *execution tree* (ET) is a labeled tree where:

1. Nodes can be labeled with *identifiers*, denoted by italicized capital letters. The special symbol "$", denoting termination, can only label leaves.
2. Branches are labeled with send, receive, or rendezvous events, denoted by lower case strings and preceded by a "!", "?", and "&", respectively. Rendezvous events represent successful interactions of corresponding send and receive events (*co-events*) having the same *names* (*name*[!*e*] = *name*[?*e*] = *e*). For example, an interaction of "!req" and "?req" produces the rendezvous event "&req".

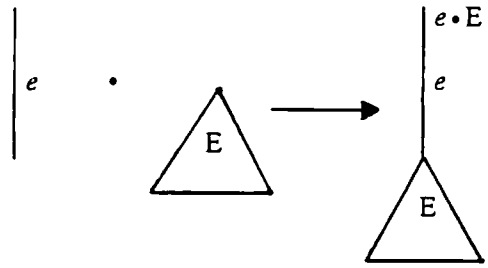
3.2.2. Execution Trees Form an Algebra

ETs form the objects of an algebra [Grat 68]. The operations of the algebra describe composition of ETs to form new ETs as follows:

Sequential composition: let *e* denote an event and E an ET, *e*•*E* denotes the tree obtained by attaching a branch labeled *e* to the tree E. See Fig. 3-2 (a).

Non-deterministic composition: let A, B be two ETs, A+B denotes the tree obtained by joining A and B at their roots. See Fig. 3-2 (b).

(a) *Sequential composition*



(b) *Non–deterministic composition*

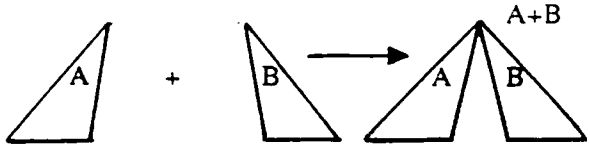


Figure 3-2: Sequential and non-deterministic compositions of ETs

Two expressions E_1 and E_2 in the algebra of ETs are said to be *equivalent*, $E_1 \equiv E_2$, if and only if, they represent the same event sequences and the choices at corresponding nodes in their ETs are the same. A formal definition of equivalence is given in appendix 3.I. The meaning of the operations in the algebra can be described, in analogy to standard algebras, using *equational axioms* as follows:

Axioms:

A1. $A+B \equiv B+A$

A2. $A+(B+C) \equiv (A+B)+C$

A3. $A+\$ \equiv A$

A4. $A+A \equiv A$

Since non-deterministic composition is commutative (A1) and associative (A2), it can be generalized to an *n-ary* operation $\sum_{i=1}^{n}$. An expression A in the algebra of ETs can then be represented canonically as a *sum of summands* $\sum_{i=1}^{n} a_i \bullet A_i$. (Note that from A3, $\$ = \sum_{i=1}^{0}$.)

The concurrent execution of two communicating ETs can be simulated by having each pair of co-events produce a rendezvous event, and considering all possible shufflings of these events. This concurrent execution can be represented by a new ET. It is formally captured by a *concurrent composition* operator "|" in the algebra of ETs, which is defined in terms of the primitive composition operators below. Let *scope*(A,B) of communication between two ETs A and B denote the set of *names* of events with which they communicate. We assume *one-to-one* addressing meaning that each message has unique sender and receiver processes. Thus, the intersection between two different *scope* sets should be always equal to $\varnothing$. (A formal definition of the scope of any two expressions is given in appendix 3.II.)

Concurrent Composition Definition:

Let $A = \sum_{i=1}^{n} a_i \bullet A_i$ and $B = \sum_{j=1}^{m} b_j \bullet B_j$, then

$$A|B = \sum_{\forall a_i,\ name(a_i) \notin scope(A,B)} a_i \bullet (A_i|B) + \sum_{\forall b_j,\ name(b_j) \notin scope(A,B)} b_j \bullet (A|B_j) + \sum_{\forall a_i \text{ and } b_j \text{ where:}} \&e \bullet (A_i|B_j)$$

(1) $name(a_i) = name(b_j) = e \in scope(A,B)$

(2) a_i and b_j are *co–events.*

Concurrent composition is easily shown to be commutative and associative. Also, from A3 and the definition above, we can deduce that $A|\$ = A$. The concurrent composition of the terminal *T*

and channel R ETs of Fig. 3-1, produces a new[1] ET: $RT \triangleq R \mid T$ depicted in Fig. 3-3. $Scope(T,R)$ is given to be equal to $\{req\}$.

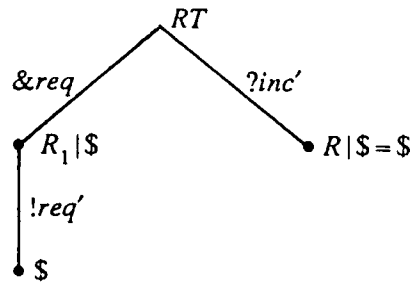


Figure 3-3: ET resulting from the concurrent composition of the two ETs of terminal T and channel R

Having presented the operations of the algebra, we can now define the syntax of an expression in the algebra of ETs. Let the set of events be denoted by ζ and the set of identifiers be denoted by Γ.

Definition 3.1 An expression E in the algebra of ETs is either: $\$$, $I \in \Gamma$, $e \bullet E$ ($e \in \zeta$), $E + E$, or, $E \mid E$.

Let the set of expressions in the algebra of ETs be denoted by χ. Upper case letters will be used to range over χ. An ET can be described by either one expression or a set of possibly recursive *equations* in the algebra. As an example of the latter, the ET describing the behavior of channel R in Fig. 3-1 corresponds to two equations: $R = ?req \bullet R_1$ and $R_1 = !req' \bullet \$$.

The algebra of ETs differs from CCS in three key respects. First, all rendezvous events in CCS are considered to be identical. We differentiate between them according to the *names* of their send and receive events since various rendezvous events typically have different performance properties. Second, addressing is many-to-many in CCS as opposed to one-to-one in the algebra of ETs. One-to-one addressing is assumed in the algebra of ETs because it simplifies the expansion of the concurrent composition of processes, and can be used to simulate many-to-many

[1]We will adopt the notation that $I_1|I_2$ corresponds to a new identifier that is a concatenation of these identifiers after sorting them in ascending order; that is, $I_1 I_2 = I_1 | I_2 = I_2 | I_1$.

addressing when needed by explicitly modeling the common mailbox as a process. Third, send and receive events in CCS can be accompanied by data transfer. Only the communication aspect of these events is considered in the algebra of ETs.

Issues such as completeness of the algebra and existence of a unique solution of expressions in the algebra are addressed in [Miln 80, Miln 81, Sand 82].

3.2.3. The Algebra of ETs is Different From the Algebra of Regular Events

The algebra of ETs is similar in several respects to the algebra of regular events [Salo 66] whose terms represent the languages accepted by finite state automata. In fact, comparing the rules of equivalence between the two algebras indicates that they are virtually the same, except for two main differences. First, the concurrent composition operation which is not included in the algebra of regular events but included in the algebra of ETs. Second, the distributive law: $a \bullet (B + C) \equiv a \bullet B + a \bullet C$ is accepted in the algebra of regular events but rejected in the algebra of ETs. This is due to the difference in the underlying definition of equivalence between the two algebras. Expressions in the algebra of regular events are equivalent if they represent the same set of event sequences. On the other hand, expressions in the algebra of ETs are equivalent if they represent the same event sequences and the choices at corresponding nodes in their ETs are the same. In the expression on the left hand side of the law, there is a choice between behavior B or C. However, on the right hand side the choice is between a and a.

3.2.4. The Algebra of ETs Meets Most Specification Requirements

The key requirements of a protocol specification method have been introduced in section 2.2.1. They state that for a specification method to adequately model functional behaviors of protocols, it should support the modeling of concurrent and non-deterministic behaviors; abstract and modular descriptions; and the modeling of control (or communication) functions and data transfer functions of protocols. Furthermore, a specification method should support features that facilitate the application of other protocol development tools such as verification and performance analysis.

The algebra of ETs supports modeling of concurrent and non-deterministic behaviors. It also supports modeling of modular descriptions of protocol processes in which details of their

operation are abstracted; only their communication behavior is considered. However, the algebra does not meet completely two of the above requirements. First, modeling of data transfer functions of protocols is not supported (this is a possible extension to the specification algebra as will be discussed in section 8.2). Second, since the primary objective of the methodology is performance analysis, only support for performance analysis of protocols is provided. In chapter 4, we will discuss how the specification algebra can be augmented with timing information needed for performance analysis.

3.3. Protocol Processes Can Be Specified Algebraically

The functional behavior of a protocol process can be specified by a *complete* set of equations in the algebra of ETs. A set of equations is complete if every identifier appearing on the right hand side of an equation also appears on the left hand side of some equation. For example, algebraic specifications of the processes (whose ETs were depicted in Fig. 3-1) in the connection establishment protocol, are given by

PROCESS T
$T = !req \bullet \$ + ?inc' \bullet \$$
END

PROCESS N
$N = ?req' \bullet \$ + !inc \bullet \$$
END

PROCESS R
$R = ?req \bullet R_1$
$R_1 = !req' \bullet \$$
END

PROCESS I
$I = ?inc \bullet I_1$
$I_1 = !inc' \bullet \$$
END

The configuration of a protocol can be specified by a list of processes and the *scope* of communication between each pair of them. A *protocol specification* is then defined to include a specification of its configuration, and algebraic specifications of its processes. The configuration of the connection establishment protocol, which is depicted in Fig. 3-4, is specified as follows:

PROTOCOL Connection Establishment: T,R,N,I

$scope(T,R) = \{req\}$
$scope(N,I) = \{inc\}$
$scope(R,N) = \{req'\}$
$scope(I,T) = \{inc'\}$

END

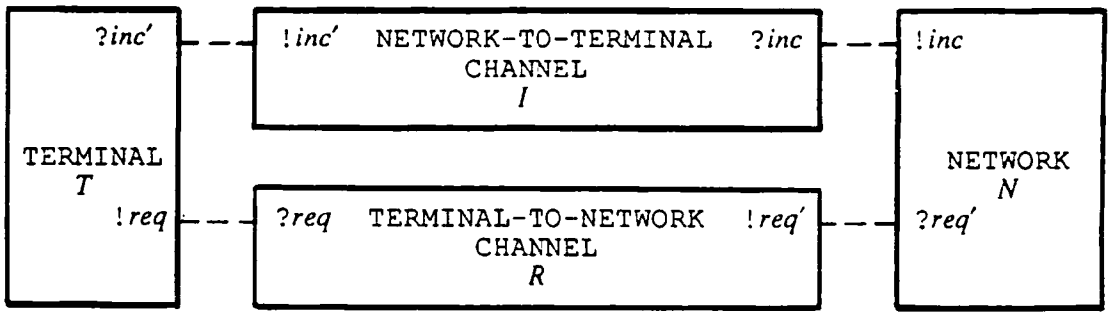


Figure 3-4: Configuration of the connection establishment protocol

3.4. Progress Errors Can Be Detected Through Concurrent Composition

Definition 3.2

A *progress error* in the concurrent execution of A and B exists if $A|B = \$$ while $A \neq \$$ and $B \neq \$$.

The resulting \$ in such cases indicates *improper termination* as opposed to proper termination which occurs when concurrently composing two terminating expressions: $S|S = S$. A progress error is an indication of either a *deadlock* error or an *unspecified reception* errorin the behavior of one or both of the behaviors being composed, as defined below. A deadlock error is due to an indefinitely unsatisfied send request, whereas an unspecified reception error is due to an indefinitely unsatisfied receive request. Let the *choice set CH* be a function: $\chi \rightarrow$ power set of ζ such that $CH(\sum_{i=1}^{n} a_i \bullet A_i) = \{a_i; i=1,...,n\}$.

Definition 3.3

If a progress error occurs while concurrently composing expressions A and B, then there is:

- a deadlock error for each receive event belonging to $CH(A) \cup CH(B)$, and
- an unspecified reception error for each send event belonging to $CH(A) \cup CH(B)$.

A protocol exhibits a deadlock[2] (unspecified reception) error if the concurrent composition of any pair of its processes exhibits a deadlock (unspecified reception) error. The concurrent composition of two processes involves the concurrent composition of several pairs of expressions in their specifications. It should be noted that based on the given specifications and the protocol designer's decision, some of the progress errors discovered may not be undesirable but can be considered proper terminations. An example of such cases is shown next.

Using the concurrent composition definition, the concurrent composition of the four processes in the connection establishment protocol, given in the previous section, is as follows:

$$RT \triangleq R|T$$
$$= \&req \bullet (R_1|\$) + ?inc' \bullet (R|\$)$$
$$R_1|\$ = !req' \bullet \$$$
$$R|\$ = \$ \text{ (progress error 1)}$$

END

$$IN \triangleq I|N$$
$$= \&inc \bullet (I_1|\$) + ?req' \bullet (I|\$)$$
$$I_1|\$ = !inc' \bullet \$$$
$$I|\$ = \$ \qquad \text{(progress error 2)}$$

END

$$INRT \triangleq RT|IN$$
$$= \&req \bullet IN|R_1 + \&inc \bullet I_1|RT$$
$$IN|R_1 = INR_1 = \&req' \bullet \$ + \&inc \bullet (I_1|R_1)$$
$$I_1|RT = I_1RT = \&inc' \bullet \$ + \&req \bullet (R_1|I_1)$$
$$I_1|R_1 = R_1|I_1 = \$ \qquad \text{(progress errors 3 and 4)}$$

From definition 3.3, the first two progress errors discovered above are found to indicate two deadlock errors since $CH(R) \cup CH(\$) = \{?req\}$ *and* $CH(I) \cup CH(\$) = \{?inc\}$. These deadlocks occur when the terminal and network are connected and the two channels are empty which is the normal final state of the protocol. Therefore, they are not undesirable errors but can be considered as proper terminations. The third and fourth progress errors discovered, however, indicate four unspecified reception errors since $CH(I_1) \cup CH(R_1) = \{!req', !inc'\}$. Unlike the deadlock errors, these are undesirable errors that occur because the specification of the protocol does not handle situations in which both the network and the terminal attempt to initialize a call concurrently (*call collisions*).

---

[2]Deadlock errors exhibited by a protocol are assumed subsequently to refer to deadlock of one or more protocol process. These deadlock errors then may not lead to a protocol deadlock as defined in section 2.3, where all the protocol processes are in deadlock.

The specification of the protocol can be revised to allow the protocol to recover from such call collisions. ETs for the revised terminal and network are shown in Fig. 3-5; the complete protocol specification is given in appendix 3.III.1. The revised specification also models successive connections where the terminal can send a terminate message (*!term*) to terminate a current connection; other connections can be initiated later. The specification of the network covers the possibility of receiving two successive terminating messages from the terminal. This is due to *premature termination of a connection*, which occurs when a call collision occurs and there is an outstanding terminate and incoming-call messages in the channels. This is caused by having a call collision occur, then the terminal terminating the connection before receiving the incoming-call message. Or, a terminate is sent when there is a chance of the network sending an incoming-call before receiving the terminal's call-request. The terminal assumes that this incoming-call message is a request for another connection and a second terminate is then required to end this second unnecessary connection.

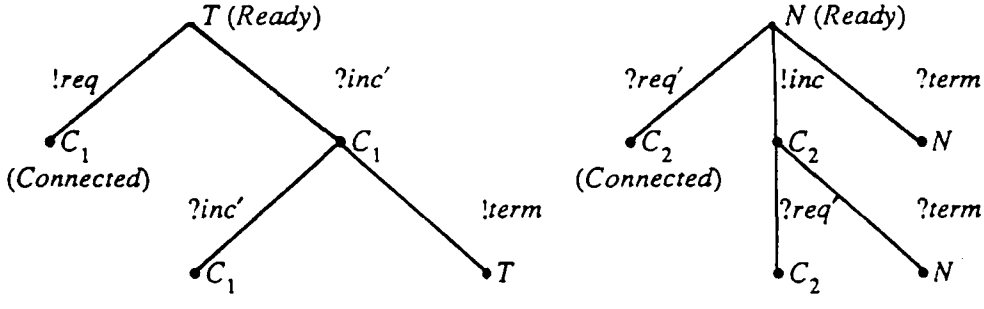


Figure 3-5: ETs of the terminal and network in the revised connection establishment protocol

3.5. Protocol Concurrent Behavior Can Be Computed Algebraically

The concurrent behavior of a protocol can be computed by concurrently composing the specifications of all the processes involved in it. For example, the concurrent behavior of the revised connection establishment protocol is obtained by concurrently composing its four processes *T*, *R*, *N*, and *I*. The resulting behavior, *INRT*, is partially shown in Fig. 3-6 and its algebraic specification is given in appendix 3.III.2. No progress errors were detected and therefore the revised protocol is free from any deadlock or unspecified reception errors. *INRT*

includes 25 equations and 48 summands. Most of these identifiers and summands are due to call collisions and premature terminations of connections. The performance of those behavior sequences belonging to *INRT* in which there are no call collisions or premature terminations will be analyzed in the chapter 4. However, it is necessary first to isolate these behaviors in order to be able to analyze them.

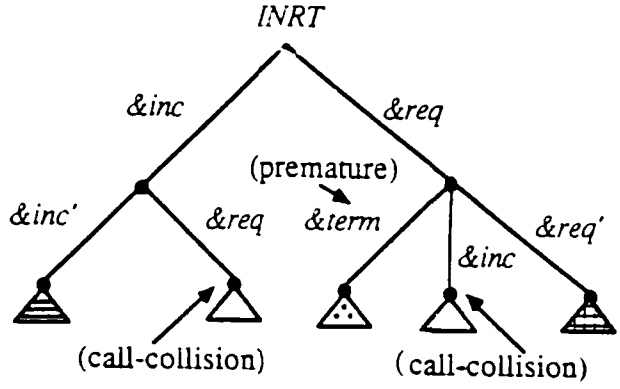


Figure 3-6: ET of the concurrent behavior *INRT*

3.6. A Protocol Designer Needs to Study Sub-Behaviors

Generally, a protocol designer may be interested not only in the entire concurrent behavior of a protocol, but also in some of its *sub-behaviors*. A sub-behavior of an ET is obtained by either pruning some of its branches or chopping some of its sub-trees by replacing them with a leaf indicating termination (\$). An example of the former are behaviors with no call collisions in the revised connection establishment protocol. An example of the latter are behaviors that terminate after the terminate message is delivered to the network (*term'*), and the four processes are in their initial state. These behaviors represent the execution of the protocol through one cycle.

Let us first classify expressions into *terminating* and *cyclic*. Let the set of *reachable identifiers* R(E) from an expression E denote the set of all identifiers that can be possibly reached in an execution of E. It can be defined recursively by $R(\sum_{i=1}^{n} a_i \bullet A_i) = \{A_1\} \cup R(A_1) \ldots \cup \{A_n\} \cup R(A_n)$, where $R(\$) = \varnothing$. An expression is said to be *terminating* if all identifiers in its reachable set are terminating. An identifier A is terminating if and only if one of its summands is of the form $a \bullet \$$, where a is any event, or of the form $a \bullet$B where B is terminating. Otherwise, A is *cyclic*. For example, the specification of the processes in the original connection establishment protocol,

given in section 3.2.1, are all terminating. They describe the behavior of the protocol for only one connection after which the processes terminate. Conversely, the specifications of the processes in the revised version of the protocol are all cyclic. They describe the behavior of the processes during the course of successive connections.

The concurrent behavior *INRT* of the revised connection establishment protocol is also cyclic describing several connections between the terminal and network processes. Suppose we are interested in its sub-behavior that describe the behavior of the protocol during the course of one cycle starting from the initial state *INRT* and ending with *&term'* when the protocol's state is *INRT*. Let this terminating behavior be denoted by $INRT_T$. The first function, *Terminate*, can be used to derive such behaviors. *Terminate* maps a given expression and some summand(s) of an expression, of the form $a \bullet A$, to another expression in which such summand(s) are changed to $a \bullet \$$. Thus, it can be used to map a cyclic expression to a terminating one.

Let Π represent the power set of a set that includes all pairs of events and identifiers (e, I), where $e \in \zeta$ and $I \in \Gamma$.

Definition 3.4

Terminate is a function: $\chi \times \Pi \rightarrow \chi$ such that

$$Terminate[\sum_{i=1}^{n} a_i \bullet A_i, P \in \Pi] = \sum_{i \neq j} a_i \bullet Terminate[A_i, P] + \sum_{\forall (a_j, A_j) \in P} a_j \bullet \$$$

For example, $INRT_T$ can be computed by *terminating INRT* with the delivery of terminate (*&term'*) to the network when all four processes are in their initial states (*INRT*). It can be specified in terms of the *Terminate* function as follows:

$$INRT_T = Terminate[INRT, \{(\&term', INRT)\}] \qquad (3.1)$$

An illustration of this mapping from *INRT* to $INRT_T$ is shown in Fig. 3-7. A complete listing of $INRT_T$ is given in appendix 3.III.3.

The *Precedence* function, defined below, will be used to map a given behavior into a sub-behavior by pruning some of the branches of its ET. This is performed based on given pairs of

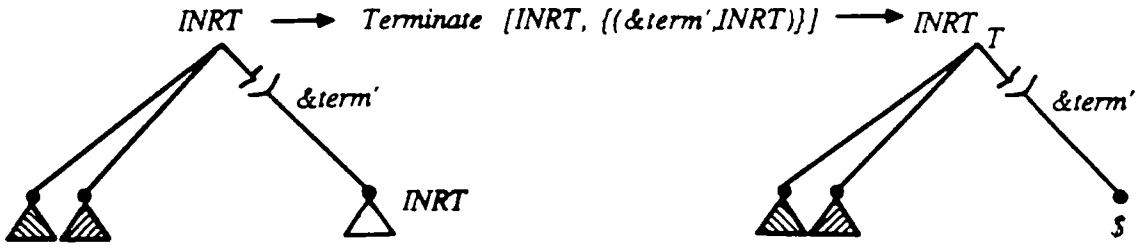


Figure 3-7: An illustration of the *Terminate* function

events such that whenever in the given behavior's ET any pair of events label outgoing branches from some node, then the branch of the second event in the pair is pruned. In such event pairs, the first event is said to *have precedence* over the second.

Let Φ denote the power set of a set of all event pairs.

Definition 3.5

Precedence is a function: $\chi \times \Phi \rightarrow \chi$ such that

$$Precedence[A=\sum_{i=1}^{n} a_i \bullet A_i, S \in \Phi] = \sum_{i=1, i \neq j}^{n} a_i \bullet Precedence[A_i, S] \qquad \forall (a_k, a_j) \in S; a_k, a_j \in CH(A)$$

The *Precedence* function can be used to compute two sub-behaviors of the concurrent behavior of the connection establishment protocol(*INRT*). In the first sub-behavior there are no call collisions, and in the second there are no premature terminations. Let $INRT_{P1}$ denote those sub-behaviors in which there are no call collisions. A call collision can be avoided if in $INRT_T$, *&req'* has precedence over *&inc*, and *&inc'* has precedence over *&req*. That is, if a call-request message has been sent by the terminal and pending delivery to the network (*&req'*), then the possibility of the network sending an incoming-call (*&inc*) is excluded. A similar explanation applies to the second precedence relation. Therefore, $INRT_{P1}$ is formally specified by

$$INRT_{P1} = Precedence[INRT_T, \{(\&inc', \&req), (\&req', \&inc)\}] \qquad (3.2)$$

An illustration of this mapping from $INRT_T$ to $INRT_{P1}$ is shown in Fig. 3-8. A complete listing of $INRT_{P1}$ is given in appendix 3.III.4.

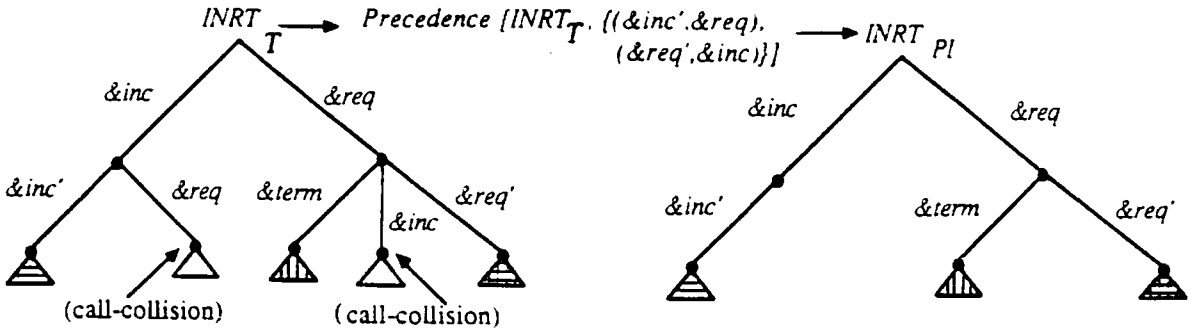


Figure 3-8: An illustration of the *Precedence* function

Let $INRT_{P2}$ denote the sub-behaviors of *INRT* in which no premature terminations of connections occur. As noted in the previous section, premature terminations are caused by those terminate messages that are issued by the terminal when there is a chance that there will be outstanding terminate and incoming-call messages in the channels at the same time. These situations are manifested in the behavior $INRT_T$ whenever events *&inc'* and *&term* are contending, or whenever events *&inc* and *&term* are contending since it might lead to the former. If the *&term* event occurs at such instants, it is premature because it always leads to behaviors where a second terminate message is required to end an unintentional second connection. $INRT_{P2}$ can thus be computed from $INRT_T$ by having both *&inc* and *&inc'* take precedence over *&term*. It is formally specified by

$$INRT_{P2} = Precedence[INRT_T, \{(\&inc,\&term),(\&inc',\&term)\}] \tag{3.3}$$

A complete listing of $INRT_{P2}$ is given in appendix 3.III.5.

Similar to the *Precedence* function, the *Restrict* function, defined below, will be also used to map a given behavior into a sub-behavior by pruning some of the branches of its ET. In the case of *Restrict*, the pruning is done based on given event-identifier pairs. Whenever in the given behavior's ET any event in a given pair labels an outgoing branch from some state other than the one denoted by the identifier associated with it, then this branch is pruned. In such even-identifier

pairs, the event is said to be *restricted* to occur only in its associated state.

Definition 3.6

Restrict is a function: $\chi \times \Pi \rightarrow \chi$ such that

$$Restrict[A = \sum_{i=1}^{n} a_i \bullet A_i, P \in \Pi] = \sum_{i \neq j} a_i \bullet Restrict[A_i, P] \qquad \forall\, (a_j, A_j) \in P \text{ and } A_j \neq A$$

For example, the ET corresponding to $A = \&a \bullet A_1 + \&b \bullet A$, *where* $A_1 = \&b \bullet A + \&a \bullet \$$, and the ET corresponding to $Restrict[A,\{(\&b,A_1)\}]$ are depicted in Fig. 3-9. Here event $\&b$ is restricted to occur only in state A_1.

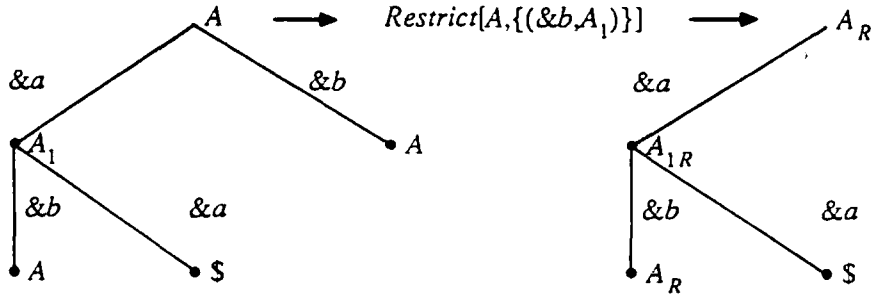


Figure 3-9: An illustration of the *Restrict* function

Note the following regarding the three functions introduced above:

- The *Precedence* function is a special case of the *Restrict* function. A behavior specified by the former with some precedence relations between pairs of events can be also specified by restricting the lower precedence events in the states where contention between event pairs does not occur. However, some behaviors (e.g., $INRT_{P1}$ in eq. 3.2) can be more naturally and concisely specified in terms of the *Precedence* function than the *Restrict* function.
- Given a protocol behavior, the functions can be used to map it to sub-behaviors that are often much smaller. A protocol designer, to save time and effort, may find it attractive in some instances to concentrate on these sub-behaviors instead of the complete protocol behavior. One example of such cases is considered in section 6.2.2.
- The specification obtained from applying one of the three functions to a process specification may have different properties from the given specification. For example, *Precedence* or *Restrict* may map a given terminating specification to a cyclic specification. Or, a progress error obtained from concurrent composition may not be reachable in the resulting specification if a branch leading to that error is

pruned, or its sub-tree is chopped, due to the application of these functions.

3.7. Summary

An algebra for specifying the communication behavior of protocol processes modeled by trees has been presented. Differences between the introduced algebra and both CCS and the algebra of regular events have been discussed. The algebra is shown to support the concurrent composition of processes. It is also shown how the concurrent behavior of a protocol can be automatically computed by concurrently composing the specifications of all the processes involved in it. During the expansion of these concurrent compositions, any deadlock or unspecified reception errors in the specifications of the protocol processes can be detected. A protocol specification has been defined to include algebraic specifications of its processes, and a specification of the *scope* of communications between them. Three functions: *Terminate*, *Precedence*, and *Restrict*, have been defined to be used in isolating interesting protocol sub-behaviors. In the next chapter, we will show how these functions can be used in specifying and analyzing the performance of a protocol.

A simple connection establishment protocol has been used to demonstrate how these concepts can be used in the first step of the methodology: to functionally specify and analyze protocols. This involves algebraic specification of the functional behavior of a protocol, computing its concurrent behavior, and possibly computing some of its sub-behaviors. If any deadlock or unspecified errors are detected during the concurrent compositions, few iterations of changing the specification of the protocol's configuration and/or its processes and then concurrently composing them again until they are free from errors, may be required. In the next chapter we will present the second step of the methodology. We will examine how the timing behavior of a protocol can be automatically extracted from its algebraic specification, and how performance can be formally specified and automatically analyzed.

Appendix 3.I. Equivalence of Expressions in the Algebra of ETs

Let derivative ∂ be a relation from $\zeta \times \chi \rightarrow \chi$ such that given an expression A, $\partial_e(A) = B$ if and only if there is a summand of A of the form $e \bullet B$, otherwise, it is undefined. Suppose we restrict expressions in the algebra of ETs to be *well formed*: expression $A = \sum_{i=1}^{n} a_i \bullet A_i$ is well formed if all events a_i, $i=1,...,n$ are distinct. Then, the derivative of such well formed expressions relative to

any event will be always unique, and the derivative would be a function of expressions and events.

Definition 3.7

Expressions A and B are *equivalent* (written as $A \equiv B$) if and only if $A \equiv_k B$, $\forall k \geq 0$, where

(i) $A \equiv_0 B$ always, and

(ii) $A \equiv_{k+1} B$, $k \geq 0$ if and only if $\forall a \in \zeta$:

(a) if $\partial_a(A)=A'$ then $\partial_a(B)=B'$ and $A' \equiv_k B'$

(b) if $\partial_b(B)=B'$ then $\partial_b(A)=A'$ and $A' \equiv_k B'$

This equivalence relation is the same as Milner's strong congruence relation defined in ([Miln 80], section 5.7). It is shown to be a *congruence* relation (theorem 5.4 [Miln 80]) meaning that if $A \equiv B$ then

(i) $a \bullet A \equiv a \bullet B$,

(ii) $A+C \equiv B+C$, and

(iii) $A \mid C \equiv B \mid C$.

Appendix 3.II. A Formal Definition of *Scope*

Definition 3.8

The *scope*(A,B) of communication between ETs A and B is defined as the set of *names* of events with which they communicate. It is assumed that the *scope* of communication between two processes is specified by the protocol designer. Let *parent*(A) denote the expression represented by the ET rooted at the parent node of A. The *scope* of two arbitrary expressions in the algebra of ETs is computed using the following rules:

(i) $scope(A,\$) = \varnothing$
(ii) $scope(I) = scope(E)$ for equation $I = E$
(iii) $scope(A,B) = scope(parent(A),parent(B))$
(iv) $scope(A,B) = scope(B,A)$
(v) $scope(A \mid B,C) = scope(A,C) \cup scope(B,C)$
(vi) $scope(A \mid B,C \mid D) = scope(A,C) \cup scope(B,C) \cup scope(A,D) \cup scope(B,D)$

Appendix 3.III. Algebraic Specifications of Behaviors of the Connection Establishment Protocol

3.III.1. Protocol Specification

The configuration of the revised protocol is depicted in Fig. 3-10. Specifications of the configuration of the protocol and its four processes follow.

PROTOCOL Connection Establishment : T,R,N,I

$scope(T,R) = \{req, term\}$
$scope(N,I) = \{inc\}$

$scope(R,N) = \{req', term'\}$
$scope(I,T) = \{inc'\}$

END

PROCESS T
$T = !req \bullet C_1 + ?inc' \bullet C_1$
$C_1 = ?inc' \bullet C_1 + !term \bullet T$
END

PROCESS N
$N = ?req' \bullet C_2 + !inc \bullet C_2 + ?term' \bullet N$
$C_2 = ?req' \bullet C_2 + ?term' \bullet N$
END

PROCESS R
$R = ?req \bullet R_1 + ?term \bullet R_2$
$R_1 = !req' \bullet R + ?term \bullet R_3$
$R_2 = !term' \bullet R$
$R_3 = !req' \bullet R_2$
END

PROCESS I
$I = ?inc \bullet I_1$
$I_1 = !inc' \bullet I$
END

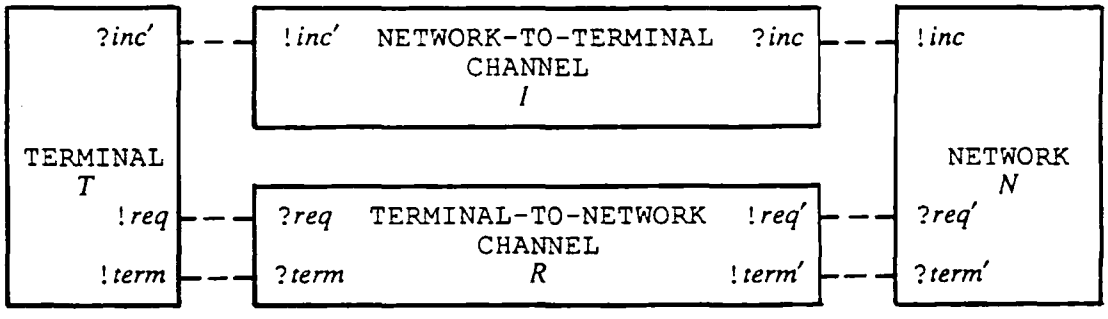


Figure 3-10: Configuration of the revised connection establishment protocol

3.III.2. Concurrent Behavior

The concurrent behavior of the revised connection establishment protocol obtained by concurrently composing its four processes: *T*, *R*, *N*, and *I* specified above is given by

$$
\begin{aligned}
INRT &= \&inc \bullet C_2RTI_1 + \&req \bullet C_1R_1NI \\
C_2RTI_1 &= \&req \bullet C_1R_1C_2I_1 + \&inc' \bullet C_1RC_2I \\
C_1R_1NI &= \&term \bullet INR_3T + \&req' \bullet C_1RC_2I + \&inc \bullet C_1R_1C_2I_1 \\
C_1R_1C_2I_1 &= \&inc' \bullet C_1R_1C_2I + \&term \bullet C_2R_3TI_1 + \&req' \bullet C_1RC_2I_1 \\
C_1RC_2I &= \&term \bullet C_2R_2TI \\
INR_3T &= \&inc \bullet C_2R_3TI_1 + \&req' \bullet C_2R_2TI \\
C_1R_1C_2I &= \&term \bullet C_2R_3TI + \&req' \bullet C_1RC_2I \\
C_2R_3TI_1 &= \&req' \bullet C_2R_2TI_1 + \&inc' \bullet C_1R_3C_2I \\
C_1RC_2I_1 &= \&inc' \bullet C_1RC_2I + \&term \bullet C_2R_2TI_1 \\
C_2R_2TI &= \&term' \bullet INRT \\
C_2R_3TI &= \&req' \bullet C_2R_2TI \\
C_2R_2TI_1 &= \&term' \bullet I_1NRT + \&inc' \bullet C_1R_2C_2I \\
C_1R_3C_2I &= \&req' \bullet C_1R_2C_2I \\
I_1NRT &= \&inc' \bullet C_1RNI + \&req \bullet C_1R_1NI_1 \\
C_1R_2C_2I &= \&term' \bullet C_1RNI \\
C_1RNI &= \&term \bullet INR_2T + \&inc \bullet C_1RC_2I_1 \\
C_1R_1NI_1 &= \&inc' \bullet C_1R_1NI + \&term \bullet I_1NR_3T + \&req' \bullet C_1RC_2I_1 \\
INR_2T &= \&inc \bullet C_2R_2TI_1 + \&term' \bullet INRT \\
I_1NR_3T &= \&inc' \bullet C_1R_3NI + \&req' \bullet C_2R_2TI_1 \\
C_1R_3NI &= \&req' \bullet C_1R_2C_2I + \&inc \bullet C_1R_3C_2I_1 \\
C_1R_3C_2I_1 &= \&inc' \bullet C_1R_3C_2I + \&req' \bullet C_1R_2C_2I_1 \\
C_1R_2C_2I_1 &= \&inc' \bullet C_1R_2C_2I + \&term' \bullet C_1RNI_1 \\
C_1RNI_1 &= \&inc' \bullet C_1RNI + \&term \bullet I_1NR_2T \\
I_1NR_2T &= \&inc' \bullet C_1R_2NI + \&term' \bullet I_1NRT \\
C_1R_2NI &= \&term' \bullet C_1RNI + \&inc \bullet C_1R_2C_2I_1
\end{aligned}
$$

3.III.3. Terminating Behavior

$INRT_T$ defined in eq. 3.1 and which represents the behavior of the connection establishment protocol for one cycle is given by:

$$
\begin{aligned}
INRT_T &= \&inc \bullet C_2RTI_{1T} + \&req \bullet C_1R_1NI_T \\
C_2RTI_{1T} &= \&req \bullet C_1R_1C_2I_{1T} + \&inc' \bullet C_1RC_2I_T \\
C_1R_1NI_T &= \&term \bullet INR_3T_T + \&req' \bullet C_1RC_2I_T \\
&\quad + \&inc \bullet C_1R_1C_2I_{1T} \\
C_1R_1C_2I_{1T} &= \&inc' \bullet C_1R_1C_2I_T + \&term \bullet C_2R_3TI_{1T} \\
&\quad + \&req' \bullet C_1RC_2I_{1T} \\
C_1RC_2I_T &= \&term \bullet C_2R_2TI_T \\
INR_3T_T &= \&inc \bullet C_2R_3TI_{1T} + \&req' \bullet C_2R_2TI_T \\
C_1R_1C_2I_T &= \&term \bullet C_2R_3TI_T + \&req' \bullet C_1RC_2I_T \\
C_2R_3TI_{1T} &= \&req' \bullet C_2R_2TI_{1T} + \&inc' \bullet C_1R_3C_2I_T \\
C_1RC_2I_{1T} &= \&inc' \bullet C_1RC_2I_T + \&term \bullet C_2R_2TI_{1T} \\
C_2R_2TI_T &= \&term' \bullet \$ \\
C_2R_3TI_T &= \&req' \bullet C_2R_2TI_T \\
C_2R_2TI_{1T} &= \&term' \bullet I_1NRT_T + \&inc' \bullet C_1R_2C_2I_T \\
C_1R_3C_2I_T &= \&req' \bullet C_1R_2C_2I_T \\
I_1NRT_T &= \&inc' \bullet C_1RNI_T + \&req \bullet C_1R_1NI_{1T} \\
C_1R_2C_2I_T &= \&term' \bullet C_1RNI_T \\
C_1RNI_T &= \&term \bullet INR_2T_T + \&inc \bullet C_1RC_2I_{1T} \\
C_1R_1NI_{1T} &= \&inc' \bullet C_1R_1NI_T + \&term \bullet I_1NR_3T_T \\
&\quad + \&req' \bullet C_1RC_2I_{1T} \\
INR_2T_T &= \&inc \bullet C_2R_2TI_{1T} + \&term' \bullet \$ \\
I_1NR_3T_T &= \&inc' \bullet C_1R_3NI_T + \&req' \bullet C_2R_2TI_{1T} \\
C_1R_3NI_T &= \&req' \bullet C_1R_2C_2I_T + \&inc \bullet C_1R_3C_2I_{1T} \\
C_1R_3C_2I_{1T} &= \&inc' \bullet C_1R_3C_2I_T + \&req' \bullet C_1R_2C_2I_{1T} \\
C_1R_2C_2I_{1T} &= \&inc' \bullet C_1R_2C_2I_T + \&term' \bullet C_1RNI_{1T} \\
C_1RNI_{1T} &= \&inc' \bullet C_1RNI_T + \&term \bullet I_1NR_2T_T \\
I_1NR_2T_T &= \&inc' \bullet C_1R_2NI_T + \&term' \bullet I_1NRT_T \\
C_1R_2NI_T &= \&term' \bullet C_1RNI_T + \&inc \bullet C_1R_2C_2I_{1T}
\end{aligned}
$$

3.III.4. Behavior With no Call Collisions

$INRT_{P1}$ specified in eq. 3.2 and which represents behaviors with no call collisions is given by

$$
\begin{aligned}
INR_{2P1} &= \&inc \bullet C_2RTI_{1P1} + \&req \bullet C_1R_1NI_{P1} \\
C_2RTI_{1P1} &= \&inc' \bullet C_1RC_2I_{P1} \\
C_1R_1NI_{P1} &= \&term \bullet INR_2T_{P1} + \&req' \bullet C_1RC_2I_{P1} \\
C_1RC_2I_{P1} &= \&term \bullet C_2R_2TI_{P1} \\
INR_2T_{P1} &= \&req' \bullet C_2R_2TI_{P1} \\
C_2R_2TI_{P1} &= \&term' \bullet \$
\end{aligned}
$$

3.III.5. Behavior With no Premature Terminations

$INRT_{P2}$ specified in eq. 3.3 and which represents behaviors with no premature termination of connections is given by

$$
\begin{aligned}
INRT_{P2} &= \&inc \bullet C_2RTI_{1\,P2} + \&req \bullet C_1R_1NI_{P2} \\
C_2RTI_{1\,P2} &= \&req \bullet C_1R_1C_2I_{1\,P2} + \&inc' \bullet C_1RC_2I_{P2} \\
C_1R_1NI_{P2} &= \&req' \bullet C_1RC_2I_{P2} + \&inc \bullet C_1R_1C_2I_{1\,P2} \\
C_1R_1C_2I_{1\,P2} &= \&inc' \bullet C_1R_1C_2I_{P2} + \&req' \bullet C_1RC_2I_{1\,P2} \\
C_1RC_2I_{P2} &= \&term \bullet C_2R_2TI_{P2} \\
C_1R_1C_2I_{P2} &= \&term \bullet C_2R_3TI_{P2} + \&req' \bullet C_1RC_2I_{P2} \\
C_1RC_2I_{1\,P2} &= \&inc' \bullet C_1RC_2I_{P2} \\
C_2R_2TI_{P2} &= \&term' \bullet \$ \\
C_2R_3TI_{P2} &= \&req' \bullet C_2R_2TI_{P2}
\end{aligned}
$$

Chapter 4

Protocol Performance Specification and Analysis

4.1. Introduction

In this chapter, an automated approach to performance analysis of protocols based on their formal algebraic specifications is introduced. Rules are devised to map an algebraic specification of a protocol, and the probability distributions of its events times, to probability and time attributes of its timing model. Protocol performance can then be *formally* specified in terms of these attributes and *automatically* analyzed. Two aspects of protocol performance are addressed: timing requirements and performance measures. Timing requirements of a protocol are conditions that have to be met by its timing behavior to ensure efficient performance. Their analysis leads to the evaluation of optimal settings of protocol performance parameters, such as the event rates. Performance measures are indications of how well a protocol performs. They are automatically analyzed using the rules for evaluating the probability and time attributes, and possibly queueing theory. The main contributions of this chapter include: developing a model for protocol timing behavior, devising rules for automatically computing key attributes of this model, and employing these rules to formally specify and automatically analyze protocol performance.

The timing behavior of protocols is modeled as a marked point process in section 4.2. Key attributes of this model are defined, and rules for evaluating them are presented in section 4.3. In section 4.4, the specification and analysis of timing requirements and performance measures of protocols using the timing attributes, is demonstrated. As an example, the performance of the connection establishment protocol of section 3.4, is specified and analyzed. An upper bound on the rate of terminating connections is computed, and the probability of call collisions is analyzed. Finally, a summary of the issues introduced in this chapter, and outline of how they can be used in the second step of the methodology: performance specification and analysis, are presented in section 4.5.

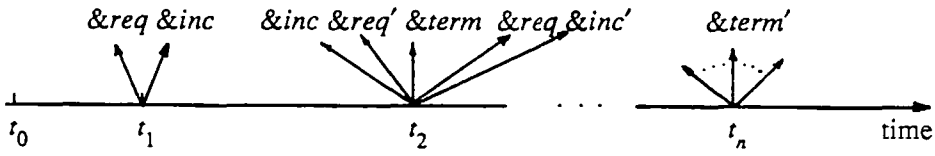
4.2. A Timing Model of Protocols

Consider the timing of a protocol behavior specified by a set of algebraic equations of the form $A = \sum_{i=1}^{n} a_i \bullet A_i$. Assume that all the expressions are *well-formed*: expression $A = \sum_{i=1}^{n} a_i \bullet A_i$ is well formed if all events a_i, $i=1,...,n$ are distinct. Subsequently, all expressions are assumed to be well-formed. The *derivative* of such well formed expressions A relative to an event a, denoted by $\partial_a(A)$, has been defined in appendix 3.I to be equal to A' if and only if there is a summand of A of the form $a \bullet A'$; otherwise, it is undefined.

Let us observe the timing behavior of A starting from some initial time t_0. At this time the choice set $CH(A) = \{a_1,...,a_n\}$ is said to be *enabled*, meaning that any event belonging to this choice set may occur at the next occurrence time t_1. If event a_i occurs at t_1, then the choice set $CH(\partial_i(A))$ is enabled. This continues until termination of A when the choice set $CH(\$) = \varnothing$ is enabled and no events can possibly occur. Generally, if an expression E_{i-1} is observed at t_{i-1} ($E_0 = A$), then any event $a \in CH(E_{i-1})$ may occur at t_i. The choice set $CH(\partial_a(E_{i-1}))$ is then enabled at t_i. Only events belonging to a choice set enabled at t_{i-1} can occur at t_i. Also, only one choice set is enabled at any time instant. Note that each occurrence time t_i has a collection of expressions and choice sets associated with it for every event that may occur at t_{i-1}.

Such protocol timing behavior can be modeled as a *marked point process* [Snyd 75] [$\{t_i, m_i\}, i \geq 0$], where t_i is the i-th occurrence time and m_i is the i-th *mark* denoting the set of possible events that may occur at t_i. Each mark m_i is a collection of choice sets $CH(E_{i-1})$ for each expression E_{i-1} associated with t_{i-1} ($m_0 = \varnothing$). For example, the occurrence times and marks of the timing of the connection establishment protocol's terminating behavior $INRT_T$ (whose ET is depicted in Fig. 3-7) is shown in Fig. 4-1. The figure describes the timing behavior of the protocol during the course of one cycle.

Let us assume that the occurrence times of a protocol timing behavior are *continuous random variables*. Computing statistics of these occurrence times is very complex because they are dependent on the marks associated with them and the event that occurred at the previous occurrence time. The computations can be simplified by assuming that the occurrence time of an event, is *independent* of the other events in it's choice set. The occurrence time of an event is measured from the time its choice set is enabled until it occurs.



$m_0=\varnothing$

$m_1=\{\{\&req,\&inc\}\}$

$m_2=\{\{\&inc,\&req',\&term\},\{\&req,\&inc'\}\}$

$m_n=\{\{\&term',...\}...\}$

Figure 4-1: The timing behavior of $INRT_T$

Suppose that expression C is observed at some time t_{i-1}. Let τ_a denote the occurrence time of event $a \in CH(C)$ measured from t_{i-1}, and $F_a(t)$ denote the probability distribution of τ_a. Then, the probability that one such event a occurs at t_i, and the mean and variance of the time until it occurs are given by the following lemma.

Lemma 4.1

1. The probability that event $a \in CH(C)$ occurs at t_i conditioned on the fact that CH(C) is enabled at t_{i-1}, is given by

$$\rho_C(a) = \int_0^\infty \prod_{a_i \in CH(C),\, a_i \neq a} [1 - F_{a_i}(t)]\, dF_a(t) \tag{4.1}$$

2. The time duration from t_{i-1} when CH(C) is enabled until $a \in CH$(C) occurs (at t_i) has a conditional mean, denoted by $\mu_C(a)$, and variance, denoted by $\sigma_C(a)$, which are given by

$$\mu_C(a) = \int_0^\infty \prod_{a_i \in CH(C)} [1 - F_{a_i}(t)]\, dt \tag{4.2}$$

$$\sigma_C(a) = \int_0^\infty t \prod_{a_i \in CH(C)} [1 - F_{a_i}(t)]\, dt - \mu_C^2(a) \tag{4.3}$$

Proof: The complete proof is given in appendix 4.I. An outline of the proof is as follows. The

events belonging to a choice set $CH(C)$ enabled at t_{i-1} are actually *contending* with each other for occurrence at t_i. However, only one of them can occur since when one occurs, for example a, it then enables the choice set $CH(\partial_a(C))$. This implicitly means that the choice set $CH(C)$ is disabled since only one choice set can be enabled at any one time. As a consequence of the contention between the events for occurrence at t_i and their mutual exclusion, the conditional probability of a occurring is given by

$$\rho_C(a) = \Pr(_a \text{ is } Min\{\tau_{a_j}, \forall a_j \in CH(C)\}) \tag{4.4}$$

This is shown in appendix 4.I to lead to the first equation in the lemma. From the contention of events for an occurrence time and their mutual exclusion, the time duration from t_{i-1} to t_i is given by

$$t_i - t_{i-1} = Min\{\tau_{a_j}, \forall a_j \in CH(C)\} \tag{4.5}$$

The mean and variance of this is shown in appendix 4.I to lead to the second and third equations in the lemma, respectively.

4.3. Attributes of a Protocol Timing Model

We are interested in distinguishing those attributes of a protocol timing model that are often required in specifying protocol performance. Let C and A be terminating and well-formed expressions such that A represents a subset of the summands of C, to be denoted by *Sum*(C). A *probability attribute* $P_C(A)$ denotes the conditional probability of A occurring relative to the sample space *Sum*(C), given that the choice set of C is enabled. Let the *time duration* of A relative to C be defined as the length of time starting from when $CH(C)$ is enabled until A terminates. A *mean-time attribute* $M_C(A)$, and a *variance-time attribute* $V_C(A)$ are the mean and variance of the duration time of A relative to C, respectively.

Note that each of these attributes is a function from terminating and well-formed expressions in the algebra of ETs, denoted by $\chi_t \subset \chi$, to positive real numbers: $\chi_t \times \chi_t \rightarrow \Re^+$, (In the case of the probability attribute, the real numbers are bounded by 0 and 1.) Theorems 4.1, 4.2, 4.3 given next define mappings from operations in the algebra of ETs to arithmetics operations on the probability, mean-time, and variance-time attributes, respectively. The proofs of the theorems are

given in appendix 4.I. Similar rules have been devised for directed graphs [Elma 64, Beiz 70] although they are defined in an algorithmic manner.

Theorem 4.1

$$\text{P1.}\quad P_C(a \bullet A) = \rho_C(a) \cdot P_{\partial_a(C)}(A)$$

$$\text{P2.}\quad P_C(\textstyle\sum_{i=1}^{n} a_i \bullet A_i) = \displaystyle\sum_{i=1}^{n} P_C(a_i \bullet A_i)$$

Theorem 4.2

$$\text{M1.}\quad M_C(a \bullet A) = \mu_C(a) + M_{\partial_a(C)}(A)$$

$$\text{M2.}\quad M_C(\textstyle\sum_{i=1}^{n} a_i \bullet A_i) = \dfrac{\displaystyle\sum_{i=1}^{n} \rho_C(a_i) \cdot M_C(a_i \bullet A_i)}{\displaystyle\sum_{i=1}^{n} \rho_C(a_i)}$$

The only assumptions made so far regarding the timing behavior of protocols are that the occurrence times are continuous random variables and that the events occurrence times of contending events are independent. No assumptions were made regarding the kind of their probability distribution. Hence, the above theorems for computing the probability and mean-time attributes of protocol timing model can be applied to any distribution of the events occurrence times for which a first moment exists. However, with such a general assumption, the rules for computing the variance-time attribute would be considerably complex; in particular, the computation of $V_C(a \bullet A)$. Therefore, for the sake of simplifying these computations, the occurrence times of events will be assumed throughout the rest of the dissertation to be *exponentially* distributed. Note, that such exponentially distributed occurrence times of event are independent on the time from which they are enabled. Consequently, a particular event would have just one value for the rate for its occurrence time.

Theorem 4.3

$$\text{V1.}\quad V_C(a \bullet A) = \sigma_C(a) + V_{\partial_a(C)}(A)$$

$$\text{V2.}\quad V_C(\sum_{i=1}^{n} a_i \bullet A_i) = \frac{\sum_{i=1}^{n} \rho_C(a_i) \cdot [V_C(a_i \bullet A_i) + M_C^2(a_i \bullet A_i)]}{\sum_{i=1}^{n} \rho_C(a_i)} - M_C^2(\textstyle\sum_{i=1}^{n} a_i \bullet A_i)$$

Let λ_a denote the exponential rate of event a. In terms of the exponential rates of the occurrence time of events, $\rho_c(a)$, $\mu_c(a)$, and $\sigma_c(a)$ of lemma 4.1 are reduced to

$$\rho_C(a) = \frac{\lambda_a}{\sum_{a_i \in CH(C)} \lambda_{a_i}} \quad \text{if and only if } a \in CH(C) \quad (4.6)$$

$$\mu_C(a) = \frac{1}{\sum_{a_i \in CH(C)} \lambda_{a_i}} \quad \text{if and only if } a \in CH(C) \quad (4.7)$$

$$\sigma_C(a) = \frac{1}{(\sum_{a_i \in CH(C)} \lambda_{a_i})^2} \quad \text{if and only if } a \in CH(C) \quad (4.8)$$

By inspection, it is clear that the rules for computing the timing attributes of given well-formed, terminating expressions respect axioms A1-A3 of the algebra of ETs given in section 3.2.2. That is, given $A \equiv B$, then $P_C(A)=P_C(B)$, $M_C(A)=M_C(B)$, and $V_C(A)=V_C(B)$. This is not true though for Axiom A4 ($A+A \equiv A$) since the timing attributes are not defined for $A+A$ which is not well-formed.

An interesting result of theorems 4.1, 4.2, and 4.3, is that given a recursively defined expression $A = X \bullet A + B$, the number of repetitions of behavior X is a random variable with a modified geometric distribution.

Corollary 4.1

Let $A = a \bullet A + B$, then

$$M_A(A) = \frac{\rho_A(a)}{[1-\rho_A(a)]} \cdot \mu_A(a) + M_A(B) \tag{4.9}$$

$$V_A(A) = \frac{\rho_A(a)}{[1-\rho_A(a)]} \cdot \sigma_A(a) + \frac{\rho_A(a)}{[1-\rho_A(a)]^2} \cdot \mu_A^2(a) + V_A(B) \tag{4.10}$$

A proof is given in appendix 4.III. The corollary can be easily generalized for any recursive expression $A = X \bullet A + B$, where X is a summation of a sequence of events.

Given a set of algebraic equations describing a protocol behavior, the rules in theorems 4.1, 4.2, and 4.3 can be used to map them into a set of linear equations. In these linear equations, the variables are the attributes and the coefficients are arithmetic expressions in the event rates. Note that since the values of mean-time attributes are required in order to compute a variance-time attribute of V2 in theorem 4.3, these means should be first computed and then used as constants in the equations of the variance-time attribute.

It should be also noted that if the rates of certain (*critical*) events in a given terminating expression are set to zero such that the expression becomes cyclic, then any timing attribute of this expression is by definition undefined. This is manifested when solving a set of linear equations in an attribute, by either having no unique solution to the equations or getting an invalid solution. In the case of computing the mean $M_X(X)$ or variance $V_X(X)$ for some expression X, if such critical events are set to zero causing X to be cyclic, then always the set of linear equations obtained in the attribute will not have a unique solution. One example is $X = a \bullet X + b \bullet \$$ and $\lambda_b = 0$. In the case of probability attribute $P_C(A)$, or conditional mean $M_C(A)$ or variance $V_C(A)$, if the rates of such critical events are zero, then either the equations in the attribute has no unique solution, or a solution is obtained but due to the violation of the attributes' definitions is invalid. For example, consider computing $P_C(A)$ for

$$C = a \bullet C + b \bullet D \qquad D = c \bullet \$ + d \bullet C + e \bullet C$$

$$A = a \bullet A + b \bullet B \qquad B = c \bullet \$ + e \bullet A$$

The critical events here, by inspection, are b and c. If $\lambda_b = 0$ or $\lambda_c = \lambda_d = 0$, then there is no

unique solution for the set of linear equations in the probability attribute. However, if only $\lambda_c = 0$, but λ_b and λ_d are non-zero, then by solving the equations we get $P_C(A) = 0$. But this solution is unacceptable since by definition, the attributes can be computed using the rules of theorems 4.1,4.2, and 4.3 only if the given expressions are terminating.

4.4. Specification and Analysis of Timing Requirements and Performance Measures

Any timing requirement or performance measure that can be specified in terms of the attributes of the timing model of protocols, can be computed using the mapping rules of theorems 4.1, 4.2, and 4.3, and lemma 4.1. Only an algebraic specification of the protocol and the exponential rates of events involved in it are used in these computations. As an example, let us analyze some aspects of the performance of the revised connection establishment protocol.

Two interesting phenomenon were exhibited by the connection establishment protocol of section 3.4. First, call collisions where both the terminal and the network attempt to concurrently initialize a connection. Second, premature terminations where after a call collision, a terminal sends a terminate message before receiving the incoming-call message from the network to initialize this same connection. Or, a terminate is sent when there is a chance of the network sending an incoming-call before receiving the terminal's call request. Then, there is a combination of an outstanding terminate and incoming-call message in the channels leading to a similar situation as the former. It was shown in section 3.4 that such premature terminations cause the terminal to erroneously interpret the incoming-call message when it arrives as a request for a second connection. By varying the rate of terminating messages, the probability of such premature terminations can be limited to a very small value. A timing requirement that captures this objective is formally specified in section 4.4.1; its analysis yields an upper bound on the rate of terminations. The probability of call collisions is also formally specified and automatically analyzed in section 4.4.2.

4.4.1. A Timing Requirement of the Connection Establishment Protocol: Minimize Probability of Premature Terminations

The *probability of premature terminations*, to be denoted by p_t, is the probability that a call collision occurs and there is an outstanding terminate message in the channel. Premature terminations of connections cause the transmission of unnecessary messages since a second terminate is always required to end the second erroneous connection. This may contribute to congestion in the network and overflow of buffers. In addition premature terminations cause unsynchronized operation of the terminal and network: the former assumes that two connections were set up (one initialized by it and the other by the network), whereas the latter assumes just one connection. Such problems are avoided in more sophisticated connection establishment protocols such as the three way handshake [Post 79] where acknowledgments are used to ensure that both processes are in the connected state before proceeding with their operation. Note that the connection establishment protocol we are studying is a one way handshake.

Our objective is then to compute the rate of terminating connections $\lambda_{\&term}$ that would limit p_t to a small value ε as specified in the following timing requirement:

$$p_t \leq \varepsilon \tag{4.11}$$

Since $INRT_{P2}$ (whose algebraic specification is given in appendix 3.III.5) represents the sub-behaviors of the protocol (whose terminating behavior $INRT_T$ is given in appendix 3.III.3) without premature terminations, p_t can be specified by

$$p_t = 1 - \mathrm{P}_{INRT_T}(INRT_{P2}) \tag{4.12}$$

p_t is plotted versus $\lambda_{\&term}$ in Fig. 4-2 for two different values of the mean communication delay between the terminal and network which is equal to $1/\lambda_{\&req'} = 1/\lambda_{\&inc'} = 1/\lambda_{\&term'}$. The figure shows that as the delay in the channels increases, p_t increases. This can be explained as follows: a large delay means more time for incoming-call to arrive at the terminal and thus a higher probability of sending a terminate before it arrives. For the given data and $\varepsilon = 0.01$, the upper bound on the rate of terminations with an accuracy of 2 decimal place is given by

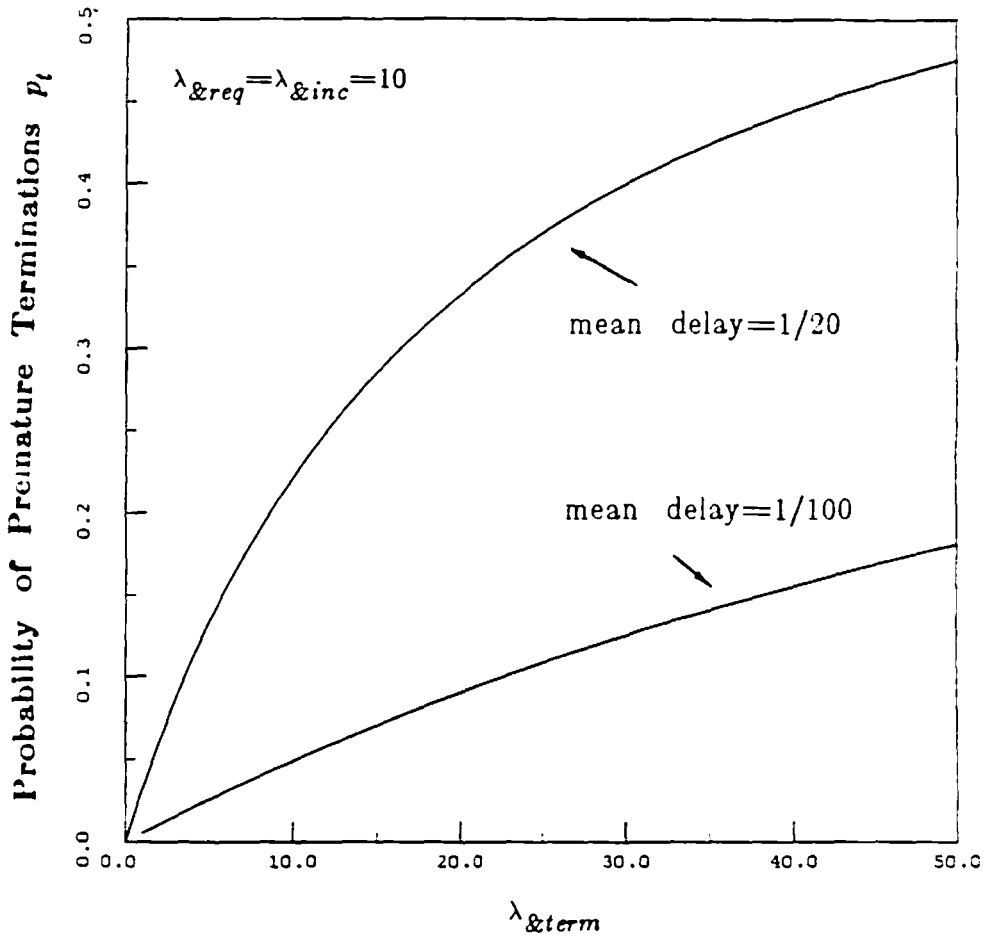


Figure 4-2: The probability of premature terminations versus $\lambda_{\&term}$ for two different values of mean delay

$\lambda_{\&term} \leq 0.3$ occurrences/sec. for mean delay = 1/20 (4.13)

$\lambda_{\&term} \leq 1.86$ occurrences/sec. for mean delay = 1/100 (4.14)

4.4.2. A Performance Measure of the Connection Establishment Protocol: Probability of Call Collisions

The *probability of call collisions*, denoted by p_c, is the probability that both the terminal and network attempt concurrently to initialize a connection, i.e., a call collision occurs. A high p_c would degrade the performance of the protocol in two respects. First, since premature terminations may occur after a call collision and in such situations a second termination of the same connection

is required, then the time to terminate a connection would increase. Second, assuming that the call-request and incoming-call messages carry information for initializing a call, such as channel number, some data transferred before the terminal and network synchronize such set-up information would be lost. Again this particular hazard is eliminated in more sophisticated connection establishment protocol.

Since $INRT_{P1}$ (whose ET is given in Fig. 3-8 and algebraic specification is given in appendix 3.III.4) represents the sub-behaviors of the protocol's terminating behavior $INRT_T$ in which no call collisions occur, p_c is formally specified by

$$p_c = 1 - \mathrm{P}_{INRT_T}(INRT_{P1}) \tag{4.15}$$

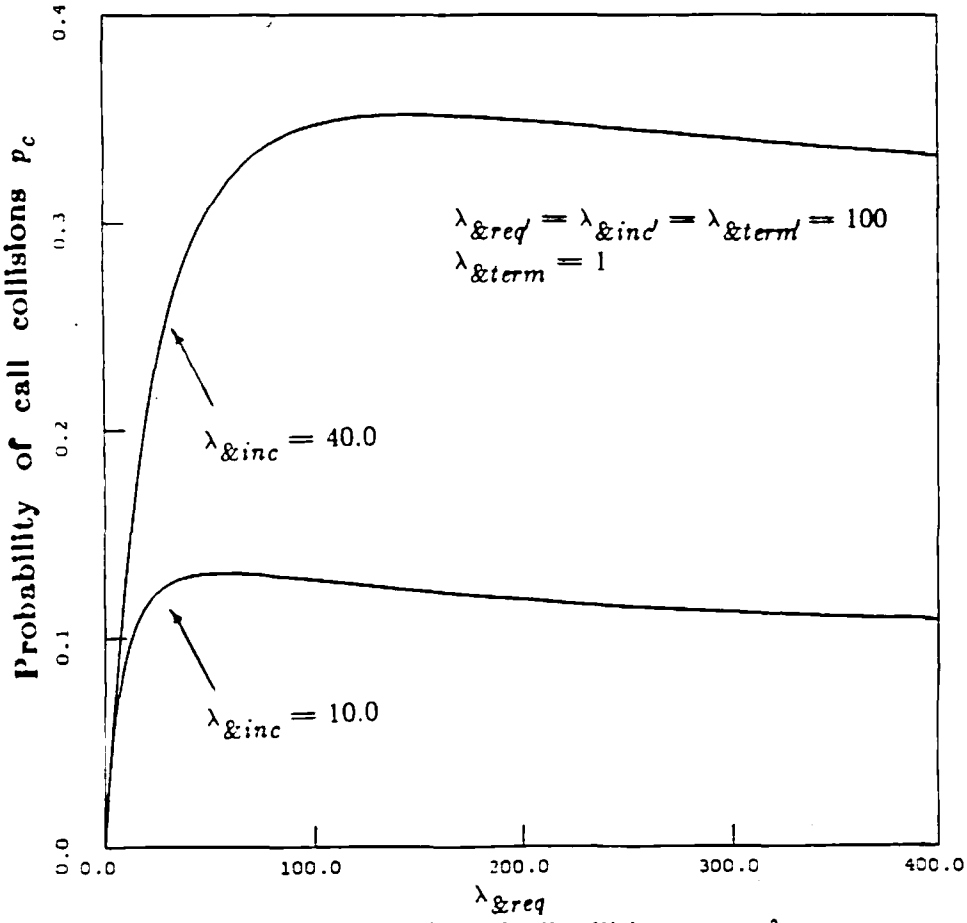


Figure 4-3: The probability of call collisions versus $\lambda_{\&req}$ for two different values of $\lambda_{\&inc}$

In Fig. 4-3, p_c is plotted versus $\lambda_{\&req}$ for two different values of $\lambda_{\&inc}$. The figure shows that the probability of collisions is saturated for a wide range of $\lambda_{\&req}$. It decreases when the ratio $\lambda_{\&req}/\lambda_{\&inc}$ is very low or very high.

4.5. Summary

Protocol timing behavior has been modeled as a marked point process. Probability, mean-time, and variance-time attributes of this timing model have been defined. Rules for mapping operations in the algebra of ETs into operations on the attributes have been given. To simplify the computations involved in evaluating the attributes, all occurrence times of events are assumed to be exponentially distributed.

Using a connection establishment protocol, the second step in the methodology of specification-based performance analysis of protocol has been described. It constitutes formally specifying protocol timing requirements and performance measures in terms of the probability, mean-time, or variance time attributes, and automatically analyzing them. This analysis uses the algebraic specification of the protocol and the exponential rates of events involved in it. A timing requirement necessary for the efficient performance of the connection protocol has been specified as an inequality relation that sets an upper bound on the probability of behaviors with no premature termination of connections. The analysis of this timing requirement has resulted in the computation of an upper bound on the rate of terminating connections. In addition, one performance measure of the protocol, the probability of call collisions has been formally specified and analyzed. It has been shown that this probability becomes large when the rates of call-requests issued by the terminal and incoming-calls issued by the network are comparably close; otherwise, it remains low.

Appendix 4.I. Proof of Lemma 4.1

From eq 4.4, the probability that event $a \in CH(C)$ occurs at some t_i, conditioned on the fact that $CH(C)$ has been enabled at t_{i-1}, is given by

$$\rho_C(a) = \Pr(\tau_a \text{ is } Min\{\tau_{a_j}, \forall a_j \in CH(C)\})$$

which can be also stated as

$$\rho_C(a) = \Pr(\tau_a < \tau_{a_j}) \quad \forall a_j \in (CH(C) - \{a\})$$

$$= \int_0^\infty \Pr(\tau_{a_j} > t \mid \tau_a = t)\, dF_a(t) \quad \forall a_j \in (CH(C) - \{a\})) \tag{4.16}$$

Using the independence assumption of the occurrence times of contending events, we get the first equation of the lemma:

$$\rho_C(a) = \int_0^\infty \prod_{j=1,\, a_j \neq a}^{n} [1 - F_{a_j}(t)]\, dF_a(t)$$

In order to prove the second and third equations of the lemma: the time duration from t_{i-1} when some $CH(C)$ is enabled until $a \in CH(C)$ occurs (at t_i) and whose mean and variance are denoted by $\mu_C(a)$ $\sigma_C(a)$, respectively, is shown in eq. 4.5 to be given by

$$t_i - t_{i-1} = Min\{\tau_{a_j},\ \forall a_j \in CH(C)\}$$

which has a probability distribution given by

$$\Pr(Min\{\tau_{a_j} \leq t,\ \forall a_j \in CH(C)\}) = 1 - \Pr(Min\{\tau_{a_j} > t,\ \forall a_j \in CH(C)\})$$
$$= 1 - \Pr(\tau_{a_j} > t,\ \forall a_j \in CH(C)) \tag{4.17}$$

From the independence assumption of the occurrence times of contending events, this is reduced to

$$1 - \prod_{a_j \in CH(C)} [1 - F_{a_j}(t)]$$

By computing the mean and variance of this time duration we get the second and third equations in the lemma, respectively.

Appendix 4.II. Proofs of Theorems 4.1, 4.2, and 4.3

- Proof of P1, M1, and V1:

  $P_C(a \bullet A)$ is the probability of a occurring and then A. Using Baye's rule. $P_C(a \bullet A)$ can be computed by multiplying the probability of a occurring, $\rho_C(a)$, with the conditional probability $P_{\partial_a(C)}(A)$ (which is conditioned on the occurrence of a since the choice set of $\partial_a(C)$ is only enabled after a occurs). This proves P1. M1 follows directly from the sequential composition of a and A. By the exponential assumption of the occurrence times of events, successive occurrence times are independent. From this and the sequential composition of a and A, V1 is proven.

- Proof of P2, M2, and V2:

  P2, M2, and V2 follow from the mutual exclusion of contending events in a choice set. M2 and V2 also follow from noting that the distribution of the time duration of $\sum_{\bullet i=1}^{\bullet n} a_i \bullet A_i$ is conditioned on the occurrence of any of the events a_i, $i=1,n$.

Appendix 4.III. Proof of Corollary 4.1

Let $A = a \bullet A + B$, $B = \sum_{j=1}^{m} b_j \bullet B_j$. The goal is to prove:

$$M_A(A) = \frac{\rho_A(a)}{[1-\rho_A(a)]} \cdot \mu_A(a) + M_A(B)$$

and

$$V_A(A) = \frac{\rho_A(a)}{[1-\rho_A(a)]} \cdot \sigma_A(a) + \frac{1}{[1-\rho_A(a)]} \cdot M_A^2(a) + V_A(B)$$

From M2 in theorem 4.2 we get

$$M_A(A) = \frac{\rho_A(a) \cdot M_A(a \bullet A) + (\sum_{j=1}^{m} \rho_A(b_j)) \cdot M_A(B)}{\rho_A(a) + \sum_{j=1}^{m} \rho_A(b_j)} \qquad (4.18)$$

Since from P2 in theorem 4.1

$$P_A(a) + \sum_{j=1}^{m} P_A(b_j]) = 1 \qquad (4.19)$$

and from M1 in theorem 4.2

$$M_A(a \bullet A) = \mu_A(a) + M_A(A) \tag{4.20}$$

then eq. 4.18 is reduced to

$$M_A(A) = \frac{\rho_A(a)}{[1-\rho_A(a)]} \cdot \mu_A(a) + M_A(B) \tag{4.21}$$

thus proving the first equation of corollary 4.1. To compute the variance in the second equation:

From V2 in theorem 4.3 we get

$$V_A(A) = \frac{\rho_A(a) \cdot [V_A(a \bullet A) + M_A^2(a \bullet A)] + (\sum_{j=1}^{m} \rho_A(b_j)) \cdot [V_A(B) + M_A^2(B)]}{\rho_A(a) + \sum_{j=1}^{m} \rho_A(b_j)} - M_A^2(A) \tag{4.22}$$

then by using M1 in theorem 4.2, V1 in theorem 4.3, eq. 4.19, and the first equation in corollary 4.1 just proven, we get

$$V_A(A) = \frac{\rho_A(a)}{[1-\rho_A(a)]} \cdot \sigma_A(a) + \frac{\rho_A(a)}{[1-\rho_A(a)]^2} \cdot \mu_A^2(a) + V_A(B) \tag{4.23}$$

thus proving the second equation of corollary 4.1.

Chapter 5

ANALYST: A Software Environment for Protocol Performance Analysis

5.1. Introduction

ANALYST is an implementation of the specification-based methodology for performance analysis of protocols presented in chapters 3 and 4. Compared to current protocol development environments, see for instance [Holz 84, Chow 85], the design of ANALYST is novel in two main respects. First, *it integrates functional and performance specification and analysis of protocols in one environment.* Since protocol performance is extracted automatically from its functional specification, which is augmented with rates of occurrence times of events, this integration allows a protocol designer to analyze protocol performance without requiring much expertise in the field. More specifically, a protocol designer is not required to engage in performance modeling of the protocol under analysis, but just to specify performance in terms of timing attributes of the protocol.

Second, *ANALYST facilitates and enhances the design process of protocols.* It supports an interactive user interface that allows the protocol designer to readily debug a protocol (and processes) specification(s) and iterate through functional and performance specification and analysis. Thus, the environment facilitates experimental protocol design where discovered design errors or results of predictions of protocol performance would necessitate variations in the protocol's (and processes') specification(s). It also provides the designer with a friendly and *uniform* user interface to the different modules that perform functional and performance analysis, i.e., the user does not have to explicitly switch from one module to the other to obtain different services. To assist the designer in understanding and debugging a protocol specification, the environment allows access to details of the steps performed automatically, and provides

information about the protocol and processes under analysis. Other protocol development environments typically support a collection of tools for functional specification and analysis of protocols. They do not support performance specification and analysis, nor provide a uniform user interface to all the supported tools.

The key functions supported by ANALYST, and how they can be used in analyzing protocol performance, are examined in the next section. A complete scenario of using ANALYST to analyze the performance of the connection establishment protocol of section 3.4, is presented in section 5.3. ANALYST's logical architecture, the key algorithms used in implementing the functions supported by each of its modules (tools), and their time and space complexities are described in section 5.4.

5.2. Using ANALYST to Analyze the Performance of Protocols

The key functions supported by ANALYST include:

- Maintenance of protocol specifications. The protocol designer can either load these from previously created files, or define them on-line. He can then edit these specifications, request information about them (such as the number of identifiers in a process specification), or save them in files.
- Concurrent composition of processes specifications as described in section 3.5. The protocol designer is notified of the presence, number, and origin of any deadlock or unspecified reception errors.
- Creation of new behaviors from given ones using the *Terminate*, *Precedence*, or *Restrict* functions defined in section 3.6. These new behaviors can be used in specifying timing requirements or performance measures of the protocol.
- Constrained concurrent composition. By this we mean that concurrent composition is constrained with the precedence or restriction relations among the events and identifiers involved. The result is not the complete concurrent behavior of the protocol, but a sub-behavior of it as specified. Thus, constrained concurrent composition can be one way of overcoming the explosion in the size of the concurrent behavior, if the designer is mostly interested in a sub-behavior of it. (A more detailed discussion is in section 5.4.2.)
- Evaluation of arithmetic expressions in which the variables can be the event rates or any of the timing attributes. Thus any timing requirement or performance measure expressed as an arithmetic expression in these variables can be analyzed. A "FOR" statement is also supported for evaluating a given arithmetic expression for several values of an index variable, which can be an event rate for example. This produces data that can be readily plotted.

The flow of activities throughout the interaction between a protocol designer and ANALYST are

depicted in Fig. 5-1. In this figure, the dashed lines denote automatically produced output and the continuous lines denote data provided by the protocol designer. Constrained concurrent composition is not shown explicitly in the figure; it is basically a combination of the two steps of concurrent composition and application of the *Precedence*, or *Restrict* functions to the concurrent behavior. Note that the step of applying the *Terminate*, *Precedence*, and *Restrict* functions is optional. The protocol designer is allowed to iterate between the steps shown in the figure.

ANALYST has been implemented in C and running on a VAX/750 under UNIX[3], 4.2BSD operating system. It has been used to automatically analyze the performance of the connection establishment protocol, the Alternating Bit protocol, and a two phase locking protocol. Performance results of the second and third protocols are given in chapters 6 and 7, respectively. A complete scenario of using ANALYST to analyze the performance of the connection establishment protocol is given next.

5.3. A Scenario of Using ANALYST for Performance Analysis of the Connection Establishment Protocol

The two main steps in the methodology for specification-based performance analysis of protocol have been described in chapters 3 and 4. They constitute functional specification and analysis followed by performance specification and analysis. In this section we show how a protocol designer follows these steps for the connection establishment protocol using ANALYST. The same steps followed have been described before when using this protocol as an example in chapters 3 and 4.

All input commands are shown in italics; otherwise, the rest are output from the environment. Comment lines are preceded by a ''#''. The exponential rate of an event *a* is represented here by ''~a''. We start by invoking the ANALYST environment.

% *analyst*

```
Welcome to ANALYST
```

ANALYST> *# ANALYST is now ready for user commands.*

[3]UNIX is a trademark of AT&T Bell Labs.

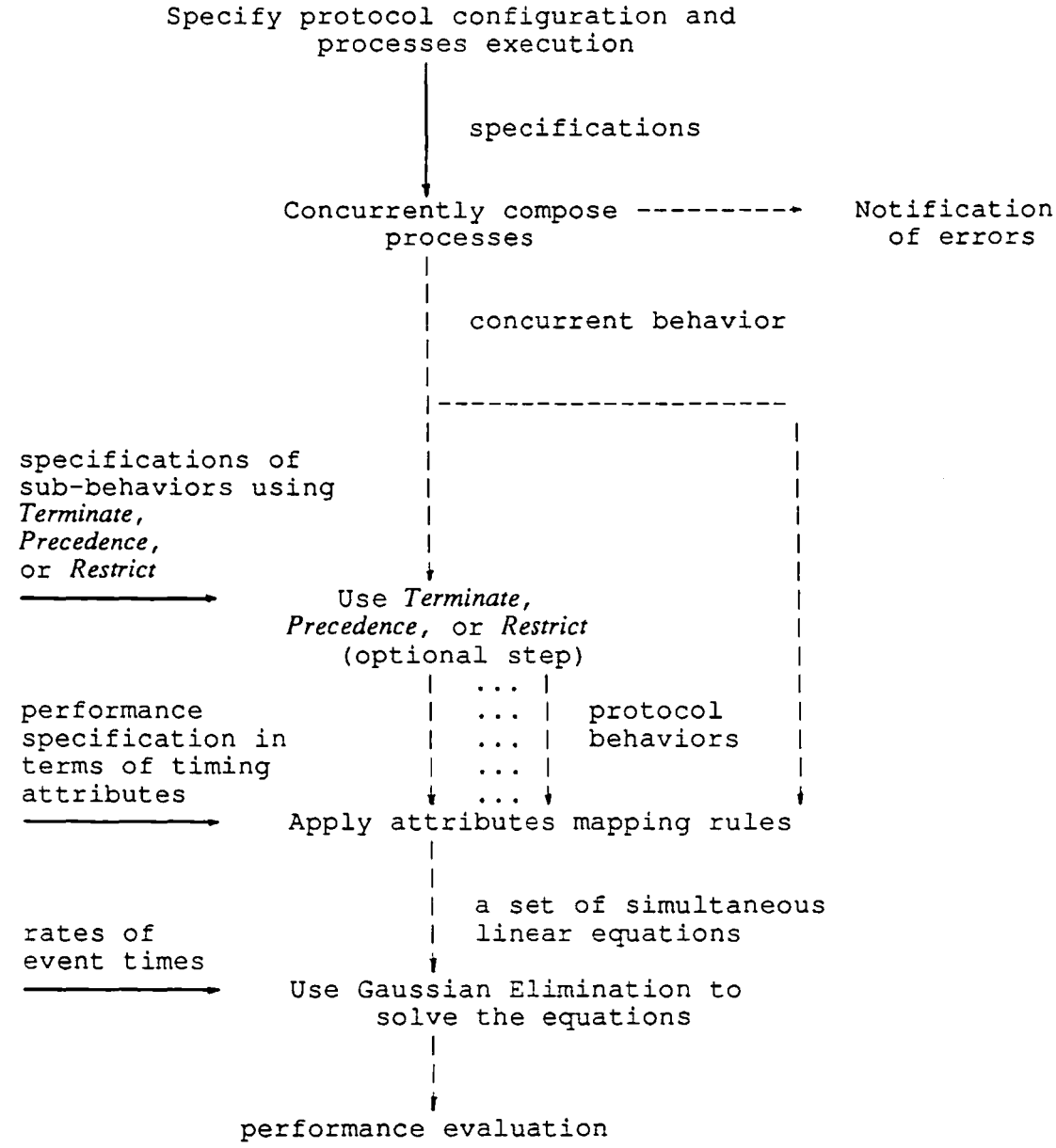


Figure 5-1: The flow of activities in using ANALYST for protocol performance analysis

5.3.1. *Functional Specification and Analysis*

5.3.1.1. *Specify Protocol and Involved Processes*

ANALYST> *# The specification of the initially described connection*

ANALYST> *# establishment protocol and the involved processes*

ANALYST> *# have been created using an editor*

ANALYST> *# and stored in files. The protocol is named CE.*

ANALYST> *# We demonstrate next that these specifications can be*

ANALYST> *# loaded from the files and the user can list them.*

ANALYST> *load CE*

CE> *# Note the change in the prompt to indicate the name of*

CE> *# the current protocol. The specification of the protocol's*

CE> *# configuration is listed next,*

CE> *# then all the processes specifications are loaded and listed.*

CE> *list*

```
PROTOCOL CE : T,R,N,I
scope(R,T) = {req}
scope(N,R) = {req'}
scope(I,N) = {inc}
scope(I,T) = {inc'}
END
```

CE> *load all*

CE> *List all*

```
PROCESS T
T = ?inc'.C1+!req.C1
C1 = $
END

Number of identifiers in process T = 2
Number of summands in process T = 2


PROCESS R
R = ?req.R1
R1 = !req'.$
END

Number of identifiers in process R = 2
Number of summands in process R = 2
```

```
PROCESS N
N = ?req'.C2+!inc.C2
C2 = $
END

Number of identifiers in process N = 2
Number of summands in process N = 2


PROCESS I
I = ?inc.I1
I1 = !inc'.$
END

Number of identifiers in process I = 2
Number of summands in process I = 2
```

5.3.1.2. *Compute Concurrent Behavior*

CE> *# We turn the SHOW variable On to follow the details of the*

CE> *# concurrent composition of the processes. Note that the detailed*

CE> *# output from the concurrent composition module is indented a*

CE> *# number of spaces according to how far it is from the top of the*

CE> *# ETs of the given processes.*

CE> *# Also note the deadlock and unspecified reception errors*

CE> *# discovered, and how they are reported with their source*

CE> *# and number.*

CE> *show on*

CE> *comp T,R*

```
scope(R,T) = {req}

compose_expressions ?req.R1 and ?inc'.C1+!req.C1
rendezvous case starting with &req
 compose_expressions R1 and C1
 compose_expressions $ and !req'.$
 shuffle case starting with !req'
  compose_expressions $ and $
shuffle case starting with ?inc'
 compose_expressions C1 and ?req.R1
 compose_expressions $ and ?req.R1

**** PROGRESS ERROR ****
** Found a deadlock error due to event ?req **
```

```
** Found 1 Deadlock Error(s) **


RT = &req.!req'.$+?inc'.$
Creating a new composite process RT

PROCESS RT
RT = &req.!req'.$+?inc'.$
END

Number of identifiers in process RT = 1
Number of summands in process RT = 2


** Found 1 Deadlock Error(s) **


CE> comp N,I

scope(I,N) = {inc}

compose_expressions ?inc.I1 and ?req'.C2+!inc.C2
rendezvous case starting with &inc
 compose_expressions I1 and C2
 compose_expressions $ and !inc'.$
 shuffle case starting with !inc'
  compose_expressions $ and $
shuffle case starting with ?req'
 compose_expressions C2 and ?inc.I1
 compose_expressions $ and ?inc.I1

**** PROGRESS ERROR ****
** Found a deadlock error due to event ?inc **

** Found 1 Deadlock Error(s) **

IN = &inc.!inc'.$+?req'.$
Creating a new composite process IN

PROCESS IN
IN = &inc.!inc'.$+?req'.$
END

Number of identifiers in process IN = 1
Number of summands in process IN = 2

** Found 1 Deadlock Error(s) **

CE> comp RT,IN

scope(IN,RT) = {req' , inc'}

compose_expressions &inc.!inc'.$+?req'.$
and &req.!req'.$+?inc'.$
shuffle case staring with &inc
 compose_expressions !inc'.$ and &req.!req'.$+?inc'.$
```

```
 rendezvous case starting with &inc'
  compose_expressions $ and $
 shuffle case starting with &req
  compose_expressions !req'.$ and !inc'.$

**** PROGRESS ERROR ****
** Found an unspecified reception error due to event !req' **
** Found an unspecified reception error due to event !inc' **

shuffle case starting with &req
 compose_expressions !req'.$ and &inc.!inc'.$+?req'.$
 rendezvous case starting with &req'
  compose_expressions $ and $
 shuffle case starting with &inc
  compose_expressions !inc'.$ and !req'.$

**** PROGRESS ERROR ****
** Found an unspecified reception error due to event !inc' **
** Found an unspecified reception error due to event !req' **

INRT = &inc.(&inc'.$+&req.$)+&req.(&req'.$+&inc.$)
Creating a new composite process INRT

PROCESS INRT
INRT = &inc.(&inc'.$+&req.$)+&req.(&req'.$+&inc.$)
END

Number of identifiers in process INRT = 1
Number of summands in process INRT = 4

** Found 4 Unspecified Reception Error(s) **
```

5.3.1.3. *Debug and Iterate*

CE> *# The specifications of the protocol and the processes are revised*

CE> *# below to avoid the design errors discovered. This time the*

CE> *# editing is done on-line.*

CE> *process T*
T=!req.C1+?inc'.C1
C1=?inc'.C1+!term.T
end
You will overwrite another process
Do you want to do that? answer with y or n : *y*

CE> *process R*
R=?req'.R1+?term.R2
R1=!req'.R+?term.R3
R2=!term'.R
R3=!req'.R2
end
You will overwrite another process

```
Do you want to do that? answer with y or n : y

CE> process N
N=!inc.C2+?req'.C2+?term'.N
C2=?req'.C2+?term'.N
end
You will overwrite another process
Do you want to do that? answer with y or n : y

CE> process I
I=!inc.I1
I1=!inc'.I
end
You will overwrite another process
Do you want to do that? answer with y or n : y

CE> scope(T,R) = {req,term}

CE> scope(R,N) = {req',term'}

CE> # Next we concurrently compose the processes again. This time

CE> # we will suppress the details of the output. Note that no errors

CE> # are reported this time.

CE> show off

CE> comp T,R,N,I

scope(R,T) = {req , term}

scope(N,RT) = {req' , term'}

scope(I,NRT) = {inc , inc'}

Creating a new composite process INRT

PROCESS INRT
INRT = &inc.C2RTI1+&req.C1R1NI
C2RTI1 = &req.C1R1C2I1+&inc'.C1RC2I
C1R1NI = &term.INR3T+&req'.C1RC2I+&inc.C1R1C2I1
C1R1C2I1 = &inc'.C1R1C2I+&term.C2R3TI1+&req'.C1RC2I1
C1RC2I = &term.C2R2TI
INR3T = &inc.C2R3TI1+&req'.C2R2TI
C1R1C2I = &term.C2R3TI+&req'.C1RC2I
C2R3TI1 = &req'.C2R2TI1+&inc'.C1R3C2I
C1RC2I1 = &inc'.C1RC2I+&term.C2R2TI1
C2R2TI = &term'.INRT
C2R3TI = &req'.C2R2TI
C2R2TI1 = &term'.I1NRT+&inc'.C1R2C2I
C1R3C2I = &req'.C1R2C2I
I1NRT = &inc'.C1RNI+&req.C1R1NI1
C2R2TI = &term'.INRT
C2R3TI = &req'.C2R2TI
C2R2TI1 = &term'.I1NRT+&inc'.C1R2C2I
```

```
C1R3C2I = &req'.C1R2C2I
I1NRT = &inc'.C1RNI+&req.C1R1NI1
C1R2C2I = &term'.C1RNI
C1RNI = &term.INR2T+&inc.C1RC2I1
C1R1NI1 = &inc'.C1R1NI+&term.I1NR3T+&req'.C1RC2I1
INR2T = &inc.C2R2TI1+&term'.INRT
I1NR3T = &inc'.C1R3NI+&req'.C2R2TI1
C1R3NI = &req'.C1R2C2I+&inc.C1R3C2I1
C1R3C2I1 = &inc'.C1R3C2I+&req'.C1R2C2I1
C1R2C2I1 = &inc'.C1R2C2I+&term'.C1RNI1
C1RNI1 = &inc'.C1RNI+&term.I1NR2T
I1NR2T = &inc'.C1R2NI+&term'.I1NRT
C1R2NI = &term'.C1RNI+&inc.C1R2C2I1
END

Number of identifiers in process INRT = 25
Number of summands in process INRT = 48
```

5.3.1.4. *Compute Protocol Sub-Behaviors*

CE> *# The terminating behavior describing the behavior of the revised*

CE> *# protocol during the course of one connection is specified next.*

CE> *# It is computed automatically. Identifiers in the new process*

CE> *# (except the initial identifier) are the same as the given*

CE> *# process with a "0" appended at the right end.*

CE> *Trm=Termin[INRT,{(&term',INRT)}]*

```
Creating a new process Trm

PROCESS Trm
Trm = &inc.C2RTI10+&req.C1R1NI0
C2RTI10 = &req.C1R1C2I10+&inc'.C1RC2I0
C1R1NI0 = &term.INR3T0+&req'.C1RC2I0+&inc.C1R1C2I10
C1R1C2I10 = &inc'.C1R1C2I0+&term.C2R3TI10+&req'.C1RC2I10
C1RC2I0 = &term.C2R2TI0
INR3T0 = &inc.C2R3TI10+&req'.C2R2TI0
C1R1C2I0 = &term.C2R3TI0+&req'.C1RC2I0
C2R3TI10 = &req'.C2R2TI10+&inc'.C1R3C2I0
C1RC2I10 = &inc'.C1RC2I0+&term.C2R2TI10
C2R2TI0 = &term'.$
C2R3TI0 = &req'.C2R2TI0
C2R2TI10 = &term'.I1NRT0+&inc'.C1R2C2I0
C1R3C2I0 = &req'.C1R2C2I0
I1NRT0 = &inc'.C1RNI0+&req.C1R1NI10
C1R2C2I0 = &term'.C1RNI0
C1RNI0 = &term.INR2T0+&inc.C1RC2I10
C1R1NI10 = &inc'.C1R1NI0+&term.I1NR3T0+&req'.C1RC2I10
INR2T0 = &inc.C2R2TI10+&term'.$
I1NR3T0 = &inc'.C1R3NI0+&req'.C2R2TI10
```

```
C1R3NI0 = &req'.C1R2C2I0+&inc.C1R3C2I10
C1R3C2I10 = &inc'.C1R3C2I0+&req'.C1R2C2I10
C1R2C2I10 = &inc'.C1R2C2I0+&term'.C1RNI10
C1RNI10 = &inc'.C1RNI0+&term.I1NR2T0
I1NR2T0 = &inc'.C1R2NI0+&term'.I1NRT0
C1R2NI0 = &term'.C1RNI0+&inc.C1R2C2I10
End

Number of identifiers in process Trm = 25
Number of summands in process Trm = 48
```

CE> *# The behavior in which no premature time-outs occur is specified next.*

CE> *A=Prec[Trm,{(&inc,&term),(&inc',&term)}]*

```
 event &inc has precedence over event &term

 event &inc' has precedence over event &term


Creating a new process A

PROCESS A
A = &inc.C2RTI100+&req.C1R1NI00
C2RTI100 = &req.C1R1C2I100+&inc'.C1RC2I00
C1R1NI00 = &req'.C1RC2I00+&inc.C1R1C2I100
C1R1C2I100 = &inc'.C1R1C2I00+&req'.C1RC2I100
C1RC2I00 = &term.C2R2TI00
C1R1C2I00 = &term.C2R3TI00+&req'.C1RC2I00
C1RC2I100 = &inc'.C1RC2I00
C2R2TI00 = &term'.$
C2R3TI00 = &req'.C2R2TI00
End

Number of identifiers in process A = 9
Number of summands in process A = 14
```

CE> *# Let us assume that the protocol designer would like to end this*

CE> *# session, save all the newly created processes, and return to work*

CE> *# later.*

CE> *quit*

```
Do you want to save current protocol CE? y

Do you want to save current processes? y

ANALYST>q
%
```

%analyst

```
Welcome to ANALYST
```

```
ANALYST> # This is a new session with ANALYST. We start by loading
ANALYST> # the previously created protocol and processes. A data file
ANALYST> # including assignments of the rates of events of the protocol
ANALYST> # has been created by an editor and saved in a file with the
ANALYST> # same name as the protocol but with extension ''dat''. Note that this data file
ANALYST> # will be loaded automatically when the protocol is loaded.
ANALYST> load CE
Data file loaded
CE> list
PROTOCOL CE : T,R,N,I,INRT,Trm,A
scope(R,T) = {req , term}
scope(N,R) = {req' , term'}
scope(I,N) = {inc}
scope(I,T) = {inc'}
END
CE> load Trm
CE> # The behavior in which no call collisions occur is specified next.
CE> B=Prec[Trm,{(&req',&inc),(&inc',&req)}]
 event &req' has precedence over event &inc
 event &inc' has precedence over event &req
Creating a new process B
PROCESS B
B = &inc.C2RTI100+&req.C1R1NI00
C2RTI100 = &inc'.C1RC2I00
C1R1NI00 = &term.INR3T00+&req'.C1RC2I00
C1RC2I00 = &term.C2R2TI00
INR3T00 = &req'.C2R2TI00
C2R2TI00 = &term'.$
END
Number of identifiers in process B = 6
Number of summands in process B = 8
```

5.3.2. *Performance Specification and Analysis*

```
CE> # Now let us start evaluating the probability of call-collisions

CE> # which is defined as 1-prob[Trm,B]. We first list the loaded

CE> # event rates, compute the probability of call-collisions for

CE> # the given values, then compute it for several values of

CE> # call-requests and incoming-calls rates.

CE> list data

~&term = 1.000000
~&term' = 100.000000
~&req' = 100.000000
~&inc' = 100.000000
~&req = 10.000000
~&inc = 10.000000

CE> 1-prob[Trm,B]

Building set of linear equations...

Solving set of linear equations...

0.090909

CE> for (~&req=0.0,10.0,1.0) 1-prob[Trm,B]

Building set of linear equations...

Solving set of linear equations...

Building set of linear equations...

Solving set of linear equations...
.
.
.
Building set of linear equations...

Solving set of linear equations...

Do you want to save the result in a file [y/n]?  n

0.000000        0.000000
1.000000        0.017265
2.000000        0.031491
3.000000        0.043384
4.000000        0.053447
5.000000        0.062049
6.000000        0.069468
7.000000        0.075916
```

```
8.000000        0.081556
9.000000        0.086519
10.000000       0.090909

CE> # Let us evaluate the probability of call collisions for a wider

CE> # range of ~&req and store the result in a file to be plotted.

CE> for (~&req=0.0,50.0,1.0) 1-prob[Trm,B]

Building set of linear equations...

Solving set of linear equations...

Building set of linear equations...

Solving set of linear equations...
.
.
.
Building set of linear equations...

Solving set of linear equations...

Do you want to save the result in a file [y/n]?  y

file name: prob-col.ans1

CE> # The rate of incoming-calls is changed next and a new set

CE> # of data for the probability of call collisions is computed.

CE> ~&inc=40.0

CE> for (~&req=0.0,50.0,1.0) 1-prob[Trm,B]

Building set of linear equations...

Solving set of linear equations...

Building set of linear equations...

Solving set of linear equations...
.
.
.
Building set of linear equations...

Solving set of linear equations...

Do you want to save the result in a file [y/n]? y
file name:  prob-col.ans2


CE> ~&inc = 10.000000
```

```
CE> # Next, we would like to evaluate the probability of premature

CE> # termination of connections defined as 1-prob[Trm,A]. Since

CE> # process B is no longer needed, it will be first freed.

CE> free B

CE> load A

CE> for (~&term=1.0,50.0,1.0) 1-prob[Trm,A]

Building set of linear equations...

Solving set of linear equations...

Building set of linear equations...

Solving set of linear equations...
.
.
.
Building set of linear equations...

Solving set of linear equations...

Do you want to save the result in a file [y/n]? y
file name: prob-prem.ans1

CE> q

Do you want to save current protocol CE? y

Do you want to save current processes? y

Process T has no specification. Not saved
Process R has no specification. Not saved
Process N has no specification. Not saved
Process I has no specification. Not saved
Process INRT has no specification. Not saved

ANALYST> q
%
```

5.4. Logical Architecture

ANALYST consists of the following four logical modules or tools, as illustrated in Fig. 5-2:

1. A parser of commands submitted by a protocol designer.
2. A compiler of expressions in the algebra of ETs, and specifications of protocol configuration and processes execution.
3. A verifier of freedom from deadlock and unspecified receptions errors.
4. A performance analyzer.

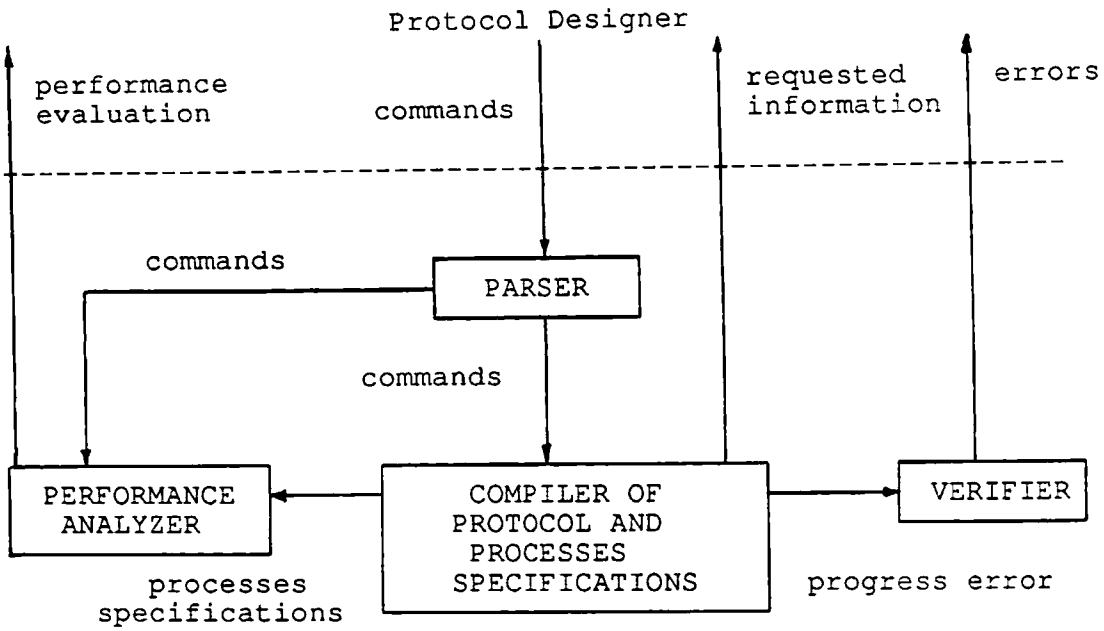


Figure 5-2: Logical architecture of ANALYST

5.4.1. Parser

The parser accepts commands from the protocol designer if they follow the command language of the environment, and then calls the appropriate module to provide the requested service. The syntax of the command language of the environment is described by a formal grammar (a Backus-Naur Form (BNF) description of the formal grammar is given in appendix 5.I). The parser was created automatically from this grammar using compiler generation tools (YACC and LEX) of UNIX as described in appendix 5.II.

5.4.2. Compiler

The functions supported by the compiler fall into three categories. First, it maintains data structures in which information about given expressions in the algebra of ETs (as defined in definition 3.1), and protocol specifications (as defined in section 3.3) are saved. These data structures are used by other modules, such as the performance analyzer. The compiler also uses these data structures in providing information and answering questions about given expressions and specifications. For example, to list a given specification, to provide the number of identifiers or summands in a given process specification, and whether a given process specification is cyclic or terminating (as defined in section 3.6). The compiler rejects expressions that are not *well-formed* (as defined in section 4.2) and processes specifications that are not *complete* (as defined in section 3.3).

Second, the compiler implements the concurrent composition definition of section 3.2.2, which informally takes two expressions and produces a rendezvous event for each pair of co-events and a complete shuffling of these rendezvous events. This is done recursively based on the structure of the composed expressions. Concurrent composition of complete process specifications, resulting in a new process describing the concurrent behavior of the processes, is also supported. Since a process specification may consist of a set of equations, the concurrent composition of two such specifications continues until all pairs of reachable identifiers in the specifications are composed. Algorithms used in implementing the concurrent composition of expressions and of processes specifications are given in appendix 5.IV.1. Let D denote the number of identifiers in an identifier table, which is an internal data structure, and d_i and s_i $(i = 1,p)$ denote the number of identifiers and summands in the specification of process i, respectively. The worst case space and time complexities for concurrently composing p processes are shown to be of $O(\prod_{i=1}^{p} d_i + \prod_{i=1}^{p} s_i)$ and $O(\prod_{i=1}^{p} s_i + D \cdot \sum_{i=1}^{p} d_i)$, respectively.

Third, the compiler supports algorithms for the application of the *Terminate*, *Precedence*, or *Restrict* functions (as defined in section 3.6) to an expression or a processes specification. Similar to the concurrent composition of processes specifications, applying one of these functions to a process specification starts off a series of applications of the function to each equation in the specification. The three functions are similar in the manner in which they apply to a process

specification in that their implementation involves inspecting each equation in the given process specification and results in the creation of a new process. As an example, the algorithm used for implementing the *Terminate* function is given in appendix 5.IV.2. Given an expression P and a set of pairs of events and identifiers, $S=\{(e_i,I_i), i=1,...,n\}$, let $Sum(P)$ denote the set of all summands in process P, and $|S|$ is the size of S. The time complexity of the algorithm is shown to be of $O(|Sum(P)|\cdot|S|+D)$.

The compiler also supports constrained concurrent composition in which the concurrent composition is guided by some constraints. These constraints are ternary relations between expressions and either sets of event pairs for the case of the *Precedence* function, or sets of pairs of events and identifiers for the case of the *Restrict* function. Constrained concurrent composition can be a means of avoiding the high space and time costs involved in computing the entire concurrent behavior of a complex real-life protocol with numerous states, summands, or processes. However, the protocol designer would only obtain a subset of the entire protocol's concurrent behavior. For example, the concurrent composition of the processes in the revised connection establishment protocol when constrained by the same *Precedence* relation in eq. (3.2) directly produces $INRT_{P1}$ which represents protocol behaviors without call-collisions (with the exception that it will be cyclic). This is a reduction in the number of identifiers from 48 to 6 (87.5%) and a reduction in the number of summands from 25 to 8 (68%).

5.4.3. Verifier

If during the course of concurrently composing two expressions, a progress error is detected (as defined in definition 3.2), the compiler calls the verifier to check for any deadlock or unspecified reception errors (as defined in definition 3.3). The verifier notifies the protocol designer of the presence of any of these errors. When concurrently composing several process specifications, the verifier keeps track of all the errors detected during the concurrent composition of every pair of expressions. It then notifies the protocol designer of the total number of deadlock and unspecified reception errors detected.

If the designer wishes to track down the exact event sequence leading to an error, he can request to see details of the step-by-step concurrent composition by setting a SHOW variable ON. In this case, the designer is notified of the events that cause each error.

5.4.4. Performance Analyzer

The performance analyzer supports the evaluation of probability, mean-time, and variance-time attributes of a protocol's timing model. Given complete sets of equations that specify some expressions A and C, and the exponential rates of the events involved in them, algorithm 5.4 given in appendix 5.IV.3 maps these equations to a set of linear equations in terms of a requested attribute. This is performed using lemma 4.1 and the rules of theorems 4.1, 4.2, and 4.3. In computing $\mathrm{P}_C(A)$, $\mathrm{M}_C(A)$, or $\mathrm{V}_C(A)$, where the specification of A involves d identifiers, the algorithm produces $d+1$ equations in any of the requested attributes. The algorithm is shown to have worst case space and time complexities of $O(d^2)$ and $O(d^3)$, respectively. A Gaussian elimination algorithm with pivoting [Rals 78] is then used for solving the set of linear equations. Given $d+1$ equations in an attribute, the time complexity of this algorithm is known to be of $O(d^3)$.

The performance analyzer also supports the computation of *arithmetic expressions* and *assignment statements*, in which the attributes and the rates of events are considered as variables. Using the assignment statements, the designer can assign values to the rates of events and possibly other variables (such as length of messages for example). Then, he can specify performance measures as arithmetic statements in these variables and request their evaluation. The analyzer also supports a FOR statement to evaluate an arithmetic expression for several values of a chosen index variable. A list of values of the index variable and attribute is produced and can be easily plotted. The current version of the analyzer does not support the analysis of inequality equations or minimization/maximization functions. Timing requirements that are specified in such formats can be analyzed manually with the assistance of the analyzer in computing any attribute value.

It should be noted that although an analytic approach is followed in evaluating the performance attributes of protocols as opposed to a simulation approach, the performance analyzer performs the evaluations numerically and not symbolically.

Appendix 5.I. A Grammar for ANALYST's Command Language

```
/* This is the precedence of the operations (in ascending */
/* order) and their associativity. */

%right EQUAL
%left  CONC
%left  CHOICE
%left  SUB
%left  MULT
%left  DIV
%left  SEQ


/* Now begins the set of grammar rules. */
/* Each rule has the form <non-terminal symbol> : <rule> */
/* A ``|'' indicates alternative rules and a successive*/
/* list of symbols within a rule indicates concatenation. */
/* Each rule ends with a ``;''.  */
/* Symbols are either terminal symbols in upper case */
/* letters or non-terminal symbols in lower case */
/* letters. */
/* Terminal symbols are returned by LEX; their definitions */
/* are given following the grammar. Non-terminal symbols */
/* are defined by rules in the grammar */


lines           :     /* empty case */
                |     lines unstr_line CR
                |     lines struct_line CR
                |     lines CR
                |     lines QUIT CR
                |     lines error CR
                ;

unstr_line      :     exp
                |     equation
                |     arith_stat
                |     SHOW ON
                |     SHOW OFF
                |     CONSTRAINT open_paren IDENTIFIER COMMA
                      IDENTIFIER close_paren EQUAL cons
                |     scope_def
                |     CONSTRAINT open_paren IDENTIFIER COMMA
                      IDENTIFIER close_paren
                |     SCOPE open_paren IDENTIFIER COMMA
                      IDENTIFIER close_paren
                |     DERIVATIVE open_paren exp COMMA event
                      close_paren
                |     CHOICE open_paren exp close_paren
                ;

exp             :     DEAD_NOP
                |     IDENTIFIER
                |     event SEQ exp
```

```
                |     exp CHOICE exp
                |     exp CONC exp
                |     open_paren exp close_paren
                ;

event           :     SEND_EVENT
                |     RCV_EVENT
                |     RND_EVENT
                ;

open_paren      :     OPEN1_PAREN
                |     OPEN2_PAREN
                |     OPEN3_PAREN
                ;

close_paren     :     CLOSE1_PAREN
                |     CLOSE2_PAREN
                |     CLOSE3_PAREN
                ;

equation        :     IDENTIFIER EQUAL exp
                ;

arith_stat      :     arith_exp
                |     VARIABLE EQUAL arith_exp
                |     FOR open_paren VARIABLE EQUAL REAL COMMA
                      REAL COMMA REAL close_paren arith_exp
                ;

arith_exp       :     VARIABLE
                |     REAL
                |     INTEGER
                |     PROBABILITY open_paren exp COMMA exp
                      close_paren
                |     MEAN open_paren exp COMMA exp
                      close_paren
                |     VARIANCE open_paren exp COMMA exp
                      close_paren
                |     arith_exp CHOICE arith_exp
                |     arith_exp SUB arith_exp
                |     arith_exp MULT arith_exp
                |     arith_exp DIV arith_exp
                |     open_paren arith_exp close_paren
                ;

cons            :     event_pair_str
                |     event_id_str
                ;

scope_def       :     SCOPE open_paren IDENTIFIER COMMA
                      IDENTIFIER close_paren  EQUAL open_paren
                      name_str close_paren

name_str        :     NAME
                |     name_str COMMA NAME
                ;
```

```
struct_line     :   prot_begin prot_actions QUIT
                ;

prot_begin      :   LOAD IDENTIFIER CR prot_spec
                |   prot_spec
                ;

prot_spec       :   PROTOCOL IDENTIFIER COLON processes_list
                    scope_list END
                ;

processes_list  :   /* empty case */
                |   processes_list COMMA IDENTIFIER
                |   IDENTIFIER
                ;

scope_list      :   /* empty case */
                |   scope_list CR
                |   scope_def
                |   scope_list scope_def
                ;

prot_actions    :   /* empty case */
                |   prot_actions p_action CR
                |   prot_actions CR
                |   prot_actions error CR
                ;

p_action        :   unstr_line
                |   proc_action
                |   data_action
                |   LIST
                ;

proc_action     :   proc_spec
                |   LOAD IDENTIFIER
                |   LOAD ALL
                |   COMPOSE comp_list
                |   IDENTIFIER EQUAL TERMINATE open_paren
                    IDENTIFIER COMMA OPEN3_PAREN
                    event_id_str CLOSE3_PAREN
                |   IDENTIFIER EQUAL PRECEDENCE open_paren
                    IDENTIFIER COMMA OPEN3_PAREN
                    event_pair_str CLOSE3_PAREN
                |   IDENTIFIER EQUAL RESTRICT open_paren
                    IDENTIFIER COMMA OPEN3_PAREN
                    event_id_str CLOSE3_PAREN
                |   LIST IDENTIFIER
                |   LIST ALL
                |   SUMS open_paren IDENTIFIER close_paren
                |   IDS open_paren IDENTIFIER close_paren
                |   CYCLIC open_paren IDENTIFIER close_paren
                    QUESTION
                |   TERMINATE open_paren IDENTIFIER
                    close_paren QUESTION
                |   FREE IDENTIFIER
```

```
                ;

proc_spec       :   proc_begin spec_lines END
                ;

proc_begin      :   PROCESS IDENTIFIER CR
                ;

spec_lines      :   /* the empty case */
                |   spec_lines equation CR
                |   spec_lines CR
                |   spec_lines error  CR
                ;

comp_list       :   /*empty case */
                |   comp_list COMMA IDENTIFIER
                |   IDENTIFIER
                ;

event_id_str    :   open_paren event COMMA IDENTIFIER
                    close_paren
                |   event_id_str COMMA open_paren event
                    COMMA IDENTIFIER close_paren
                ;

event_pair_str  :   open_paren event COMMA event close_paren
                |   event_pair_str COMMA open_paren event
                    COMMA event close_paren
                ;

data_action     :   LOAD DATA IDENTIFIER
                |   DATA
                |   LIST DATA
                ;
```

Appendix 5.II. Using UNIX Programming Development Tools in Producing ANALYST's Parser

YACC (Yet Another Compiler-Compiler) [John 79] and LEX (LEXical analyzer generator) [Lesk 79], which are programming development tools supported by UNIX, were used in automatically producing the parser. YACC is a parser generator that accepts some formal grammar written in BNF describing the syntax of some language and produces a parser of statements in that language. The rules of the grammar are accompanied by actions which are invoked when an instance of a rule in the grammar is recognized in the input commands being parsed. LEX can be used by the YACC program to read the input commands and divide them into syntactical units referred to as *terminal symbols*, which are then passed to the YACC program.

The input to LEX are regular expressions defining the formats of these terminal symbols. The rules in the grammar input to YACC are written in terms of these terminal symbols and *non-terminal symbols* which are defined by other rules in the grammar.

YACC and LEX have been used to generate the parser for the command language of ANALYST, as illustrated in Fig. 5-3. The command language is specified as a formal grammar (its BNF description is in appendix 5.I), in which each rule specifies some service supported by the environment. The actions accompanying the rules provide this service. These actions are written in the C language. The syntax of the terminal symbols used in ANALYST's command language is given in appendix 5.III.

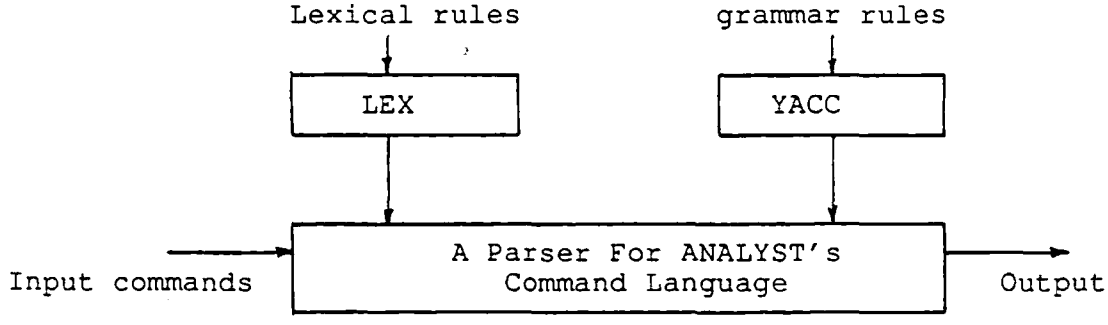


Figure 5-3: Using YACC and LEX to generate a parser of ANALYST's command language

Appendix 5.III. Definitions of Terminal Symbols in ANALYST's Command Language

```
/* In what follows, terminal symbols are defined as regular */
/* expressions. These definitions are input to LEX and a */
/* C function is produced. */
/* This C function is called by the parser (produced by */
/* YACC from the above given grammar) to transform the */
/* input commands into a sequence of terminal symbols. */
/* In each definition, "c" matches the character ``c'', */
/* [c1 c2 ... cn] matches any */
/*  ``ci'' (i=1,n), and [c1] [c2] */
/*  matches a concatenation of characters */
/* ``c1'' and ``c2''. ]? */
```

```
/* indicates an optional matching of the character ``c'' */
#.*\n                       /* comments */
[\ ]+                       /* tabs and blank spaces */
"\n"                        return(CR);
"("                         return(OPEN1_PAREN);
")"                         return(CLOSE1_PAREN);
"["                         return(OPEN2_PAREN);
"]"                         return(CLOSE2_PAREN);
"{"                         return(OPEN3_PAREN);
"}"                         return(CLOSE3_PAREN);
"+"                         return(CHOICE);
"-"                         return(SUB);
"*"                         return(MULT);
"/"                         return(DIV);
"|"                         return(CONC);
"."                         return(SEQ);
"$"                         return(DEAD_NOP);
"="                         return(EQUAL);

[Pp][Rr][Ee][Cc]?[Ee]?[Dd]?[Ee]?[Nn]?[Cc]?[Ee]?
                            return(PRECEDENCE);
[Rr][Ee][Ss]?[Tt]?[Rr]?[Ii]?[Cc]?[Tt]?
                            return(RESTRICT);
[Ff][Rr][Ee]?[Ee]?          return(FREE);
[Tt][Ee][Rr][Mm][Ii][Nn]?[Aa]?[Tt]?[Ee]?
                            return(TERMINATE);
[Cc][Yy][Cc]?[Ll]?[Ii]?[Cc]?
                            return(CYCLIC);
[?]                         return(QUESTION);
","                         return(COMMA);
":"                         return(COLON);
```

```
[Ii][Dd][Ss]?              return(IDS);

[Ss][Uu][Mm]?[Ss]?         return(SUMS);

[Ss][Cc][Oo]?[Pp]?[Ee]?    return(SCOPE);

[Dd][Aa][Tt]?[Aa]?         return(DATA);

[Aa][Ll][Ll]?              return(ALL);

[Cc][Oo][Mm][Pp]?[Oo]?[Ss]?[Ee]?
                           return(COMPOSE);

[Pp][Rr][Oo][Cc][Ee]?[Ss]?[Ss]?
                           return(PROCESS);

[Pp][Rr][Oo][Tt][Oo]?[Cc]?[Oo]?[Ll]?
                           return(PROTOCOL);

[Ll][Oo][Aa]?[Dd]?         return(LOAD);

[Ll][Ii][Ss]?[Tt]?         return(LIST);

[Ee][Nn][Dd]               return(END);

[Cc][Hh][Oo]?[Ii]?[Cc]?[Ee]?
                           return(CHOICE);

[Dd][Ee][Rr]?[Ii]?[Vv]?[Aa]?[Tt]?[Ii]?[Vv]?[Ee]?
                           return(DERIVATIVE);

[Cc][Oo][Nn]?[Ss]?[Tt]?[Rr]?[Aa]?[Ii]?[Nn]?[Tt]?[Ss]?
                           return(CONSTRAINT);

[Ss][Hh][Oo]?[Ww]?         return(SHOW);

[Pp][Rr]?[Oo]?[Bb]?[Aa]?[Ll]?[Ii]?[Tt]?[Yy]?
                           return(PROBABILITY);

[Mm][Ee]?[Aa]?[Nn]?        return(MEAN);

[Vv][Aa]?[Rr]?[Ii]?[Aa]?[Nn]?[Cc]?[Ee]?
                           return(VARIANCE);

[Oo][Ff][Ff]?              return(OFF);

[Oo][Nn]                   return(ON);

[Ss][Oo][Rr]?[Tt]?         return(SORT);

[Ee][Xx][Ii]?[Tt]?         return(QUIT);

[Qq][Uu]?[Ii]?[Tt]?        return(QUIT);

[Ff][Oo][Rr]               return(FOR);
```

```
[0-9]+"."[0-9]*(E)?        return(REAL);
[0-9]+                     return(INTEGER);
[a-z][a-z0-9'`]*           return(NAME);
[!][a-z][a-z0-9'`]*        return(SEND_EVENT);
[?][a-z][a-z0-9'`]*        return(RCV_EVENT);
[&][a-z][a-z0-9'`]*        return(RND_EVENT);
[#~][a-z][a-z0-9'`]+       return(VARIABLE);
[A-Z][A-Za-z0-9&]*         return(IDENTIFIER);
```

Appendix 5.IV. Key Algorithms used in ANALYST

5.IV.1. Algorithms for Concurrent Composition of Expressions and Processes:

Algorithms used in implementing the concurrent composition of expressions and of processes. *Compose-Expressions* and *Compose-Processes*, respectively, are given next. In *Compose-Expressions*, the function *verify* is called if a progress error is detected. The algorithm for implementing this function is straightforward following definitions 3.2 and 3.3.

Before presenting the algorithms, a brief description of the main data structures used is required. Three key data structures maintained by the compiler include an *identifier table*, a *protocol structure*, and a *process table*. The identifier table is a linked list of identifier records in which the identifier and the corresponding expression are saved. In the protocol structure, information about the current protocol are saved including its name and a linked list of the names of processes involved in it. The *scope* of communication between pairs of processes are saved in the identifier table where identifier records have optional entries for a communicating identifier (the name of the communicating process) and the corresponding *scope*. The process table includes the process name and a linked list of the identifiers involved in it.

Algorithm 5.1 *Compose-Expressions*

Input:	Two expressions A and B.
Output:	A composite expression produced by the concurrent composition of the two input expressions following concurrent composition definition.

Method:

1. Initialize the return expression to the empty string. Set the current scope to be equal to *scope*(A,B).
2. For every summand $a_i \bullet A_i$ of expression A do the following:
   i. (*Shuffle case*) If a_i is not an element of the current scope, then concatenate "$+a_i \bullet (A_i|B)$" to the return string and go to step 2(i). Otherwise, go to 2(ii).
   ii. (*Rendezvous case*) If there exists a summand $b_j \bullet B_j$ of B such that b_j is not an element of the current scope, a_i and b_j are *co-events*, and $e = name(a_i) = name(b_j)$, then concatenate "$+\&e \bullet (A_i|B_j)$" to the return string.
3. For every summand $b_j \bullet B_j$ of B:
   (*Shuffle case*) if b_j is not an element of the current scope, then concatenate "$+b_j \bullet (A|B_j)$" to the return string.
4. If the return string is equal to the empty string, then replace it by "$". If A≠$ and B≠$, then there is a progress error; call function *verify* with the two expression A and B as input arguments to detect any deadlock or unspecified reception errors. Return with the return string.

The expansion of the concurrent composition of the input expressions for only one step was considered in the algorithm. In the implementation of the algorithm, the expansion is carried further if the two expressions to be concurrently composed next are found not to be recursive. Otherwise, the protocol designer should then explicitly request the expansion of each step to obtain the complete expansion. This a procedure is performed automatically if the protocol designer requests the concurrent composition of process specifications.

Algorithm 5.2 *Compose-Processes*

Input: A list of processes names whose specifications are to be concurrently composed.

Output: A new composite process specification, whose name is a concatenation of the input processes names.

Method:

To concurrently compose two processes P_1 and P_2:

1. Create a new entry in the process table for the new concurrent process P_1P_2 (concatenating the identifiers after sorting their names in ascending order). Assign the identifier P_1P_2 as the first element in its list of identifiers. Set the current *scope* to *scope*(P_1,P_2) as set in the identifier table.
2. Initialize I_1 to P_1 and I_2 to P_2. Create a list of reachable-identifiers that includes identifier pairs. Add the pair of identifiers I_1 and I_2 to it and mark them.
3. Create a new entry for identifier I_1I_2 in the identifier table and assign the empty

string to it.

4. Look up the expressions assigned to the current identifier I_1 and I_2 in the identifier table and call *Compose-Expressions* (see algorithm 5.1) to concurrently compose them. Scan the resulting expression for any <identifier-name>|<identifier-name>. Replace all such strings with a concatenation of the identifiers and add the identifiers to the list of reachable-identifiers. Assign the resulting expression to $I_1 I_2$ in the identifier table.
5. If all pairs of identifiers on the reachable-identifiers list are marked, then return. Otherwise assign the next unmarked pair of identifiers on the list to I_1 and I_2. Mark this pair and go to step 3.

In order to concurrently compose several process specifications, the above algorithm is called several times. Given a list of processes, the first pair of processes in the list are concurrently composed using the algorithm, then the resulting composite process is concurrently composed with the third process on the list. This procedure is repeated until the list of processes is exhausted.

The concurrent composition of even simple process specifications often leads to an explosion in the size of the resulting composite specification. Let us compute the space and time complexities of computing such concurrent behaviors; that is, the time and space complexities of algorithm 5.2.

Consider a protocol involving p processes. Let d_i and s_i $(i = 1,p)$ denote the number of identifiers and summands in the specification of process i, respectively. The space complexity for computing the concurrent behavior of a protocol involving p communicating processes is then of $O(\prod_{i=1}^{p} d_i + \prod_{i=1}^{p} s_i)$. The first term corresponds to the space allocated for the reachable-identifiers lists. The second term corresponds to the size of the resulting composite process. The time complexity is of $O(\prod_{i=1}^{p} s_i + D \cdot \sum_{i=1}^{p} d_i)$, where D is the number of identifiers in the identifier table. The first term represents the time spent in concurrent composition in step 4 since all combinations of the summands in the processes may have to be considered. The second term represents the time spent in searching for the identifiers of the processes in the identifier table using a simple linear search. If a hashing algorithm is used instead, then D would be reduced to a constant.

These are worst case complexities when every identifier in a process is reachable from every other identifier, and there are no rendezvous interactions produced. On the average, the number of summands and identifiers in a composite specification are usually much less than that computed

by the product form above. For example, $\prod_{i=1}^{4} s_i$ for the revised connection establishment protocol is equal to 240. However, the actual number of summands in the concurrent behavior of the protocol is only 48; that is about 80% less. Also, $\prod_{i=1}^{4} d_i$ is equal to 48, but the actual number of identifiers in the concurrent behavior is 25; that is about 50% less.

5.IV.2. Algorithm for the *Terminate* function:

Applying the *Terminate*, *Precedence*, and *Restrict* functions to a process P starts off a series of applications of the function to each equation in the process specification. Algorithms for implementing these functions are similar in many respects. An algorithm for implementing the *Terminate* function is given below as a representative. The same data structures described in appendix 5.IV are used here. Identifiers in the produced process specification have the same names as those in the given process specification with a ''0'' concatenated at the right end.

Algorithm 5.3 *Terminate*

Input: A process name P, a set $S=\{(e_i,I_i), i=1,...,n\}$ of pairs of events and identifiers, and a new process name N.

Output: A terminating process, named N, following definition 3.4.

Method:

1. Create a new entry in the process table for the new concurrent process N. Assign the identifier N as the first element in its list of identifiers. Set the current identifier I to P.
2. Look up the expression assigned to the current identifier I in the identifier table.
3. Create a new entry in the identifier table for a new identifier $I0$ (except initially where it is named with the new process name N) and assign the empty string to it.
4. For every summand $a_i \bullet A_i$ in the expression assigned to I, if $(a_i,A_i) \notin S$ then concatenate the string assigned to $I0$ with ''$+a_i \bullet A_i$''. Otherwise, if $(a_i,A_i) \in S$ then concatenate the string assigned to $I0$ with ''$+a_i \bullet \$$''.
5. If the the string assigned to $I0$ is the empty string, then replace it by ''$''. If I is the last identifier in the identifier list of process P then return, otherwise, goto step 6.
6. Let the current identifier I be the next identifier in the identifier list of process P and goto step 2.

Let *Sum*(P) denote the set of all summands in the specification of process P, and D denote the number of identifiers in the identifier table. The time complexity of this algorithm is then of

$O(|Sum(P)| \cdot |S|+D)$, where $|Sum(P)| \cdot |S|$ is the time spent in step 4 since for every summand in process P all the elements of the set S are examined, and D is the time spent in step 2 searching the identifier table using a simple linear search.

5.IV.3. Algorithm for Building Set of Linear Equations in a Timing Attribute

In this algorithm, it is assumed that the user has assigned values to the exponential rates of events in the given expressions. These rates are stored in an array called variable-list. In the case of the variance-time attribute, it is assumed that this algorithm has been called once before to compute the mean-time attributes of the identifiers involved in the given expression. These are then used as constants that are fetched from a pre-stored array.

Algorithm 5.4 *Build-Set-of-Attribute-Equations*

Input: Two complete sets of algebraic equations describing some behaviors A and C, and a request to compute either $P_C(A)$, $M_C(A)$, or $V_C(A)$. All expressions in the given specifications are well-formed, and have summands of the form $a \bullet A$, where A is an identifier.

Output: The coefficient matrix (X) and the right hand side vector (Y) of the set of linear equations (XA=Y) in a requested attribute (A). If the given specification of A includes d identifiers, then the dimension of X is a $(d+1)\times(d+1)$ matrix, and the dimension of both A and Y is $(d+1)\times 1$. The addition of 1 to d is for the termination symbol \$.

Method:

1. Check if A is cyclic. If yes, then return ''undefined -- expression is cyclic''. Else, go to step 2.
2. Initialize all entries of Y to zero, diagonal entries of X to 1, and all other entries of X to zero. Initialize the current index k to 1, and the current pair of identifiers A_k and C_k to A and C, respectively. Initialize a list of identifier-pairs to include A and C. Mark this entry in the list.
3. Look up the expressions assigned to A_k and C_k. Compute the choice set $CH(C_k)$. Look up in the variable-list the rates of all the events in this set, and assign their sum to a sum-of-rates variable.
4. For each summand $a \bullet A'$ of A_k, do the following:
   i. Compute $\partial_a (C_k)$. If its value is undefined, then return with ''undefined -- A is not a summand of C''. Otherwise, assign its value to C' and go to step 4 (ii).
   ii. Look up the pair A', C' in the identifier-pairs list; add it to the list if not found. Let j be the index of it in the list.
   iii. Add to coefficient x_{kj} of matrix X the value ''-λ_a/sum-of-rates''. If the requested attribute is mean-time, then add to y_k the value

"λ_a/(sum-of-rates^2)". Else, if the requested attribute is variance-time, then add to y_k the value:

$$\text{"}\lambda_a/(\text{sum-of-rates}^3)+[\lambda_a/(\text{sum-of-rates})]\cdot\{1/(\text{sum-of-rates}^2)+[2/(\text{sum-of-rates})]\cdot M_{C'}(A')+M^2_{C'}(A')\}\text{"}$$

5. If the requested attribute is the variance-time, then add to y_k the value "$-M_C(A)^2$".
6. If all identifier pairs on the identifier-pairs list are marked, then return. Otherwise, increment k, and assign the next unmarked identifier pair on the list to A_k and C_k. Mark this pair and go to step 3.

In the worst case, the given expression C is the same as A, and the number of summands in the set of algebraic equations describing A is equal to d^2, where d denotes the number of identifiers in the equations. The worst case space complexity of the algorithm is then of $O(d^2)$ since d^2+d units of space are required for the matrix and vector. The worst case time complexity of the algorithm is of $O(d^3)$ since the time spent in step 4 (ii) is in the order of the number of summands in A multiplied by $\lceil d/2 \rceil$ (the time spent in linearly searching the identifier-pairs list).

Part III

Applications

Chapter 6

Performance Analysis of the Alternating Bit Protocol

6.1. Introduction

In this chapter we apply the methodology and tools developed in the previous chapters to study the performance of the Alternating Bit protocol [Bart 69]. The Alternating Bit protocol is a simple data transfer protocol that belongs to the family of protocols employed in the data link layer of the ISO hierarchy of Fig. 1-1. It has been used in some actual networks such as the Advanced Research Project Agency (ARPA) network and the European Informatics Network (EIN) [Davi 79]. It is also widely considered as a benchmark among researchers in the area of protocol development tools.

The function of the protocol is to ensure reliable transfer of messages between a sender and a receiver communicating through an unreliable channel. The sender sends a message and waits for its acknowledgment to arrive from the receiver before sending another message (send-and-wait flow control). If the sender does not get an acknowledgment within a certain time-out period, it would retry transmission. A binary numbering scheme (where a control bit of 0 or 1 is used) is used to differentiate between new and old messages or acknowledgments.

The protocol behavior is identical for each cycle with control bit 0 or 1. Therefore, a simplified version of the protocol, a send-and-wait protocol, in which there are no message numbers is first considered in the next section. The complete Alternating Bit protocol is then considered in section 6.3. Two performance problems of the send-and-wait protocol, and one performance problem of the Alternating Bit protocol are addressed. *First, the effect of varying the time-out rate on the performance of the send-and-wait protocol is analyzed.* A too short time-out period causes the sender to flood the medium unnecessarily with retransmissions. A too long time out period causes the recovery from a message or acknowledgment loss to be unnecessarily delayed. This trade-off

involves a few performance parameters:

1. Time-out rate $\lambda_{\&tout}$: the rate at which the sender retransmits.
2. Probability of premature time-out p_p: the probability that the sender will time out prior to a loss occurring. It is used as a measure of unnecessary retransmissions.
3. Mean roundtrip delay t_d: the mean time from sending a message until the successful arrival of its acknowledgment.

Typically, the time-out period is a constant equal to the mean of the roundtrip delay. This setting aims at responding efficiently to loss while avoiding premature time-outs. However, it has been shown to be adequate only if the variance of the roundtrip delay is very small [Suns 75]. We propose an alternative approach for computing an optimal value of the time-out rate by maximizing the ratio $(1-p_p)/t_d$, which is analogous to the *power* measure used to study delay throughput trade-off in queueing theory [Schw 80]. This timing requirement would yield an optimal value of the time-out rate even if the variance of the roundtrip delay is not small. It also does not require a constant setting of the time-out period; an exponential distribution is assumed in this study.

Second, two performance measures of the send-and-wait protocol: maximum throughput and mean waiting time, are formally specified and automatically analyzed. Maximum throughput denotes the maximum rate of delivering messages to the receiver when the sender has always a message to send. The mean waiting time denotes the average of the time duration that a message arriving at the source of messages has to be queued until the sender can service it. This queueing delay is due to the send-and-wait nature of the protocol. These two measures are specified for an approximation of the send and wait protocol in which time-outs occur only after a loss. Results obtained are shown to agree remarkably well with past results reported in the literature using manual analysis.

Third, a mean cycle time performance measure is defined for the complete Alternating Bit protocol to capture the effect of improper settings of the time-out rate on the timing behavior of the protocol. Yemini and Kurose [Yemi 82] have shown that if the rate of loss in the channels is equal to zero, then a non-zero time-out rate would cause the number of message transmissions to increase eventually to ∞. They did not, however, analyze these timing errors for an arbitrary value of the rates of time-out and loss in the channels. In section 3, the mean cycle time measure is

formally specified, and the effect of variations in the rate of loss, and the upper bound on the number of messages in the channels on it, are analyzed.

6.2. A Send-and-Wait Protocol

6.2.1. Functional Specification and Analysis

6.2.1.1. An Algebraic Specification

A send-and-wait (SW) protocol can be used for message transfer between a sender process S and a receiver process R communicating through two half-duplex, FIFO, and unreliable channels: M (S to R for messages) and A (R to S for acknowledgments). An upper bound of 2 messages/acknowledgments in the channels is assumed. (The sensitivity of the results to this assumption is analyzed in section 6.2.2.1.) The sender process is a composite process resulting from an original sender concurrently composed with a source of messages and a time-out timer. Two rendezvous events: *&get* (get a message from the source) and *&tout* (time-out interrupt), are obtained from these compositions, respectively. The two channels also result from concurrent composition with a loss process producing the rendezvous events *&lm* (loss of a message) and *&la* (loss of an acknowledgment).

ETs describing the execution of the four processes involved in the protocol are shown in Fig. 6-1. Initially, the sender and receiver processes are ready for data transfer, and the two channels are empty. Channel M simply receive messages (*?msg*) and either delivers (*!msg'*) or loses (*&lm*) them. Channel A behaves similarly (change *msg* to *ack* and *lm* to *la*). When the sender process S gets a message (*&get*) from the source, it sends it (*!msg*), and waits for its acknowledgment (*?ack'*) before transmitting another message. If time-out occurs before that, the sender retransmits the same message and waits again for acknowledgment or time-out. The receiver process R loops through the following behavior: when it receives a message (*?msg'*) from channel M, it sends an acknowledgment (*!ack*) to channel A and becomes ready again. Note that this protocol is similar to the send-and-wait protocol of chapter 2, with the exception that here acknowledgments can be lost and time-outs can occur before a loss. The configuration of the protocol is depicted in Fig. 6-2. Algebraic specifications of the processes are given in appendix 6.I.1.

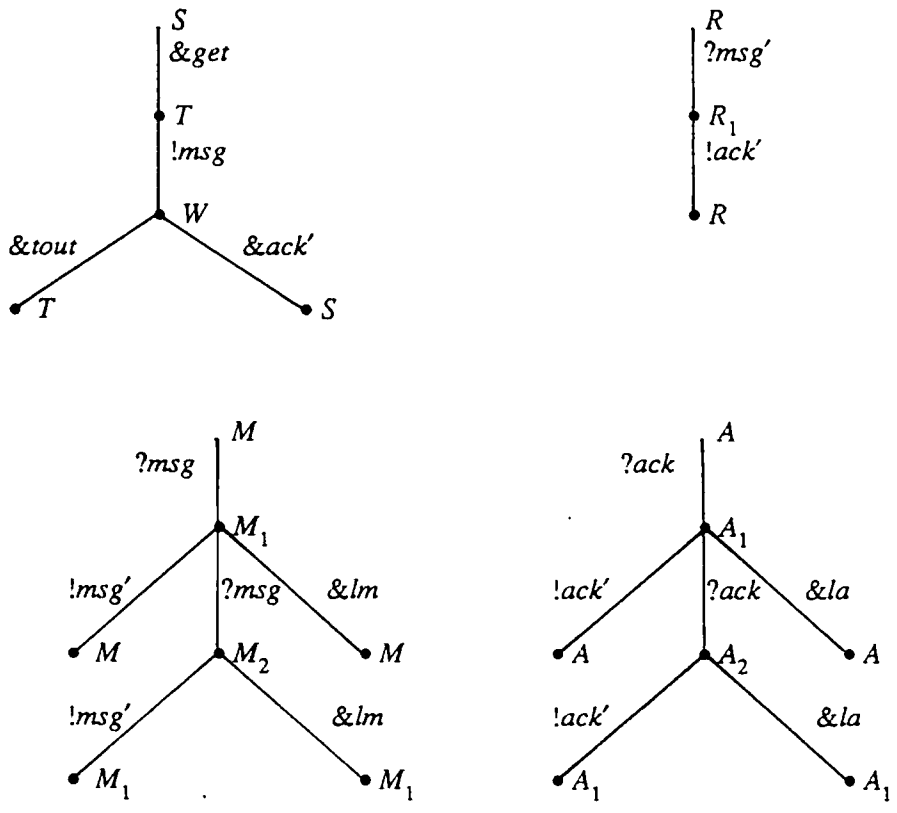


Figure 6-1: ETs describing the execution of the processes in the send-and-wait protocol

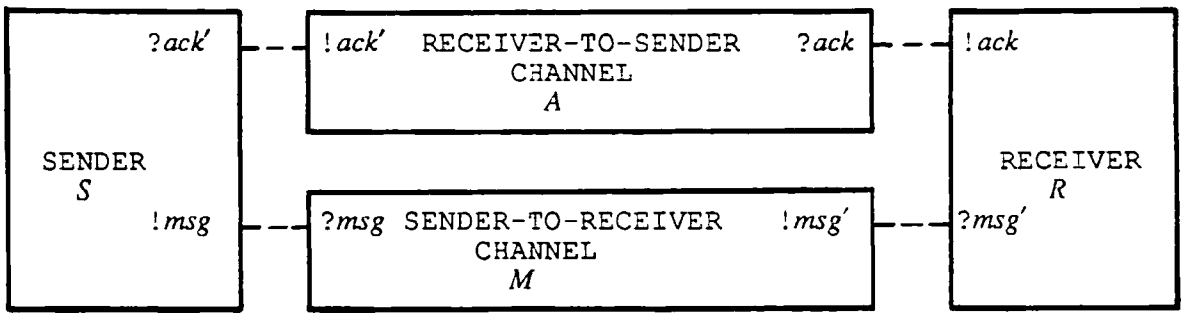


Figure 6-2: Configuration of the send-and-wait protocol

6.2.1.2. The Concurrent Behavior

The concurrent behavior of the protocol, *AMSR*, computed by concurrently composing the above given specifications of its four processes, is given in appendix 6.I.2. *AMSR* is a cyclic behavior that describes the execution of the protocol through several cycles. Each cycle starts with the sender getting a message from the source (*&get*) and ends with the first acknowledgment delivered to the sender (*&ack'*). That is, at the start of a cycle the sender process is at state *S* but the other processes can possibly be at any state. Let these global states of the protocol be denoted by ''***S'', where each ''*'' matches any state of the receiver, and the two channels. Note that other than the initial state *AMSR*, all such states would lead to erroneous behaviors in which old messages and acknowledgments in the protocol can not be distinguished from new ones. But then this is expected since the protocol specification does not consider such situations.

One interesting behavior of the protocol is that which starts at *AMSR* and terminates with the delivery of acknowledgment at the sender (*&ack'*) when the protocol is at any state ''***S''. This behavior, to be denoted by $AMSR_T$, is a representative of the time a message and its acknowledgments occupy protocols that use the send-and-wait flow control mechanism, such as the Alternating Bit. $AMSR_T$ can be specified in terms of the *Terminate* function (see definition 3.4) as follows:

$$AMSR_T = Terminate[AMSR, \{(\&ack', ***S)\}] \tag{6.1}$$

The ET of $AMSR_T$ is shown in Fig. 6-3; its algebraic specification is given in appendix 6.II.3. $AMSR_T$ includes 37 equations each with, on the average, three summands. Most of these are due to unnecessary retransmissions of messages caused by premature time-outs.

Let us compute the sub-behavior, to be denoted by $AMSR_P$, of $AMSR_T$ in which there are no premature time-outs. A premature time-out can be avoided if whenever in $AMSR_T$ a time-out is contending with any other event in the protocol, then the possibility of this time-out occurring is excluded. The reason is that any time-out that occurs before a loss or before allowing a message or its acknowledgment to reach its destination is premature. In other words, events *&lm*, *&ack*, *&la*, *&msg*, *&msg'*, and *&ack'* should have precedence over *&tout*. Therefore, using the *Precedence* function (see definition 3.5), $AMSR_P$ is formally specified by

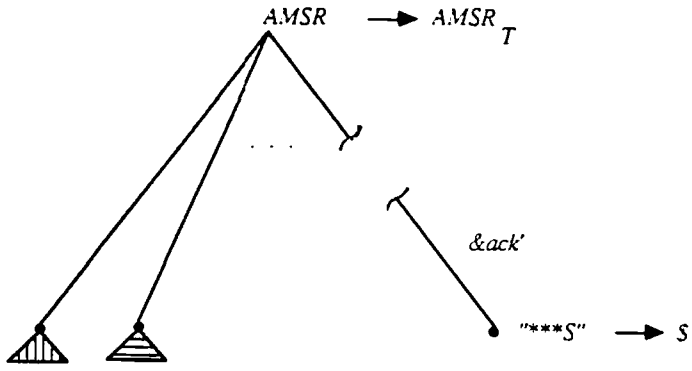


Figure 6-3: ET of the terminating behavior of the send-and-wait protocol

$$AMSR_P = Precedence[AMSR_T\{(\&lm,\&tout),(\&ack,\&tout),(\&la,\&tout),(\&msg,\&tout),(\&msg',\&tout),(\&ack',\&tout)\}] \quad (6.2)$$

The ET describing the execution of $AMSR_P$ is shown in Fig. 6-4. The figure shows three time-out branches that were pruned from $AMSR_P$ as dashed lines. The algebraic specification of $AMSR_P$ is given in appendix 6.I.4. There is a reduction in the number of identifiers in $AMSR_P$, compared to $AMSR_T$, from 37 to just 6. Thus, if the protocol designer is only interested in those behaviors of the protocol with no premature time-outs, he can consider the simpler behavior of $AMSR_P$ instead of $AMSR_T$. $AMSR_T$ and $AMSR_P$ will be used next in specifying the performance of the protocol.

6.2.2. Performance Specification and Analysis

The timing behavior of $AMSR_T$ is depicted in Fig. 6-5 (b). The behavior starts at t_0 when a message arrives at the source. A message arriving at the source when the protocol system is busy is queued for a *waiting time* duration τ_w before being served. The protocol system is busy if the sender is waiting for the acknowledgment of a previously sent message. The *roundtrip delay* τ_d is the time starting with sending a message at the sender until receiving its acknowledgment. Let t_w, and t_d denote the mean of the waiting time and roundtrip delay, respectively.

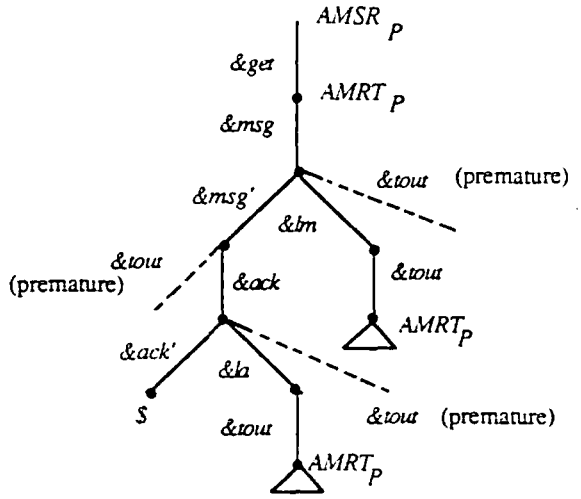


Figure 6-4: ET describing the execution of the send-and-wait protocol with no premature time-outs

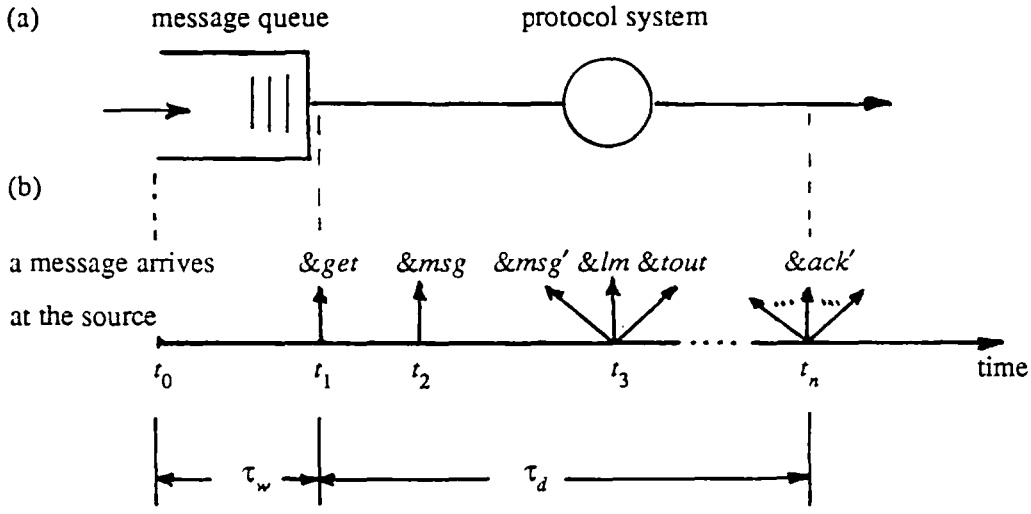


Figure 6-5: (a) Queueing model of the send-and-wait protocol

(b) Timing behavior of $AMSR_T$

The rates of the various events of the protocol can be explained as follows. $\lambda_{\&msg}$ and $\lambda_{\&ack}$ represent message and acknowledgment transmission rates, respectively. For a transmission channel bandwidth b, and assuming that the length of messages and acknowledgments are exponentially distributed with mean l, $\lambda_{\&msg}=\lambda_{\&ack}=b/l$. $\lambda_{\&msg'}$ and $\lambda_{\&ack'}$ represent the average communication delay of messages and acknowledgments in the channels, respectively. $\lambda_{\&lm}$ and $\lambda_{\&la}$ represent the rate of losing messages and acknowledgments, respectively. $\lambda_{\&tout}$ represents the exponential rate of occurrence of time-outs.

The data used throughout this chapter are variations from those used by Molloy [Moll 81] in analyzing a simplified version of the send-and-wait protocol (in which time-out only occurs after a loss). The events' rates are set as follows: $\lambda_{\&msg}=\lambda_{\&ack}=9.375$, $\lambda_{\&msg'}=\lambda_{\&ack'}=74.22$, $\lambda_{\&lm}=\lambda_{\&la}=3.91$, and $\lambda_{\&tout}=1.0$ occurrences/sec. Molloy computed the mean roundtrip delay for this one set of data; the result he obtained using stochastic Petri nets will be shown to agree with ours in section 6.2.2.2.

6.2.2.1. Computation of Optimal Time-out Rate

Time-out is used in the SW protocol to recover from loss of messages and acknowledgments. For the protocol to perform efficiently, the time-out rate has to be set such that both the probability of premature time-outs p_p, and the mean roundtrip delay t_d, are minimized. A minimal probability of premature time-outs ensures that the number of unnecessary retransmissions is minimized. A minimal roundtrip delay ensures that time-out occurs promptly after a loss. However, these two goals are contradictory as shown next. The trade-off between them is studied, and a balanced timing requirement of the protocol is then specified.

Since $AMSR_p$ depicted in Fig. 6-4 represents the behaviors of the protocol in which no premature time-outs occur, then p_p can be specified by

$$p_p = 1 - \mathrm{P}_{AMSR_T}(AMSR_p) \tag{6.3}$$

p_p is plotted against the time-out rate for three different rates of loss in Fig. 6-6. The figure indicates that a change in the rate of loss has a negligible effect on the probability of premature time-outs. This can be explained as follows. Consider the choice sets involved in $AMSR_p$ of Fig. 6-4 in which there is a choice between loss and time-out. In these cases there is also the choice of

&msg' or *&ack'* which have a much higher rate then the rates of loss used in the graph. Therefore, the probability of premature time-outs is mostly unaffected by a change in the rate of loss if the communication delay in the channels is small relative to $1/\lambda_{\&lm}$, which is the most likely case.

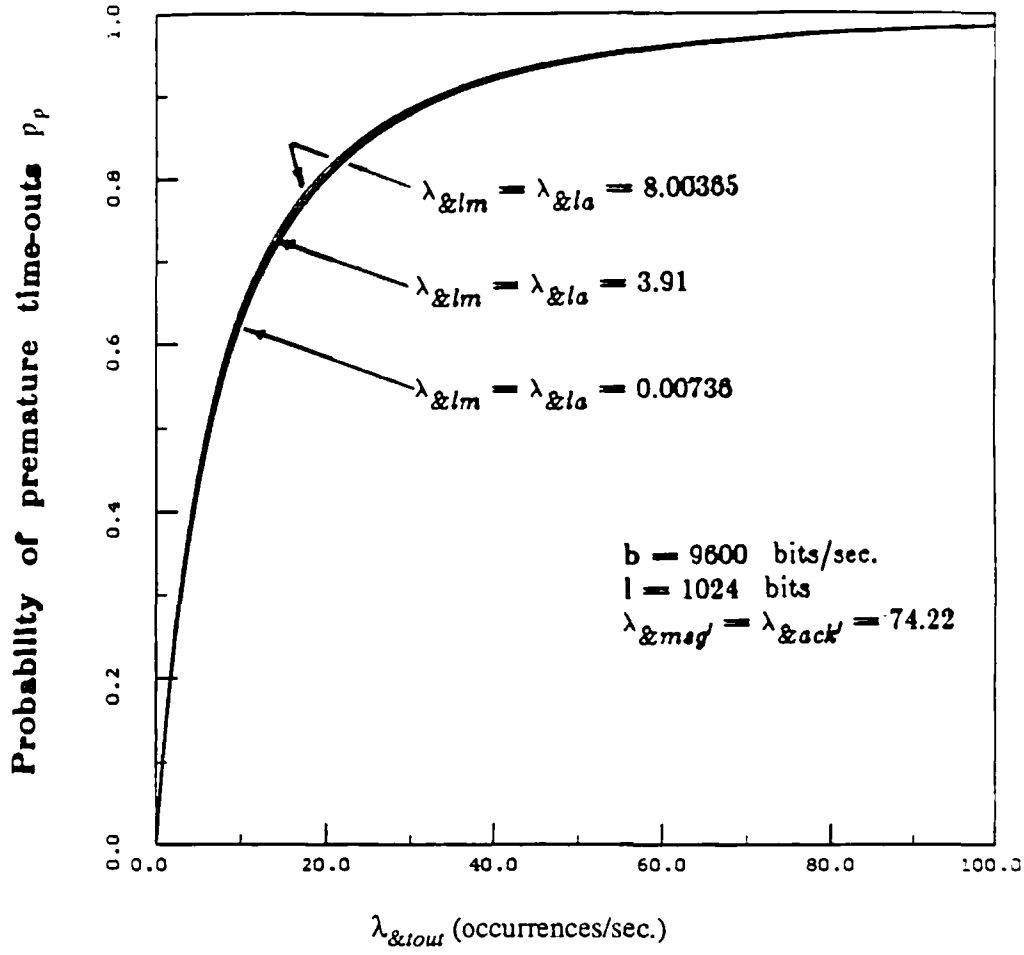


Figure 6-6: The probability of premature time-outs versus the time-out rate for three different rates of loss

The mean roundtrip delay t_d has been defined to represent the mean time duration of the protocol behavior that starts with sending a message at the sender and ends with receiving its acknowledgment. The protocol state corresponding to sending the first copy of a message is $AMRT_T$. t_d can be then specified by

$$t_d = \mathrm{M}_{AMRT_T}(AMRT_T) \tag{6.4}$$

t_d is plotted against the time-out rate for three different rates of loss in Fig. 6-7. For small values of the time-out rate, t_d decreases with an increase in the time-out rate because it takes less time to recover from a loss situation. However, for large values of the time-out rate, the mean roundtrip delay starts to increase. This interesting phenomenon can be attributed to a combination of two factors. First, as is described in the specification of the sender process, an arriving acknowledgment does not preempt a retransmission due to a time-out. Therefore, for a very high time-out rate, there will be a delay until the sender realizes an acknowledgment has arrived. Second, and most important, is the delay of retransmissions because the channels are full with 2 messages/acknowledgment. This delay increases and its effect becomes more noticeable for small values of the rate of loss since the probability of a premature time-out is large. This phenomenon can be avoided if the sender's specification is changed such that it can accept acknowledgments after a time-out occurs and before it retransmits.

Fig. 6-6 and Fig. 6-7 indicate a trade-off between the two performance goals of the protocol for small and medium range values of the time-out rate. This trade-off is also evident in Fig. 6-8 in which by varying the rate of time-out $\lambda_{\&tout}$, the mean roundtrip delay, t_d, is plotted against the probability of premature time-outs p_p, for two different rates of loss.

Power [Schw 80] defined as $(1-p_p)/t_d$ can be used as a balanced performance measure in this case. The objective is then to compute the optimal value of the time-out rate in order to meet the following timing requirement of the protocol:

$$SW\text{–}Timing\text{–}Requirement: \quad Maximize \ \frac{(1-p_p)}{t_d}$$

The optimal time-out rate for three different values of the rate of loss are listed in table 6-1. Time-out rates are computed with an accuracy of at least 1 decimal digit.

6.2.2.2. Specification and Analysis of Mean Waiting Time and Maximum Throughput

Assume that the time-out rate $\lambda_{\&tout}$ is set equal to 0.2 occurrences/sec. The probability of $AMSR_P$ relative to $AMSR_T$ is then more than 90% even for a wide range of the other rates of events in the protocol. Then, the former behavior can be considered as an approximation of latter behavior. This approximation will be assumed throughout this section. The mean roundtrip delay t_d would then be respecified as

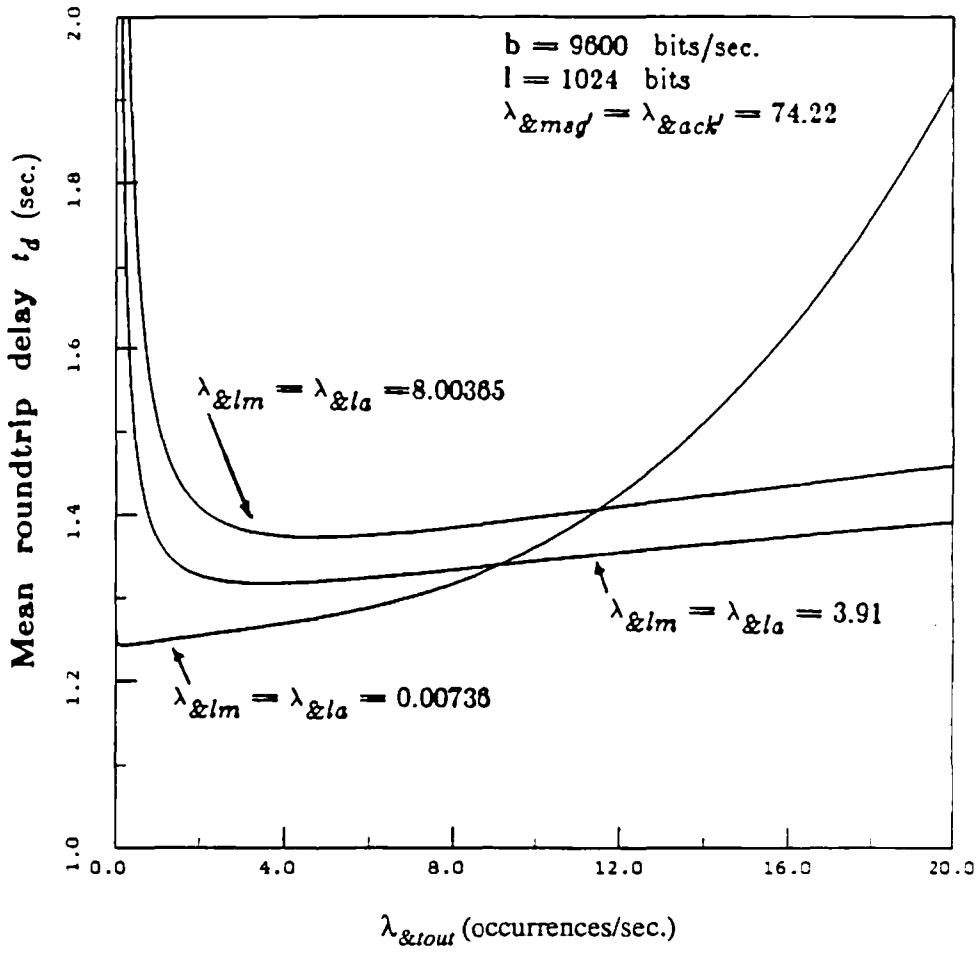


Figure 6-7: The mean roundtrip delay versus the time-out rate for three different rates of loss

$$t_d = \mathrm{M}_{AMRT_P}(AMRT_P) \tag{6.5}$$

For the same data given before, Molloy computed t_d for this protocol using stochastic Petri nets to be 0.3662 sec/message. Using ANALYST t_d is equal to 0.36618 sec/message.

The Pollaczek-Khinchine formula [Klei 75] gives the mean waiting time of Fig. 6-5 to be

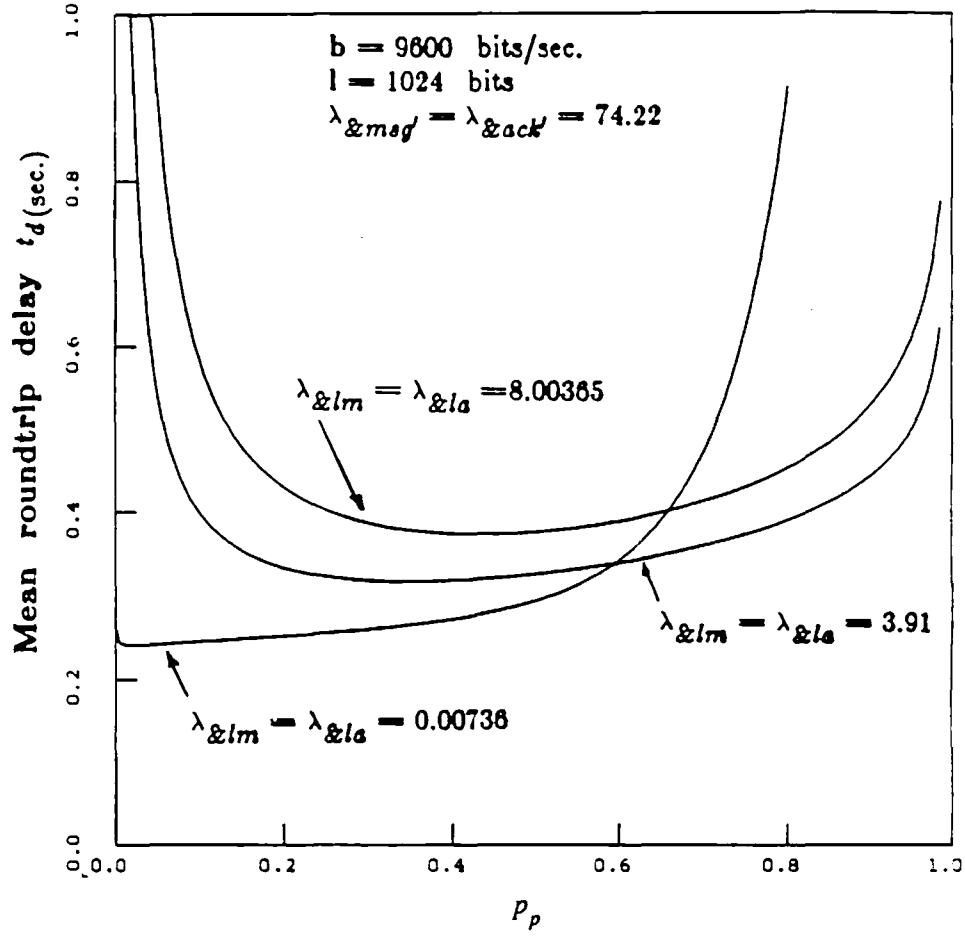


Figure 6-8: The mean roundtrip delay versus the probability of premature time-outs

Rate of loss $\lambda_{\&lm}=\lambda_{\&la}$ occurrences/sec.	Optimal time-out rate $\lambda_{\&tout}$ occurrences/sec.
0.00736	0.07
3.91	1.4
8.00365	1.9

Table 6-1: Optimal time-out rate of the send-and-wait protocol for three different rates of loss

$$t_w = \frac{\lambda [t_d^2 + \sigma_d]}{2[1 - \lambda t_d]} \tag{6.6}$$

where λ denotes the rate of message arrivals, and σ_d denotes the variance of the roundtrip delay τ_d starting from state $AMRT_P$ and thus is given by

$$\sigma_d = V_{AMRT_P}(AMRT_P) \tag{6.7}$$

Now let us analyze the effect of varying the bit error probability p_{ber} of the channels, which is the probability of at least one bit-error (an incorrect message or acknowledgment is considered lost in our specification), on the protocol's waiting time. p_{ber} is related to the rate of loss $\lambda_{\&lm}$ as shown below.

Given p_{ber}, then

$$\Pr(\text{message loss}) = 1 - (1 - p_{ber})^l \tag{6.8}$$

and from $AMSR_P$ in appendix 6.I.4

$$\Pr(\text{message loss}) = \rho_{AM_1RW_P}(\&lm) \tag{6.9}$$

which using Lemma 4.1 is equal to

$$\Pr(\text{message loss}) = \frac{\lambda_{\&lm}}{\lambda_{\&msg'} + \lambda_{\&lm}} \tag{6.10}$$

Thus, for a given p_{ber} and $\lambda_{\&msg'}$, eq. 6.8 and eq. 6.10 above can be used to compute the corresponding $\lambda_{\&lm}$. The same applies to the rate of acknowledgment loss assuming that p_{ber} for both channels is the same.

A plot of the mean waiting time t_w against p_{ber} for several values of channel bandwidth is given in Fig. 6-9. The figure shows that t_w is not affected by a change in p_{ber} for small values of p_{ber}, but then it increases sharply for large values of p_{ber} due to the delay incurred in the retransmissions. A similar result has been obtained by manual analysis [Tows 79].

Maximum throughput TH is the average transmission rate of useful data between the sender and

receiver (i.e., excluding any acknowledgments or retransmissions required by the protocol) achieved when the sender always has a new message to send [Bux 80]. Since for each roundtrip delay, only one message can be delivered to the receiver, *TH* is given by

$$TH = l/t_d \tag{6.11}$$

A plot of *TH* against mean message length l for several p_{ber} in Fig. 6-10 shows a degradation in *TH* for large values of l. This is due to the increase in probability of channel loss for large l, as indicated in eq. 6.8. As p_{ber} decreases this degradation is evident for very long messages and *TH* saturates for a range of l. A similar result has been reported by Bux et al., [Bux 80] from manual analysis of the effect of changing the channels error bit probability on maximum throughput of a more complex data link protocol: a class of HDLC procedures.

In summary, for terrestrial channels (where p_{ber} is very small e.g., 10^{-10}), maximum throughput of the given data link protocol only suffers degradation at large message lengths. The mean waiting time of the protocol is also not affected by any slight change in bit error probability. However, for satellite channels with higher bit error probability, all the protocol parameters should be considered.

6.3. The Alternating Bit Protocol

6.3.1. Functional Specification and Analysis

6.3.1.1. An Algebraic Specification

The configuration of the Alternating Bit (AB) protocol is depicted in Fig. 6-11. The protocol involves four processes: a Sender *S*, a Receiver *R*, a sender-to-receiver channel *M*, and a receiver-to-sender channel *A*. As in the above send-and-wait protocol, the channels are assumed to be FIFO and unreliable. Also, the sender and channel processes are results from concurrent compositions with a time-out process, and a loss process, respectively. The former composition produces the time-out event (*&tout*), and the latter composition produces loss events for messages and acknowledgments with both values of control bit (*&lm*0, *&lm*1, *&la*0, both *&la*1). It is assumed that the protocol system is operating under full load, i.e., the sender always has a message

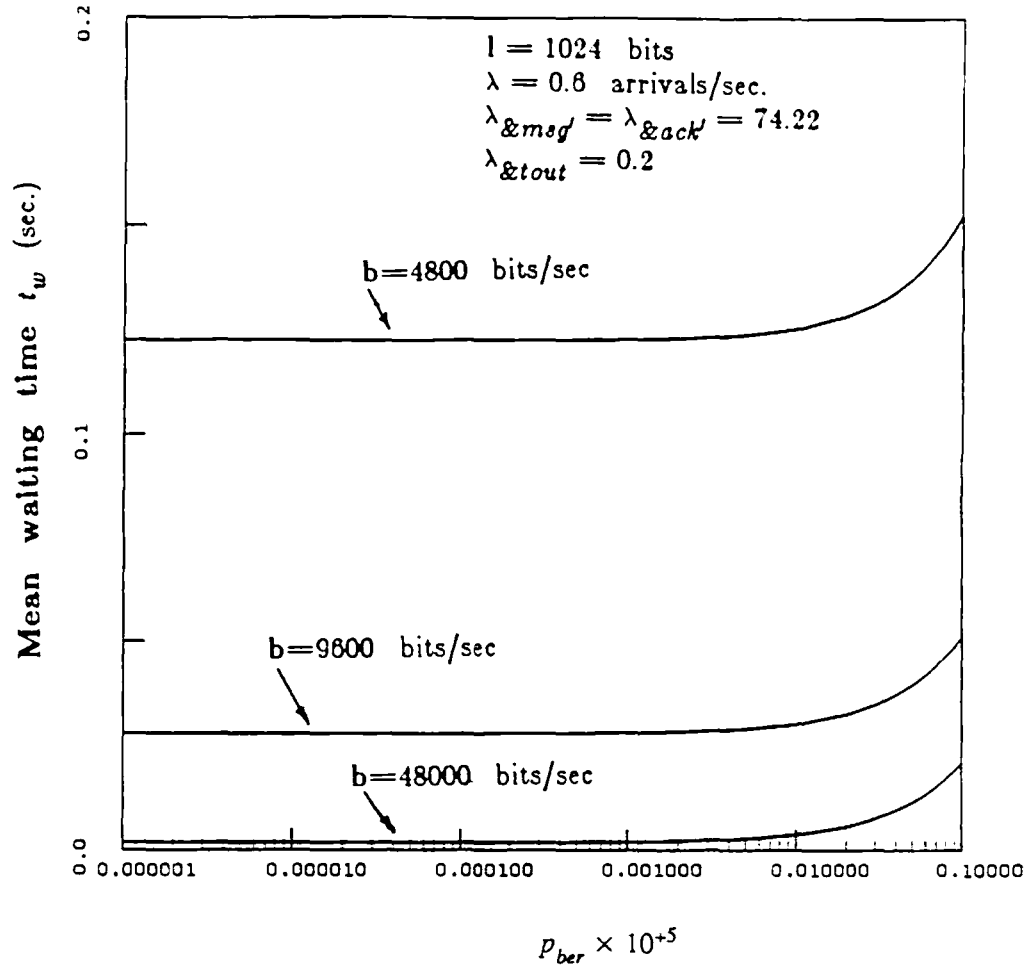


Figure 6-9: The mean waiting time versus bit error probability for various channel bandwidths

to send.

ETs describing the execution of the sender and receiver processes are shown in Fig. 6-12. The sender and receiver processes are assumed to be initially synchronized. That is, the sender is ready to send a message with a control bit 0 and the receiver is expecting a copy of this message to be delivered to it. The sender starts by sending such a message and waits for one of three events to occur. If it receives the expected acknowledgment with control bit 0, it sends the next message in its buffer with a control bit 1. Otherwise, if it receives an old acknowledgment with control bit 1 or a time-out occurs, the same message with control bit 0 is retransmitted. After sending the

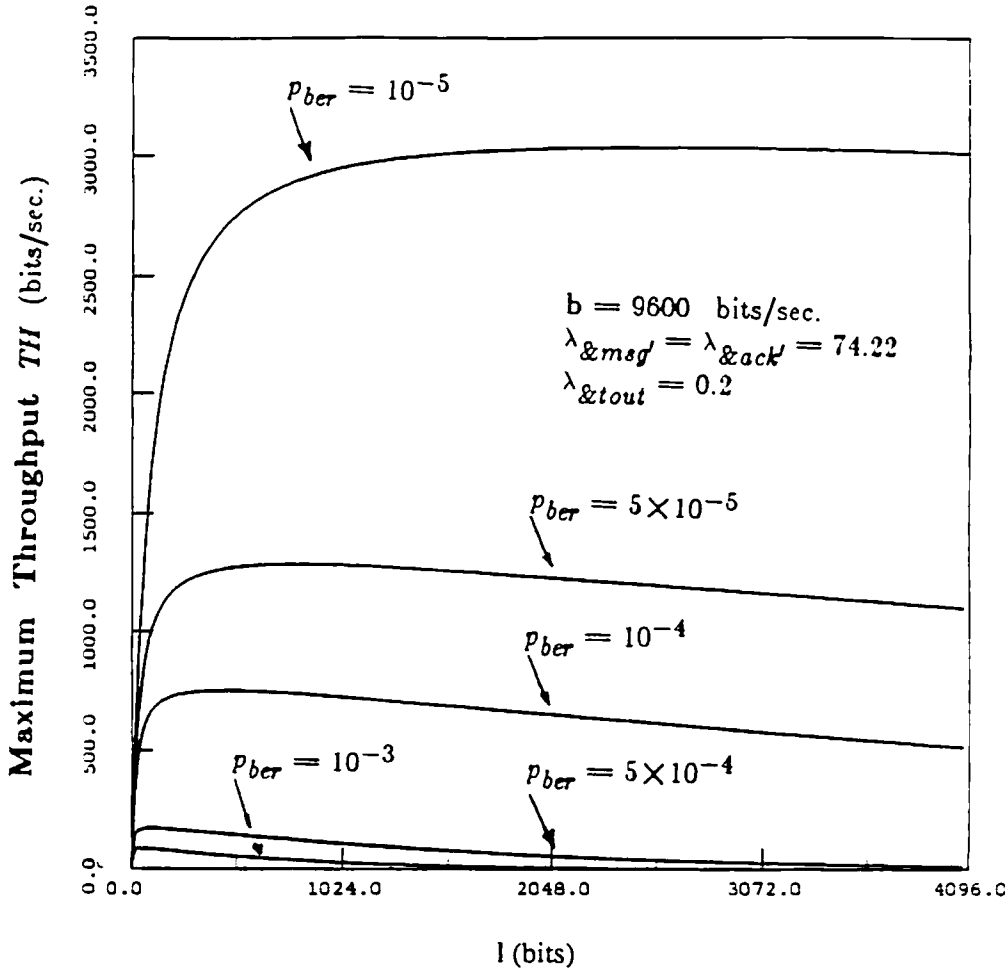


Figure 6-10: Maximum Throughput versus message and acknowledgment lengths for various values of bit error probability

message with control bit 1, the above procedure is repeated with the value of the control bit reversed (i.e., 0 changed to 1 and 1 changed to 0). The receiver's behavior is basically to send acknowledgments for every message received. The values of the control bit of these acknowledgments are the same as those of the received messages.

Generalized specification of the channels that allow an arbitrary bound of I on the number of messages/acknowledgments are given in appendix 6.II.1 as part of the protocol specification. The basic behavior of the channels is similar to the channels employed in the data link protocol discussed in the previous section with the exception that the channels can handle messages/acknowledgments with a 0 or 1 control bit. The channels at any time can only have a

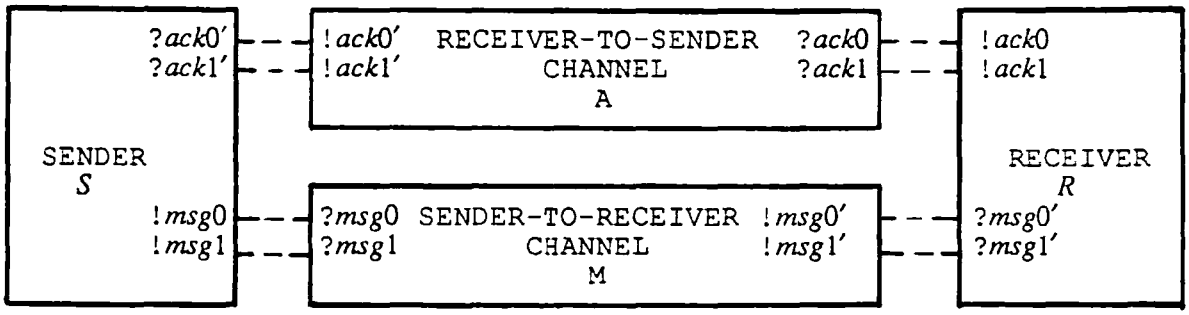


Figure 6-11: Configuration of the Alternating Bit protocol

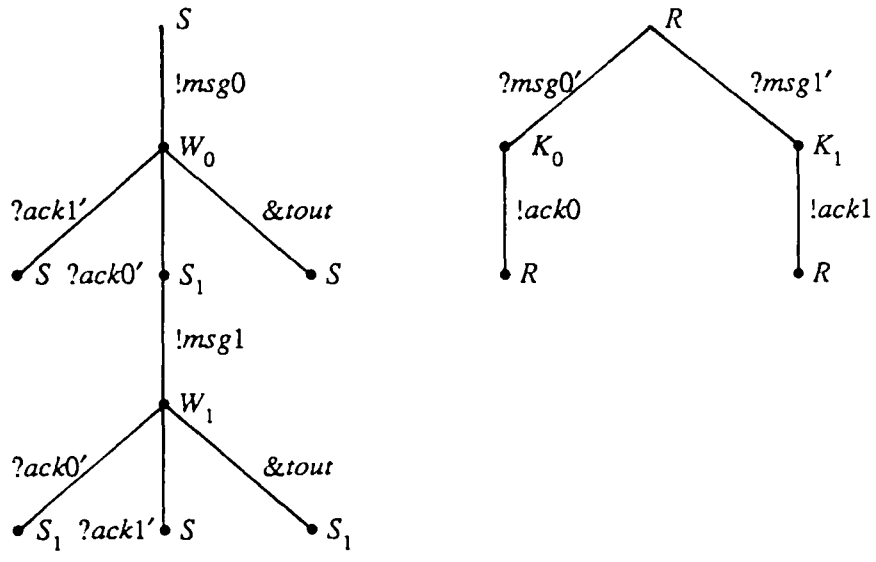


Figure 6-12: ETs describing the execution of the sender and receiver in the Alternating Bit protocol

queue of messages (acknowledgments) with a control bit of 0 followed by a queue of messages (acknowledgments) with a control bit of 1, or vice versa. The reason is the send-and-wait nature of the protocol and the FIFO nature of the channels.

It should be noted that having the sender and the receiver react to receiving an old acknowledgment and an old message, as described above, is not necessary for the correct

functioning of the protocol. The arrival of old messages/acknowledgments can be safely ignored. Such reactions, though, provide a faster error recovery in cases when message or acknowledgments are lost in the channels. Since the channels are FIFO, a receipt of an old message or acknowledgment indicates that the more recently sent one has been lost. The effect of these extra messages and acknowledgments on the performance of the protocol will be analyzed in section 6.3.3.

6.3.1.2. The Concurrent Behavior

The concurrent behavior of the protocol, *AMSR*, for I equal to 1 is given in appendix 6.II.2. From algorithm 5.2 in appendix 5.IV.2 for concurrent composition, the number of equations in the concurrent behavior in the worst case is equal to the product of the number of identifiers in the protocol's processes. By inspection of the protocol specification in appendix 6.II.1, there are 4, 3, 2I+1, and 2I+1 identifiers in *S*, *R*, *M*, and *A* respectively. Therefore, the number of equations included in *AMSR* is bounded by $12(2I+1)^2$. The actual number of equations in *AMSR* for I equal to 1, 2, and 3, is given in Table 6-2.

I	Number of equations
1	70
2	238
3	558

Table 6-2: The number of equations in the concurrent behavior of the Alternating Bit protocol for several values of I

The table shows a drastic increase in the number of equations in the protocol's concurrent behavior. This can be attributed to two factors. The first is premature time-outs which had a similar effect in the SW protocol discussed in the previous section. The second is having extra messages and acknowledgments occupying the protocol system. These are caused by the sender's response to the receipt of an old acknowledgment and the receiver's response to the receipt of an old message. Note that there would be no extra messages and acknowledgments if there were no premature time-outs. The effect of these two factors on the size of the concurrent behavior of the protocol increases as the upper bound on the number of messages in the channels increases.

This same phenomenon has been described by Yemini and Kurose [Yemi 82]. They have shown that it is in its worst form when the channels rates of loss is equal to zero, but the time-out rate is not. In this case, by following the specification of the processes, one finds that the number of message transmissions with control bit equal to 1 (0) is at least equal to the number of previous message transmissions with control bit equal to 0 (1). However, each time a time-out occurs, the former increases by 1 over the latter. This means that eventually the number of message transmissions increases to ∞. Next, we propose a performance measure that would capture this timing error and analyze the effect of the time-out rate, loss rate, and upper bound of number of messages in the channel, on it.

6.3.2. Performance Specification and Analysis

6.3.2.1. Specification and Analysis of Mean Cycle Time

Consider the time the protocol takes to complete one cycle starting at the initial state *AMSR*, and ending with the delivery of acknowledgment with control bit 1 (*&ack*1′) and all four processes are in their initial states. The mean of this *cycle time* is a good measure of the delays caused by time-outs and extra transmissions which have the effect of increasing the time until the protocol system returns to its initial state. It also measures the duration of the cycle repeated by the protocol behavior.

The terminating behavior, $AMSR_T$, which represents the behavior of each of these cycles can be computed by applying the *Terminate* function to the protocol's concurrent behavior, *AMSR*, such that it terminates with event *&ack*1′. It is given by

$$AMSR_T = Terminate[AMSR,\{(\&ack1', AMSR)\}] \tag{6.12}$$

The mean cycle time t_c is then formally specified by

$$t_c = \mathrm{M}_{AMSR_T}(AMSR_T) \tag{6.13}$$

Let us compute those behaviors of the protocol, to be denoted by $AMSR_P$, in which there are no premature time-outs. The mean cycle time of this ideal behavior will be compared with that of *AMSR* below. Similar to the SW protocol of the previous section, the ideal behavior of the

protocol can be obtained by having all events in the protocol, except time-out, take precedence over time-out. This ideal behavior, with no premature time-outs, and therefore, no extra messages or acknowledgments, is specified by

$$AMSR_P = Precedence[AMSR_T, \{(\&lm0,\&tout),(\&ack0,\&tout), (\&la0,\&tout),(\&msg0,\&tout), (\&msg0',\&tout),(\&ack0',\&tout), (\&lm1,\&tout),(\&ack1,\&tout), (\&la1,\&tout),(\&msg1,\&tout), (\&msg1',\&tout), (\&ack1',\&tout)\}] \quad (6.14)$$

The algebraic specification of $AMSR_P$ is given in appendix 6.II.3.

t_c is plotted against the time-out rate $\lambda_{\&tout}$ in Fig. 6-13 and Fig. 6-14 for a loss rate ($\lambda_{\&lm0}=\lambda_{\&lm1}=\lambda_{\&la0}=\lambda_{\&la1}$) equal to 3.91 occurrences/sec and 1.0 occurrences/sec , respectively. In both figures, three different values of I = 1, 2, and 3, are considered. Also included in the two figures are t_c for the protocol's ideal behavior with no premature time-outs. Both figures show that for small values of the time-out rate (in which case the effect of premature time-outs is not yet evident), t_c decreases as the rate of time-out increases since this means a faster recovery from loss situations. Variations in the value of I has a negligible effect on t_c for these small time-out rates. Comparing the two figures shows that variations in t_c occurs at smaller values of the time-out rate when the rate of loss is smaller. The reason for this is apparently that as the rate of loss decreases, a premature time-out occurs for a smaller value of the time-out rate.

The two figures also show that t_c starts to increase as the time-out rate increases and in fact explodes for large values of the time-out rate. This is partly due to the delay of retransmissions when the channels are full as explained in section 6.2.2.1. However, if this was the only reason, then an increase in I should have caused a decrease in t_c. But this is not the case. Therefore, another reason for the increase of t_c is the increasing effect of premature time-outs and extra transmissions on extending the duration of the protocol's cycle time. The increase becomes larger for larger values of I and smaller values of rate of loss in the channels.

The following insights have been gained from the above analysis of the cycle time of the AB protocol:

1. Increasing I results in an explosion in the cycle time for large values of the time-out

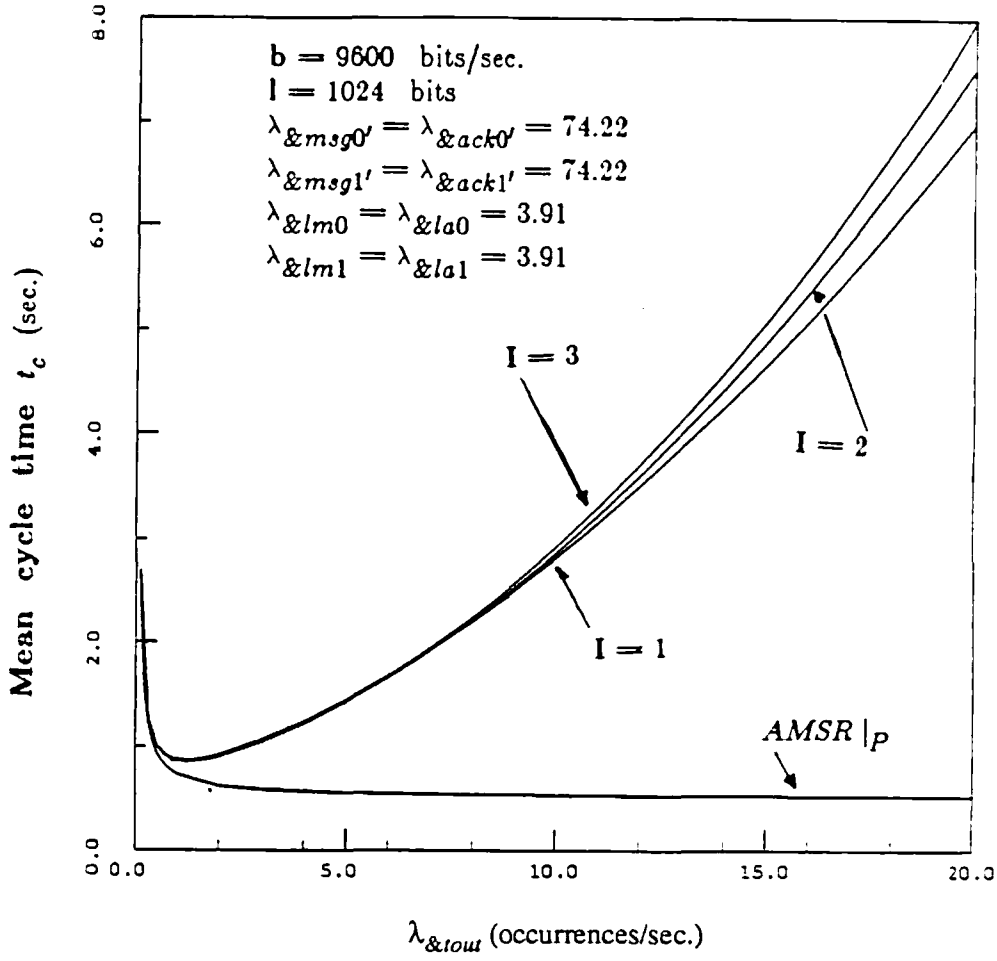


Figure 6-13: The mean cycle time versus the time-out rate for three values of I with rate of loss equal to 3.91 occurrence/sec.

rate because the number of message retransmissions increases. However, for small values of the time-out rate the cycle time is not significantly affected by a change in I.

2. Decreasing the rate of loss in the channels also causes an increase in the cycle time of the protocol especially for large values of the time-out rate. This is due to the delay incurred by retransmissions when the channels are full.

Assuming that the time-out is set such that $AMSR_P$ is a suitable approximation of $AMSR_T$, then the AB protocol's behavior is similar to the behavior of the SW protocol examined in the previous section repeated twice (for values 0 and 1 of the control bit). This becomes also apparent from comparing the behavior of the two protocols given in appendix 6.I.4 and appendix 6.II.3.

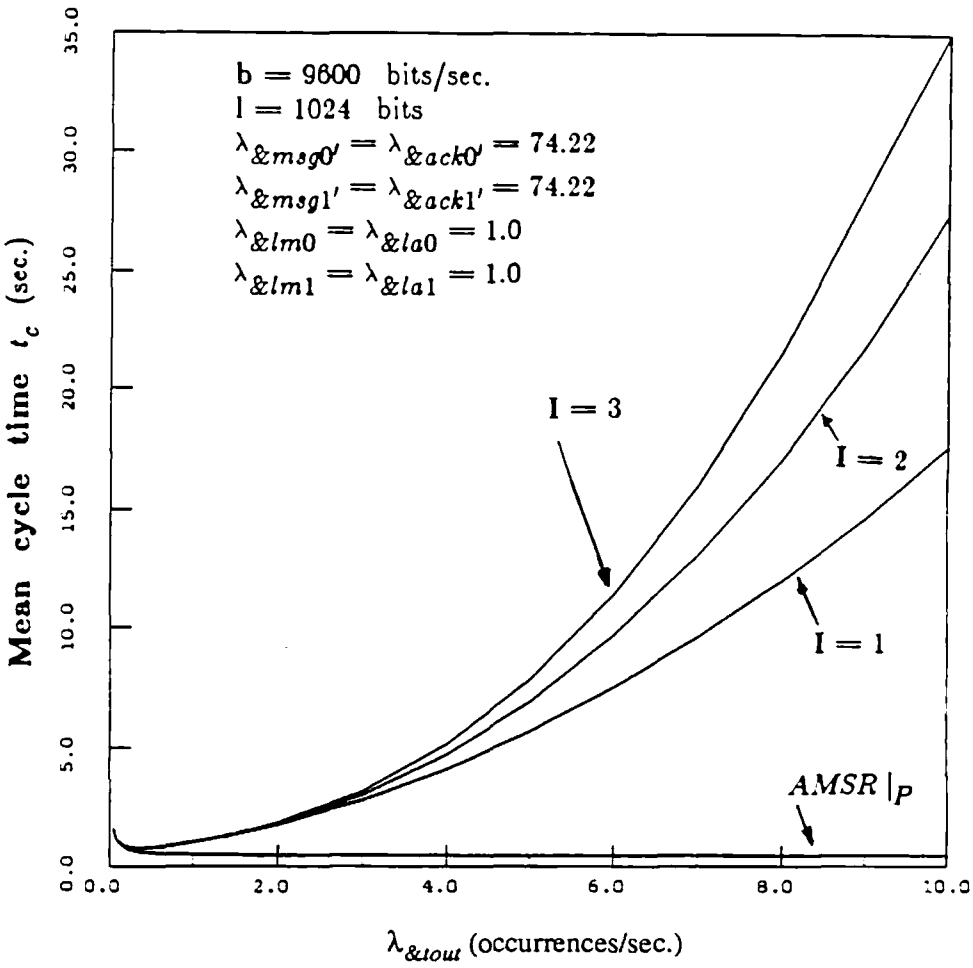


Figure 6-14: The mean cycle time versus the time-out rate for three values of I with the rate of loss equal to 1.0 occurrence/sec.

Therefore, in this case the results of the analyses of maximum throughput and mean waiting time of SW protocol in section 6.2.2.2 would also be valid for the AB protocol.

6.4. Summary

An automated performance analyses of a send-and-wait and the Alternating Bit protocols using ANALYST, has been provided. An optimal setting of the time-out rate of the send-and-wait protocol has been computed. Maximum throughput and mean waiting time performance measures of the protocol have been formally specified and analyzed. Results from these analysis have been

shown to agree remarkably well with results reported in the literature using a manual approach.

The cycle time of the Alternating Bit protocol has been defined and formally specified. It has been shown that it is an adequate measure of the performance of the protocol that captures the effects of premature time-outs and extra transmissions. These two aspects of the behavior of the protocol has been shown previously by Yemini and Kurose to result in a timing error. We provided for the first time a performance analysis of these effects. The effects of the rate of loss, and the upper bound on the number of messages/acknowledgments in the channels on the mean cycle time have been analyzed.

The performance analyses of the send-and-wait and Alternating Bit protocols provided in this chapter are novel in three respects. First, a specification-based performance analysis of these protocol is provided for the first time. Second, a general timing requirement of the send and wait protocol has been specified, and optimal settings of the time-out rate have been computed. Third, a performance analysis of the timing error exhibited by the Alternating Bit protocol reported previously in the literature has been provided.

Appendix 6.I. Algebraic Specifications of the Behaviors of the Send-and-Wait Protocol

6.I.1. Protocol Specification

The specification of the configuration of SW protocol and algebraic specifications of its processes are as follows:

PROTOCOL SW: S,M,R,A

$scope(S,M) = \{msg\}$ $\quad$ $scope(M,R) = \{msg'\}$
$scope(R,A) = \{ack\}$ $\quad$ $scope(A,S) = \{ack'\}$

END

PROCESS S
$S = \&get \bullet T$
$T = !msg \bullet W$
$W = \&tout \bullet T + ?ack' \bullet S$
END

PROCESS R
$R = ?msg' \bullet R_1$
$R_1 = !ack \bullet R$

END

PROCESS M
$M = ?msg \bullet M_1$
$M_1 = !msg' \bullet M + \&lm \bullet M + ?msg \bullet M_2$
$M_2 = !msg' \bullet M_1 + \&lm \bullet M_1$
END

PROCESS A
$A = ?ack \bullet A_1$
$A_1 = !ack' \bullet A + \&la \bullet A + ?ack \bullet A_2$
$A_2 = !ack' \bullet A_1 + \&la \bullet A_1$
END

6.I.2. Concurrent Behavior

$AMSR = \&get \bullet AMRT$
$AMRT = \&msg \bullet AM_1RW$
$AM_1RW = \&msg' \bullet AMR_1W + \&lm \bullet AMRW + \&tout \bullet AM_1RT$
$AMR_1W = \&ack \bullet A_1MRW + \&tout \bullet AMR_1T$
$AMRW = \&tout \bullet AMRT$
$AM_1RT = \&msg' \bullet AMR_1T + \&lm \bullet AMRT + \&msg \bullet AM_2RW$
$A_1MRW = \&ack' \bullet AMSR + \&la \bullet AMRW + \&tout \bullet A_1MRT$
$AMR_1T = \&ack \bullet A_1MRT + \&msg \bullet AM_1R_1W$
$AM_2RW = \&msg' \bullet AM_1R_1W + \&lm \bullet AM_1RW + \&tout \bullet AM_2RT$
$A_1MRT = \&la \bullet AMRT + \&msg \bullet A_1M_1RW$
$AM_1R_1W = \&ack \bullet A_1M_1RW + \&lm \bullet AMR_1W + \&tout \bullet AM_1R_1T$
$AM_2RT = \&msg' \bullet AM_1R_1T + \&lm \bullet AM_1RT$
$A_1M_1RW = \&ack' \bullet AM_1RS + \&la \bullet AM_1RW + \&msg' \bullet A_1MR_1W + \&lm \bullet A_1MRW + \&tout \bullet A_1M_1RT$
$AM_1R_1T = \&ack \bullet A_1M_1RT + \&lm \bullet AMR_1T + \&msg \bullet AM_2R_1W$
$AM_1RS = \&msg' \bullet AMR_1S + \&lm \bullet AMSR + \&get \bullet AM_1RT$
$A_1MR_1W = \&ack' \bullet AMR_1S + \&la \bullet AMR_1W + \&ack \bullet A_2MRW + \&tout \bullet A_1MR_1T$
$A_1M_1RT = \&la \bullet AM_1RT + \&msg' \bullet A_1MR_1T + \&lm \bullet A_1MRT + \&msg \bullet A_1M_2RW$
$AM_2R_1W = \&ack \bullet A_1M_2RW + \&lm \bullet AM_1R_1W + \&tout \bullet AM_2R_1T$
$AMR_1S = \&ack \bullet A_1MRS + \&get \bullet AMR_1T$
$A_2MRW = \&ack' \bullet A_1MRS + \&la \bullet A_1MRW + \&tout \bullet A_2MRT$
$A_1MR_1T = \&la \bullet AMR_1T + \&ack \bullet A_2MRT + \&msg \bullet A_1M_1R_1W$
$A_1M_2RW = \&ack' \bullet AM_2RS + \&la \bullet AM_2RW + \&msg' \bullet A_1M_1R_1W + \&lm \bullet A_1M_1RW + \&tout \bullet A_1M_2RT$
$AM_2R_1T = \&ack \bullet A_1M_2RT + \&lm \bullet AM_1R_1T$
$A_1MRS = \&la \bullet AMSR + \&get \bullet A_1MRT$
$A_2MRT = \&la \bullet A_1MRT + \&msg \bullet A_2M_1RW$
$A_1M_1R_1W = \&ack' \bullet AM_1R_1S + \&la \bullet AM_1R_1W + \&ack \bullet A_2M_1RW + \&lm \bullet A_1MR_1W + \&tout \bullet A_1M_1R_1T$
$AM_2RS = \&msg' \bullet AM_1R_1S + \&lm \bullet AM_1RS + \&get \bullet AM_2RT$
$A_1M_2RT = \&la \bullet AM_2RT + \&msg' \bullet A_1M_1R_1T + \&lm \bullet A_1M_1RT$
$A_2M_1RW = \&ack' \bullet A_1M_1RS + \&la \bullet A_1M_1RW + \&msg' \bullet A_2MR_1W + \&lm \bullet A_2MRW + \&tout \bullet A_2M_1RT$
$AM_1R_1S = \&ack \bullet A_1M_1RS + \&lm \bullet AMR_1S + \&get \bullet AM_1R_1T$

$$A_1M_1R_1T = \&la \bullet AM_1R_1T + \&ack \bullet A_2M_1RT + \&lm \bullet A_1MR_1T + \&msg \bullet A_1M_2R_1W$$
$$A_1M_1RS = \&la \bullet AM_1RS + \&msg' \bullet A_1MR_1S + \&lm \bullet A_1MRS + \&get \bullet A_1M_1RT$$
$$A_2MR_1W = \&ack' \bullet A_1MR_1S + \&la \bullet A_1MR_1W + \&tout \bullet A_2MR_1T$$
$$A_2M_1RT = \&la \bullet A_1M_1RT + \&msg' \bullet A_2MR_1T + \&lm \bullet A_2MRT + \&msg \bullet A_2M_2RW$$
$$A_1M_2R_1W = \&ack' \bullet AM_2R_1S + \&la \bullet AM_2R_1W + \&ack \bullet A_2M_2RW + \&lm \bullet A_1M_1R_1W + \&tout \bullet A_1M_2R_1T$$
$$A_1MR_1S = \&la \bullet AMR_1S + \&ack \bullet A_2MRS + \&get \bullet A_1MR_1T$$
$$A_2MR_1T = \&la \bullet A_1MR_1T + \&msg \bullet A_2M_1R_1W$$
$$A_2M_2RW = \&ack' \bullet A_1M_2RS + \&la \bullet A_1M_2RW + \&msg' \bullet A_2M_1R_1W + \&lm \bullet A_2M_1RW + \&tout \bullet A_2M_2RT$$
$$AM_2R_1S = \&ack \bullet A_1M_2RS + \&lm \bullet AM_1R_1S + \&get \bullet AM_2R_1T$$
$$A_1M_2R_1T = \&la \bullet AM_2R_1T + \&ack \bullet A_2M_2RT + \&lm \bullet A_1M_1R_1T$$
$$A_2MRS = \&la \bullet A_1MRS + \&get \bullet A_2MRT$$
$$A_2M_1R_1W = \&ack' \bullet A_1M_1R_1S + \&la \bullet A_1M_1R_1W + \&lm \bullet A_2MR_1W + \&tout \bullet A_2M_1R_1T$$
$$A_1M_2RS = \&la \bullet AM_2RS + \&msg' \bullet A_1M_1R_1S + \&lm \bullet A_1M_1RS + \&get \bullet A_1M_2RT$$
$$A_2M_2RT = \&la \bullet A_1M_2RT + \&msg' \bullet A_2M_1R_1T + \&lm \bullet A_2M_1RT$$
$$A_1M_1R_1S = \&la \bullet AM_1R_1S + \&ack \bullet A_2M_1RS + \&lm \bullet A_1MR_1S + \&get \bullet A_1M_1R_1T$$
$$A_2M_1R_1T = \&la \bullet A_1M_1R_1T + \&lm \bullet A_2MR_1T + \&msg \bullet A_2M_2R_1W$$
$$A_2M_1RS = \&la \bullet A_1M_1RS + \&msg' \bullet A_2MR_1S + \&lm \bullet A_2MRS + \&get \bullet A_2M_1RT$$
$$A_2M_2R_1W = \&ack' \bullet A_1M_2R_1S + \&la \bullet A_1M_2R_1W + \&lm \bullet A_2M_1R_1W + \&tout \bullet A_2M_2R_1T$$
$$A_2MR_1S = \&la \bullet A_1MR_1S + \&get \bullet A_2MR_1T$$
$$A_1M_2R_1S = \&la \bullet AM_2R_1S + \&ack \bullet A_2M_2RS + \&lm \bullet A_1M_1R_1S + \&get \bullet A_1M_2R_1T$$
$$A_2M_2R_1T = \&la \bullet A_1M_2R_1T + \&lm \bullet A_2M_1R_1T$$
$$A_2M_2RS = \&la \bullet A_1M_2RS + \&msg' \bullet A_2M_1R_1S + \&lm \bullet A_2M_1RS + \&get \bullet A_2M_2RT$$
$$A_2M_1R_1S = \&la \bullet A_1M_1R_1S + \&lm \bullet A_2MR_1S + \&get \bullet A_2M_1R_1T$$

6.I.3. Terminating Behavior

$$AMSR_T = \&get \bullet AMRT_T$$
$$AMRT_T = \&msg \bullet AM_1RW_T$$
$$AM_1RW_T = \&msg' \bullet AMR_1W_T + \&lm \bullet AMRW_T + \&tout \bullet AM_1RT_T$$
$$AMR_1W_T = \&ack \bullet A_1MRW_T + \&tout \bullet AMR_1T_T$$
$$AMRW_T = \&tout \bullet AMRT_T$$
$$AM_1RT_T = \&msg' \bullet AMR_1T_T + \&lm \bullet AMRT_T + \&msg \bullet AM_2RW_T$$

$$A_1MRW_T = \&ack' \bullet \$ + \&la \bullet AMRW_T + \&tout \bullet A_1MRT_T$$
$$AMR_1T_T = \&ack \bullet A_1MRT_T + \&msg \bullet AM_1R_1W_T$$
$$AM_2RW_T = \&msg' \bullet AM_1R_1W_T + \&lm \bullet AM_1RW_T + \&tout \bullet AM_2RT_T$$
$$A_1MRT_T = \&la \bullet AMRT_T + \&msg \bullet A_1M_1RW_T$$
$$AM_1R_1W_T = \&ack \bullet A_1M_1RW_T + \&lm \bullet AMR_1W_T + \&tout \bullet AM_1R_1T_T$$
$$AM_2RT_T = \&msg' \bullet AM_1R_1T_T + \&lm \bullet AM_1RT_T$$
$$A_1M_1RW_T = \&ack' \bullet \$ + \&la \bullet AM_1RW_T + \&msg' \bullet A_1MR_1W_T + \&lm \bullet A_1MRW_T + \&tout \bullet A_1M_1RT_T$$
$$AM_1R_1T_T = \&ack \bullet A_1M_1RT_T + \&lm \bullet AMR_1T_T + \&msg \bullet AM_2R_1W_T$$
$$A_1MR_1W_T = \&ack' \bullet \$ + \&la \bullet AMR_1W_T + \&ack \bullet A_2MRW_T + \&tout \bullet A_1MR_1T_T$$
$$A_1M_1RT_T = \&la \bullet AM_1RT_T + \&msg' \bullet A_1MR_1T_T + \&lm \bullet A_1MRT_T + \&msg \bullet A_1M_2RW_T$$
$$AM_2R_1W_T = \&ack \bullet A_1M_2RW_T + \&lm \bullet AM_1R_1W_T + \&tout \bullet AM_2R_1T_T$$
$$A_2MRW_T = \&ack' \bullet \$ + \&la \bullet A_1MRW_T + \&tout \bullet A_2MRT_T$$
$$A_1MR_1T_T = \&la \bullet AMR_1T_T + \&ack \bullet A_2MRT_T + \&msg \bullet A_1M_1R_1W_T$$
$$A_1M_2RW_T = \&ack' \bullet \$ + \&la \bullet AM_2RW_T + \&msg' \bullet A_1M_1R_1W_T + \&lm \bullet A_1M_1RW_T + \&tout \bullet A_1M_2RT_T$$
$$AM_2R_1T_T = \&ack \bullet A_1M_2RT_T + \&lm \bullet AM_1R_1T_T$$
$$A_2MRT_T = \&la \bullet A_1MRT_T + \&msg \bullet A_2M_1RW_T$$
$$A_1M_1R_1W_T = \&ack' \bullet \$ + \&la \bullet AM_1R_1W_T + \&ack \bullet A_2M_1RW_T + \&lm \bullet A_1MR_1W_T + \&tout \bullet A_1M_1R_1T_T$$
$$A_1M_2RT_T = \&la \bullet AM_2RT_T + \&msg' \bullet A_1M_1R_1T_T + \&lm \bullet A_1M_1RT_T$$
$$A_2M_1RW_T = \&ack' \bullet \$ + \&la \bullet A_1M_1RW_T + \&msg' \bullet A_2MR_1W_T + \&lm \bullet A_2MRW_T + \&tout \bullet A_2M_1RT_T$$
$$A_1M_1R_1T_T = \&la \bullet AM_1R_1T_T + \&ack \bullet A_2M_1RT_T + \&lm \bullet A_1MR_1T_T + \&msg \bullet A_1M_2R_1W_T$$
$$A_2MR_1W_T = \&ack' \bullet \$ + \&la \bullet A_1MR_1W_T + \&tout \bullet A_2MR_1T_T$$
$$A_2M_1RT_T = \&la \bullet A_1M_1RT_T + \&msg' \bullet A_2MR_1T_T + \&lm \bullet A_2MRT_T + \&msg \bullet A_2M_2RW_T$$
$$A_1M_2R_1W_T = \&ack' \bullet \$ + \&la \bullet AM_2R_1W_T + \&ack \bullet A_2M_2RW_T + \&lm \bullet A_1M_1R_1W_T + \&tout \bullet A_1M_2R_1T_T$$
$$A_2MR_1T_T = \&la \bullet A_1MR_1T_T + \&msg \bullet A_2M_1R_1W_T$$
$$A_2M_2RW_T = \&ack' \bullet \$ + \&la \bullet A_1M_2RW_T + \&msg' \bullet A_2M_1R_1W_T + \&lm \bullet A_2M_1RW_T + \&tout \bullet A_2M_2RT_T$$
$$A_1M_2R_1T_T = \&la \bullet AM_2R_1T_T + \&ack \bullet A_2M_2RT_T + \&lm \bullet A_1M_1R_1T_T$$
$$A_2M_1R_1W_T = \&ack' \bullet \$ + \&la \bullet A_1M_1R_1W_T$$

$$+ \&lm \bullet A_2MR_1W_T + \&tout \bullet A_2M_1R_1T_T$$
$$A_2M_2RT_T = \&la \bullet A_1M_2RT_T + \&msg' \bullet A_2M_1R_1T_T$$
$$+ \&lm \bullet A_2M_1RT_T$$
$$A_2M_1R_1T_T = \&la \bullet A_1M_1R_1T_T + \&lm \bullet A_2MR_1T_T$$
$$+ \&msg \bullet A_2M_2R_1W_T$$
$$A_2M_2R_1W_T = \&ack' \bullet \$ + \&la \bullet A_1M_2R_1W_T$$
$$+ \&lm \bullet A_2M_1R_1W_T + \&tout \bullet A_2M_2R_1T_T$$
$$A_2M_2R_1T_T = \&la \bullet A_1M_2R_1T_T + \&lm \bullet A_2M_1R_1T_T$$

6.I.4. Behavior With no Premature Time-outs

$$AMSR_P = \&get \bullet AMRT_P$$
$$AMRT_P = \&msg \bullet AM_1RW_P$$
$$AM_1RW_P = \&msg' \bullet AMR1W_P + \&lm \bullet AMRW_P$$
$$AMR1W_P = \&ack \bullet A_1MRW_P$$
$$AMRW_P = \&tout \bullet AMRT_P$$
$$A_1MRW_P = \&ack' \bullet \$ + \&la \bullet AMRW_P$$

Appendix 6.II. Algebraic Specifications of the Behaviors of the Alternating Bit Protocol

6.II.1. Protocol Specification

The specification of the configuration of Alternating Bit protocol and algebraic specifications of its processes are as follows:

PROTOCOL AB: S,M,R,A

$scope(S,M) = \{msg0, msg1\}$ $scope(M,R) = \{msg0', msg1'\}$
$scope(R,A) = \{ack0, ack1\}$ $scope(A,S) = \{ack0', ack1'\}$

END

PROCESS S
$S = !msg0 \bullet W_0$
$W_0 = ?ack0' \bullet S_1 + \&tout \bullet S + \&ack1' \bullet S$
$S_1 = !msg1 \bullet W_1$
$W_1 = ?ack1' \bullet S + \&tout \bullet S_1 + \&ack0' \bullet S_1$
END

PROCESS R
$R = ?msg0' \bullet K_0 + ?msg1' \bullet K_1$
$K_0 = !ack0 \bullet R$
$K_1 = !ack1 \bullet R$

END

Generalized specifications of channels M and A with an arbitrary bound of I on the number of messages/acknowledgments are given below. For channel M, the queues of messages with a

control bit of 0 and 1 are denoted by μ_0 and μ_1, respectively. Similarly, for channel A, the queues of acknowledgments with a control bit of 0 and 1 are denoted by α_0 and α_1, respectively. Two operations on these queues are used: ''+'' for adding a message to a queue, and ''–'' for removing a message from a queue. Note that the channels at any time can only have a queue of messages (acknowledgments) with a control bit of 0 followed by a queue of messages (acknowledgments) with a control bit of 1, or vice versa. The reason is the send-and-wait nature of the protocol and the FIFO nature of the channels.

PROCESS M

$$M = ?msg0 \bullet M_0 + ?msg1 \bullet M_1$$

$$(|\mu_0| > 0)\ M_{\mu_0,\mu_1} = !msg0' \bullet M_{\mu_0-0,\mu_1} + \&lm0 \bullet M_{\mu_0-0,\mu_1} + \&lm1 \bullet M_{\mu_0,\mu_1-1} + ?msg1 \bullet M_{\mu_0,\mu_1+1} \quad \text{if } |\mu_0+\mu_1| < I$$

$$(|\mu_1| > 0)\ M_{\mu_1,\mu_0} = !msg1' \bullet M_{\mu_1-1,\mu_0} + \&lm0 \bullet M_{\mu_1,\mu_0-0} + \&lm1 \bullet M_{\mu_1-1,\mu_0} + ?msg1 \bullet M_{\mu_1+1,\mu_0} \quad \text{if } |\mu_0+\mu_1| < I$$

END

PROCESS A

$$A = ?ack0 \bullet A_0 + ?ack1 \bullet A_1$$

$$(|\alpha_0| > 0)\ A_{\alpha_0,\alpha_1} = !ack0' \bullet A_{\alpha_0-0,\alpha_1} + \&la0 \bullet A_{\alpha_0-0,\alpha_1} + \&la1 \bullet A_{\alpha_0,\alpha_1-1} + ?ack1 \bullet A_{\alpha_0,\alpha_1+1} \quad \text{if } |\alpha_0+\alpha_1| < I$$

$$(|\alpha_1| > 0)\ A_{\alpha_1,\alpha_0} = !ack1' \bullet A_{\alpha_1-1,\alpha_0} + \&la0 \bullet A_{\alpha_1,\alpha_0-0} + \&la1 \bullet A_{\alpha_1-1,\alpha_0} + ?ack1 \bullet A_{\alpha_1+1,\alpha_0} \quad \text{if } |\alpha_0+\alpha_1| < I$$

END

6.II.2. Concurrent Behavior

This is for the case of an upper bound of I = 1 on the number of messages in the channels.

$$AMSR = \&msg_0 \bullet AM_0W_0R$$
$$AM_0W_0R = \&msg0' \bullet AK_0MW_0 + \&lm_0 \bullet AMW_0R + \&tout \bullet AM_0RS$$
$$AK_0MW_0 = \&ack_0 \bullet A_0MW_0R + \&tout \bullet AK_0MS$$
$$AMW_0R = \&tout \bullet AMSR$$
$$AM_0RS = \&msg0' \bullet AK_0MS + \&lm_0 \bullet AMSR$$
$$A_0MW_0R = \&ack0' \bullet AMS_1R + \&la_0 \bullet AMW_0R + \&tout \bullet A_0MRS$$
$$AK_0MS = \&ack_0 \bullet A_0MRS + \&msg_0 \bullet AK_0M_0W_0$$
$$AMS_1R = \&msg_1 \bullet AM_1W_1R$$
$$A_0MRS = \&la_0 \bullet AMSR + \&msg_0 \bullet A_0M_0W_0R$$
$$AK_0M_0W_0 = \&ack_0 \bullet A_0M_0W_0R + \&lm_0 \bullet AK_0MW_0 + \&tout \bullet AK_0M_0S$$
$$AM_1W_1R = \&msg1' \bullet AK_1MW_1 + \&lm1 \bullet AMW_1R + \&tout \bullet AM_1S_1R$$
$$A_0M_0W_0R = \&ack0' \bullet AM_0S_1R + \&la_0 \bullet AM_0W_0R + \&msg0' \bullet A_0K_0MW_0 + \&lm_0 \bullet A_0MW_0R + \&tout \bullet A_0M_0RS$$
$$AK_0M_0S = \&ack_0 \bullet A_0M_0RS + \&lm_0 \bullet AK_0MS$$
$$AK_1MW_1 = \&ack_1 \bullet A_1MW_1R + \&tout \bullet AK_1MS_1$$
$$AMW_1R = \&tout \bullet AMS_1R$$
$$AM_1S_1R = \&msg1' \bullet AK_1MS_1 + \&lm1 \bullet AMS_1R$$
$$AM_0S_1R = \&msg0' \bullet AK_0MS_1 + \&lm_0 \bullet AMS_1R$$
$$A_0K_0MW_0 = \&ack0' \bullet AK_0MS_1 + \&la_0 \bullet AK_0MW_0 + \&tout \bullet A_0K_0MS$$
$$A_0M_0RS = \&la_0 \bullet AM_0RS + \&msg0' \bullet A_0K_0MS + \&lm_0 \bullet A_0MRS$$
$$A_1MW_1R = \&ack1' \bullet AMSR + \&la1 \bullet AMW_1R + \&tout \bullet A_1MS_1R$$
$$AK_1MS_1 = \&ack_1 \bullet A_1MS_1R + \&msg_1 \bullet AK_1M_1W_1$$
$$AK_0MS_1 = \&ack_0 \bullet A_0MS_1R + \&msg_1 \bullet AK_0M_1W_1$$
$$A_0K_0MS = \&la_0 \bullet AK_0MS + \&msg_0 \bullet A_0K_0M_0W_0$$
$$A_1MS_1R = \&la1 \bullet AMS_1R + \&msg_1 \bullet A_1M_1W_1R$$
$$AK_1M_1W_1 = \&ack_1 \bullet A_1M_1W_1R + \&lm1 \bullet AK_1MW_1 + \&tout \bullet AK_1M_1S_1$$
$$A_0MS_1R = \&la_0 \bullet AMS_1R + \&msg_1 \bullet A_0M_1W_1R$$
$$AK_0M_1W_1 = \&ack_0 \bullet A_0M_1W_1R + \&lm1 \bullet AK_0MW_1 + \&tout \bullet AK_0M_1S_1$$
$$A_0K_0M_0W_0 = \&ack0' \bullet AK_0M_0S_1 + \&la_0 \bullet AK_0M_0W_0 + \&lm_0 \bullet A_0K_0MW_0 + \&tout \bullet A_0K_0M_0S$$
$$A_1M_1W_1R = \&ack1' \bullet AM_1RS + \&la1 \bullet AM_1W_1R + \&msg1' \bullet A_1K_1MW_1 + \&lm1 \bullet A_1MW_1R + \&tout \bullet A_1M_1S_1R$$
$$AK_1M_1S_1 = \&ack_1 \bullet A_1M_1S_1R + \&lm1 \bullet AK_1MS_1$$
$$A_0M_1W_1R = \&ack0' \bullet AM_1S_1R + \&la_0 \bullet AM_1W_1R + \&msg1' \bullet A_0K_1MW_1 + \&lm1 \bullet A_0MW_1R + \&tout \bullet A_0M_1S_1R$$
$$AK_0MW_1 = \&ack_0 \bullet A_0MW_1R + \&tout \bullet AK_0MS_1$$
$$AK_0M_1S_1 = \&ack_0 \bullet A_0M_1S_1R + \&lm1 \bullet AK_0MS_1$$
$$AK_0M_0S_1 = \&ack_0 \bullet A_0M_0S_1R + \&lm_0 \bullet AK_0MS_1$$
$$A_0K_0M_0S = \&la_0 \bullet AK_0M_0S + \&lm_0 \bullet A_0K_0MS$$
$$AM_1RS = \&msg1' \bullet AK_1MS + \&lm1 \bullet AMSR$$
$$A_1K_1MW_1 = \&ack1' \bullet AK_1MS + \&la1 \bullet AK_1MW_1 + \&tout \bullet A_1K_1MS_1$$
$$A_1M_1S_1R = \&la1 \bullet AM_1S_1R + \&msg1' \bullet A_1K_1MS_1 + \&lm1 \bullet A_1MS_1R$$
$$A_0K_1MW_1 = \&ack0' \bullet AK_1MS_1 + \&la_0 \bullet AK_1MW_1 + \&tout \bullet A_0K_1MS_1$$

$$A_0MW_1R = \&ack0' \bullet AMS_1R + \&la_0 \bullet AMW_1R + \&tout \bullet A_0MS_1R$$
$$A_0M_1S_1R = \&la_0 \bullet AM_1S_1R + \&msg1' \bullet A_0K_1MS_1 + \&lm1 \bullet A_0MS_1R$$
$$A_0M_0S_1R = \&la_0 \bullet AM_0S_1R + \&msg0' \bullet A_0K_0MS_1 + \&lm_0 \bullet A_0MS_1R$$
$$AK_1MS = \&ack_1 \bullet A_1MRS + \&msg_0 \bullet AK_1M_0W_0$$
$$A_1K_1MS_1 = \&la1 \bullet AK_1MS_1 + \&msg_1 \bullet A_1K_1M_1W_1$$
$$A_0K_1MS_1 = \&la_0 \bullet AK_1MS_1 + \&msg_1 \bullet A_0K_1M_1W_1$$
$$A_0K_0MS_1 = \&la_0 \bullet AK_0MS_1 + \&msg_1 \bullet A_0K_0M_1W_1$$
$$A_1MRS = \&la1 \bullet AMSR + \&msg_0 \bullet A_1M_0W_0R$$
$$AK_1M_0W_0 = \&ack_1 \bullet A_1M_0W_0R + \&lm_0 \bullet AK_1MW_0 + \&tout \bullet AK_1M_0S$$
$$A_1K_1M_1W_1 = \&ack1' \bullet AK_1M_1S + \&la1 \bullet AK_1M_1W_1 + \&lm1 \bullet A_1K_1MW_1 + \&tout \bullet A_1K_1M_1S_1$$
$$A_0K_1M_1W_1 = \&ack0' \bullet AK_1M_1S_1 + \&la_0 \bullet AK_1M_1W_1 + \&lm1 \bullet A_0K_1MW_1 + \&tout \bullet A_0K_1M_1S_1$$
$$A_0K_0M_1W_1 = \&ack0' \bullet AK_0M_1S_1 + \&la_0 \bullet AK_0M_1W_1 + \&lm1 \bullet A_0K_0MW_1 + \&tout \bullet A_0K_0M_1S_1$$
$$A_1M_0W_0R = \&ack1' \bullet AM_0RS + \&la1 \bullet AM_0W_0R + \&msg0' \bullet A_1K_0MW_0 + \&lm_0 \bullet A_1MW_0R + \&tout \bullet A_1M_0RS$$
$$AK_1MW_0 = \&ack_1 \bullet A_1MW_0R + \&tout \bullet AK_1MS$$
$$AK_1M_0S = \&ack_1 \bullet A_1M_0RS + \&lm_0 \bullet AK_1MS$$
$$AK_1M_1S = \&ack_1 \bullet A_1M_1RS + \&lm1 \bullet AK_1MS$$
$$A_1K_1M_1S_1 = \&la1 \bullet AK_1M_1S_1 + \&lm1 \bullet A_1K_1MS_1$$
$$A_0K_1M_1S_1 = \&la_0 \bullet AK_1M_1S_1 + \&lm1 \bullet A_0K_1MS_1$$
$$A_0K_0MW_1 = \&ack0' \bullet AK_0MS_1 + \&la_0 \bullet AK_0MW_1 + \&tout \bullet A_0K_0MS_1$$
$$A_0K_0M_1S_1 = \&la_0 \bullet AK_0M_1S_1 + \&lm1 \bullet A_0K_0MS_1$$
$$A_1K_0MW_0 = \&ack1' \bullet AK_0MS + \&la1 \bullet AK_0MW_0 + \&tout \bullet A_1K_0MS$$
$$A_1MW_0R = \&ack1' \bullet AMSR + \&la1 \bullet AMW_0R + \&tout \bullet A_1MRS$$
$$A_1M_0RS = \&la1 \bullet AM_0RS + \&msg0' \bullet A_1K_0MS + \&lm_0 \bullet A_1MRS$$
$$A_1M_1RS = \&la1 \bullet AM_1RS + \&msg1' \bullet A_1K_1MS + \&lm1 \bullet A_1MRS$$
$$A_1K_0MS = \&la1 \bullet AK_0MS + \&msg_0 \bullet A_1K_0M_0W_0$$
$$A_1K_1MS = \&la1 \bullet AK_1MS + \&msg_0 \bullet A_1K_1M_0W_0$$
$$A_1K_0M_0W_0 = \&ack1' \bullet AK_0M_0S + \&la1 \bullet AK_0M_0W_0 + \&lm_0 \bullet A_1K_0MW_0 + \&tout \bullet A_1K_0M_0S$$
$$A_1K_1M_0W_0 = \&ack1' \bullet AK_1M_0S + \&la1 \bullet AK_1M_0W_0 + \&lm_0 \bullet A_1K_1MW_0 + \&tout \bullet A_1K_1M_0S$$
$$A_1K_0M_0S = \&la1 \bullet AK_0M_0S + \&lm_0 \bullet A_1K_0MS$$
$$A_1K_1MW_0 = \&ack1' \bullet AK_1MS + \&la1 \bullet AK_1MW_0 + \&tout \bullet A_1K_1MS$$
$$A_1K_1M_0S = \&la1 \bullet AK_1M_0S + \&lm_0 \bullet A_1K_1MS$$

6.II.3. Behavior With no Premature Time-outs

$$AMSR_P = \&msg0 \bullet AM_0W_0R_P$$
$$AM_0W_0R_P = \&msg0' \bullet AK_0MW_{0P} + \&lm0 \bullet AMW_0R_P$$
$$AK_0MW_{0P} = \&ack0 \bullet A_0MW_0R_P$$
$$AMW_0R_P = \&tout \bullet AMSR_P$$
$$A_0MW_0R_P = \&ack0' \bullet AMS_1R_P + \&la0 \bullet AMW_0R_P$$
$$AMS_1R_P = \&msg1 \bullet AM_1W_1R_P$$
$$AM_1W_1R_P = \&msg1' \bullet AK_1MW_{1P} + \&lm1 \bullet AMW_1R_P$$
$$AK_1MW_{1P} = \&ack1 \bullet A_1MW_1R_P$$
$$AMW_1R_P = \&tout \bullet AMS_1R_P$$
$$A_1MW_1R_P = \&ack1' \bullet \$ + \&la1 \bullet AMW_1R_P$$

Chapter 7

Performance Analysis of a Two Phase Locking Protocol

7.1. Introduction

In a distributed data base system, data items are distributed among several sites. User processes, at possibly different sites, execute *transactions* that are allowed to concurrently access and modify the data items. Clearly, such concurrent access has to be controlled in order to maintain a consistent state of the data base. Locking is one policy that has been used for that purpose. In a locking protocol, access to a data item is exclusive for the transaction that owns its lock. Eswaren, et. al., [Eswa 76] have shown that consistency is maintained by locking protocols if transactions do not request new locks after releasing a lock.

In this chapter we apply the methodology and tools developed to study the performance of a two phase locking (2PL) protocol used for concurrency control in data base systems [Bern 79]. In a 2PL protocol, each transactions passes through a *growing phase*, *commits*, and then goes through a *shrinking phase*. In the growing phase, a transaction goes through a loop of performing some processing actions. Whenever it needs a data item, it sends a locking request to the concerned data item, then continues processing after its request is granted. The growing phase ends when the transaction commits i.e., all its actions are guaranteed even if the transaction later aborts (due to failure of its process, for example). In the shrinking phase, a transaction releases all acquired locks in the same order in which they were acquired and *terminates*.

The 2PL protocol ensures consistency of the data items, but it does not guarantee absence of deadlock situations. Such situations may arise between two transactions if each is waiting for a lock acquired by the other. Deadlock can be avoided if each transaction locks all data items required by a transaction before initiating it (*static locking*). Otherwise, a mechanism has to be employed to recover from deadlock situations. The typical mechanism used for deadlock

detection and recovery involves elaborate computations and checks of wait-for graphs [Mena 79]. In this study, time-outs, as suggested in [Balt 82, Ceri 84], are used for deadlock recovery. That is, if a time-out occurs while a transaction is waiting for lock acquisition, it suspects that it is involved in a deadlock, aborts, and restarts. The choice of a time-out rate that would ensure that the probability of a time-out occurring unnecessarily is minimal and that a time-out occurs promptly after a deadlock is clearly an important performance problem.

The first performance problem of the two phase locking protocol to be examined in this chapter is to analyze the effect of varying the time-out rate on the performance of the protocol. No previous work has been reported in that respect. A too large time-out rate would cause a transaction to unnecessarily abort and restart thus decreasing the effective throughput of the protocol (thrashing effect). A too small time-out rate would cause a transaction to wait for a long time after a deadlock situation has occurred before aborting and restarting. As a consequence, the response time of the transaction would be degraded. Similar to the send-and-wait protocol of section 6.2, this trade-off involves few performance parameters:

1. Time-out rate $\lambda_{\&t}$: the rate at which the transaction aborts.
2. Probability of unnecessary time-out p_t: the probability that a transaction will time out without being involved in a deadlock. It is used as a measure of unnecessary aborts and restarts.
3. Transaction mean response time t_r: the mean time from the start of a new transaction until it commits and releases all its locks, including aborts and restarts.

By varying the time-out rate, the trade-off between the two objectives of a minimal probability of unnecessary time-outs and mean response time of a transaction, is demonstrated. Then, similar to the approach followed in chapter 6, a balanced objective is chosen to maximize the *power* measure of transactions running on a process defined as $(1-p_t)/t_r$. An optimal value of the time-out rate that meets this timing requirement is thus computed. Note that being able to compute an optimal setting of the time-out rate suggests that it may be an alternative for the elaborate computations involved in the deadlock detection mechanism of the wait-for graphs. The advantage of the former over the latter is that much less time is involved in the detection.

Several performance measures of the two phase locking protocol, such as the probability of deadlock, are also of key importance. Work reported on analyzing such performance measures of

the protocol used either simulation or a manual analytic approach, see for instance [Poti 80, Shum 81, Ches 83, Mitr 84, Morr 84]. *The second objective of this chapter is then to use the methodology and the developed tools to automatically analyze a transaction's mean response time and probability of deadlock.* The effects of varying several performance parameters of the protocol, such as the access rates of different data items or the length of transactions, on these performance measures are examined.

The rest of this chapter is organized as follows. In the next section, an algebraic specification of the protocol is provided, its concurrent behavior is computed, and sub-behaviors of this concurrent behavior that are required for performance specification are specified and derived. In section 7.3, the timing behavior of the protocol is examined, and the two performance problems described above are studied. A summary of the results presented in this chapter is given in section 7.5.

7.2. Functional Specification and Analysis

7.2.1. An Algebraic Specification

Consider a distributed data base system with M logical processes, and N distinct data items each with a scheduler process associated with it. Let M denote the set $\{i;\ i=1,...,M\}$, and N denote the set $\{j;\ j=1,...,N\}$. The communications between a process P_i and a data item D_j are depicted in Fig. 7-1. There are three ports through which they interact: a port a_{ij} for messages to acquire a new lock to the data item, a port l_{ij} for messages to grant a lock, and a port r_{ij} for messages to release a lock. No distinction is drawn between read and write locks.

Simplified versions of the ET of a process and a data item scheduler are depicted in Fig. 7-2 and Fig. 7-3, respectively. In these figures, events denoting communications between a process P_i and data item scheduler D_j are described by a subscript ij. Internal events (produced from previous concurrent compositions) in a process P_i are associated with subscript i (except $\&p_{ij}$ representing transactions on process P_i deciding it needs to lock data item j).

The execution of P_i, as shown in Fig. 7-2, starts by running a new transaction which may execute some internal processing and then decides it needs a lock to data item j ($\&p_{ij}$). A transaction attempts to lock a data item only when it is required during the growing phase (*dynamic locking*).

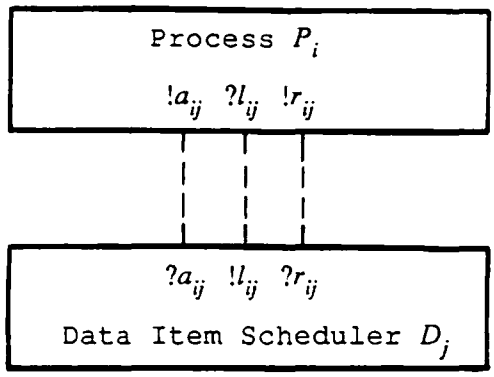
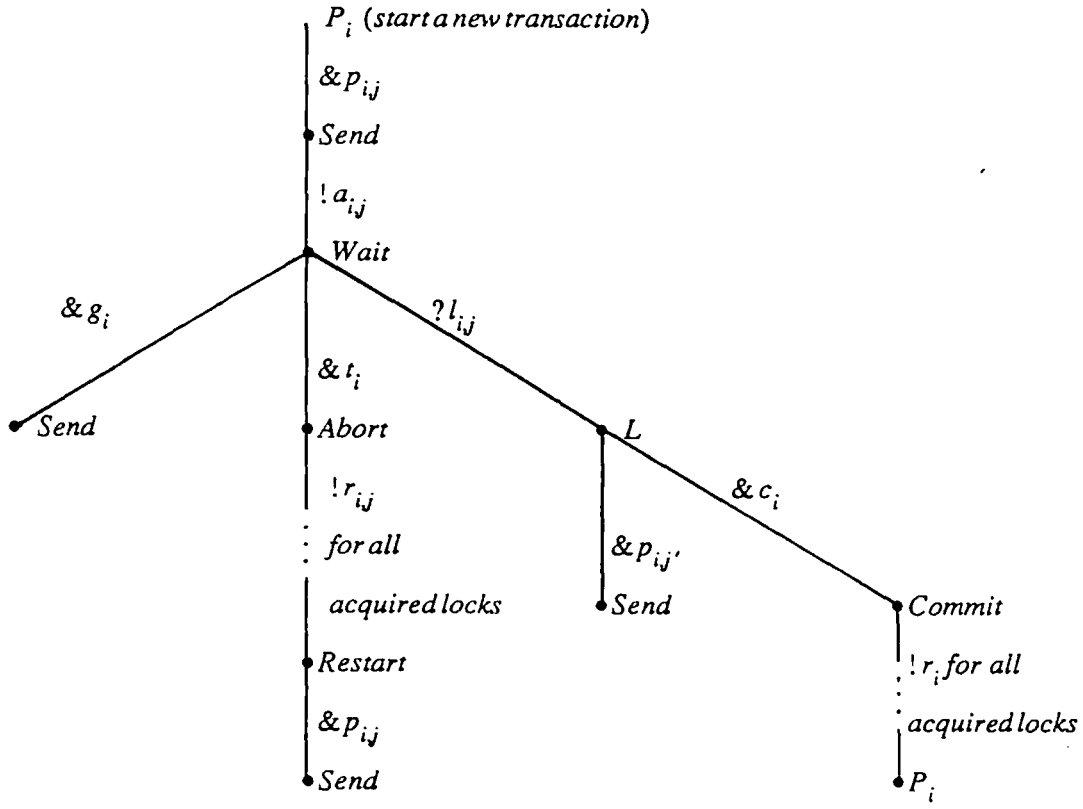


Figure 7-1: Communications between a process and a data item scheduler in a distributed data base system

It sends a request to it ($!a_{ij}$) and waits for a grant of its request ($?l_{ij}$) upon which it either continues processing and acquiring more locks, or decides to commit ($\&c_i$). If after a certain waiting period the locking request is not granted, the transaction decides to try again ($\&g_i$) and sends another request. However, if the time-out occurs ($\&t_i$) before the locking request is granted, the transaction suspects that it is involved in a deadlock, aborts, and restarts it. When aborting or committing, a transaction releases all the acquired locks ($!r_{ij}$) in the same order in which they have been acquired. We assume that there is always a transaction waiting to be executed on each process; therefore, after a transaction commits and terminates a new transaction is started immediately. In addition, we assume that the behavior of a restarted transaction is independent of that of the previously aborted transactions.

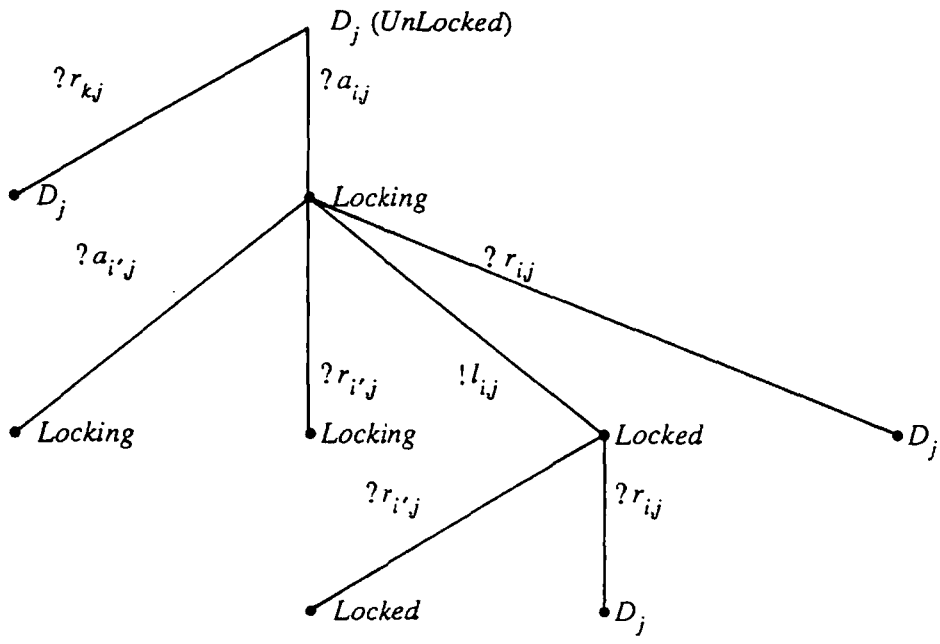
The behavior of a data item scheduler D_j, as shown in Fig. 7-3, starts at a state in which it is waiting for a locking request. The first locking request it receives ($?a_{ij}$) is granted and the scheduler is then in a locking state. Subsequent locking requests received while it is still locking are ignored. A grant of the first received locking request ($!l_{ij}$) is sent to the source process and the data item is locked. The data item remains locked until it receives a release request from the transaction on that process ($?r_{ij}$). Release requests received from transactions on other processes, which are aborting, are ignored.



where $j \neq j'$; $j,j' \in N$; and $i \in M$

Figure 7-2: A simplified ET of a process in the 2PL protocol

Algebraic specifications of a process and a data item scheduler are given in appendix 7.I.1. These specifications follow the simpler corresponding ETs in Fig. 7-2 and Fig. 7-3, but with the following additions. First, since a transaction may be involved in deadlock only if it has already locked one data item, time-out is not allowed when a transaction is waiting for its first lock. Second, identifiers of a process specification, except for the initial identifier, are associated with an ordered list of acquired, and awaited, data item numbers. This allows the order of acquiring locks to be remembered and thus to release them in that order in the shrinking phase. In addition, identifiers of a data item scheduler specification are associated with the process number running the transaction owning its lock in order to distinguish between release requests to respond to when the data item is locked. A glossary of the identifiers used in the algebraic specifications, and the



where $i \neq i'$; $i,i',k \in M$; and $j \in N$

Figure 7-3: A simplified ET of a data item scheduler in the 2PL protocol

states they represent, are given in Table 7-1. Identifiers are associated with subscripts denoting the identity of the process or data item scheduler whose behavior they describe.

7.2.2. The Concurrent Behavior

The concurrent behavior C of the specified 2PL protocol with M processes and N data item schedulers is given by

$$C = P_1 \mid P_2 \mid ... \mid P_M \mid D_1 \mid D_2 \mid ... \mid D_N \tag{7.1}$$

The concurrent behavior for M=1 and N=2 is given in appendix 7.I.2. Generally, the time and space complexities of obtaining the concurrent behavior of the protocol are shown to be of $O(N!^{M} \cdot M^{2N})$.

These explosive time and space complexities are due largely to that every process has a different

P_i	process P_i is starting a new transaction.
$S_i\ j_1 j_2 \ldots j_n$	transaction on process P_i has acquired locks for data items $j_1\ j_2 \ldots j_{n-1}$ and has decided to send a locking request to data item j_n.
$W_i\ j_1 j_2 \ldots j_n$	transaction on process P_i has acquired locks for data items $j_1\ j_2 \ldots j_{n-1}$ and is waiting for lock of data item j_n.
$L_i\ j_1 j_2 \ldots j_n$	transaction on process P_i has acquired locks for data items $j_1\ j_2 \ldots j_n$.
$C_i\ j_1 j_2 \ldots j_n$	transaction on process P_i has decided to commit.
$A_i\ j_1 j_2 \ldots j_n$	transaction on process P_i is aborting.
R_i	transaction on process P_i is restarting.
D_j	data item D_j is unlocked.
$E_j\ i$	data item D_j is being locked by transaction on process P_i.
$F_j\ i$	data item D_j is locked by transaction on process P_i.

where $n \leq \mathbf{N}$

Table 7-1: Glossary of identifiers used in the specification of the two phase locking protocol

identifier to describe its behavior for every possible sequence of locks acquired. Every data item scheduler has also a different identifier for every possible process running a transaction that may lock it. In addition, there are behaviors in which transactions unnecessarily time-out and abort even though they are not involved in deadlock, and unnecessarily retry for awaited locks even though they are not yet available. Consequently, generating the concurrent behavior of the 2PL protocol with large numbers of communicating processes and data items is very expensive. Subsequently in this chapter, we will examine only cases of both M and N equal to at most 2. Even in this case the concurrent behavior includes 580 equations which makes its algebraic specification too large to list.

The concurrent behavior C is cyclic describing the execution of several successive transactions on the processes in the data base. Four sub-behaviors of C will be derived next to be used in performance specification. The first two will be used in specifying a transaction's mean response time and the probability that it would unnecessarily time-out. The last two will be used in specifying the probability that a transaction becomes involved in a deadlock.

The first behavior is the terminating behavior C_{term}, which starts at C and ends with the transaction executing on process P_1 releasing its last lock ($\&r_{11}$ or $\&r_{12}$) and terminating. This behavior describes the execution of one transaction from start (when transactions on the other process also starting and the data items are available) until termination, and the effects of the execution of the other transaction on it. The termination of a transaction running on process P_1 is represented in the concurrent behavior C by identifier "$*P_1**$"[4], where '*' matches any identifier of the other processes and data items schedulers. Therefore, using the *Terminate* function (see definition 3.4), C_{term} as depicted in Fig. 7-4 can be specified by

$$C_{term} = Terminate[C, \{(\&r_{11}, *P_1**), (\&r_{12}, *P_1**)\}] \tag{7.2}$$

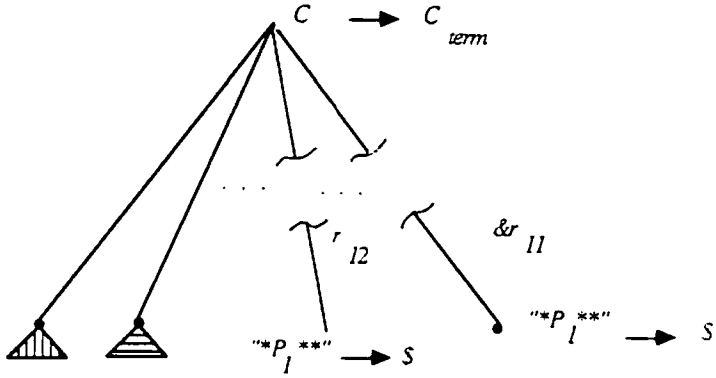


Figure 7-4: ET of the terminating behavior C_{term} of the two phase locking protocol

The second behavior, to be denoted by C_1, that we are interested in deriving represents those sub-behaviors of C_{term} in which the executions of the transaction running on P_1 is restricted such that it times-out ($\&t_1$) only if it is involved in a deadlock. Two identifiers in C_{term}, correspond to the protocol being in a deadlock state: "$F_1\,1\,W_1\,12\,F_2\,2\,W_2\,21$" and "$F_1\,2\,W_1\,21\,F_2\,1\,W_2\,12$". For the first identifier, the transaction on process P_1 has locked data item D_1 and is waiting to acquire the lock to D_2, whereas the transaction on process P_2 has locked data item D_2 locked and is waiting to acquire the lock to D_1. The same description applies to the second identifier with the exception that the data items are interchanged. Using the *Restrict* function (see definition 3.6), C_1 as depicted in Fig. 7-5 is then specified by

[4]Global identifiers of the 2PL protocol are assumed to be a concatenation of the identifiers of D_1, P_1, D_2, and P_2.

$$C_1 = Restrict[C_{term}, \{(\&t_1, F_1\,1\ W_1\,12\ F_2\,2\ W_2\,21),\ (\&t_1, F_1\,2\ W_1\,21\ F_2\,1\ W_2\,12)\}] \tag{7.3}$$

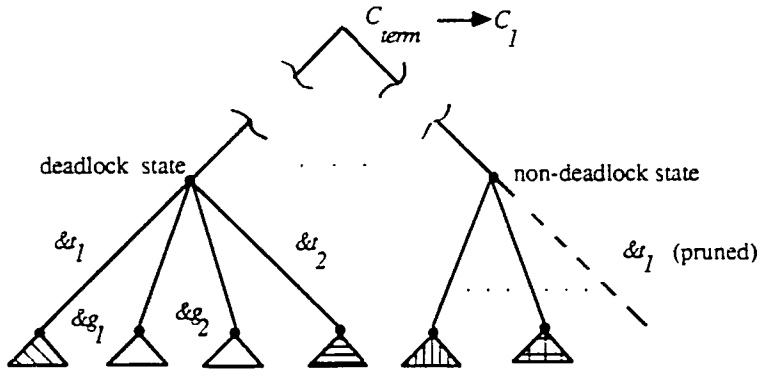


Figure 7-5: ET describing the execution of the two phase locking protocol without unnecessary time-outs of transactions running on P_1

The third behavior, to be denoted by C_{dead}, is derived from C by terminating when a deadlock occurs. Thus, the deadlock behavior of the protocol without giving a chance for time-outs to resolve these deadlocks can be examined. C_{dead} is given by

$$C_{dead} = Terminate[C, \{(\&a_{12}, F_1\,1\ W_1\,12\ F_2\,2\ W_2\,21),\ (\&a_{11}, F_1\,2\ W_1\,21\ F_2\,1\ W_2\,12),\ (\&a_{21}, F_1\,1\ W_1\,12\ F_2\,2\ W_2\,21),\ (\&a_{22}, F_1\,2\ W_1\,21\ F_2\,1\ W_2\,12)\}] \tag{7.4}$$

An illustration of the mapping from C to C_{dead} is shown in Fig. 7-6.

Now let us derive the fourth behavior, to be denoted by C_2, which includes only those events sequences in which a transaction running on P_1 would be involved in a deadlock. C_2 can be derived from C_{dead} by restricting the locking requests $\&a_{11}$ and $\&a_{12}$ such that the transaction on process P_1 does not lock the two available data items and therefore there would be no possibility of deadlock. That is, $\&a_{11}$ should not occur if the protocol is in state "$D_1 S_1\,12{**}$", and $\&a_{11}$ should not occur if the protocol is in state "$*S_1\,12\ D_2*$". Also, committing ($\&c_1$) and time-outs ($\&t_1$), should not be allowed to avoid committing or restarting before allowing deadlock to occur.

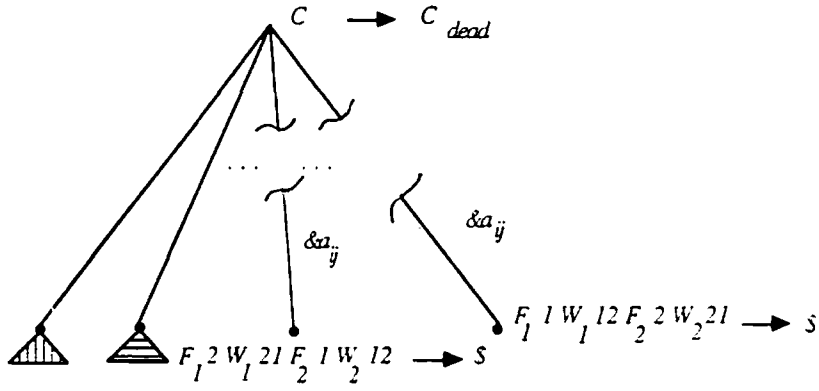


Figure 7-6: ET of the terminating behavior C_{dead} of the two phase locking protocol

If $\overline{\Gamma}$ denotes the set of all identifiers in C_{dead}, then C_2 is given by

$$C_2 = Restrict[C_{dead}, \{(\&a_{11}, I \in (\overline{\Gamma} - D_1 S_1 21 **)), (\&a_{12}, I \in (\overline{\Gamma} - * S_1 12 D_2 *)), (\&c_1, I \in \varnothing), (\&t_1, I \in \varnothing)\}] \quad (7.5)$$

7.3. Performance Specification and Analysis

The rates of the events included in C are described as follows:

$\lambda_{\&p_{ij}}$ rate of transactions on process P_i accessing data item D_j.

$\lambda_{\&t_i}$ rate of time-out of transactions on process P_i.

$\lambda_{\&g_i}$ rate of polling of transactions on process P_i for awaited lock.

$\lambda_{\&c_i}$ rate of committing of transactions on process P_i.

$\lambda_{\&a_{ij}}$ rate of transmission plus communication delay of locking request from transactions on process P_i to scheduler D_j.

$\lambda_{\&l_{ij}}$ rate of transmission plus communication delay of a lock grant from scheduler D_j to transactions on process P_i.

$\lambda_{\&r_{ij}}$ rate of transmission plus communication delay of a release request from transactions on process P_i to scheduler D_j.

Let the mean delay incurred in the transmission and communication of a lock request, lock grant, or release request be denoted by $\delta_{i,j} = 1/\lambda_{\&a_{i,j}} = 1/\lambda_{\&l_{i,j}} = 1/\lambda_{\&r_{i,j}}$ for any i, j. Each process is assumed to be running transactions belonging to the same class, and therefore has the same rates

of events for various transactions running on it. The time-out rates for the two transaction classes will be assumed equal. Thus, the probability that either of the transactions involved in a deadlock would abort is the same. Note that one transaction class could be given a priority over the other by setting its time-out rate to be less, and thus in a deadlock situation the other transaction would be more likely to abort.

In the next section, the trade-off involved in setting the time-out rate of a transaction class is studied, and an optimal setting is computed. The probability of deadlock and the mean response time of a transaction are then specified and analyzed in section 7.3.2.

7.3.1. Computation of Optimal Time-out Rate

Time-out is used in the 2PL protocol to recover from deadlock situations. For the protocol to perform efficiently, the time-out rate has to be set such that both the probability of unnecessary time-outs p_t and the transaction's mean response time t_r are minimized. A minimal probability of unnecessary time-outs ensures minimal thrashing of a transaction where it enters a cycle of abortions and restarts. A minimal mean response time ensures that a transaction times-out promptly after a deadlock occurs. However, these two goals are contradictory as shown next. The trade-off between them is studied, and a balanced timing requirement of the protocol is then specified.

Since C_1, whose ET is depicted in Fig. 7-5, represents the sub-behaviors of C_{term} in which no unnecessary time-outs occur, then p_t can be specified by

$$p_t = 1 - \mathrm{P}_{C_{term}}(C_1) \tag{7.6}$$

The transaction's mean response time t_r corresponds to the mean-time attribute of behavior C_{term} whose duration represents the time from the start of a new transaction on P_1 until that transaction commits and releases all its acquired locks. This time duration includes delays incurred by abortions and restarts due to time-outs. t_r can be then specified by

$$t_r = \mathrm{M}_{C_{term}}(C_{term}) \tag{7.7}$$

It should be noted that p_t and t_r specified above are for the case when the transaction on P_1 starts only in the initial global state ($D_1P_1D_2P_2$). In the general case, unconditional p_t and t_r can be specified by considering terminating behaviors given in eq. 7.2 from any state "*P_1**". Then if the probabilities of being in these states relative to the the cyclic behavior of the protocol C are given, the theorem of total probability can be used to compute unconditional p_t and t_r. The methodology, though, does not support the evaluation of such state probabilities; this is an issue for future research as is outlined in section 8.2.

By varying the time-out rate, t_r is plotted versus p_t in Fig. 7-7. The figure shows the trade-off between the two performance objectives of the protocol: minimizing both t_r and p_t. Similar to the approach followed in chapter 6, the *power* measure defined as $(1-p_t)/t_r$ can be used to specify a balanced timing requirement for a transaction class as follows:

$$2PL\text{–}Timing\text{–}Requirement \; : \quad Maximize \; \frac{(1-p_t)}{t_r}$$

The optimal time-out setting for the data of Fig. 7-7 is equal to 4.6 occurrences/sec (with accuracy of 1 decimal digit). This if for the transaction class of process P_1. In order to compute the optimal time-out settings for transaction classes of all processes, an iteration procedure must be employed in which each transaction class computes its optimal time-out rate and then the others follow. Convergence of such an iteration is an open research problem.

7.3.2. Specification and Analysis of Probability of Deadlock

The probability of deadlock p_d is defined as the probability that a transaction, without any restarts, would be involved in a deadlock. Other transactions running on other processes are allowed to commit or abort. Recall that C_2, specified in eq. 7.5, represents the sub-behavior of C_{dead} that lead to deadlock. p_d can then be given by

$$p_d = \mathrm{P}_{C_{dead}}(C_2) \tag{7.8}$$

This specification of p_d is conditioned on having the transaction start only when other processes and data items schedulers are in their initial state. An unconditional p_d can be specified in the same manner discussed in the previous section.

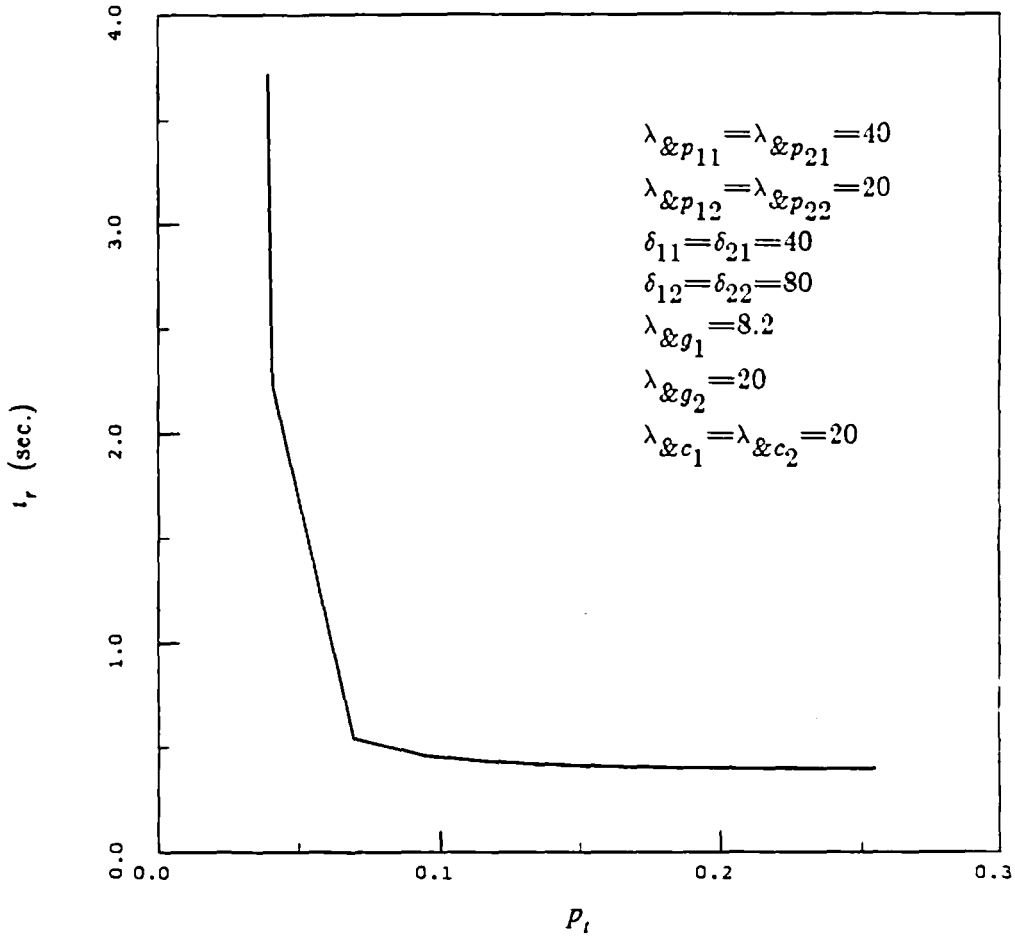


Figure 7-7: Transaction mean response time versus probability of unnecessary time-outs

p_d is plotted in Fig. 7-8 versus the rate of committing $\&c_1$, for several values of the rate of committing of the transaction class on P_2 ($\&c_2$). As the rate of committing of a transaction class increases the transactions belonging to it are shorter. Such transactions are less likely to need to lock all the data items available in the data base. The figure shows that the probability of deadlock increases sharply as the length of transactions increase, especially if long transactions are running on both processes.

In Fig. 7-9, the probability of deadlock is plotted against mean delay δ_{11} for various rates of access $\lambda_{\&p_{11}} = \lambda_{\&p_{12}}$. Increasing the access rates leads to a smaller time spent in processing actions, to be denoted by t_{pc}. The two rates are maintained equal to analyze the effect of varying

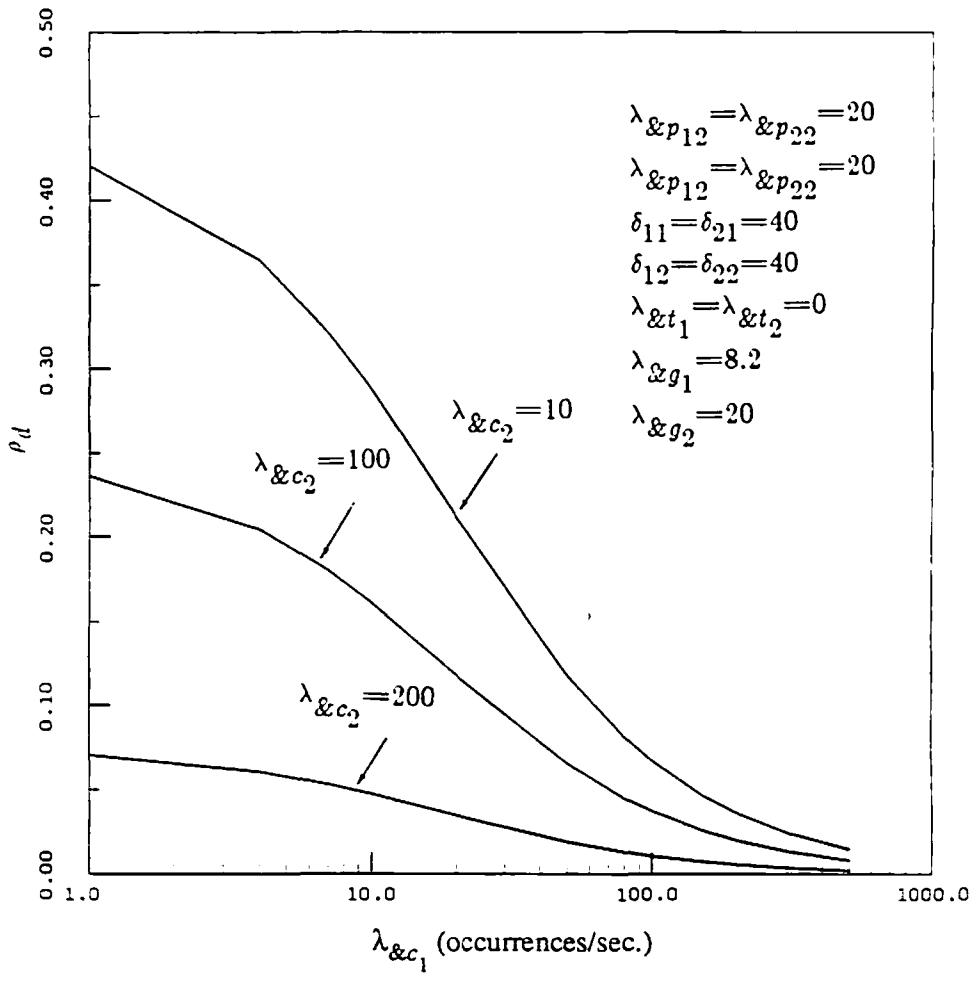


Figure 7-8: Probability of deadlock versus commit rate $\lambda_{\&c_1}$ for various $\lambda_{\&c_2}$

t_{pc} on the probability of deadlock while holding the access ratio constant. The figure shows that as δ_{11} increases, the probability of deadlock increases and saturates for very large values. A large mean delay means that a lock request sent by a transaction takes a long time to reach the data item during which the other transaction may have the chance to lock it, thus increasing the probability of deadlock. However, for a mean delay that is already larger than the mean delay between the other transaction and the data items, this increase disappears. Additionally, as the processing time t_{pc} increases, the probability of deadlock decreases because of the higher probability that the transaction decides to commit instead of needing another lock.

In Fig. 7-10, the effect of varying the access ratio on the probability of deadlock is demonstrated.

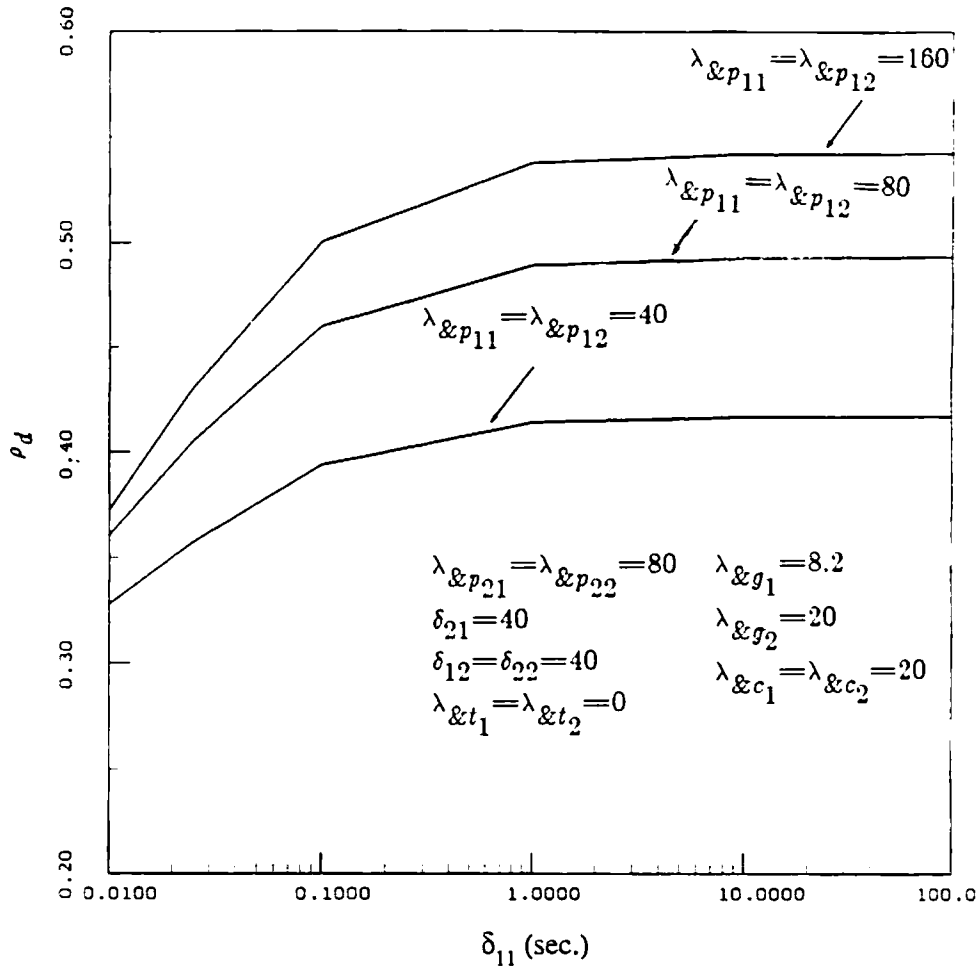


Figure 7-9: Probability of deadlock versus mean delay δ_{11} for various $\lambda_{\&p_{11}}=\lambda_{\&p_{12}}$

It is shown that as the access ratio increases, the probability of deadlock decreases. The reason is that by increasing the access ratio, the transactions running on process P_1 would more likely need to lock only one data item, and thus the probability of it being involved in a deadlock decreases.

In summary of the above three figures, the probability of a transaction becoming involved in a deadlock is small when any of the following is true:

1. The transaction itself is short.
2. The transaction's mean delay is small.
3. The transaction's processing time is small.
4. The transaction's rate of accessing data items are not comparable in value to each

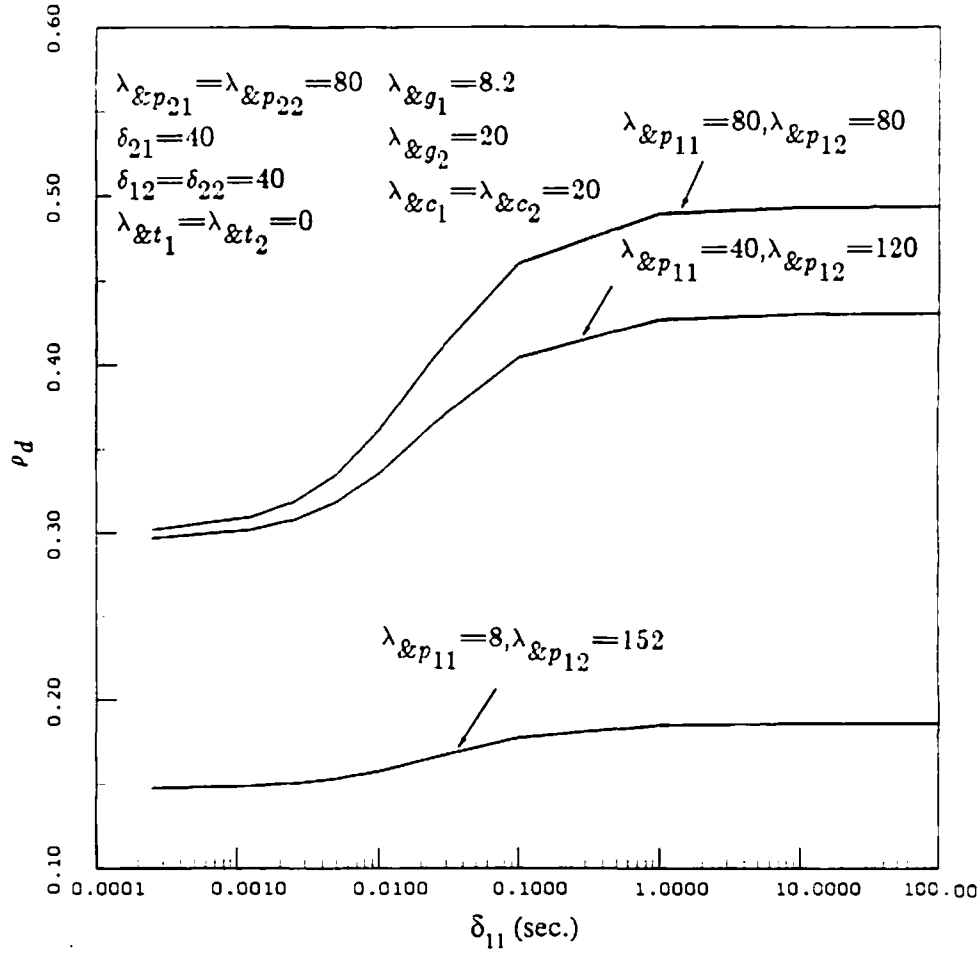


Figure 7-10: Probability of deadlock versus δ_{11} for various $\lambda_{\&p_{11}}/\lambda_{\&p_{12}}$

other.

7.3.3. Analysis of Mean Response Time

The mean response time t_r specified in eq. 7.7 is plotted in Fig. 7-11 versus the commit rate $\lambda_{\&c_1}$ for various access rates $\lambda_{\&p_{11}}=\lambda_{\&p_{12}}$. As expected, the mean response time decreases as the commit rate increases since transactions are shorter. Increasing the access rates, which means smaller processing time t_{pc}, results in a lower mean response time. However, for very large access rates, the rate of decrease of t_r is much less than for small access rates. This can be explained by noting that the probability of the transactions running on P_1 committing is given by $\lambda_{\&c_1}/(\lambda_{\&c_1}+\lambda_{\&p_{11}}+\lambda_{\&p_{12}}+\lambda_s)$, where λ_s represents a sum of the rates of some other contending

events. Therefore, for very large access rates, the rate of decrease in the mean response time is less since the effect of a change in $\lambda_{\&c_1}$ on the probability of committing is much less.

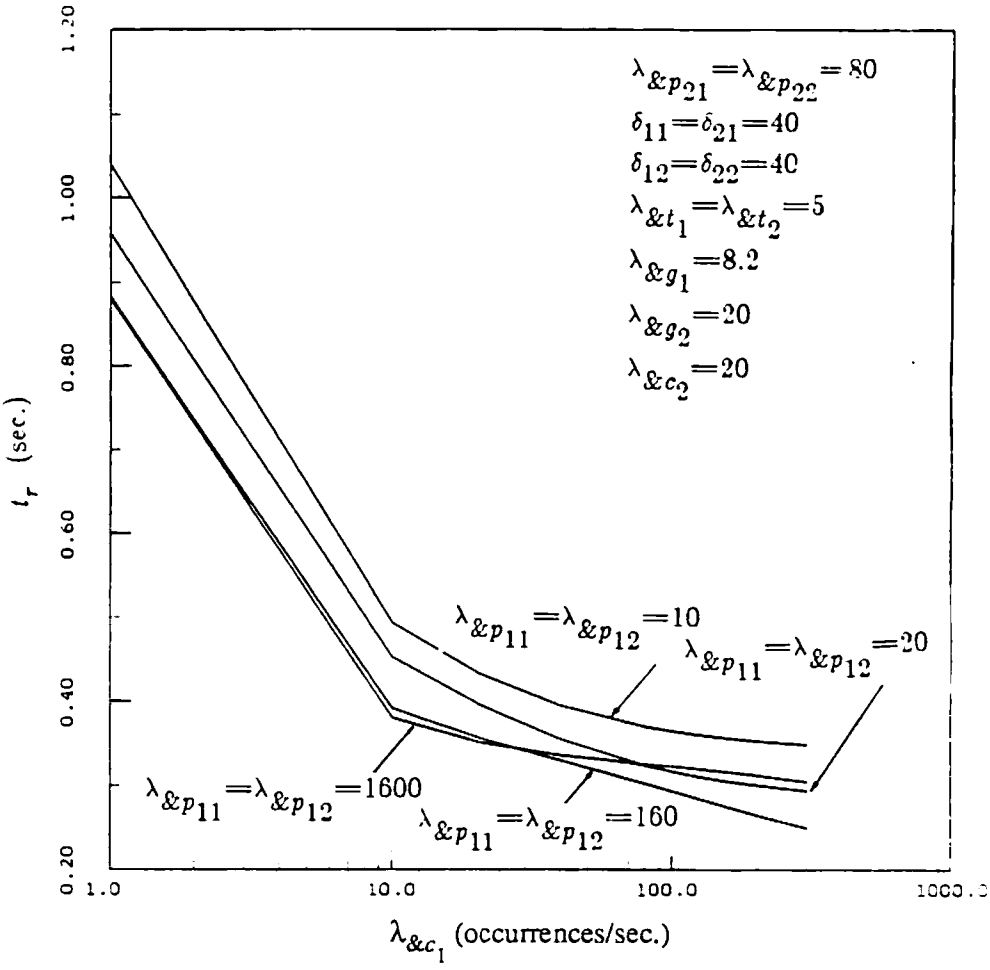


Figure 7-11: Transaction mean response time versus commit rate $\lambda_{\&c_1}$

In Fig. 7-12, the mean response time is plotted against mean delay δ_{11} for M=2,N=1; M=1,N=2: and M=2,N=2 where $\lambda_{\&p_{11}}$=1600 and $\lambda_{\&p_{12}}$=80. The mean response time of M=2,N=1 is obtained by setting rates of all rendezvous events in C_{term} between one data item and other processes to 0. Similarly, for the case of M=1,N=2. The figure shows that the mean response time for M=1,N=2 is larger than the other two cases due to less interference between transactions. The mean response time for M=2,N=2, where $\lambda_{\&p_{11}}$=1600 and $\lambda_{\&p_{12}}$=80, is very close to that of M=2,N=1, especially when δ_{11} is high, since one of the data items is much more likely to be

accessed than the other.

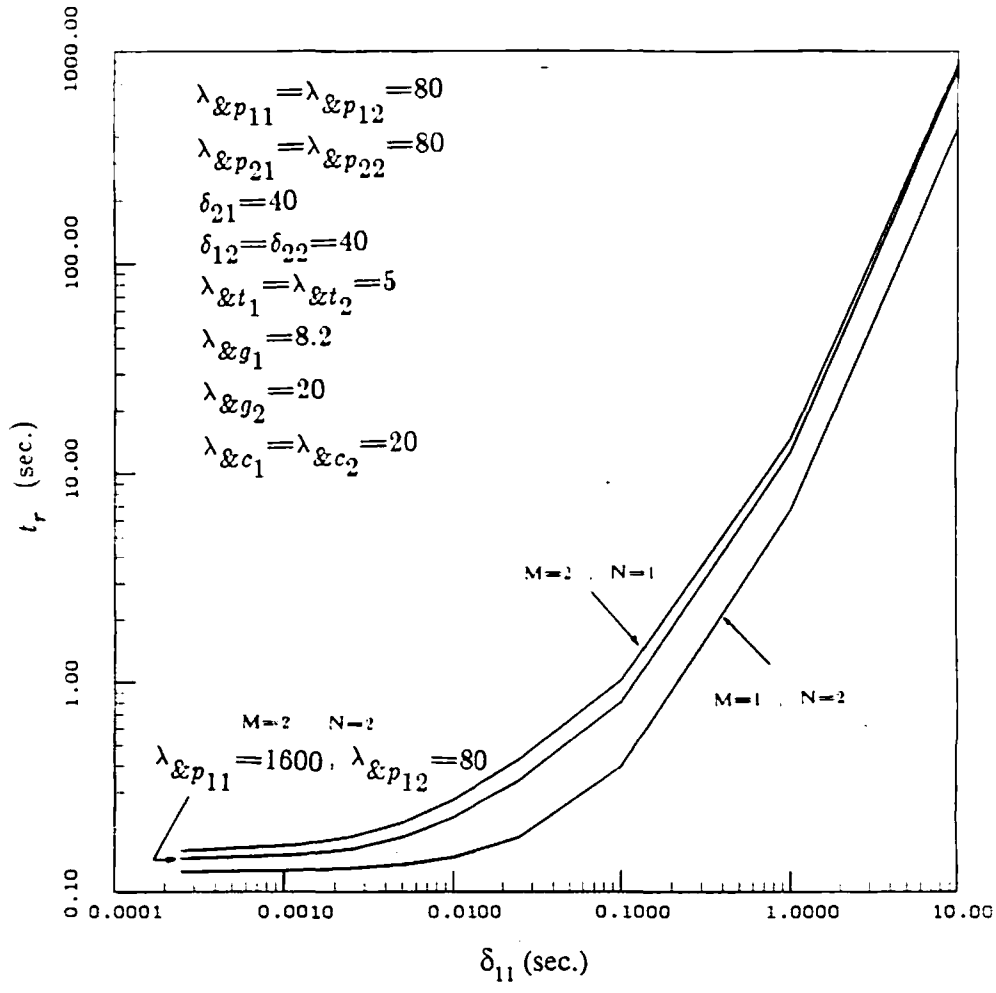


Figure 7-12: Transaction mean response time versus δ_{11}

7.4. Summary

An automated performance analysis of a two phase locking protocol has been presented. Algebraic specifications of processes running user transactions and data item schedulers involved in the protocol have been provided, and their concurrent behavior automatically has been computed. Factorial time and space complexities for the computation of such concurrent behavior have been demonstrated. Consequently, at most two processes and data items have been considered. Nevertheless, the performance results presented provide insights into the performance

of the protocol involving many processes and data items schedulers when the conflicts are mostly between two processes and data items.

A timing requirement necessary for the efficient performance of a transaction class running on one process has been specified, and an optimal time-out rate has been computed. In addition, the probability of deadlock and the mean response time of a transaction have been specified and analyzed. The effects of several performance parameters of the protocol on these performance measures have been demonstrated. The given specifications of the various performance measures of a transaction, assumed that it starts when the protocol is in its initial state. An extension of the methodology in which a state probability attribute is added to the timing attributes has been suggested to allow for the specification of unconditional performance measures of a transaction.

The performance analysis of the 2PL protocol presented in this chapter is novel in two main respects. First, it is the first specification-based performance analysis of this protocol. Second, for the first time, an optimal setting of the time-out rate of one transaction class has been computed. This suggests that time-out may be a feasible mechanism for deadlock detection. Note that the use of time-outs for deadlock detection involves local decisions to restart a transaction, i.e., there is a minimal overhead in the response time compared with other detection mechanisms which involve elaborate computations and checks of wait-for graphs.

Appendix 7.I. Algebraic Specifications of the Behaviors of the Two Phase Locking Protocol

7.I.1. Protocol Specification

A specification of the configuration of the 2PL protocol and algebraic specifications of a typical process and data item scheduler are as follows.

PROTOCOL 2PL: $P_1, P_2, \ldots, P_M, D_1, D_2, \ldots, D_N$

$scope(P_i, D_j) = \{a_{ij}, l_{ij}, r_{ij}\}$ $\quad 1 \le i \le M$ and $1 \le j \le N$

END

PROCESS P_i $\qquad 1 \le i \le M$

$$P_i = \sum_{j=1}^{N} \&p_{ij} \bullet S_i j \tag{7.9}$$

$$S_i j_1 j_2 \cdots j_n = !a_{ij_n} \bullet W_i j_1 j_2 \cdots j_n \tag{7.10}$$

$$W_i j = ?l_{ij_n} \bullet L_i j + \&g_i \bullet S_i j \tag{7.11}$$

$$W_i j_1 j_2 \cdots j_n = ?l_{ij_n} \bullet L_i j_1 j_2 \cdots j_n + \&g_i \bullet S_i j_1 j_2 \cdots j_n + \&t_i \bullet A_i j_1 j_2 \cdots j_n \tag{7.12}$$

$$L_i j_1 j_2 \cdots j_n = \sum_{k \in N - \{j_1, \ldots j_n\}} \&p_{ik} \bullet S_i j_1 j_2 \cdots j_n k + \&c_i \bullet C_i j_1 j_2 \cdots j_n \tag{7.13}$$

$$C_i j_1 j_2 \cdots j_n = !r_{ij_1} \bullet C_i j_2 \cdots j_n \tag{7.14}$$

$$C_i j = !r_{ij} \bullet P_i \tag{7.15}$$

$$A_i j_1 j_2 \cdots j_n = !r_{ij_1} \bullet A_i j_2 \cdots j_n \tag{7.16}$$

$$A_i j = !r_{ij} \bullet R_i \tag{7.17}$$

$$R_i = \sum_{j=1}^{N} \&p_{ij} \bullet S_i j \tag{7.18}$$

END

PROCESS D_j $1 \leq j \leq N$

$$D_j = \sum_{i=1}^{M} ?a_{ij} \bullet E_j i + \sum_{i=1}^{M} ?r_{ij} \bullet D_j \quad (7.19)$$

$$E_j i = \sum_{i=1}^{M} ?a_{ij} \bullet E_j i + \sum_{k \in M-\{i\}} ?r_{kj} \bullet E_j i + !l_{ij} \bullet F_i j + ?r_{ij} \bullet D_j \quad (7.20)$$

$$F_j i = \sum_{k \in M-\{i\}}^{M} ?a_{ik} \bullet F_j i + \sum_{k \in M-\{i\}} ?r_{kj} \bullet F_j i + ?r_{ij} \bullet D_j \quad (7.21)$$

END

The number of identifiers and summands in a process' specification are given in Table 7-2. Also, the number of identifiers and summands in a data item scheduler's specification are given in Table 7-3. By inspection, the total number of identifiers and summands in a process specification are both of $O(N!)$. Similarly, the total number of identifiers and summands in a data item specification are of $O(M)$ and $O(M^2)$, respectively.

Equation	Number of identifiers	Number of summands
7.9	1	N
7.10	N+N(N−1)+...N!	N+N(N−1)+...N!
7.11 and 7.12	N+N(N−1)+...N!	3(N+N(N−1)+...N!)−N
7.13	N+N(N−1)+...N!	N(1+(N−1))+(N−1)(1+(N−2)) +...N!
7.14 and 7.15	N+N(N−1)+...N!	N+N(N−1)+...N!
7.16 and 7.17	N+N(N−1)+...N!	N+N(N−1)+...N!
7.18	1	N

Table 7-2: Number of identifiers and summands in a process' specification of the two phase locking protocol

Equation	Number of identifiers	Number of summands
7.19	1	2M
7.20	M	M(2M+1)
7.21	M	M(2M−1)

Table 7-3: Number of identifiers and summands in the data item scheduler's specification of the two phase locking protocol

7.I.2. Concurrent Behavior

This is for case of M=1 and N=2.

$$D_1P_1D_2 = \&p_{11} \bullet D_1S_1\,1D_2 + \&p_{12} \bullet D_1S_1\,2D_2$$
$$D_1S_1\,1D_2 = \&a_{11} \bullet D_2E_1\,1W_1\,1$$
$$D_1S_1\,2D_2 = \&a_{12} \bullet D_1W_1\,2E_2\,1$$
$$D_2E_1\,1W_1\,1 = \&l_{11} \bullet D_2F_1\,1L_1\,1 + \&g_1 \bullet D_2E_1\,1S_1\,1$$
$$D_1W_1\,2E_2\,1 = \&l_{12} \bullet D_1L_1\,2F_2\,1 + \&g_1 \bullet D_1S_1\,2E_2\,1$$
$$D_2F_1\,1L_1\,1 = \&c_1 \bullet C_1\,1F_1\,1D_2 + \&p_{12} \bullet D_2F_1\,1S_1\,12$$
$$D_2E_1\,1S_1\,1 = \&a_{11} \bullet D_2E_1\,1W_1\,1$$
$$D_1L_1\,2F_2\,1 = \&c_1 \bullet C_1\,2D_1F_2\,1 + \&p_{11} \bullet D_1S_1\,21F_2\,1$$
$$D_1S_1\,2E_2\,1 = \&a_{12} \bullet D_1W_1\,2E_2\,1$$
$$C_1\,1F_1\,1D_2 = \&r_{11} \bullet D_1P_1D_2$$
$$D_2F_1\,1S_1\,12 = \&a_{12} \bullet E_2\,1F_1\,1W_1\,12$$
$$C_1\,2D_1F_2\,1 = \&r_{12} \bullet D_1P_1D_2$$
$$D_1S_1\,21F_2\,1 = \&a_{11} \bullet E_1\,1W_1\,21F_2\,1$$
$$E_2\,1F_1\,1W_1\,12 = \&l_{12} \bullet F_1\,1L_1\,12F_2\,1 + \&t_1 \bullet E_2\,1F_1\,1T_1\,12 + \&g_1 \bullet E_2\,1F_1\,1S_1\,21$$
$$E_1\,1W_1\,21F_2\,1 = \&l_{11} \bullet F_1\,1L_1\,21F_2\,1 + \&t_1 \bullet E_1\,1T_1\,21F_2\,1 + \&g_1 \bullet E_1\,1S_1\,21F_2\,1$$
$$F_1\,1L_1\,12F_2\,1 = \&c_1 \bullet C_1\,12F_1\,1F_2\,1$$
$$E_2\,1F_1\,1T_1\,12 = \&r_{11} \bullet D_1T_1\,2E_2\,1$$
$$E_2\,1F_1\,1S_1\,12 = \&a_{12} \bullet E_2\,1F_1\,1W_1\,12$$
$$F_1\,1L_1\,21F_2\,1 = \&c_1 \bullet C_1\,21F_1\,1F_2\,1$$
$$E_1\,1T_1\,21F_2\,1 = \&r_{12} \bullet D_1P_1D_2$$
$$E_1\,1S_1\,21F_2\,1 = \&a_{11} \bullet E_1\,1W_1\,21F_2\,1$$
$$C_1\,12F_1\,1F_2\,1 = \&r_{11} \bullet C_1\,2D_1F_2\,1$$
$$D_1T_1\,2E_2\,1 = \&r_{12} \bullet D_1P_1D_2$$
$$C_1\,21F_1\,1F_2\,1 = \&r_{12} \bullet D_1P_1D_2$$

Following algorithm 5.2 in appendix 5.IV.2 for concurrent composition, and using the figures of Tables 7-2 and 7-3, the time and space complexities of obtaining the concurrent behavior of the

2PL protocol are both of $O(N!^{M}.M^{2N})$.

Part IV

Conclusions

Chapter 8

Summary and Directions for Future Research

8.1. Summary

Contributions of this research can be summarized into three main categories:

1. *The development of a methodology that supports formal specification and automatic analysis of two aspects of protocol performance: timing requirements and performance measures.* Rules that map an algebraic specification of a protocol, and the exponential rates of its events times, to probability, mean-time, and variance-time attributes of its timing behavior have been devised. Timing requirements and performance measures of a protocol that can be formally specified in terms of attributes of its timing model are thus automatically analyzed. The analysis of timing requirements yields optimal settings of the protocol's performance parameters, whereas the analysis of its performance measures provides an assessment of the efficiency of its performance.

2. *The design and development of ANALYST: a software environment that supports automated performance analysis of protocols.* Compared to current protocol development environments, see for instance [Holz 84, Chow 85], the design of ANALYST has been shown to be novel in two main respects. First, it integrates functional and performance specification and analysis of protocols. Since protocol performance is extracted automatically from its functional specification, this integration allows a protocol designer to analyze protocol performance without requiring much expertise in the field. More specifically, a protocol designer is not required to engage in performance modeling of the protocol under analysis, but just to specify performance in terms of timing attributes of the protocol. Second, it facilitates and enhances the design process of protocols. It supports an interactive user interface that allows the protocol designer to readily debug a protocol and iterate through functional and performance specification and analysis thus facilitating experimental protocol design. It also provides the designer with a friendly and uniform user interface to the different modules that perform functional and performance analysis, i.e., the user does not have to explicitly switch from one module to the other to obtain different services.

3. *The automated derivation of performance analysis and optimum timing of a connection establishment protocol, the Alternating Bit protocol, and a two phase locking protocol.* In the case of the simple connection establishment protocol, an upper bound on the rate of terminating connections has been computed in order to limit the probability of unsynchronized operation of the connecting parties, and the probability of call collisions has been analyzed. A cycle time performance measure for the Alternating Bit protocol that captures a well-known timing error related to the time-out rate has been specified and analyzed. An optimal time-out rate of a simplified version of the protocol has been computed, and its maximum throughput

and mean delay have been analyzed producing results that agreed remarkably well with those obtained manually by other researchers. An automated performance analysis of the two phase locking protocol has demonstrated that time-outs may be an alternative to elaborate checks for detecting deadlocks. An optimal setting of the time-out rate has been computed, and the protocol's probability of deadlock and mean response time have been analyzed.

8.2. Directions for Future Research

Directions for future research can be divided into two categories: those related to the methodology and those related to the applications.

1. *Methodology*: The specification algebra and the timing model used in the methodology can be extended along several avenues. The main limitation of the specification algebra has been shown to be the state explosion suffered when computing the concurrent behavior of a real life protocol. This was evident in the case of the two phase locking protocol studied in chapter 7 which typically involves numerous processes and data items. To overcome this limitation, the algebra can be extended such that events and identifiers can be associated with parameters. For example, a transmission channel that allows any number of messages to be resident simultaneously can be specified with the number of messages as a parameter in a way similar to writing balance equations in queueing theory. Gouda [Goud 86] has recently used such *parameterized specifications* to avoid the state explosion problem when verifying protocols. It would be an interesting and promising research problem to examine how they can be also used for specification-based performance analysis. The specification algebra can be also extended to support the modeling of different kinds of addressing besides one-to-one such as one-to-many or broadcast addressing. The extensions just outlined would allow the specification algebra, and the methodology, to be applied to a wider spectrum of protocols.

   The timing model and its attributes can be extended in three respects. First, other attributes of protocol timing models can be considered. Given a cyclic expression, a steady state probability attribute can be defined and used in specifying performance measures relative to a local process when there is a need to average it over several global states. This state probability attribute can be then used to compute unconditional transaction's mean response time, optimal time-out rate, and probability of deadlock for the two phase locking protocol. Another timing attribute that may be useful is the Laplace transform of the occurrence times. It can be used to describe their probability distribution and not just statistics of these times. Second, allowing for non-exponentially distributed event occurrence times in the case of evaluating probability or mean-time attributes has to be demonstrated. Although the exponential assumption may be often acceptable, yet there are events whose times defy such an assumption and thus relaxing it would allow for more realistic analysis of protocols employing such events. Third, further study of timing requirements of protocols in terms of when they are needed, and whether they have some common formats, would be interesting.

   The software environment, ANALYST, can be also extended along four main lines. First, since the specification algebra describes communication trees, a state-of-the-art graphical interface would be highly useful and attractive. It would provide the protocol designer with a visual representation of protocols that is probably easier to understand than the algebraic representation. Second, other protocol specification

methods that can be readily translated into the specification algebra, such as finite state automata [Miln 81] and Petri nets [Boud 84, Golt 84], can be supported. The specification algebra would then serve as a canonical representation of protocols on which other tools in the environment are based. Consequently, such a protocol development environment would be no more specification-based as is typical of current environments as noted in chapter 2. Third, an automated analysis of various formats of timing requirements can be supported. Fourth, two other tools: *specification-based simulation* and *probabilistic verification* of protocols can be added to the set of tools supported by ANALYST. Specification-based simulation would be a valuable tool for validating performance results obtained from the analytic performance analysis tool. In this work, the verification of deadlock and unspecified reception errors have been considered. It may be useful for the protocol designer to know the probability of such erroneous behaviors. He can then weigh the advantages and the costs of correcting such errors. Also, the verification of other general protocol design errors such as channel overflow can be supported.

2. *Applications*: In addition to the low-level protocols considered in this research, a valuable application would be a protocol with a window mechanism since this mechanism is employed frequently by low-level protocols for flow control. Furthermore, more high-level protocols with their special functions should be considered. The methodology can be also applied to various protocols used in local area and integrated service digital networks.

References

[Ande 84] D.Anderson and L.Landweber.
Protocol Specification By Real-Time Attribute Grammars.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification.* North-Holland, June, 1984.

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

[Azem 78] P.Azema, J.Ayache, and B.Berthomieu.
Design and Verification of Communication Procedures: A Bottom-Up Approach.
In *Proceedings of the Third International Conference on Software Engineering*, pages 168-174. 1978.

[Balt 82] R.Balter, P.Berard, and P.Decitre.
Why Control of the Concurrency Level in Distributed Systems is More Fundamental Than Deadlock Management.
In *Proceedings of Symposium on Principles of Distributed Computing*, pages 183-193. ACM, 1982.

[Bart 69] K.Bartlett, R.Scantlebury, and P.Wilkinson.
A Note on Reliable Full-Duplex Transmission over Half-Duplex Lines.
CACM 12(5):260-261, May, 1969.

[Baue 82] W.Bauerfeld.
A Hybrid Model for Protocols and Services: Verification and Simulation by a Modified Depth-First Search Algorithm.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 451-464. May, 1982.

[Beiz 70] B.Beizer.
Analytical Techniques for the Statistical Evaluation of Program Running Time.
In *Fall Joint Computer Conference*, pages 519:524. ACM-IEEE, 1970.

[Berg 82] H.Berg, W.Boebert, W.Franta, and T.Moher.
Formal Methods of Program Verification and Specification.
Prentice-Hall, 1982.

[Bern 79] P.Bernestein, D.Shipman, and W.Wong.
Formal Aspects of Serializability in Data Base Concurrency Control.
IEEE Transactions on Software Engineering SE-5:203-216, May, 1979.

[Bert 82] G.Berthelot and R.Terrat.
Petri Nets Theory for the Correctness of Protocols.
IEEE Transaction on Communications COM-12:2476-2505, December, 1982.

[Bill 82] J.Billington.
Specification of the Transport Service Using Numerical Petri Nets.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 77-100. May, 1982.

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

[Boch 77b] G.Bochmann and R.Chung.
A Formalized Specification of HDLC Classes of Procedures.
In *Proceedings of the NTC*, pages 03A:2_1-03A:2_11. December, 1977.

[Boch 78] G.Bochmann.
Finite State Description of Communication Protocols.
Computer Networks 2:361-372, October, 1978.

[Boch 79] G.Bochmann and T.Joachim.
Development and Structure of an X.25 Implementation.
IEEE Transactions on Software Engineering SE-5(5):423-439, September, 1979.

[Boch 80a] G.Bochmann.
A General Transition Model for Protocols and Communication Services.
IEEE Transactions on Communications COM-28(4):643-650, April, 1980.

[Boch 80b] G.Bochmann and C.Sunshine.
Formal Methods in Communication Protocol Design.
IEEE Transactions on Communications COM-28(4):624-631, April, 1980.

[Boch 83] G.Bochmann.
Distributed Systems Design.
Springer-Verlag, 1983.

[Boch 84] G.Bochmann.
Formal Description Techniques for OSI: An Example.
In *Proceedings of INFOCOM*. IEEE, 1984.

[Bolo 84] T.Bolognesi and H.Rudin.
On the Analysis of Time-Dependent Protocols by Network Flow Algorithms.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification.* North-Holland, 1984.

[Boud 84] C. Boudol, G. roucairol, and R. Simon.
Petri Nets and Algebraic Calculi of Processes.
Technical Report, INRIA, 1984.

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

[Bran 82] D.Brand and W.Joyner.
Verification of HDLC.
IEEE Transactions on Communications COM-30(5):1136-1142, May, 1982.

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

[Brin 86] E.Brinksma.
A Tutorial on LOTOS.
In *Proceedings of the Fifth IFIP International Workshop on Protocol Specification, Testing and Verification*. North-Holland, 1986.

[Bux 80] W.Bux, K.Kummerle, and H.Truong.
Balanced HDLC Procedures: A Performance Analysis.
IEEE Transactions on Communications COM-28(11):1889-1898, November, 1980.

[Bux 82] W.Bux and K.Kummerle.
Data Link-Control Performance: Results Comparing HDLC Operational Modes.
Computer Networks 6:37-51, 1982.

[Ceri 84] S.Ceri and G.Pelagatti.
Distributed Data Bases: Principles and Systems.
McGraw-Hill Computing Science Series, 1984.

[Ches 83] A.Chesnais and E.Gelenbe.
On the Modeling of Parallel Access to Shared Data.
Communication of the ACM 26(3):196-202, March, 1983.

[Chow 85] C. Chow.
A Discipline for Verification and Modular Construction of Communication Protocols.
PhD thesis, Computer Science Dept., University of Texas at Austin, December, 1985.

[Dant 80] A.Danthine.
Protocol Representation with Finite State Models.
IEEE Transactions on Communications COM-28(4):632-643, April, 1980.

[Davi 79] D.Davies, D.Barber, W.Price, and C.Solomonides.
Computer Networks and Their Protocols.
John Wiley & Sons, New York, 1979.

[Diaz 82] M.Diaz.
Modeling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 465-510. May, 1982.

[Dick 80a] G.Dickson.
State Transition Diagrams for One Logical Channel of X.25.
In *Switching and Signalling Branch Paper 23,Australian Telecommunications Commission*. July, 1980.

[Dick 80b] G.Dickson.
Formal Specification Technique for Data Communication Protocol X.25 Using Processing State Transition Diagrams.
Australian Telecommunication Research 14(2), 1980.

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

[ECMA 80] ECMA/TC23/80/18.
3rd. Draft of Transport protocol.
Technical Report, European Computer Manufacturer Association, 1980.

[Elma 64] S.Elmaghraby.
An Algebra For the Analysis of Generalized Activity Networks.
Management Science 10(3):494-514, April, 1964.

[Eswa 76] K.Eswaren et al.
On the Notions of Consistency and Predicate Locks in a Relational Database System.
Communications of the ACM 19(11), 1976.

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

[Gele 78] E.Gelenbe.
Performance Evaluation of the HDLC Protocol.
Computer Networks 2:409-415, 1978.

[Genr 79] H.Genrich and K.Lautenbach.
The Analysis of Distributed Systems by Means of Predicate/Transition Nets.
Semantics of Concurrent Computation, Evian, G. Kahn (ed), Lecture Notes in Computer Sciences.
Springer-Verlag, 1979, pages 123-146.

[Golt 84] U. Golt and A. Mycroft.
On the Relationship of CCS and Petri Nets.
Technical Report, Lehrstuhl fur Informatik, 1984.

[Good 78] D.Good and R.Cohen.
Verifiable Communications Processing in Gypsy.
In *Compcon*, pages 28-35. 1978.

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

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

[Goud 84b] M.Gouda and Y. Yu.
Protocol Validation by Maximal State Exploration.
IEEE Transactions on Communications COM-32:94-97, January, 1984.

[Goud 86] M. Gouda and A. Sastry.
Broadcasting Finite State Machines: For Modeling LAN Protocols.
In *Proceedings of INFOCOM'86*, pages 58-66. IEEE, 1986.

[Grat 68] G.Gratzer.
Universal Algebra.
Springer-Verlag, 1968.

[Gutt 78] J.Guttag, E.Horowitz, and D.Musser.
Abstract Data Types and Software Validation.
CACM 21(12):1048-1064, December, 1978.

[Hail 80] B.Hailpern and S.Owicki.
Verifying Network Protocols Using Temporal Logic.
In *NBS Trends and Applications Symposium*, pages 18-28. May, 1980.

[Haje 78] J.Hajek.
Automatically Verified Data Transfer Protocol.
In *Proceedings of the Fourth International Computer Communications Conference*, pages 749-756. September, 1978.

[Hara 77] J.Harangozo.
An Approach to Describing a Link Level Protocol with a Formal Language.
In *Proceedings of the Fifth Data Communications Symposium*, pages 4.37-4.49. September, 1977.

[Hoar 69] C.Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12(10):576-583, October, 1969.

[Holz 82] G.Holzmann.
A Theory For Protocol Validation.
IEEE Transactions on Computers , August, 1982.

[Holz 84] G.Holzmann.
The Pandora System: An interactive System for the Design of Data Communication Protocol.
Computer Networks 8:71-79, 1984.

[ISO 83] ISO TC97/SC16 N1347 .
A FDT based on an extended state transition model.
Technical Report, ISO, July, 1983.

[ISO 85] ISO TC97/SC16 WG1 Subgroup C .
LOTS - Description of the Temporal Ordering Specification Language.
Technical Report, ISO, 1985.

[John 79] S.Johnson.
Yacc: Yet Another Compiler-Compiler.
Technical Report, AT&T Bell Labs, 1979.

[Jurg 84] W.Jurgensen and S. Vuong.
Formal Specification and Validation of ISO Transport Protocol Components, Using Petri Nets.
In *Proceedings of SIGCOMM Symposium.* ACM, 1984.

[Kell 76] R.Keller.
Formal Verification of Parallel Programs.
Communications of the ACM 19(7), July, 1976.

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

[Koba 78] H.Kobayashi.
Modeling and analysis: An Introduction to System Performance Evaluation Methodology.
Addison-Wesley Pub. Co, 1978.

[Krit 84] P.Kritzinger.
Analyzing the Time Efficiency of a Communication Protocol.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification.* North-Holland, 1984.

[Krog 78] S.Krogdahl.
Verification of a Class of Link-Level Protocols.
BIT 18:436-448, 1978.

[Kuro 82] J.Kurose.
The Specification and Verification of a Connection Establishment Protocol Using Temporal Logic.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 43-62. May, 1982.

[Lam 82] S.Lam and A.Shankar.
An Illustration of Protocol Projections.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification.* 1982.

[Lamp 77] L.Lamport.
Proving The Correctness of Multiprocess Programs.
IEEE Transactions on Software Engineering SE-3:125-143. 1977.

[Lamp 80] L.Lamport.
'Sometime' is Sometimes 'Not Never'.
In *Proceedings of the ACM POPL Conference*, pages 174-185. 1980.

[Lamp 83] L.Lamport.
Specifying Concurrent Program Modules.
ACM Transactions on Programming Languages and Systems 5(2):190-222, April, 1983.

[Lesk 79] M.Lesk and E.Schmidt.
Lex - A Lexical Analyzer Generator.
Technical Report, AT&T Bell Labs, 1979.

[Lond 80] R.London and L.Robinson.
The Role of Verification Tools and Techniques.
Software Development Tools, W.Riddle and R.Fairley ed.
Springer-Verlag, 1980, pages 206-212.

[Mann 81] Z.Manna and A.Pneuli.
Verification of Concurrent Programs: The Temporal Framework.
Technical Report STAN-CS-81-836, Stanford University, June, 1981.

[Mena 79] D.Menasce and R.Muntz.
Locking and Deadlock Detection in Distributed Data Bases.
IEEE Transactions on Software Engineering SE-5(3):195-202, May, 1979.

[Merl 76] P.Merlin and D.Farber.
Recoverability of Communication Protocols - Implications of a Theoretical Study.
IEEE Transactions on Communications COM-24:1036-1043, September, 1976.

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

[Miln 81] R.Milner.
A Complete Inference System for a Class of Regular Behaviors.
Technical Report, University of Edinburgh, September, 1981.

[Mitr 84] D.Metra and P.Weinberger.
Probabilistic Models of Database Locking: Solutions, Computational Algorithms, and Asymptotics.
Journal of the ACM 31(4):854-878, October, 1984.

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

[Morr 84] R.Morris and W.Wong.
Performance Analysis of Concurrency Control Algorithms With Nonexclusive Access.
In *Proceedings of Performance'84*, pages 87-99. Elsevier Science Publishers B.V. (North Holland), 1984.

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

[Noun 85] N. Nounou and Y.Yemini.
Development Tools for Communication Protocols: A Survey.
Technical Report, Computer Science Department, Columbia university, February, 1985.

[Nutt 72] G.Nutt.
Evaluation Nets for Computer System Performance analysis.
AFIPS Conference Proceedings 41,Part 1:279-286, 1972.

[Pete 77] J.Peterson.
Petri Nets.
ACM Computing Surveys 9(3):224-252, September, 1977.

[Pnue 77] A.Pnueli.
The Temporal Logic of Programs.
In *The Eighteen Annual Symposium on Foundations of Computer Science*, pages 46-57. October, 1977.

[Post 76] J.Postel and D.Farber.
Graphic Modeling of Computer Communications Protocols.
In *Proceedings of the Fifth Texas Conference on Computing Systems*, pages 66-67. 1976.

[Post 79] J. Postel, ed.
Transmission Control Protocol (TCP).
Technical Report, ISI, Marina Del Rey, Ca., 1979.

[Poti 80] D.Potier and Ph.Leblanc.
Analysis of Locking Policies in Database Management Systems.
Communications of the ACM 23(23):584-593, October, 1980.

[Prad 79] B.Chezaviel-Pradin.
Un Outil Grphiquw Interactif pour la Validation des Systemes a Evolution Parallele Decrits par Reseaux de Petri.
PhD thesis, Universite Paul Sabatier, December, 1979.

[Rals 78] A. Ralston and P. Rabinowitz.
A First Course in Numerical Analysis.
McGraw-Hill Book Company, 1978.

[Razo 84] R.Razouk.
The Derivation of Performance Expressions for Communication Protocols from Timed Petri Net Models.
In *Proceedings of the SIGCOMM Symposium*, pages 210-217. ACM, June, 1984.

[Regh 82] H.Reghbati.
Performance Analysis of Message-Based Systems.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 321-324. May, 1982.

[Reis 82] M.Reiser.
Performance Evaluation of Data Communication Systems.
In *Proceedings of the IEEE*, pages 171-196. February, 1982.

[Ridd 80] W.Riddle and R.Fairley.
Introduction.
Software Development Tools, W.Riddle and R.Fairley ed.
Springer-Verlag, 1980, pages 1-8.

[Rock 81] A.Rockstrom and R.Sarraco.
SDL CCITT Specification and Description Language.
In *Proceedings of the NTC*, pages G6.3.1-G6.3.5. 1981.

[Rubi 82] J.Rubin and C.West.
An Improved Protocol Validation Technique.
Computer Networks 6:65-73, 1982.

[Rudi 84] H.Rudin.
An Improved Algorithm for Estimating Protocol Performance.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification*. North-Holland, 1984.

[Rudi 85] H.Rudin.
An Informal View of Formal Protocol Specification.
IEEE Communications Magazine 23(2):46-52, March, 1985.

[Sabn 82a] K.Sabnani and M.Schwartz.
Verification of a Multidestination Protocol Using Temporal Logic.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 21-42. may, 1982.

[Sabn 82b] K.Sabnani.
Multidestination Protocols for Satellite Broadcast Channels.
PhD thesis, Columbia University, 1982.

[Salo 66] A.Salomaa.
Two Complete Axiom Systems for the Algebra of Regular Events.
jacm 13(1):158:169, 1966.

[Sand 82] M. Sanderson.
Proof Techniques in CCS.
PhD thesis, University of Edinburgh, November, 1982.

[Schi 80] S.Schindler.
Algebraic and Model Specification Techniques.
In *Proceedings of the Hawaii International Conference on System Sciences.* 1980.

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

[Schw 80] M. Schwartz.
Routing and Flow Control in Data Networks.
Technical Report, IBM Watson Research Center, 1980.

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

[Schw 81b] R.Schwartz and P.Melliar-Smith.
Temporal Logic Specification of Distributed Systems.
In *Proceedings of the IEEE Distributed Computer Systems Conference*, pages 446-454. 1981.

[Schw 82] R.Schwartz and P.Melliar-Smith.
From State Machines to Temporal Logic: Specification Methods for Protocol Standards.
IEEE Transaction on Communications COM 12:2476-2505, December, 1982.

[Schw 83] R.Schwartz, P.Melliar-Smith and F.Vogt.
Interval Logic: A Higher-Level Temporal Logic for Protocol Specification.
In *Proceedings of the Third IFIP International Workshop on Protocol Specification, Testing and Verification.* North-Holland, 1983.

[Shan 84] A.Shankar and S.Lam.
Specification and Verification of Time-Dependent Communication Protocols.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification.* North-Holland, 1984.

[Shum 81] A.Shum and P.Spirakis.
Performance Analysis of Concurrency Control Methods In Database Systems.
In *Proceedings of Performance'81*, pages 2-19. Elsevier Science Publishers B.V. (North Holland), 1981.

[Sloa 83] L.Sloan.
Mechanisms That Enforce Bounds on Packet Lifetimes.
ACM Transactions on Computer Systems 1(4):311-330, November, 1983.

[Snyd 75] D.Snyder.
Random Point Processes.
Wiley-Interscience, 1975.

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

[Suns 75] C.Sunshine.
Interprocess Communication Protocols for Computer Networks.
PhD thesis, Stanford University, Digital Systems Laboratory TR 105, December, 1975.

[Suns 81] C.Sunshine.
Formal Modeling of Communication Protocols.
In *Proceedings of the First International INWG/NPL Workshop : Protocol Testing - Towards Proof?*, pages 29-58. 1981.

[Suns 82a] C.Sunshine, D.Thompson, R.Erickson, S.Gerhart, and D.Shwabe.
Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models.
IEEE Transactions on Software Engineering SE-8(5):460-489, September, 1982.

[Suns 82b] C.Sunshine.
Experience with Automated Verification Systems.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification.* 1982.

[Suns 83] C.Sunshine.
Experience with Automated Verification Systems.
In *Proceedings of the Third IFIP International Workshop on Protocol Specification, Testing and Verification.* 1983.

[Symo 80] F.Symons.
Representation, Analysis & Verification of Communication Protocols.
Technical Report 7380, Australian Telecommunication Research, 1980.

[Teng 78] A.Teng and M.Liu.
A Formal Model for Automatic Implementation and Logical Validation of Network Communication Protocol.
In *NBS Computer Networking Symposium*, pages 114-123. 1978.

[Tows 79] D.Towsley and J.Wolf.
On the Statistical Analysis of Queue Lengths and Waiting Times for Statistical Multiplexors with ARQ Retransmission Schemes.
IEEE Transactions on Communications COM-27(4):693-702, April, 1979.

[Vogt 82] F.Vogt.
Event-Based Temporal Logic Specifications of Services and Protocols.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 63-74. May, 1982.

[Wass 81] A.Wasserman.
Tutorial: Software Development Environments.
Software Development Tools, W.Riddle and R.Fairley ed.
IEEE Computer Society, 1981, pages 1-2.

[West 78a] C.West.
An Automated Technique of Communications Protocol Validation.
IEEE Transactions on Communications COM-26(8):1271- 1275, August, 1978.

[West 78b] C.West.
General Technique for Communications Protocol Validation.
IBM Journal of Research and Development 22(4):393-404. July, 1978.

[West 78c] C.West and P.Zafiropluo.
Automated Validation of a Communications Protocol: the CCITT X.21 Recommendation.
IBMJRD 22(1):60-71, January, 1978.

[West 82] C.West.
Applications and Limitations of Automated Protocol Validation.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification.* 1982.

[Wolp 82] P.Wolper.
Specification and Synthesis of Communicating Processes using an Extended Temporal Logic.
In *Proceedings of the Ninth Symposium on Principles of Programming Languages.* January, 1982.

[X.21 76] CCITT.
Recommendation X.21 (Revised).
Technical Report, Geneva, Switzerland, March, 1976.

[X.25 80] CCITT.
Recommendation X.25 Packet Switch Data Transmission Services.
Technical Report, Geneva, Switzerland, 1980.

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

[Yu 79] L.Yu and J.Majthia.
An Analysis of One Direction of Window Mechanism.
IEEE Transactions on Communications COM-27(5):778-788, May, 1979.

[Zimm 80] H.Zimmermann.
The ISO Model of Architecture for Open System Interconnection.
IEEE Transactions on Communications COM-28(4), April, 1980.

Index