Development Tools For Communication Protocols

Nihal Nounou and Yechiam Yemini
Computer Science Department
Columbia University
New York, New York 10027

February 1985

CUCS-160-85

ABSTRACT

The past decade has witnessed a surge in research efforts aimed at developing tools to aid the designer of communication protocols. Most of these efforts have been directed towards designing individual tools. Recently, however, there has been a growing interest in building development environments that support an integrated set of such tools. This paper presents a survey of commonly used protocol development tools. Two categories of protocol development tools are examined: construction tools to successively refine communication protocols from specifications to working systems and validation tools to assess whether the refinements meet functional and performance protocol objectives. Construction tools surveyed include tools for specification, synthesis, and implementation. Validation tools surveyed include tools for formal verification, performance analysis and testing. A simple send-and-wait protocol is used as an example throughout the paper.

Table of Contents

1 Introduction 1
2 Specification Tools 3
2.1 Requirements of Specification Tools for Protocols 4
2.2 Survey of Specification Tools 5
2.2.1 Finite State Machines 5
2.2.2 State Machine Models 7
2.2.3 Formal Grammars and Sequence Expressions 8
2.2.4 Petri Net-Based Models 10
2.2.5 Algebraic Specifications 12
2.2.6 Temporal Logic Specification 13
2.2.7 Procedural Languages 15
2.3 A Taxonomy for Specification Tools 15
3 Protocol Synthesis Tools 16
4 Implementation Tools 19
5 Verification Tools 21
5.1 State Exploration 22
5.2 Assertion Proof 24
6 Performance Analysis Tools 24
6.1 Specification and Verification of Protocol Timing Requirements 25
6.2 Evaluation of Performance Measures 26
6.2.1 Analytic Tools 27
6.2.2 Simulation 29
7 Testing Tools 30
7.1 Logical Architectures for Testing 31
7.2 Test Sequences Selection 31
8 Conclusions 33

List of Figures

Figure 1:	Illustration of protocol layers	1
Figure 2:	A local view of a protocol layer	2
Figure 3:	A protocol specification for the send-and-wait protocol using FSM's (a) Sender (b) Receiver (c) Medium	6
Figure 4:	A service specification for the send-and-wait protocol using FSM's	6
Figure 5:	A partial state machine specification of the sender process of a modified send-and-wait protocol with binary sequence numbers	8
Figure 6:	A formal grammar specification for the sender process of the send-and-wait protocol	9
Figure 7:	A send-and-wait protocol specification using petri nets	11
Figure 8:	A state-based temporal logic specification for the sender process of the send-and-wait protocol	14
Figure 9:	An illustration of the proposed taxonomy of specification tools	17
Figure 10:	A reachability graph for the send-and-wait protocol	23
Figure 11:	A modified reachability graph for the send-and-wait protocol	28
Figure 12:	Transfer time vs. arrival rate of the send-and-wait protocol	29
Figure 13:	Logical architecture for testing	32
Figure 14:	Physical architecture including the portable unit	32

1 Introduction

In a *computer network*, distributed processes can communicate and share information through message-exchange. Such communication involves a rather complex set of problems since the distributed processes are allowed to concurrently access shared resources and to proceed asynchronously. Moreover, they may be executed by heterogeneous processors, and their communication channels are often unreliable -- they might lose, duplicate, reorder, and/or corrupt messages. *Communication protocols* are thus required to regulate the communication between distributed processes in a computer network. They constitute a set of rules and a set of message formats. The reader is referred to [Tane 81] for a tutorial on protocols.

The International Standards Organization (ISO) has proposed a reference model of protocol architecture for Open Systems Interconnection (OSI) (described in [Zimm 80]). The model has seven hierarchical layers illustrated in Fig. 1; protocols at layers 1 through 4 are referred to as *low-level protocols* and those at layers 5 through 7 as *high-level protocols*. 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*. A protocol designer is also concerned with the internal structure and operation of the layer's black box which is illustrated in Fig. 2.

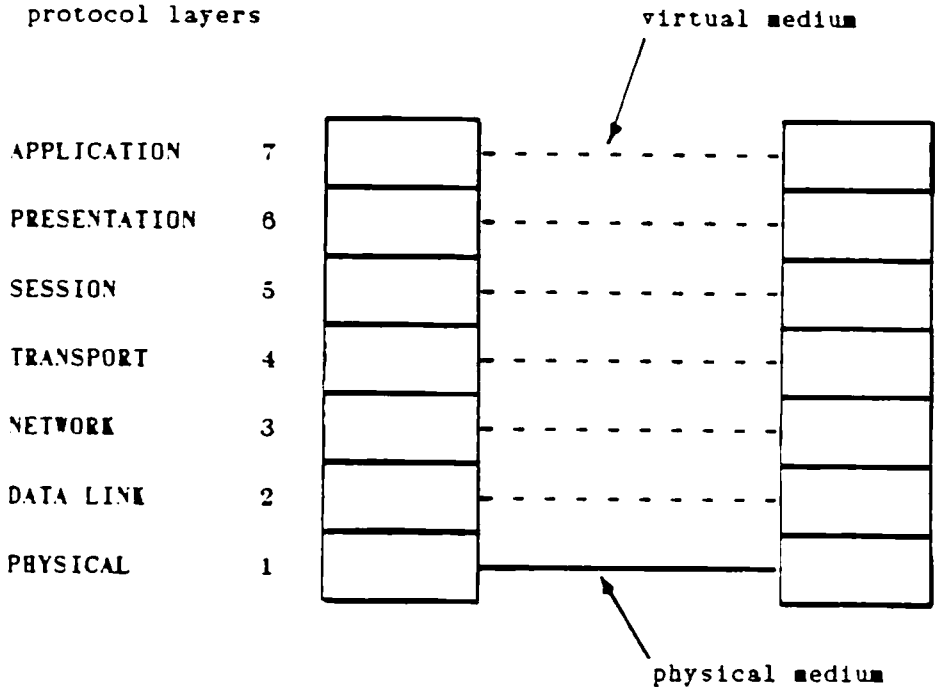


Figure 1: Illustration of protocol layers

In this figure each protocol process (also referred to in the literature as component, module, entity, and party) resides typically at a different site and communicates with other *peer* (i.e., neighboring) 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

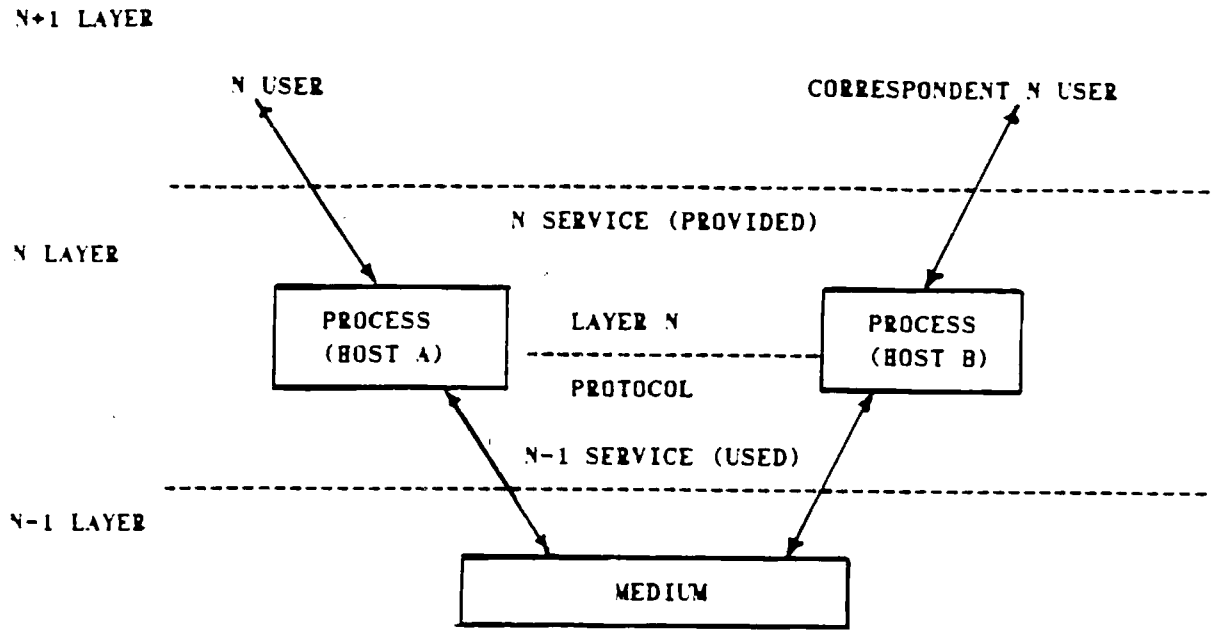


Figure 2: A local view of a protocol layer

to as the *protocol specification*. Finally, the protocol specification refined into actual code describing 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 a phased approach to their development. In such a framework of phased development (for details see in particular [Boeh 76, Lehm 80, Oste 80]) there are three main phases: *service statement*, *protocol design*, and *implementation*. *Development tools* are required to support the evolution of protocols from specifications into working systems. This survey covers two kinds of protocol development tools: *construction tools* for developing and refining protocol specifications and *validation tools* to assess how a specification meets its functional (e.g., deadlock freedom) and performance (e.g., maximizing throughput) design objectives. 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, communicating through unreliable transmission mediums, and whose behavior is often time-dependent. These features affect protocol development tools in various ways. First, the communication nature of protocols becomes the prime concern underlying the tools. The basic objective of protocol validation tools, for example, is to assure robustness and efficiency of the communication between the protocol processes. Second, timing requirements as well as functional requirements of protocol behavior should be considered in the various tools. This integration provides a more realistic and relatively simpler description of protocol behavior than when only functional requirements are considered. Third, tools used for general software systems might have varied importance for protocols. One example of a tool that is rarely used for general software, but that is becoming highly desirable for protocols is *certification* of independently developed protocol implementations to ensure that they comply with a standard, and thus will be able to internetwork.

Recently there has been a growing interest in building protocol development environments that integrate the various development tools required throughout the entire protocol development. An ideal protocol development environments should encompass a comprehensive set of tools and a methodology for their use. Therefore, a prime objective of this survey is to examine the **complete** set of commonly used protocol development tools. Other surveys concerned with only subsets of the tools examined in this paper can be found in [Boch 80a, Dant 80, Hail 81, Suns 81, Diaz 82, Schw 82, Suns 83].

The paper is organized as follows: in sections 2 through 4 we survey the *construction tools* including specification, synthesis, and implementation, respectively. Sections 5 through 7 are surveys of *validation tools* including formal verification, performance analysis, and testing respectively. Finally, in section 8 we present some conclusions and remarks on possible directions for future research.

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. High-level languages are used for describing implementation specifications. These will not be discussed here; throughout the rest of the paper 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); 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 might be required if the specification tool used in the protocol environment is changed.

Throughout this section and subsequent sections, a simple send-and-wait protocol will 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 medium. There are three distributed processes involved in the protocol: a sender S, a receiver R, and a transmission medium 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 medium 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 buffered. To recover from cases of message and acknowledgment 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 medium, after which it delivers it to a destination D and then sends an acknowledgment a to the sender through the medium. For the sake of simplicity, it is assumed that the medium 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 medium 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 medium'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.

In the following section, requirements of specification tools for protocols are outlined, the various specification tools are surveyed in section 2.2, and a taxonomy of the these tools is proposed in section 2.3.

2.1 Requirements of Specification Tools for Protocols

The key requirements of a specification tool to adequately express protocols include the following.

1. Supporting abstract descriptions such that implementation-dependent parts can be left unspecified.
2. Supporting modeling of concurrency.
3. Supporting 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. Supporting the description of the two categories of functions involved in protocols: *control functions* involving connection initialization and inter-process synchronization, and *data transfer* functions involving processing of messages texts and related issues such as message sequence numbering.
5. Supporting modular descriptions to facilitate readability and ease of use of specifications.

Since specification tools often are the basis of other development tools, they must also include the following features to facilitate their application:

1. Executability of the specification to facilitate its direct simulation, and the automation of the implementation process.
2. Providing constructs for expressing functional properties of protocols, thus facilitating their automated formal verification.
3. Supporting the specification of the timing requirements of protocols. Since the behavior of protocol is often time-dependent, their correct functioning might depend on certain timing requirements. For example, the specification of the value of the time-out period in a protocol with such a feature greatly affects its function. If the the time-out period is too short, the network would be flooded with duplicate messages and the protocol would enter an infinite cycle of time-outs.
4. Providing constructs for expressing performance properties of protocols (including properties of transmission mediums such as bit error probability and desired performance such as bounds on throughput and delay measures), thus facilitating automated performance analysis.
5. Supporting the clear definition of the interfaces between the protocol layer concerned and the layers above and below to allow for separate testing of the implementation of each protocol layer.

The extent to which a specification tool exhibits the first set of requirements is examined in section 2.2. In section 2.3 we examine the extent with which the various classes of specification tools based on the proposed taxonomy in that section support the second set of requirements.

2.2 Survey of Specification Tools

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×state→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 an command according to the input and its current state representing the history of past commands. FSM's were used in early work on specification of protocols [Bart 69, Suns 75].

Consider using FSM's 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 interleaving 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 buffered 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 FSM's 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 FSM's in which case only channels with a low bound on the number of messages can be feasible assumed. Even then their FSM specifications are considerably more complex than in the second approach.

Following the latter approach, specifications of the three communicating processes in the send-and-wait protocol are shown in Fig. 3. 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 overbar denoting send events or events with an underbar 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. A service specification for the same protocol is shown in Fig. 4 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.

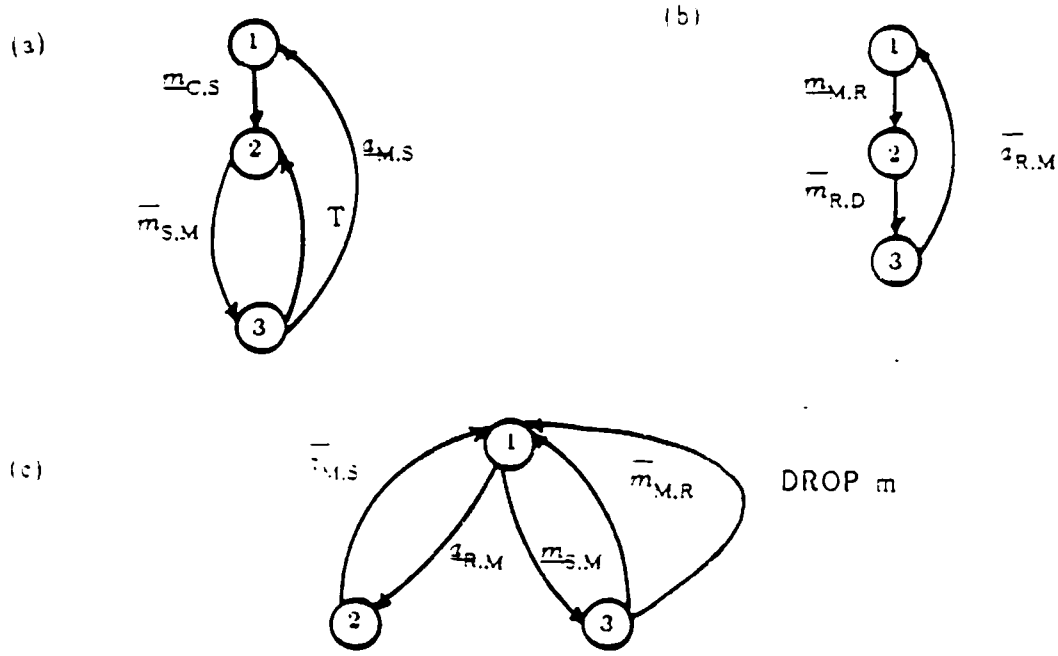


Figure 3: A protocol specification for the send-and-wait protocol using FSM's (a) Sender (b) Receiver (c) Medium

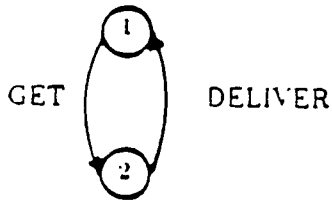


Figure 4: A service specification for the send-and-wait protocol using FSM's

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 in the following section, alleviate most of these limitations.

2.2.2 State Machine Models

State machines are FSM's 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 83a]) of the sender process of this extended protocol, is given in Fig. 5. 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 could have a predicate stating that it should be not 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 Gesei [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 subprocesses which are then specified as state machines.

A state machine model proposed by the ISO TC97/SC16/WG1 subgroup B on formal description techniques (FDT) [Iso 83a, Boch 84] employs Pascal-like constructs in extending FSM's. 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 were reported by Schwabe [Schw 81a], Divito [Divi 82] and Shankar and Lam [Shan 84]. These efforts are distinguished 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 could be especially desirable in the specification of high level protocols. Divito uses buffer 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; a requirement that is needed for the correct functioning of many network layer protocols (those at layer 3 in Fig. 1).

```
module Sender

var
  state : (state1, state2, state3);
       (* same states labels as in Fig. 3(a) *)


  corrupted : boolean;

  next-message-to-send : 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-message-to-send)
     from state3
        begin
          next-message-to-send := (next-message-to-send + 1) mod 2;
        end;
     to state1
     .
     .
     .

end module Sender
```

Figure 5: A partial state machine specification of the sender process of a modified send-and-wait protocol with binary sequence numbers

Combining the two formalisms of FSM's 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.

2.2.3 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, 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. 6. It is a direct translation of its FSM in Fig. 3(a) with terminal symbols (represented by upper-case letters) denoting input commands and non-terminal symbols (represented by lower-case letters) denoting states.

$G = \{V, T, S, P\}$,

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 6: A formal grammar specification for the sender process of the send-and-wait protocol

Since regular grammars and FSM's are equivalent, they share the same limitations. The state explosion problem is manifested here as an explosion in the number of production rules. To overcome this problem, Harangozo [Hara 77] 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] used a context-free grammar, which provides more expressive power than regular grammars. They also uses a shuffle operation to integrate grammars defining processes in the same protocol layer by computing all possible interleavings of their behavior, and a substitution operation 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. 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 is

described in one expression where no nonterminal symbols are used. The sender process of the send-and-wait protocol can be specified as a sequence expression given by

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

where operations "*" , "–" , and "+" denote the Kleene star, sequential composition, and nondeterministic choice operations, respectiveiy.

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

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. PN's are in many ways similar to FSM's, 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 FSM's, PN's 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 represents 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 PN's to model protocols can be found in [Post 78, Azem 78, Dant 80]. A PN specification of the send-and-wait protocol is given in Fig. 7. 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 loss of a message or an acknowledgment and the assumption that acknowledgments are not lost.

Similar to FSM's, PN's cannot adequately model complex data transfer of protocol without suffering from explosion of the net size, or timing considerations such as time-out. Two major extensions to PN's that add to their power in modeling protocols lead to *hybrid PN's* and *timed PN's*. The price for these extensions is more complex validation.

Hybrid Petri Nets

Hybrid petri nets are extended PN's in which tokens can have identities and transitions can have predicates and actions associated to them. Adding predicates to PN's produces *predicate/transition* nets formalized by Genrich and Lautenbach [Genr 79], where transitions

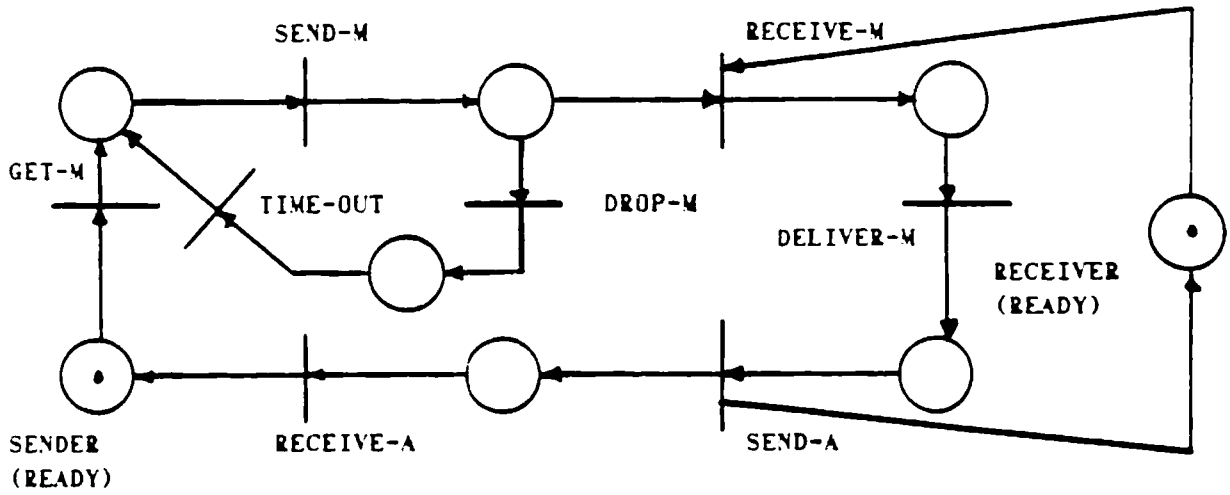


Figure 7: A send-and-wait protocol specification using petri nets

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] 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 receiving of a message m with certain parameters is described in a predicate, and the sending 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. 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 were used in modeling protocols [Boch 77a, Azem 78, Baue 82]. NPN's 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 used NPN to model a Transport service [Bill 82].

Timed Petri Nets

A Timed PN is a PN extended to support some description of time. Timed PN's that have been used for protocols include *time PN's* (TPN's) introduced by Merlin [Merl 76] and *stochastic PN's* (SPN's) 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] used a combination of TPN's and Nutt's evaluation nets [Nutt 72] (a kind of abbreviated PN) to model the Transport protocol of the Cyclade network.

SPN's are PN's 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 model is extracted from the PN graph describing the global protocol behavior; in a markov model all transitions should be either exponentially or geometrically distributed. The random representation of time involved in protocol events is used in SPN's to allow for quantitative performance analysis.

2.2.5 Algebraic Specifications

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 protocol behavior expressions 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.

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 d represent a receive port for message incoming from the source. The sender specification S is described recursively as follows.

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

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 proposed by the ISO TC97/SC16/WG1 subgroup C [Iso 83b, Brin 84]. Holzmann [Holz 82] also introduced a CCS-variant algebraic model with a division operation used to represent send events and message buffers used to allow for asynchronous inter-process communication. Another CCS-variant model introduced by Nounou [Noun 84] associates probability and time attributes with protocol behavior expressions to allow for the specification of protocol timing behaviors as well as their functional behaviors. This allows protocol timing requirements to be specified as will be described in section 6.1.

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 specification 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]).

2.2.6 Temporal Logic Specification

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 could 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* "$\Box$" and *eventually* "$\Diamond$". 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 the state-based temporal logic method 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. 8 (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. The

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

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

A3. S_o=p and $\Diamond$ S_o=q=p implies $\Diamond$(S_o=q=p and at SEND-M) **Until** (S_i=q=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. 3(a).

Figure 8: A state-based temporal logic specification for the sender process of the send-and-wait protocol

axioms in Fig. 8 have the following interpretations. Axiom A1 states that a message value remains in S_o until both its successful acknowledgment has been received and a new message has been 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 considers 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 use 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 number of messages sent equals number of messages received could 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 can not 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 were used in specifying and verifying the three way handshake connection protocol. Shankar and Lam [Shan 84] use 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] uses a history variable to represent the sequence of past events and thus establish the required context. In another event-based approach, Wolper [Wolp 82] 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 could not be expressed in temporal logic such as stating a proposition that is to hold in every other state in a sequence.

2.2.7 Procedural Languages

In a procedural language, the unit of specification 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 method. Examples of such works can be found in [Sten 78, 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 *buffer histories* to record all send and receive operations executed on a system's buffer. One limitation of specifications employing buffer histories, is the difficulty in modeling unreliable communication mediums [Divi 82] since processes communicate through message buffers 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 buffer.

While procedural languages are a natural choice for coding implementation specifications, there has been much controversy regarding their use for specification in early design phases. The shortcoming of using procedural languages for specification lies in their detailed descriptions of a systems' operation. This makes it rather difficult to specify the abstract requirements of protocols 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, could argue that such languages, with their rich expressive power, support the specification of both control and data transfer functions of protocols.

2.3 A Taxonomy for Specification Tools

As a summary of this section, we propose a taxonomy of specification tools that will be helpful in judging the extent by which a specification tool meets the second set of requirements given in section 2.1. The first three are requirements of specification tools to be executable, to support the specification of desired properties of protocols, and to support the specification of performance parameters of protocol behavior. The fourth requirement of

providing clear descriptions of interfaces between protocol layers can be met by a service specification that describes both the service used and the service provided by the protocol layer concerned.

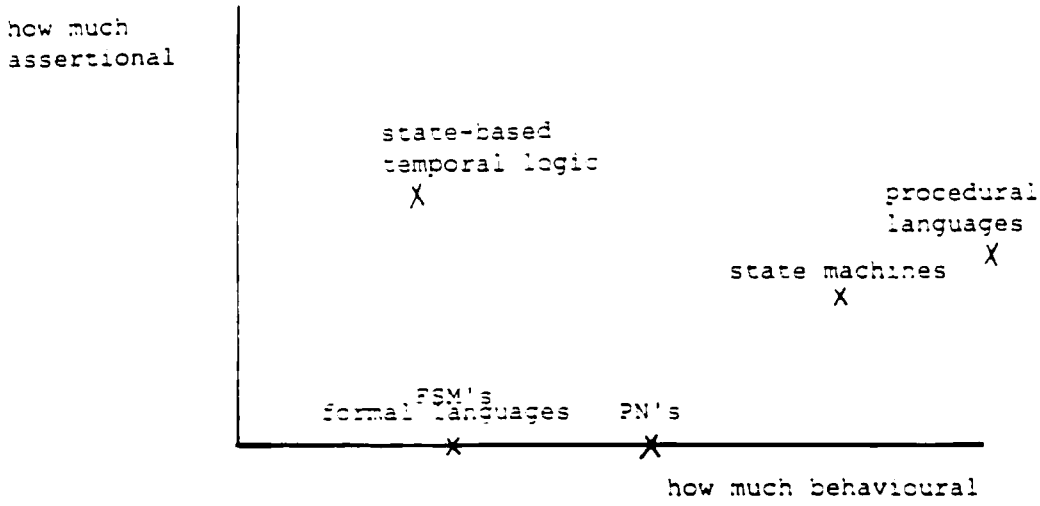
We classify specification tools along two axis. Based on the first classification, they are either *state-based* or *event-based*. The underlying model of a protocol in state-based tools is concerned with the states through which the protocol passes during its operation and with the events that cause 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 (as will be explained in section 4). 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. As will become clear in the following sections, 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.

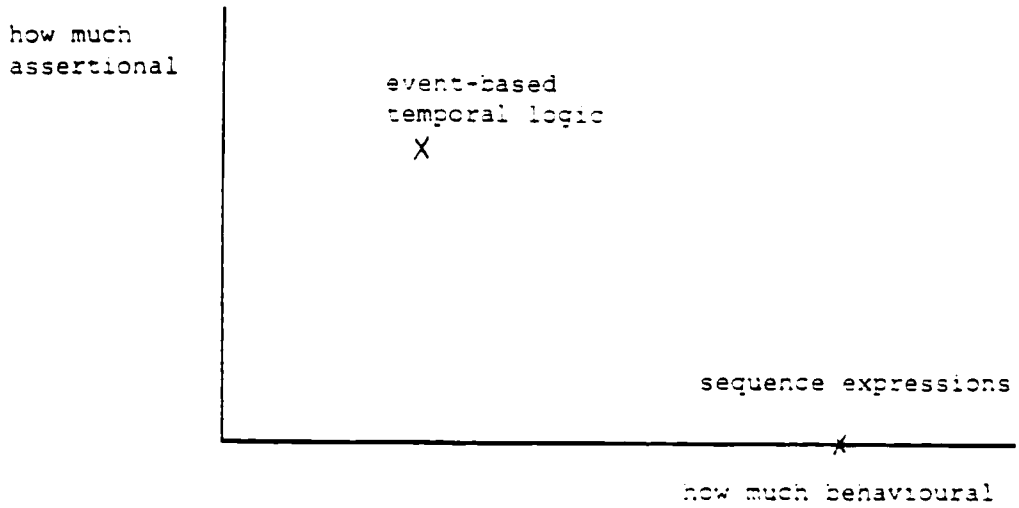
Most specification tools actually exhibit features belonging to both the behavioral and assertional classes. Also, each of these classes constitute a spectrum of specification tools. The extent to which a specification tool is behavioral depends on how much support it provides for the specification of 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. 9 the relative positions of the various specification tools covered in this section.

3 Protocol Synthesis Tools

The job of composing a specification for an entire protocol system is quite complex. Furthermore, given such a protocol, the problem of formally verifying that it is free from certain design errors has shown to be generally undecidable (see [Bran 83]). Towards simplifying the complexity of specifying entire protocol systems, some research has been directed towards synthesizing complete specifications of protocols, which are specifications that include all the communicating processes involved, from incomplete ones. In some of these efforts the produced specifications are also guaranteed to be free from certain design errors and thus avoid the possibly undecidable formal verification problem. The various synthesis approaches vary primarily in the kinds of design errors considered, the maximum number of communicating processes in a protocol that are supported, and the features of the transmission channel that are assumed. However, they all take advantage of the duality inherent in the interactions among protocol processes where a message sent by one process should be received at another communicating process.



(a) State-based specification tools



(b) Event-based specification tools

Figure 9: An illustration of the proposed taxonomy of specification tools

Zafiropulo, et al. [Zafi 80] have proposed an interactive and incremental synthesis technique. in which the protocol local processes are modeled as communicating FSM's with error-free FIFO channels. In each increment of interaction between the protocol designer and the synthesis program, the designer provides a sending interaction of one of the communicating processes. The program uses the already synthesized, partially constructed FSM's and a set of rules to find the state at which the receiving process can accept the sent interaction. It then prompts the designer for the state which the receiving process would enter upon

receiving the found reception. The synthesis algorithm uses a set of three production rules that find the receive interactions in such a manner as to prevent the designer from creating *unspecified receptions* and *nonexecutable interactions*. An unspecified reception indicates that a message reception that can take place is missing in the specification. A nonexecutable interaction is a reception or a transmission interaction that is included in the specification but that cannot be exercised under normal operating conditions. The designer is also notified of the presence of *state deadlocks* and *state ambiguities*. A state deadlock occurs when each and every process has no possible transition out of its current state. A state ambiguity occurs when one process can coexist in a certain state with more than one state in any other process provided that all channels are empty.

The synthesis algorithm accepts information from the designer and uses it in incrementally building trees that trace all possible executions of each process' FSM. The algorithm is in control of the incremental construction of the protocol. It must decide at which point to stop the growth of the execution trees; that is, when continuation cannot reveal any new information about the protocol. If all channel capacities are finite, or if there are only two processes with not more than one unbounded channel, then the termination of the algorithm is guaranteed [Bran 80]. Otherwise the trees can grow indefinitely and heuristics must be used to decide when to terminate their growth. For example, if channels were unbounded and there was a transmission loop in one of the FSM's, then the execution tree corresponding to this FSM can grow indefinitely. The complexity of the termination problem is the major limitation of this approach. The initial work done on the synthesis algorithm has been limited to only two communicating processes. In an attempt to generalize the algorithm for more than two processes [Bran 80], it was found that a different set of rules (still three rules) should be used. However, a proof of the production rules being necessary and sufficient only exists for the case of two processes.

Gouda and Yu [Goud 84a] proposed another synthesis methodology that accepts the complete specification of one process and produces a mirror-like specification of its communicating process. Similar to the work of Zafiropulo, et al. specifications are given as communicating FSM's, and the synthesized specifications are guaranteed to be free from the same design errors. It also computes the smallest bound on the number of messages in transit in the channels at any one time. The synthesis methodology consists of two algorithms. The first algorithm takes as input one process specification P_1 and produces two processes Q_1 and Q_2. Q_1 is computed from P_1 by adding some receiving transitions to it. Q_2 is then computed such that the communication between Q_1 and Q_2 is deadlock-free, bounded, with no unspecified receptions, no nonexecutable receptions, and no state ambiguities. The first step of this algorithm constructs a process Q_2 whose behavior mirrors that of input processes specification P_1. That is, they have the same states and transitions with the conversion of each sending (receiving) transition in P_1 to a receiving (sending) transition in Q_2 with the same label. A loss of synchronization leading to deadlock, though, might happen if some of the states in P_1 have outgoing transitions which are both sending and receiving since both P_1 and Q_2 might traverse sending transitions. To resolve this synchronization problem, correction transitions are added to P_1 to produce Q_1, and also included in Q_2. This restricts the communication pattern of the synthesized specification to a pattern of the communicating processes proceeding until a loss of synchronization is detected upon which they backup by following their correcting transitions.

The second algorithm takes, as an input, Q_1 and Q_2 and computes the smallest size for each of the two channels between Q_1 and Q_2. The communication channels are assumed to be error-free and FIFO, and the number of processes supported by the algorithms is limited to

two. An advantage of this synthesis approach is that each of the two algorithms takes a deterministic time of $O(st)$, where s is the size of the state space and t is the number of transitions in the input process specification.

Bochmann and Merlin [Boch 83b] describe a synthesis approach that in contrast to the two described above, does not produce error-free specifications. It has, though, the unique feature of employing the service specification of the protocol in the synthesis procedure. Both the duality principle of communication between processes and the fact that the combined communication of a protocol layer process should provide its service are used in the synthesis procedure. The synthesis algorithm takes as input the service specification as well as the specifications of the protocol layer (n-1) communicating processes and determines the specification of the remaining process (provided one is possible). The process specifications are given as sequence expressions and inter-process communication is modeled by direct coupling.

A formula is used in generating the specification of the remaining process. The specification produced is maximal in the sense that it includes the largest number of execution sequences possible, and thus corresponds to the most general process (including possibly redundant transitions). Also, it might reach deadlock when interacting with the other processes. The approach does not guarantee that all execution sequences specified for the system will be produced by the interaction of the n subprocesses. If this is the case, then there exists no process that together with the given (n-1) process can provide the required system service. The communication channel is modeled as process in the layer, and the approach could support any number of interacting processes.

This synthesis approach can be applied to the send-and-wait protocol as follows. Given the specifications of the Service to be provided, the Medium, and the Sender of the protocol, the approach can produce the specification of the receiver process.

4 Implementation Tools

An implementation tool is a construction tool (a compiler in effect) that transforms a protocol specification into code. While low-level protocols in the ISO hierarchy are often implemented in firmware, high-level protocols are implemented in software. For an example of the former, the reader is referred to [Goud 76]. In this section we will limit our discussion to software implementations of protocols.

Clearly, one would like protocol implementation tools to be automated in order to minimize both the effort involved and the probability of errors. This depends not only on the protocol specification tool used but also on the programming language used for implementation, and on the complexity of the protocol. Subsequently, we first examine the extent to which the various specification tools facilitate the automation of the implementation process and the general approaches employed. We then examine some implementation choices encountered when translating protocols given in any specification tool.

In our proposed taxonomy in section 2.3, we classified specification tools into behavioral and assertional tools along one axis and into state-based and event-based tools along another. Behavioral specifications, such as state machines and petri net-based tools, lend themselves more easily to direct translations into implementation than assertional specifications, such as temporal logic. This is because the former describe how the execution of a protocol proceeds, while the latter are concerned with requirements of protocol operation and not

with how the requirements are achieved. Furthermore, event-based specification are more difficult to translate into implementations than state-based specification because they are concerned with the outcomes of the protocol operation and not with how the outcomes are produced. In summary, state-based, behavioral specifications are the most suitable for direct translations into implementations.

Let us next discuss some works on implementing protocols specified in the various specification tools. Procedural specifications are clearly the easiest to be transformed into code because they are the richest in terms of expressing both the syntax and detailed semantics of protocol operation. The resulting implementation would probably be in the same language used for specification, with the addition of implementation specifics such as buffer management functions.

The typical approach for implementing a FSM specification, as described in [Boch 82], is to translate it to a looping program, with each cycle of the loop executing a transition. The loop would consist of a set of conditional statements with each testing for one kind of input interaction. Note that this construct is basically Dijkastra's guarded command [Dijk 75]. For each of these cases another set of conditional statements would test the major state of the module and compute the next state accordingly. State machine specifications and hybrid petri net specifications, which combine state transition specifications with high-level language statements, can be translated into code by simply transforming the state transition parts as described above and using the high-level statements as they are or with minor variations in the implementation. Bochmann, et al. [Boch 79] transformed manually a state machine specification of the X.25 protocol into an implementation in a Concurrent Pascal. Blumer and Tenney [Blum 82] in translating a state machine specification of the National Bureau of Standards' (NBS) transport protocol into C implementations, were able to produce 40% of the implementation automatically.

Sequence expressions, which belong to the event-based specification class, can not directly be directly translated into implementation, but need to be first transformed into a behavioral specification. This is similar to the derivation of a FSM that would generate a given regular expression. In implementing sequence expressions, which have much in common with regular expressions, Schindler, et al. [Schi 81] uses a two pass compiler to derive a Flow Control Graph (FCG) from the specification and then checks whether this graph is equivalent to some extended finite state machine (EFSM). If so, a PASCAL implementation of this EFSM is generated in the second pass.

Yelowitz, et al. [Yelo 82] describe an experiment of manually implementing AFFIRM algebraic specifications with its underlying abstract data types and state machine models in the Ada programming language. Abstract data types, state variables, and events in AFFIRM are mapped into types, objects, and tasks in Ada, respectively. In order to describe concurrency of the implementation of local processes, a feature not supported by AFFIRM, a special synchronization task that does not correspond to any AFFIRM event is added to the Ada implementation. Any task corresponding to an AFFIRM event has to get permission before proceeding with its actions, and upon completion thereof, notifies the synchronization task. Then, the synchronization task can be used to implement any desired imitation of, or even true, concurrency.

Finally, there are issues underlying any implementation tool, which preclude completely automated implementations. Human intervention in protocol implementations is required for two purposes. First, to add the implementation dependent parts, and message coding.

Second, the implementor often has to make certain choices based on the specific protocol being implemented. For example, whether to implement the protocol modules as part of the operating system or as cooperating user processes, and how will the different modules interact: using shared memory, or using some kind of interrupt mechanism, are two possible choices.

5 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 *implementation verification* [Boch 80a].

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 could be

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 such as include freedom from unspecified receptions, nonexecutable interactions, and state deadlocks (as defined in section 3). Other general properties include *progress* and *absence of medium overflow*. Progress means absence of cyclic behavior (also called tempo-blocking) where the protocol enters an infinite cycle accomplishing no useful work. Absence of medium overflow means that the number of messages in transit in the medium is always less 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 the problem of design verification.

5.1 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 9(a). State exploration of the concurrent behavior 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 general desirable properties of the protocol such as progress, absence of tempo-blocking and medium 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. 10. All send events in the graph are followed by the corresponding receive event indicates absence of unspecified receptions, and all the transitions in the FSM specification of the communicating processes in Fig. 3 have corresponding links in the reachability graph indicates 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. 3. The system would then deadlock at state 5 in Fig. 10 if the medium loses a message. Note that in producing the graph of Fig. 10, 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. 10 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, as will be discussed in more detail in section 6.

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. 4 states 1 and 2 are implemented by states 1 and 8 in Fig. 10 respectively, and events GET and DELIVER in Fig. 3 correspond to $\underline{m}_{C,S}$ and $m_{R,D}$ in Fig. 10 respectively.

The principal advantage of state exploration is that it could 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

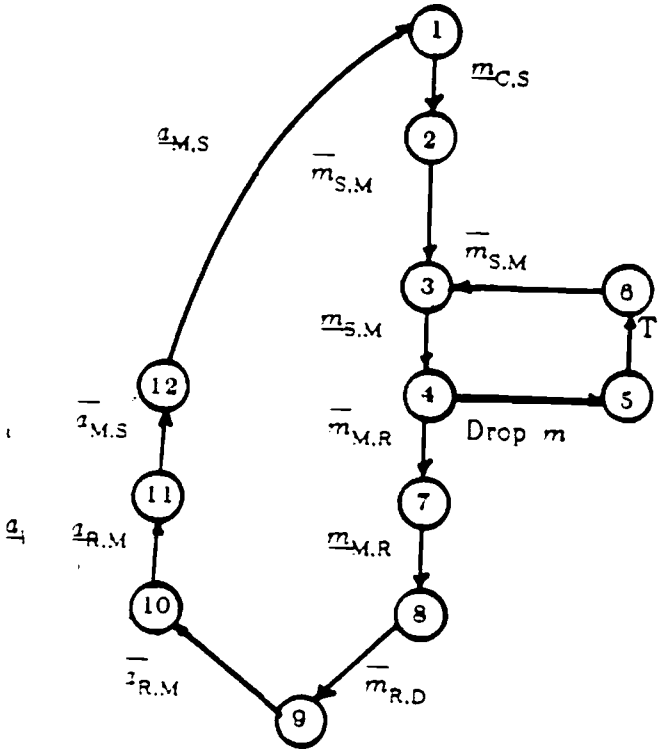


Figure 10: A reachability graph for the send-and-wait protocol

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 FSM's, is generally undecidable [Bran 83] except for a restricted class of communicating FSM's [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], limiti..g the number of messages in the medium [West 82, Noun 84], 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, using some equivalence relation to minimize the reachability graph [Rubi 82]. In addition, instead of verifying the compiete global behavior of a protocol, considerable simplification could 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 can not 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.

5.2 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 2.3. 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 could 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 verification systems used for protocols, the reader is referred to [Suns 82b, Suns 83].

6 Performance Analysis Tools

Performance analysis of protocols includes *specification and verification of timing requirements*, and *evaluation of performance measures*. The behavior of protocols depends on timing requirements, and so these requirements should be specified and verified in order to ascertain correct behavior. The efficiency of protocol behavior is decided through the evaluation of its key performance measures. 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 performance models of both the protocol and the communication medium. operating environment, are required. The latter 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), the following medium characteristics should be specified: bandwidth, bit error probability, topology, medium configuration (i.e., half or full duplex), and the maximum bound on the number of messages in transit at any one time.

A performance model of a protocol could be either formulated directly based on its operation, 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 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, Rudi 84, Noun 84]. Nounou and Yemini combine the specification of event times and probability in a marked point process model of protocol performance [Noun 84]. Third, some statistics for message lengths should be provided. These are typically considered as constants or represented by their distributions.

6.1 Specification and Verification of Protocol Timing Requirements

Protocol timing requirements are predicates stating the correct timing relationships between protocol events. Consider, for example, a retransmission on time-out protocol such as the send-and-wait protocol. The correct functioning of the protocol depends, among other things, on the requirement that time-out would occur after a message loss only with a very small probability. 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.

Consequently, it becomes apparent that the classical correctness paradigm of safety and liveness is not enough. Verification of safety properties might 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 was found in the alternating bit protocol [Bart 69], which has been proven safe and live [Yemi 82]. It was shown that the protocol would never achieve its eventual goal if the time-out rate is not properly set. Thus the ultimate goal of verification tools should be to unify verification of protocol timing requirements with the verification of their functional requirements.

Early work on the specification of timing requirements was done by Merlin [Merl 76] using time PN's (see section 2.2.4). A bounds representation of time was 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.

Nounou and Yemini use a time constraint relation "$\prec$" to define correct orderings of protocol events involved in its global behavior. Consider the example of the send-and-wait protocol without the assumption that time-out occurs only after message loss. Let the time-out event be denoted by τ_T, message loss by τ_{ld}, and the global behavior of the protocol by G. The time constraint $\tau_{ld} \prec_G \tau_T$ states that whenever in G there is a choice between τ_{ld} and τ_T, then the probability of τ_T occurring is zero. This would ensure that there are no premature time-outs. (The reader is referred to [Noun 84] for the complete time constraint.) G can be divided into a set of behaviors satisfying the given time constraint and another set that does not. A behavior satisfies a time constraint whenever there is a choice between the involved events, the event on the right hand side of the time constraint occurs. Let G_{TC} denote such a set of behaviors. The protocol's timing requirement could be then given as

$$\text{Probability}[G_{TC}] \geq 1-\epsilon$$

where ϵ is a small probability error. The timing requirement states that the subset of protocol behaviors $[G_{TC}]$, in which time-outs occur only after messages are lost, happens with a very high probability. Using rules for evaluating behavior probability, the probability of G_{TC} can be evaluated as a function of the time-out rate. As a result, an upper bound of the time-out rate for a given ϵ is computed. A distribution representation of In this approach event times were represented by their probability distribution.

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". Time constraints of the form "event e will occur within a certain elapsed time interval" are stated as safety properties and verified accordingly.

6.2 Evaluation of Performance Measures

Key protocol performance 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*.

6.2.1 Analytic Tools

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. 10 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. 11. The problem can be stated as follows: given a 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, evaluate the mean value of delay d between state 2 to 8 in Fig. 11.

Recall from section 1 our assumption that time-out only occurs 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

$$p = 1 - (1 - p_b)^l \qquad (6.1)$$

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

The mean delay is given by

$$E[d] = p/(1-p)\,(t_T + t_s) + 3t_s + 2t_d \qquad (6.2)$$
$$= 0.357 \text{ sec/message}$$

and the second moment of d is

$$E[d^2] = p/(1-p)\,(2t_T^2 + 2t_s^2) + 2p^2/(1-p)^2\,(t_T + t_s)^2 + 6t_s^2 + 4t_d^2 \qquad 6.3$$
$$= 0.09$$

Derivations of equations 6.2 and 6.3 are given in appendix I. Assume that messages arrive at state 2 in Fig. 11 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]:

$$T = E[d] + (\lambda\, E[d^2])/(2[1-\lambda\, E[d]]) \qquad (6.4)$$

In Fig. 12, 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.

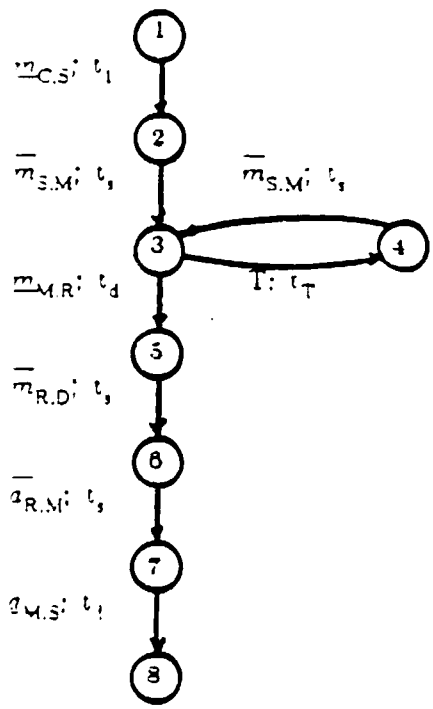


Figure 11: A modified reachability graph for the send-and-wait protocol

Examples of specification-based performance evaluation tools include works by Molloy [Moll 81] and Nounou and Yemini [Noun 84]. 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 process, that is isomorphic to the set, is generated. The steady-state probabilities of the Markov process can be 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. Nounou and Yemini associate probability and time attributes with protocol behaviors which are specified algebraically. Using rules for evaluating the attributes from the distributions of inter-event times, behavior attributes can be determined. These attributes can be used in defining and computing such measures as throughput and delay. Unlike the previous approach, there is no inherent restriction on the distribution of event times. 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 might be in some cases too restrictive as is the case, for example, with the markovian property of the resulting performance model of SPN's.

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 directly determined from a human understanding of its operation.

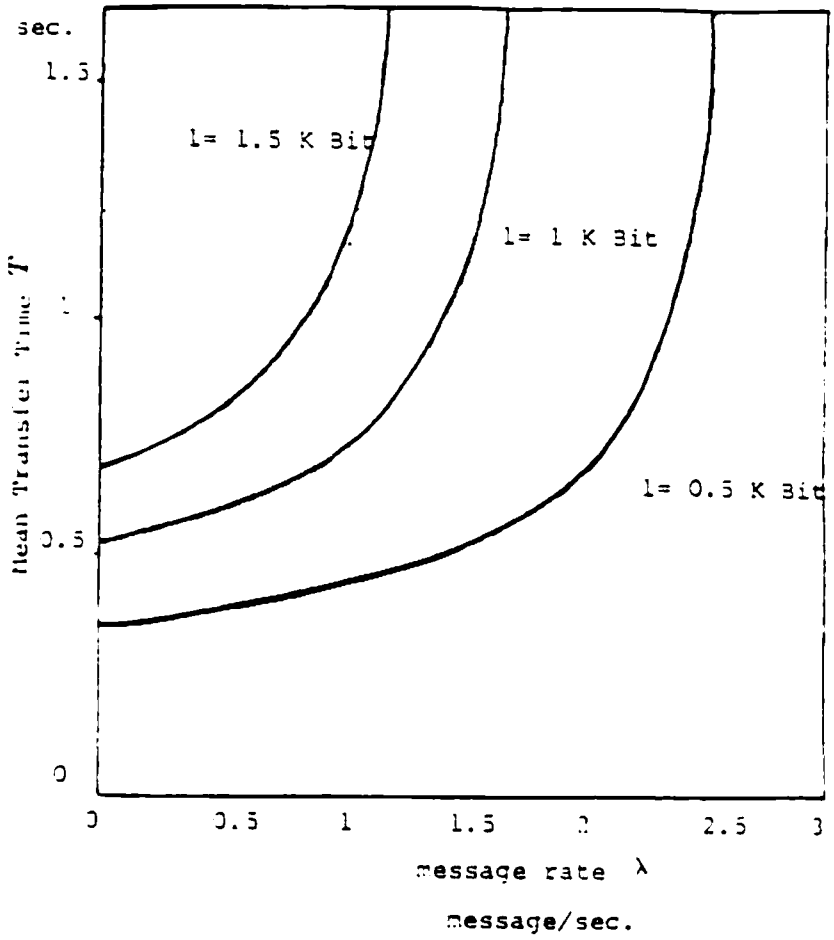


Figure 12: Transfer time vs. arrival rate of the send-and-wait protocol

6.2.2 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 could 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. 9, a method that is easily executed is one that could also be easily transformed into an implementation. Therefore, the same discussion in section 4 on the ease of implementing a protocol specification also applies here. 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. A direct simulation of the HDLC procedures was 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 could be alleviated through a large number of simulation runs.

7 Testing Tools

Testing is a validation tool that can be used to examine whether a protocol implementation satisfies the functional requirements set by its standard, measure its performance and assess its robustness in recovering from exceptional conditions. Exhaustive testing basically aims at exercising all possible behaviors of the protocol under consideration. This, however, is not realistic for most real-life protocols which typically exhibit a large set of possible behaviors. Therefore, part of the protocol testing problem is to find a way of identifying the most probable protocol behaviors and thus produce testing results which are within a certain range of accuracy. Consequently, testing as a validation tool is weaker than formal verification because it does not guarantee correctness and is less rigorous than analytic methods of performance analysis because it can only provide measurements for specific performance parameters. Nevertheless, testing is a valuable validation tool required to confirm that the implementation under test (IUT) satisfies the standard implementation of the protocol and thus ensure that different implementations of the protocol will be able to internetwork.

In the context of the ISO hierarchically layered architecture, a protocol module at layer N has two interfaces: the N interface through which service requests to layer N are provided, and the N-1 interface through which layer N requests services from layer N-1. In order to test an implementation of such a protocol, one must test its response to erroneous as well as correct requests across each of these two interfaces. An incorrect request at the N interface indicates an incorrect service request, but an incorrect response at the N-1 interface could result from either an incorrect response from the remote peer module or an error in the transmission of a correct response through the communication medium. All these possibilities must be covered in testing an IUT.

Testing of protocols can be either *direct* or *remote*. In direct testing the IUT is tested in a simulated environment where correct and faulty responses from the lower protocol layer are simulated, and the results compared with those of a standard reference implementation. In remote testing, an IUT is tested in its normal operating environment, where it is at one end of the network and some reference implementation of the protocol is at the other end. The reference implementation is driven by the protocol tester and the operation of the implementation under test is observed remotely. Note that testing in the second approach is probably more complete and more detailed than the first approach. This is at the cost of increased complexity however.

Several groups around the world are currently involved in proposals for testing centers that would be responsible for carrying out the remote tests and accordingly provide certificates describing the performance of a client's (an implementor of a protocol implementation) IUT on them. The groups include the National Physics Laboratory (NPL) group in England [Bart 80, Rayn 82], the Agency de l'Informatique (ADI) in France [Ansa 81, Ansa 82], the Gesellschaft fuer Mathematik und Datenverarbeitung (GMD) in Germany [Falt 83], and the National Bureau of Standards (NBS) in the USA [Nigh 82]. Other specialized protocol testing architectures for certain network architectures have been proposed. For example, the X.25 testing facilities for the Datapac network [Weir 78], an architecture for testing IBM's systems network architecture (SNA) protocols [Cork 83], and a BX.25 (an X.25 compatible protocol developed at Bell Labs) certification facility [Meli 82]. We will restrict our discussion to general testing architectures.

We examine next the two main issues pertinent to testing: logical architectures for testing and techniques for selecting test sequences.

7.1 Logical Architectures for Testing

Within the framework of the ISO model, a common logical testing architecture is given in Fig. 13. In this architecture the peer protocol implementation (PPI) of the IUT is a combination of a reference implementation and a protocol-data-units generator (see Fig. 13). The PPI at layer N together with reference implementations for layers 4,5,....,N-1 are located at the test center, while the IUT is at the implementor's site. Both ends are connected to an X.25 network which provides the first three network layers[1] . The protocol-data-units generator is responsible for generating correct N level service requests, requests for the generation of N-th level protocol errors, indications of undetected N-th level protocol errors, and acts as an encoder and decoder of both valid and invalid (N-1) service. The PPI and the protocol-data-units generator are driven by a test driver (TD) at the testing center. The test responder (TR) is the software module which acts as the user of the N service, and whose operation is totally predictable so that the results of the tests depend only on the behavior of the IUT. The TD and TR communicate through a non-standard protocol.

Based on this architecture, the various groups mentioned above differ in the following respects. At GMD, the TR function is performed manually thus making testing inexpensive for the implementor but slow and error-prone. At NBS, the TR is the same as the TD except that all send (receive) requests are changed to receive (send) requests. In this case no special TD-TR protocol is required. At both ADI and NPL, the full architecture is supported with the difference that the TR at ADI can handle multiple connections through the IUT which is necessary in testing protocols with multiplexing functions, whereas at NPL the TR handles only one connection at a time which has the advantage of a simpler TR. Multiple connections at NPL are handled by parallel instantiations of the same TR design.

In order to assess the IUT, it is necessary to test its response to erroneous and correct requests across both the N and N-1 interfaces. However, if the N-1 service of the protocol being tested is not end-to-end (as in the case of the packet-level of the X.25), then it is not possible to control it remotely. Therefore, a portable box is introduced between the communication medium and the implementor's system (see Fig. 14) in the testing architectures established at NPL and ADI. It is used to detect any errors introduced by the sub-network and introduce errors in it upon request from the testing center.

Clearly, making testing independent of the protocol being tested as much as possible is highly desirable so that only minimum variations need to be made when a protocol at another network layer is tested. This can be achieved by minimizing the protocol dependent parts of the architecture, and automating the process of test sequences selection. The only part of the testing architecture that needs be protocol dependent is the protocol-data-units generator, especially the part for testing normal and faulty N service. This dependency could be minimized by automating that part of the generator such that it is derived from some specification of the protocol.

7.2 Test Sequences Selection

A test sequence is an input request to the IUT generated by the TD or TR. Since the source of the IUT is typically not provided by the implementor, the selection of test sequences at the testing center can only be derived from the service and protocol

---

[1]Only end-to-end protocols above X.25 are tested in such architectures

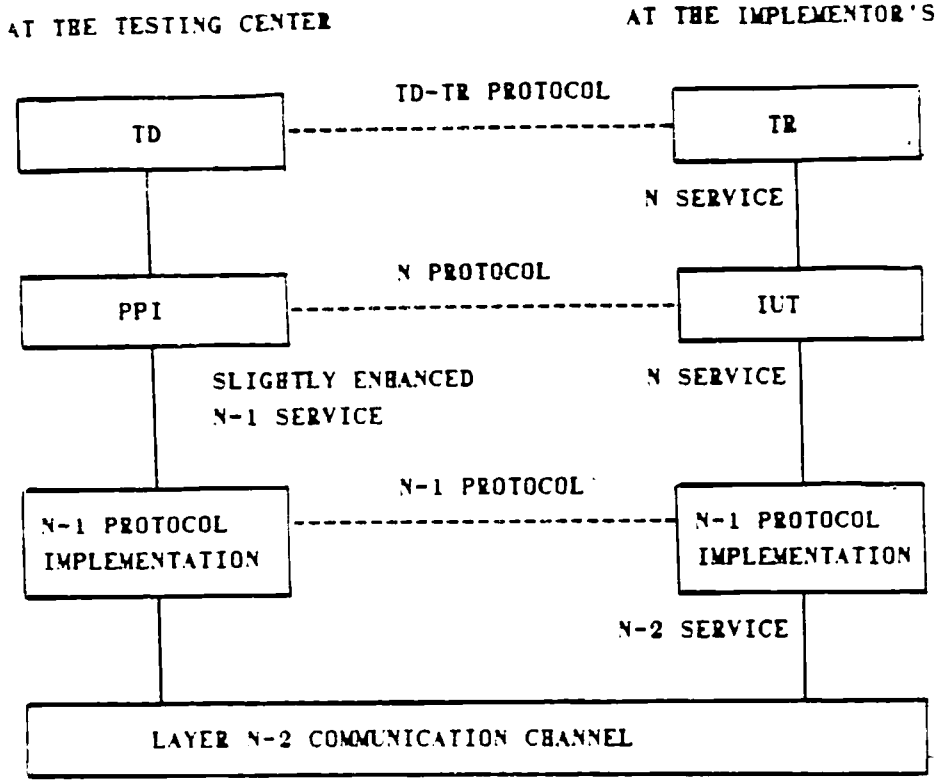


Figure 13: Logical architecture for testing

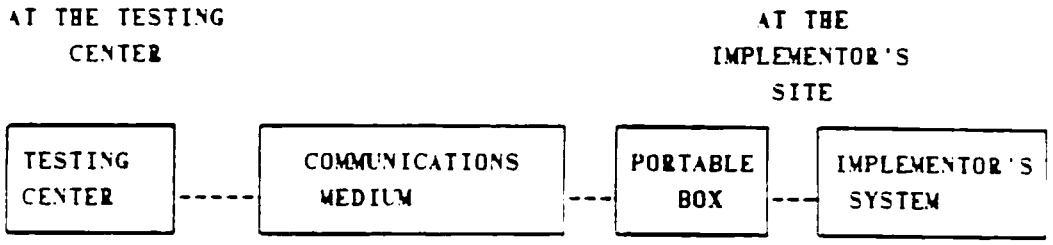


Figure 14: Physical architecture including the portable unit

specification of the protocol under consideration. Test sequences could be specified simply as sequences of commands, as state tables describing the various states of testing and the events and associated actions for each state, or using a test specification language that might be then translated into state tables [Rayn 82].

Testing is said to be *complete* if all the possible requests that could be applied to the IUT are covered by the test sequences. Unfortunately, theoretical results [Piat 80] show that without knowledge of the protocol internal state the size (measured as the number of distinct sequential inputs applied to the IUT) of a complete test sequence has an upper bound of $O(n^2)$ where n is the size of the state set of the protocol reference model. Otherwise, with an access to the protocol internal state this figure comes down to $O(n^2)$.

These bounds could be very large for complex protocols such as those involving sequence numbers.

However, there are other methods for near complete tests sequence selection [Sari 82, Ural 83]. As an example, we will use the *transition tours* method described by Sarikaya and Bochmann [Sari 82] to calculate a test sequence for the send-and-wait protocol. This method is used to derive test sequences from a protocol specified formally as a state machine but using only its FSM part. A transition tour sequence is an input sequence starting at the initial state and covering all the transitions at least once. The length of the sequence for our protocol example (see Fig. 11) is 8 and the sequence is given by

$$\underline{m}_{C,S}\ \overline{m}_{S,M}\ T\ \overline{m}_{S,M}\ \underline{m}_{M,R}\ \overline{m}_{R,D}\ \overline{a}_{R,M}\ \underline{a}_{M,S}$$

In general, the upper bound on the sequence length is $q + (q-1)(n-1)$, where q is the number of possible transitions. This is the worst case where a traversal of all (n-1) states is required to include each transition in the test sequence. This method detects all operation errors (errors in the output function of the state machine), but it does not detect all transfer errors (errors of the next state function).

8 Conclusions

In surveying the various protocol development tools, their dependency on the specification tool used has been demonstrated. Based on our taxonomy of specification tools described in section 2.3, we can conclude that behavioral specifications are better suited for synthesis, implementation, performance analysis and testing tools. Assertional specifications, on the other hand, offer better support for verification tools. Belonging to the latter class are temporal logic specifications which can adequately describe both static and temporal requirements of protocol behavior. We expect future proposals of specifications tools to combine the temporal logic framework with other specification models. In addition, since specification-based performance analysis tools are starting to attract much interest, specification tools should offer better support for the specification of protocol timing requirements and performance measures.

Most of the past research on protocol validation tools has ignored the specification and verification of such protocol timing requirements. We believe that such requirements are essential for the correct functioning of most protocols. Integrating the analysis of timing requirements in functional validation tools, i.e., verification and testing tools, would exclude unrealistic protocol behavior and thus simplify their functional validation.

In addition to the surveyed works on individual protocol development tools, there has been recently a growing interest in integrating them into development environments. An ideal development environment should provide a consistent user interface to the various tools supported. Also, recognizing the visual attraction, clarity and wide acceptance of graphical descriptions of protocols, we expect the user interfaces to employ state-of-the-art technology in supporting such descriptions. Technological advances of graphical display devices with colors, multiple window displays, high resolution, and numerous pointing aids (e.g., tablet, mouse and light pen) can be used to aid the protocol developer in constructing and validating complex real-life protocols. The integration of specification-based development tools in environments would facilitate both the functional and performance validation of

protocols starting from early development phases. Thus the costs incurred in iterations through the development phases after post-implementation detection of errors, would be reduced. Furthermore, as more protocol standards are developed, more experience will be required in the application of current and future tools and environments to these standards.

[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.

[Ansa 81] J.Ansart.
Test and Certification of Standardised Protocols.
In *Proceedings of the First International INWG/NPL Workshop : Protocol Testing - Towards Proof?*, pages 119-126. 1981.

[Ansa 82] J.Ansart.
GENEPI/A -A Protocol Independent System for Testing Protocol Implementation.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification.* 1982.

[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.

[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.

[Bart 80] K.Bartlett and D.Rayner.
The Certification of Data Communication Protocols.
In *NBS Trends and Applications Conference*, pages 12-17. May 29, 1980.

[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.

[Berg 82] H.Berg, W.Boebert, W.Franta, and T.Moher.
Formal Methods of Program Verification and Specification.
Prentice-Hall, 1982.

[Bert 82] G.Berthelot and R.Terrat.
Petri Nets Theory for the Correctness of Protocols.
IEEE Transaction on Communications COM-12:2476-2505, Decmber, 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.

[Blum 82] T.Blumer and R.Tenney.
A Formal Specification Technique and Implementation method for Protocols.
Computer Networks 6:201-217, 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 and C.Sunshine.
Formal Methods in Communication Protocol Design.
IEEE Transactions on Communications COM-28(4):624-631, April, 1980.

[Boch 80b] G.Bochmann.
A General Transition Model for Protocols and Communication Services.
IEEE Transactions on Communications COM-28(4):643-650, April, 1980.

[Boch 82] G.Bochmann et al.
Some Experience with the Use of Formal Specifications.
IEEE Transaction on Communications COM-12:2476-2505, Decmber, 1982.

[Boch 83a] G.Bochman.
Distributed Systems Design.
Springer-Verlag, 1983.

[Boch 83b] G.Bochmann and P.Merlin.
On the Construction of Communication Protocols.
ACM Transactions on Programming Languages and Systems 5-1:1-25, January, 1983.

[Boch 84] G.Bochmann.
Formal Description Techniques for OSI: An Example.
In *Proceedings of INFOCOM.* IEEE, 1984.

[Boeh 76] B.Boehm.
Software Engineering.
IEEE Transaction on Computer C-25(12):1226:1241, 1976.

[Bolo 84] T.Bolognesi and H.Rudin.
On the Analysis of Time-Dependent Protocls by Network Flow Algorithms.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification*. North-Holland, 1984.

[Bran 78] D.Brand and W.Joyner,Jr.
Verification of Protocols Using Symbolic Execution.
Computer Networks 2:351-360, October, 1978.

[Bran 80] D.Brand and P.Zafiropulo.
Synthesis of Protocols for an Unlimited Number of Processes.
In *NBS Trends and Applications Symposium*, pages 29-40. May, 1980.

[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 84] E.Brinksma and G.Karjoth.
A Specification of the OSI Transport Service in LOTOS.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification*. North-Holland, 1984.

[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.

[Cork 83] R.Cork.
The Testing of Protocols in SNA Products - an Overview.
In *Proceedings of the Third IFIP International Workshop on Protocol Specification, Testing and Verification*. North-Holland, 1983.

[Dant 80] A.Danthine.
Protocol Representation with Finite State Models.
IEEE Transactions on Communications COM-28(4):632-643, April, 1980.

[Diaz 82] M.Diaz.
Modeling and Analysis of Communication and Coopeartion 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 29,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.

[Dijk 75] E.Dijkstra.
Guarded Commands, Nondeterminacy and Formal Derivation of Programs
Communications of the ACM :453-457, August, 1975.

[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.

[Falt 83] U.Faltin et al.
TESDI Manual: Testing and Diagnosis Aid for Higher Level Protocols.
In *IFV-IK-RZ, GMD, Darmstadt, Germany*. 1983.

[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.
Semantics of Concurrent Computation, Evian, G. Kahn (ed), Lecture Notes in Computer Sciences. : The Analysis of Distributed Systems by Means of Predicate/Transition Nets.
Springer-Verlag, 1979, pages 123-146.

[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 76] M.Gouda and E.Manning.
Protocol Machine: A Concise Formal Model and its Automatic Implementation.
In *Proceedings of the Third ICCC*, pages 346-350. 1976.

[Goud 84a] M.Gouda and Y. Yu.
Synthesis of Communicating Finite-State Machines with guaranteed Progress.
IEEE Transactions on Comunications COM-32(7):779-788, July, 1984.

[Goud 84b] M.Gouda and Y. Yu.
Protocol Validation by Maximal State Exploration.
IEEE Transactions on Comunications COM-32:94-97, January, 1984.

[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.

[Hail 81] B.Hailpern.
Specifying and Verifying Protocols Represented as Abstract Programs.
IBM Journal of Research and Development RC 8674 (37908), February, 1981.

[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 Tranactions on Computers , August, 1982.

[Iso 83a] ISO TC97/SC16 N1347 .
A FDT based on an extended state transition model.
Technical Report, ISO, July, 1983.

[Iso 83b] ISO TC97/SC16 N1347 .
Draft Tutorial Document on Temporal Ordering Specification Language.
Technical Report, ISO. August, 1983.

[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 Perfromance 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.

[Lehm 80] M.Lehman.
Programs, Life Cycles, and Laws of Software Evolution.
In *Proceedings of the IEEE*, pages 1060-1075. September, 1980.

[Lond 80] R.London and L.Robinson.
Software Development Tools, W.Riddle and R.Fairley ed. : The Role of Verification Tools and Techniques.
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 Unversity. June, 1981.

[Meli 82] J.Melici.
The BX.25 Certification Facility.
Computer Networks 6:319-329, 1982.

[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.

[Moll 81] M.Molloy.
On the Integration of Delay and Throughput Measures in Distributed Processing Models.
PhD thesis, Univ. of California Los Angeles, 1981.

[Muss 80] D.Musser.
Abstarct data Type Specifications in the AFFIRM System.
IEEE Transactions on Software Engineering SE-6(1), January, 1980.

[Nigh 82] J.Nightingale.
Protocol Testing Using A Reference Implementation.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification.* 1982.

[Noun 84] N.Nounou and Y.Yemini.
Algebraic Specification-Based Performance Analysis of Communication Protocols.
In *Proceedings of the Fourth IFIP International Workshop on Protocol Specification, Testing and Verification.* North-Holland, June, 1984.

[Nutt 72] G.Nutt.
Evaluation Nets for Compuetr System Perfromance analysis.
AFIPS Conference Proceedings 41,Part 1:279-286, 1972.

[Oste 80] L.Osterweil.
Software Development Tools, W.Riddle and R.Fairley ed. : A Software Lifecycle Methodology and Tool Support.
Springer-Verlag, 1980, pages 82-118.

[Pete 77] J.Peterson.
Petri Nets.
ACM Computing Surveys 9(3):224-252, September, 1977.

[Piat 80] T.Piatkowski.
Remarks on ADCCP Validation and Testing Techniques.
NBS Trends and Applications Symposium , May 29, 1980.

[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.

[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.

[Rayn 82] D.Rayner ed.
A System for Testing Protocol Implementations.
Technical Report 9/82, NPL, August, 1982.

[Razo 84] R.Razouk.
The Derivation of Perfromance 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.
Perfromance Evaluation of Data Communication Systems.
In *Proceedings of the IEEE,* pages 171-196. February, 1982.

[Ridd 80] W.Riddle and R.Fairley.
Software Development Tools, W.Riddle and R.Fairley ed. : Introduction.
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 G8.3.1-G8.3.5. 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.

[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.

[Sari 82] B.Sarikaya and G.Bochmann.
Some Experience with Test Sequence Generation for Protocols.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification.* 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 81a] D.Schwabe.
Fromal 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 12:2478-2505. Decmber, 1982.

[Schw 83] R.Schwartz, P.Melliar-Smith and F.Vogt.
Interval Logic: A Higher-Level Teporal 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.

[Sloa 83] L.Sloan.
Meachanisms That Enforce Bounds on Packet Lifetimes.
ACM Transactions on Computer Systems 1(4):311-330. November, 1983.

[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 Sytems 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.

[Tane 81] A.Tanenbaum.
Network Protocols.
Computing Surveys 13(4):453-489, December, 1981.

[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 Multiplexers with ARQ Retransmission Schemes.
IEEE Transactions on Communications COM-27(4):693-702, April, 1979.

[Ural 83] H.Ural and R.Probert.
User-Guided Test Sequence Generation.
In *Proceedings of the Third IFIP International Workshop on Protocol Specification, Testing and Verification*. North-Holland, 1983.

[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.
IEEE Computer Society, 1981, pages 1-2.

[Weir 78] F.Weir, W.Prater, and X.Dam.
X.25 Test Facilities on Datapac.
In *Proceedings of the Fourth ICCC*, pages 273-279. September, 1978.

[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 Limitaions 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.
Recomendation X.25 Packet Switch Data Transmission Services.
Technical Report, Geneve, Switzerland, 1980.

[Yelo 82] L.Yellowitz, S.Gerhart and G.Hilborn.
Modeling a Netwprk Protocol in AFFIRM and Ada.
In *Proceedings of the Second IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 435-450. May, 1982.

[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.

[Zafi 80] P.Zafiropulo, C.West, H.Rudin, D.Cowan, and D.Brand.
Towards Analyzing and Synthesizing Protocols.
IEEE Transactions on Communications COM- 28(4):651-661, April, 1980.

[Zimm 80] H.Zimmermann.
The ISO Model of Architecture for Open System Interconnection.
IEEE Transactions on Communications COM-28(4), April, 1980.