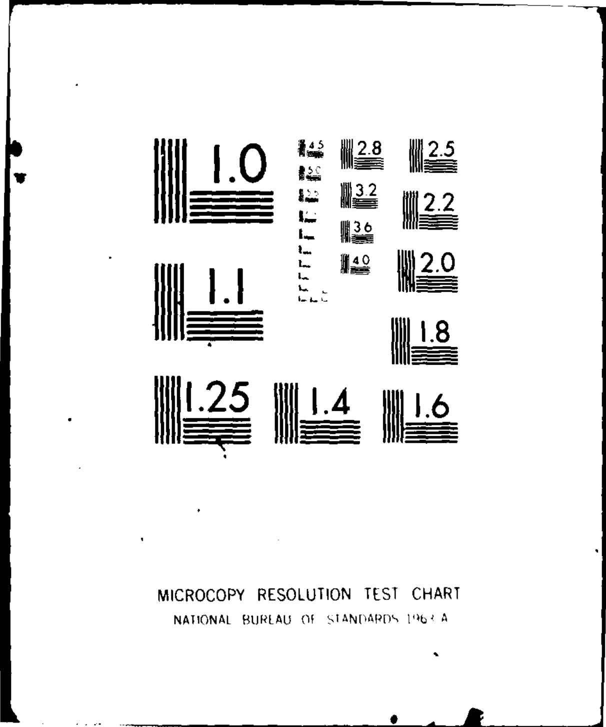
AD-A098 576 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 17/2
SPECIFICATION AND VERIFICATION OF COMMUNICATION PROTOCOLS IN AF--ETC(U)
MAR 81 D H THOMPSON, C A SUNSHINE DAHC15-75-C-0308
UNCLASSIFIED ISI/RR-81-88 NL

END
DATE
FILMED
8-81
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

ISI/RR-81-88
March 1981

LEVEL

David H. Thompson
Carl A. Sunshine
Roddy W. Erickson
Susan L. Gerhart
Daniel Schwabe

AD A098576

Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models

DTIC ELECTE MAY 6 1981 C

DTIC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/California 90291
(213) 822-1511

81 5 06 033

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ISI/RR-81-88	2. GOVT ACCESSION NO.: AD-A098	3. RECIPIENT'S CATALOG NUMBER: 376
4. TITLE (and Subtitle): Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models		5. TYPE OF REPORT & PERIOD COVERED: Research report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): David H. Thompson, Susan L. Gerhart, Carl A. Sunshine, Daniel Schwabe, Roddy W. Erickson		8. CONTRACT OR GRANT NUMBER(s): DAHC15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE: March 1981
		13. NUMBER OF PAGES: 65
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): -------		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

----

18. SUPPLEMENTARY NOTES

----

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstract data types, algebraic axiomatic specifications, Alternating Bit protocol, natural-deduction theorem-proving, protocols, specification, state transition models, verification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1473, 1 JAN 73 — EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

It is becoming increasingly important that communication protocols be formally specified and verified. This report describes a particular approach--the state transition model--using a collection of mechanically supported specification and verification tools incorporated in a running system called Affirm. Although developed for the specification of abstract data types and the verification of their properties, the formalism embodied in Affirm can also express the concepts underlying state transition machines. Such models easily express most of the events occurring in protocol systems, including those of the users, their agent processes, and the communication channels. The report reviews the basic concepts of state transition models and the Affirm formalism and methodology, and describes their union. A detailed example, the Alternating Bit Protocol, illustrates various properties of interest for specification and verification. Other examples explored using this formalism are briefly described and the accumulated experience is discussed.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Avail and/or
Dist Special
A

ISI/RR-81-88
March 1981

David H. Thompson
Carl A. Sunshine
Roddy W. Erickson
Susan L. Gerhart
Daniel Schwabe

Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research is supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

This document is approved for public release and sale; distribution is unlimited.

CONTENTS

1. INTRODUCTION 1

1.1. State Transition Models 2
1.2. Specification and Verification in *Affirm* 3
1.2.1. Data Abstraction 3
1.2.2. Theorem Proving 5
1.3. Protocols 7
1.3.1. Protocol Specification 7
1.3.2. Protocol Verification 8
1.4. Related Work 9

2. AN OVERVIEW OF OUR METHOD OF PROTOCOL SPECIFICATION AND VERIFICATION 10

3. A SERVICE SPECIFICATION FOR A SIMPLE MESSAGE SYSTEM 12

3.1. State Variables 12
3.2. State Transitions 12
3.3. Behavior of the Simple Message System 13
3.4. Converting State Transition Specifications to *Affirm* 14
3.4.1. State Transition Function → Constructor 14
3.4.2. State Variable → Selector 14
3.4.3. Transition Definition → Set of Axioms 14
3.5. The *Affirm* Representation 15
3.6. Properties of a Specification 16
3.7. Alternative Notations 18

4. VERIFICATION ISSUES 19

4.1. Verifying Properties of a Specification 19
4.2. Verifying the Protocol against the Service Specification 20
4.3. Verifying a Program against the Protocol Specification 22

5. DETAILED EXAMPLE: THE ALTERNATING BIT PROTOCOL 23

5.1. A Brief Description of the Protocol 23
5.2. A State Transition Machine for the Alternating Bit Protocol 24
5.2.1. Data Types Used in the Specification 24
5.2.2. State Variables 25
5.2.3. State Transition Functions 26
5.3. The *Affirm* Representation 27
5.4. Verifying the Protocol against the Service Specification 28
5.4.1. Safety 28
5.4.2. Liveness 31
5.5. Protocol Properties and Invariants 32
5.6. Implementation 32

6. FURTHER APPLICATIONS 34

6.1. Stenning's Data Transfer Protocol 34
6.2. Transport Service 34
6.3. Selective Repeat Transport Protocol 35
6.4. Connection Establishment Protocol 35

7. PROBLEMS AND EXTENSIONS 36

7.1. Composition of Specifications 36
7.2. Concurrency 36
7.3. Exceptions 37
7.4. Specification and Verification of Systems with More than Two Interacting Entities 37
7.5. Higher Level Protocols 38

8. CONCLUSION 39

APPENDIX I. DATA TRANSFER SERVICE SPECIFICATION 40

APPENDIX II. THE PROTOCOL REPRESENTATION 42

APPENDIX III. SERVICE AXIOMS → PROTOCOL THEOREMS 48

III.1. The Correspondence between the Service and the Protocol 48
III.2. Correspondence of States Between Service and Protocol 48
III.3. Example: Mapping Two Service Axioms into Protocol Theorems 49
III.4. Effects on State Variables by User Operations 50
III.5. Effects on State Variables by Spontaneous Operations 51
III.6. The next-state Transitions for all Operations 51

APPENDIX IV. IMPLEMENTING PROCEDURES AND ASSERTIONS 53

IV.1. Asserted Procedures for the Sender 53
IV.2. Asserted Procedures for the Receiver 54
IV.3. Definitions for the Assertions 55
IV.4. Context in Which the Assertions are Defined 56

REFERENCES 57

FIGURES

Figure 1-1: A simple message system 3
Figure 1-2: The internal structure of the service machine 8
Figure 2-1: The steps in protocol verification 11
Figure 5-1: The protocol state transition machine 24
Figure 5-2: The correspondence between service and protocol-level state variables 29

1. INTRODUCTION

When we send electronic mail, funds, or programs to another site, we expect many things to happen: the message should be delivered to a particular site and not to others; only one copy of the message should be delivered; the delivery should be timely; the receipt should be acknowledged; and so on. In computer science terms, these properties are often called *safety* (correct delivery), *liveness* (effective work being done), and *performance* (work being done fast enough). The social importance of guaranteeing these properties for electronic media cannot be over-valued: our dependence on such systems increases daily.

Over the past few years, the Internetwork Concepts Research project at ISI has been studying the overall problem of protocol verification, as well as the design of correct protocols. Simultaneously, the ISI Program Verification project has been developing a general-purpose specification and verification system called ***Affirm***. This report presents the results of joint research over a year's time. Specific accomplishments include increased understanding of an underlying formalism (state transition models), rendering of such models in the specification language of ***Affirm***, experimenting with various ways of expressing the three properties mentioned above so that they can be proved for state transition specifications, study of several levels of specification (all the way from the user services down to the programming language implementation), an in-depth study of a particular protocol (the Alternating Bit protocol), and a survey of a number of other protocols. Our overall accomplishment is a general method of specifying and verifying certain aspects of protocols, supported by mechanical assistance. Most of our work has focused on safety properties, rather than liveness and performance properties.

Because we expect at least one of the three areas of communication protocols, state transition machines, and abstract data types to be new to most readers, we have included an introduction to each of these topics in this chapter. The main bulk of the report presents a rather simple example of the integration of these concepts. Thus the emphasis is on *methodology* rather than the results obtained for a particular protocol. Later work [42] will present extensive concrete results on protocols of more practical interest.

Our general method of protocol specification and verification is summarized in Chapter 2. Details of the specification method are illustrated in Chapter 3. Verification issues are considered in Chapter 4. The method is applied to the Alternating Bit protocol in Chapter 5. Chapter 6 summarizes some of the results obtained with more complex protocols. Extensions and problems are analysed in Chapter 7. Our conclusions are presented in Chapter 8.

1.1. State Transition Models

A variety of methods for modeling the behavior of systems in terms of state transitions have been developed, including finite state automata (FSA) and abstract machines. The key components of these models are as follows.

1. A set of *commands* (also called *inputs* or *events*).

2. One or more *state variables*, collectively called the *state*.

3. A *transition* function

   (command $\times$ state) $\rightarrow$ state.

4. An *initial* state (assigning initial values to all the state variables).

Each command is a single *state transition function* mapping the current state into a new state. Generally, commands are considered atomic operations that are processed sequentially: no concurrent commands are allowed.

A state transition machine operates by starting in its initial state. At unspecified times, the state is transformed by one of the state transition functions (or an input "appears," and is used by the overall transition function to effect a state change). The machine may be designed to operate forever, or may have a specified set of *final* states. When one of these states is reached the machine is considered to have halted.

Within these basic guidelines, there are a number of possible variations. State variables may be defined as value-returning functions. The commands may have parameters. The effects of commands may be made visible to the outside world (i.e., the users of the machine) by defining some of the state variables to be visible, or by producing explicit outputs as additional effects of an operation. *Exceptional* conditions may be specified where a given command has no effect on the state of the system except to produce an error indication or output to the invoking user. If the data types of the state variables are unbounded (e.g., a queue), the model may not have a finite number of states.

State transition models are often written graphically, with circles representing states and arcs representing transitions. Each arc is labeled with the command causing the transition. Outputs produced are also written on the arcs if needed. Fig. 1-1 gives an example of a state transition model for a very simple message system allowing only a single message in transit from sender to receiver. (This example is explained further in Chapter 3.)

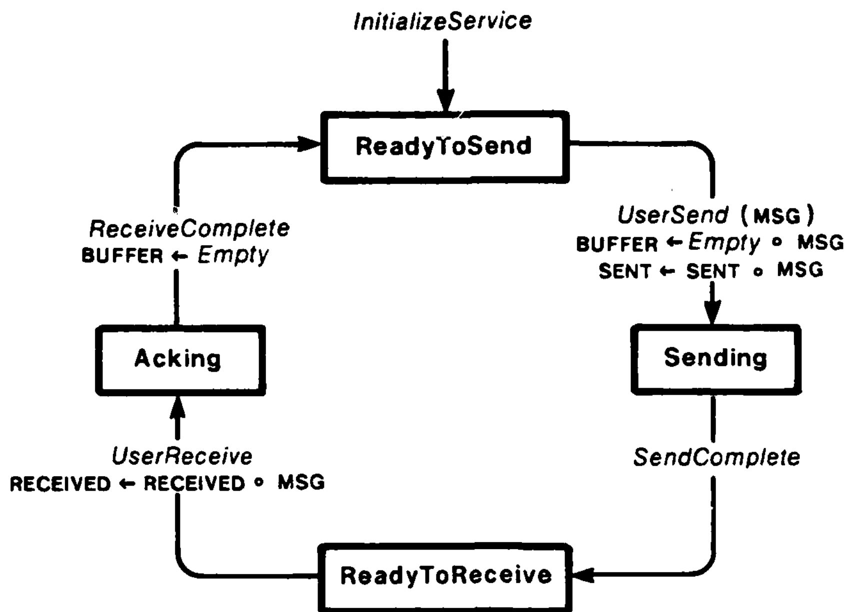


Figure 1-1: A simple message system

1.2. Specification and Verification in Affirm

Affirm [31, 9, 50] is an experimental system for the algebraic specification and verification of user-defined abstract data types. The heart of the system is a natural deduction theorem prover for the interactive proof of data type properties. (These properties are stated in the predicate calculus extended with data types.) Programs, written in a variant of ***Pascal*** extended with data types, may be verified using the inductive assertion method [8]. Additional features include tools for the analysis of algebraic specifications, a library of useful data types, and user interface facilities. Experience includes extensive experimentation with data type specifications, verification of small programs, the specification and partial proof of a large file-updating module, and the proof of high-level properties of protocols and security kernels.

The specification and theorem-proving portions of ***Affirm*** are relevant to the current discussion.

1.2.1. Data Abstraction

As Guttag has explained [14, 15, 16], a data type is specified by first defining three sets of functions:

1. *Constructors*. These functions create values of the type. Their range is the data type being specified. All values of the type can be described in terms of a some functional composition of these functions.

2. *Extenders* (or *Modifiers*). These functions also have the data type being specified as their range, but in contrast to the constructors, they are not needed to express values of the data type. (These functions can be expressed in terms of the constructors.)

3. *Selectors* (or *Predicates*). These functions yield values of types other than the one being specified. The general term is *selector*, but functions yielding values of type *Boolean* are often termed *predicates*.

For example, the constructors of a queue are *NewQueue* (the empty queue) and *Add* (appends an element to a queue). Example extender functions are *Remove* (deletes the first element from a queue) and *Append* (concatenates two queues). Example selector functions are *Front* and *Length*. Example predicates are *in* and *nodups* (asks whether there are any duplicate elements).

```
declare q, q1, q2: QueueOfInteger;
declare i: Integer;

interfaces NewQueueOfInteger, q Add i, Remove(q), Append(q1, q2): QueueOfInteger;

interfaces Front(q), Length(q): Integer;

interface i in q: Boolean;
```

The effect of such a specification is to view values of the type in terms of the constructors that build them. All selectors and extenders are defined in terms of these constructors. For example, the queue of integers

<1, 2, 3>

is represented (in infix form) as

((NewQueueOfInteger Add 1) Add 2) Add 3

Thus the first part of a specification gives the names of all operations, their domains, and their ranges (e.g., the syntax of the type).

The second part of a data type specification provides semantics for the operations. Extenders and selectors are defined by equational axioms relating how each function behaves when applied to each of the constructors. (Constructor functions are treated as primitive, unspecified operations.) These axioms look like equations but are treated by ***Affirm*** as left-to-right rewriting rules. Various methods are used to check the consistency and completeness of the axioms [30, 31]. For example, some axioms from the type *QueueOfInteger* are:

```
axioms
        Remove(NewQueueOfInteger) = = NewQueueOfInteger,
        Remove(q Add i) = = if q = NewQueueOfInteger
                            then q
                            else Remove(q) Add i,
```

```
Length(NewQueueOfInteger) = = 0,
Length(q Add i) = = Length(q) + 1;

Append(q, NewQueueOfInteger) = = q,
Append(q1, q2 Add i) = = Append(q1, q2) Add i,
```

An important use of these data type specifications is to obtain levels of abstraction, in particular, to avoid low-level implementation details. For example, in our specification of a queue we don't care whether it is implemented with an array or via pointers and a linked list. Of course, implementation details do constrain the abstraction, e.g., by space limitations, but this is a separate problem. A standard method for relating implementations to their abstractions is the *representation* (or *abstraction*) function *rep* mapping from implementation to abstraction [22, 52]. For example, we might define a function

```
rep(a, lb, ub) = = if lb > ub
                   then NewQueue
                   else rep(a, lb, ub-1) Add a[ub]
```

to map from an array a over the sequence of (integer) indices lb to ub into queues.

The proof of correctness for an implementation involves showing that all abstract operations of interest have code that computes, via the *rep* function, the proper function. For example, we might have a procedure

```
procedure RemoveImplementation(var a: Array; var lb, ub: Integer);
  pre wf(a, lb, ub);
  post wf(a, lb, ub) and rep(a, lb, ub) = Remove(rep(a', lb', ub'))
   ... body of procedure ...
```

where the primed notation *x'* denotes the initial value of *x* at the start of the procedure. The expression "wf(a, lb, ub)" is the *implementation* (or *concrete*) invariant *well-formed*, a predicate showing that the variables of the implementation will always map into some abstract object. In the inductive assertion method, the interpretation of the pre and post conditions is as follows. If the precondition holds for the variables at entry to the procedure, then the postcondition will hold for the variables at procedure exit. Note that there is no statement that the procedure terminates.

1.2.2. Theorem Proving

Typical data type properties might include "*the length of the concatenation of two queues is the sum of their lengths*," stated as

Length(q1 Append q2) = Length(q1) + Length(q2)

and "*The length of any queue is always nonnegative*":

Length(q) $\geq$ 0

Such properties are proved by induction based on the constructors of the data type, that is, using *structural induction*. For our queue example, the induction schema uses the inference rule

$$\frac{P(NewQueueOfInteger),\ (all\ q,\ i\ (P(q) \supset P(q\ Add\ i)))}{(all\ q\ (P(q)))}$$

In other words, we prove the property P for *NewQueueOfInteger* and then, assuming it for some queue *q*, prove P for *q* with any element *i* appended to it (*q Add i*). These two proofs suffice to prove P for all *q*.

Affirm 's style of theorem proving is interactive. The user develops the proof; the system's role is to follow the user's commands and provide various kinds of necessary information and checking. It does not attempt to search for a proof. ***Affirm*** simplifies propositions using the data type axioms (as rewrite rules), with built-in simplification procedures for the predicate calculus. The user can ask the system to employ induction, split into subgoals, substitute equalities, and apply lemmas; experimentation with various strategies is often necessary before finding a proof. This experimentation and backtracking is supported with a model of the proof as a forest of proof trees, and with numerous display and query features.

The overall effect is that the user follows the usual mathematical proof methods, but ***Affirm*** carries out the mechanics of the proof (down to the axioms or assumptions). Of course, proofs are not ironclad: there might be a bug (in either our code or the underlying Interlisp system),[1] or the user might make an invalid assumption. ***Affirm*** is used to produce better, not guaranteed perfect, proofs. Such proofs should also be readable (when properly structured in terms of lemmas) and read to be believed.

A more serious problem is that of ascertaining that we have proved (or are trying to prove) what we really want proven. Experience has shown repeatedly that propositions we thought were theorems were not; this quickly led us to the conclusion that "the purpose of proving (with ***Affirm***) is to turn a conjecture into a theorem."

---

[1] To our knowledge, ***Affirm*** has never generated an invalid proof; we consider it unlikely that an error would produce just the right behavior to validate an incorrect theorem, particularly since the user would probably note associated strange behavior. The usual result of a bug is to prevent a valid proof from proceeding. However, soundness cannot be guaranteed.

1.3. Protocols

In order to apply state transition models and abstract data types to communication protocols, we must first understand specification and verification problems in the protocol domain. The meaning of protocol specification and verification will be described in terms of a model first introduced in [47].

1.3.1. Protocol Specification

A user's interest in a protocol lies in what kind of *services* it provides. Usually the service involves interactions with other entities (such as users or programs) in order to get certain functions performed. For example, one user may wish to interact with another (remote) user by performing various functions such as *SendMessage*. How these functions are actually performed by the protocol is not really of concern; only the end result matters.

Users, then, can regard the protocol as a black box, to which one gives a series of commands in order to get certain services performed. The description of this machine is termed the *service specification*. One theorem we may wish to prove about a service specification is that the messages received constitute an initial subsequence of the messages sent (i.e., messages are not delivered in the wrong order, or garbled, nor are messages spontaneously delivered if they were not sent).

In general, the components used to provide the service can also be regarded as black boxes in their own right. In the case of protocols there is always more than one entity interacting (because we are dealing with distributed systems). In order to provide a given service, it is necessary to have several *stations* (at least one for each physical site) interacting with each other via some *transmission machine* (see Fig. 1-2). The pattern of their interactions constitutes the protocol.

This transmission machine is just another level of protocol. Thus we can see a hierarchy of abstract machines developing. In this *uses* hierarchy (following Parnas [36]), each protocol level makes use of the services provided by the lower level. Within each level, there is an *implementation* hierarchy where the service is logically implemented by the abstract protocol specification. The protocol is implemented in turn by an actual program. Thus for each protocol level N, the following information must be provided:

1. A *service specification*, describing the services provided by the level to the users above, at level $N + 1$;

2. A *protocol specification*, describing the interaction of the objects in this level in a precise way (assuming services provided by the level below, level $N - 1$); and

3. A *program* implementing each station in the level (of course, the program may vary from station to station).

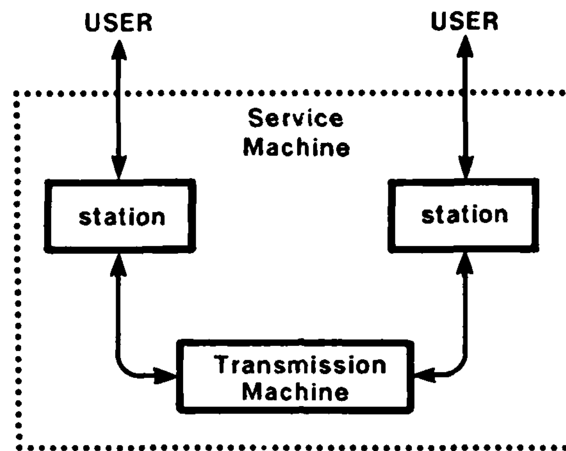


Figure 1-2: The internal structure of the service machine

This characterization follows closely the model for open system interconnection being proposed by the International Organization for Standardization [23].

1.3.2. Protocol Verification

In the context of the model introduced in the previous subsection, we say that *protocol verification* is *a formal demonstration that the logical design of the protocol (the interaction of the stations within one layer) satisfies the service specification of that layer.*

Note that this will depend on the assumed properties (the service specification) of the layer below.

The ultimate task in protocol verification is to demonstrate that an actual program is a valid implementation of the protocol specification. That is, when one has reached a low enough level of abstraction in the specification, it is possible to take an actual program that purportedly implements the protocol, and show that it is correct with respect to the specification. This is no different from traditional program verification.

In order to gain greater confidence that specifications are suitable for their intended use, it is useful to prove properties of a single specification. For example, we might want to show that the sequence of messages delivered is equal to the sequence of messages sent. Liveness properties such as freedom from deadlock or eventual termination are also often proved for a single specification. We will discuss these issues at greater length in Section 3.6.

Thus we have three major types of protocol verification problems in each layer of a system:

1. Verification of the *protocol* against its *service*;

2. Verification of an *implementation* against the *protocol*; and

3. *Independent* verification of desired properties of the service, protocol, and program.

1.4. Related Work

To our knowledge, this work is the first combination of state transition machine, protocol, and axiomatic specification notions. However, a large body of work exists in each of these areas individually, and to a lesser extent for each pair.

A variety of methods have been used to specify communication protocols, including Petri nets (and related graph models), formal languages, sequencing expressions, I/O histories, and programming languages. However, the variations on state transition machine methods discussed in Section 1.1 seem to be most popular. Much of this work is either limited in expressive power (e.g., finite state automata) or lacking a solid theory and automated tools for verification. Sunshine [48] provides a survey and comparison of this work.

In the area of abstract data types, a large body of work also exists [14, 15, 10, 11, 28]. Usually state transition machine (or abstract machine) model approaches and axiomatic approaches are viewed as mutually exclusive alternatives [18, 4, 26]. A number of state transition machine models have been proposed [34, 39, 37, 4, 38, 27]. Several variations of axiomatic methods have also been developed [16, 25, 12]. The notion of specifying state transition machines axiomatically seems relatively unexplored, although Flon and Misra [7] hint at it.

We have drawn heavily on the following concepts:

- Hierarchical layering and cooperating remote stations within a layer from the protocol domain [47, 23];

- Verification of the properties of a specification [15, 18, 32, 38, 6, 19, 20, 35]; and

- Verification that a lower level system properly implements a higher level one [40, 37, 17, 13], or that the two systems are behaviorally equivalent [4, 45].

Of course, we have had to adapt these concepts to the new environment resulting from the merger of protocol, state transition machine, and axiomatic specification concerns.

2. AN OVERVIEW OF OUR METHOD OF PROTOCOL SPECIFICATION AND VERIFICATION

Our method of specifying and verifying protocols can be summarized as follows:

1. Produce a *service specification*. If a state transition machine description of the service already exists, translate it into an ***Affirm*** representation. Otherwise directly state the service specification as a state transition specification in ***Affirm***.

2. Validate that the service specification at least partially meets the requirements of the user (either the ultimate user or another layer). Typically this involves proving some invariant properties of the specification, e.g., what gets sent by the user at one station gets delivered to the user at the other station in the same order.

3. Produce the *protocol specification*. Again, if a state transition machine representation exists, simply translate it into an ***Affirm*** representation.

4. Verify that the protocol specification implements the service specification. This is a two-step process.

    a. First, define a correspondence (a *rep* function) between the state variables of the two specifications.

    b. Then show that the *axioms* of the service specification, when reformulated using the corresponding data structures of the protocol specification, are *theorems* provable from the axioms of the protocol specification.

    A further validation involves independently stating the service requirements in terms of the state variables of the protocol specification, and then proving that the protocol specification satisfies these requirements.

5. Specify an algorithm implementing the protocol specification.

6. Verify that the algorithm implements the protocol.

Chapters 3, 4, and 5 discuss these steps in some detail. Figure 2-1 displays the relationship of the elements involved in protocol specification and verification.

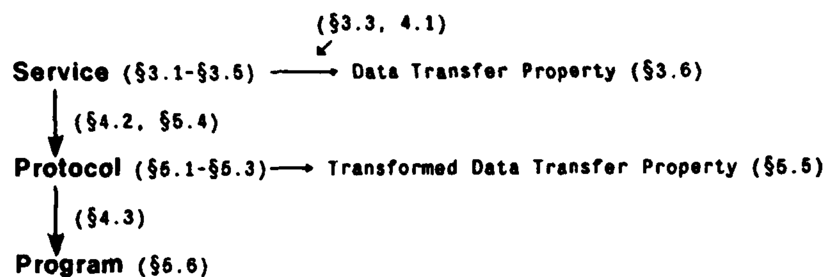


Figure 2-1: The steps in protocol verification

The references prefaced by "§" are pointers to relevant sections of this paper
Vertical lines mean *implemented by*;
Horizontal lines mean *invariant of*.

3. A SERVICE SPECIFICATION FOR A SIMPLE MESSAGE SYSTEM

Perhaps the simplest data transfer service provides for transmission of one message at a time from a fixed *sender* to a fixed *receiver*. The sender must wait until the previous message is received before sending the next one. There is no possibility of message loss, duplication, or corruption. The system is shown graphically in Figure 1-1. The next section provides an informal English description of the state transition machine. We will show how it can be represented in ***Affirm*** in the following sections.

3.1. State Variables

There are only a few state variables, each performing a simple function. (Each state variable has an associated data type, as shown.)

State: *ControlState*
: The current status of the service. This state variable simply cycles through the four values of the enumerated type *ControlState*. The four values of the type are *ReadyToSend*, *Sending*, *ReadyToReceive*, and *Acking* (Acknowledging). The state variable *State* is tested by most state transition functions as a general applicability test: the transition function will not change the state unless this variable has the appropriate value.

Sent: *QueueOfMessage*
: The queue of messages that have been sent to the receiver. One of the properties to prove about this service is that the queue of messages sent equals the queue of messages received (except for possibly the very last message of the *Sent* queue, which may not have been received yet).

Received: *QueueOfMessage*
: The queue of messages that have been received by the receiver.

Buffer: *QueueOfMessage*
: The queue of messages that have been sent by the sender but not yet received by the receiver. This state variable represents the *channel* of a real protocol. In the current protocol, this queue is either empty, or has exactly one message in it, the one just sent (but of course we have to prove it, not just say it!).

The types of the state variables are assumed to be explicitly defined (e.g., type *ControlState*), or are assumed to have a standard definition (as is the case with type *QueueOfMessage*).

3.2. State Transitions

A few of the state transition functions would be requested by a user, while others would appear to the user to occur spontaneously. For example, the user would explicitly request the *UserSend* operation, but the *SendComplete* operation, corresponding to the event "message pops out of the

channel at the receiver's end," would appear to be spontaneous to the user. These spontaneous transitions are included to explicitly model the delay involved in sending a message. We consider this to be an important aspect of the service.

InitializeService
Initializes the state variables. *Sent*, *Received*, and *Buffer* are all initialized to the empty queue, and *State* is initialized to *ReadyToSend*.

UserSend(message)
Only applicable if *State* is *ReadyToSend*; otherwise, this operation is a no-op. Adds message to the *Sent* queue, adds message to *Buffer*, and sets *State* to *Sending*.

SendComplete
A spontaneous event (the user cannot directly request it). Applicable only if *State* is *Sending*, i.e., there is an outstanding *Send* operation to be completed. Sets *State* to *ReadyToReceive*.

UserReceive
Applicable only if *State* is *ReadyToReceive*. The message at the front of the *Buffer* queue is added to *Received*, indicating passage of the message to the user. *State* is then updated to *Acking*--an abstraction of the process of sending an acknowledgment to the sender, telling of the receipt of the message.

ReceiveComplete
A spontaneous event, corresponding to the event "sender receives acknowledgment of message receipt." Applicable only if *State* is *Acking*. A message is removed from *Buffer*, and *State* is updated to *ReadyToSend*, indicating the cycle is complete.

3.3. Behavior of the Simple Message System

The state machine starts by performing the *InitializeService* command. The system then repeatedly cycles through the four states *ReadyToSend*, *Sending*, *ReadyToReceive*, and *Acking*. Each of these four states has only two successor states: itself (when a command that is not applicable is issued, in which case there's no change), and the next in the cycle. (Of course, at any time the *InitializeService* command can be re-issued, in which case the machine is reset to its initial state.)

As the system cycles through the four states, it maintains an invariant: the sequence of messages sent equals the concatenation of the sequence of messages received and the single message currently being sent (if there is one).[2] This and similar properties are called *service requirements*. If the state transition machine is specified correctly, these properties are straightforward to verify.

[2] Almost. We will discuss the correct formulation of this property later.

3.4. Converting State Transition Specifications to Affirm

The ***Affirm*** representation of a state transition machine is basically just a representation of the *state vector* of the state machine. Each state variable forming one part of the machine's state vector becomes a selector function. Each state transition function (command) becomes a constructor. There are usually no extender functions in this scheme. The axioms simply state how each state variable is modified by each state transition function.

3.4.1. State Transition Function → Constructor

Each state transition function (command) of the state transition machine becomes a constructor of an ***Affirm*** type.

```
state machine SimpleMessageSystem;

declare s: SimpleMessageSystem;
declare m: Message;

constructors
    InitializeService, UserSend(s, m), SendComplete(s), UserReceive(s), ReceiveComplete(s): SimpleMessageSystem;
```

Each constructor has as its range the type being defined. And each of the constructors (except the initialization function) is given a parameter of the type being defined. This parameter represents the entire state of the system. Thus state or event histories can easily be represented as compositions of the *constructor functions*. For example, the sequence of commands representing a machine cycle

InitializeService; UserSend(m); SendComplete; UserReceive; ReceiveComplete

would simply be

ReceiveComplete(UserReceive(SendComplete(UserSend(InitializeService, m))))

3.4.2. State Variable → Selector

Each state variable of the state transition machine becomes a selector function in the ***Affirm*** specification. In the ***Affirm*** specification, each function will take a parameter of the type being defined. Thus each state variable is simply an extraction function of the state vector.

```
selector State(s): ControlState;

selectors
    Buffer(s), Sent(s), Received(s): QueueOfMessage;
```

3.4.3. Transition Definition → Set of Axioms

The preceding subsections paved the way by defining the domain and range information of the constructors and selectors. Now we must define their semantics. It will become quite clear why each function carries along the "state" parameter: it provides a natural way of describing a transition. We

will demonstrate the method by writing the axioms for the state variable *Sent*. From Section 3.2, we know that the state variable *Sent* is modified by the *InitializeService* operation, possibly modified by the *UserSend* operation, and not modified by the remaining operations *SendComplete*, *UserReceive*, and *ReceiveComplete*.

```
axioms
   1. Sent(UserSend(state, message))
      = = if State(state) = ReadyToSend
            then Sent(state) Add m
            else Sent(state),
   2. Sent(SendComplete(state)) = = Sent(state),
   3. Sent(UserReceive(state)) = = Sent(state),
   4. Sent(ReceiveComplete(state)) = = Sent(state),
   5. Sent(InitializeService) = = NewQueueOfMessage;
```

Axioms 2, 3, and 4 simply state that the operations have no effect on the state variable. For example, axiom 2 says "the value of the state variable *Sent* after a state transition from state *state* to state *SendComplete(state)* is equal to the value of *Sent* in state *state*." Similarly, axiom 1 says "if the state variable *State* in state *state* is *ReadyToSend*, then the operation *UserSend* will have an effect on the state variable *Sent*; otherwise it won't." This method of constructing a specification ensures that the specification will be complete--the effects of each command on each state variable are detailed.

3.5. The Affirm Representation

The following is a stylized representation of ***Affirm*** input, for the sake of readability. State transition functions that leave a state variable unchanged are not explicitly specified; the convention is "not specified, not modified." The actual ***Affirm*** input is displayed in Appendix I.

```
state machine SimpleMessageSystem;

declare s: SimpleMessageSystem;
declare m: Message;

constructors InitializeService, UserSend(s, m), SendComplete(s), UserReceive(s), ReceiveComplete(s);
selectors Buffer(s), Sent(s), Received(s): QueueOfMessage;
selector State(s): ControlState;

axioms {InitializeService}
   State(InitializeService) = = ReadyToSend,
   Buffer(InitializeService) = = NewQueueOfMessage,
   Sent(InitializeService) = = NewQueueOfMessage,
   Received(InitializeService) = = NewQueueOfMessage;
```

```
axioms {UserSend}
    State(UserSend(s, m)) = = if State(s) = ReadyToSend
                              then Sending
                              else State(s),
    Buffer(UserSend(s, m)) = = if State(s) = ReadyToSend
                               then Buffer(s) Add m
                               else Buffer(s),
    Sent(UserSend(s, m)) = = if State(s) = ReadyToSend
                             then Sent(s) Add m
                             else Sent(s);
axioms {SendComplete}
    State(SendComplete(s)) = = if State(s) = Sending
                               then ReadyToReceive
                               else State(s);
axioms {UserReceive}
    State(UserReceive(s)) = = if State(s) = ReadyToReceive
                              then Acking
                              else State(s),
    Received(UserReceive(s)) = = if State(s) = ReadyToReceive
                                 then Received(s) Add Front(Buffer(s))
                                 else Received(s);
axioms {ReceiveComplete}
    State(ReceiveComplete(s)) = = if State(s) = Acking
                                  then ReadyToSend
                                  else State(s),
    Buffer(ReceiveComplete(s)) = = if State(s) = Acking
                                   then Remove(Buffer(s))
                                   else Buffer(s);
end {SimpleMessageSystem} ;
```

3.6. Properties of a Specification

To increase our confidence that the state transition machine we have specified is a reasonable one, we can formulate certain properties we expect to hold during the machine's operation. These service requirements may be proved using structural induction as described in Section 1.2.2. We present an example of such *service requirements* for the simple data transfer service.

A useful safety property for this service might be:

Sent = *Received* join *Transit*

stating that the messages received are equal to the messages sent except for any still in transit. We must be careful in our definition of *Transit* to take into account the state *Acking* when the message is still in *Buffer*, but has been received. The exact theorem in ***Affirm*** would be:

```
theorem DataTransferService, all s (Sent(s) = Received(s) join Transits(s));

define Transit(s) = = if State(s) = Acking
                      then NewQueueOfMessage
                      else Buffer(s);
```

This theorem has been proved in ***Affirm***.

Another form of the service requirement might be

$$(State(s) = ReadyToSend) \supset (Sent(s) = Received(s))$$

stating that input exactly equals output whenever the system returns to its "idle" state. This turns out to be a special case of the more general theorem above.

Liveness properties for this simple machine are relatively trivial. It is fairly obvious that the allowed progression of states involves a single fixed cycle (ignoring rejected operations having no effects), where a single message is transferred during each cycle. First, the meaning of "ignore rejected operations" is formalized, as follows:

```
interface StripNoOps(s): SimpleMessageSystem;

axioms
    StripNoOps(InitializeService) = = InitializeService,
    StripNoOps(UserSend(s, m)) = = if State(s) = ReadyToSend
                                   then UserSend(StripNoOps(s), m)
                                   else StripNoOps(s),
    StripNoOps(SendComplete(s)) = = if State(s) = Sending
                                    then SendComplete(StripNoOps(s))
                                    else StripNoOps(s),
    StripNoOps(UserReceive(s)) = = if State(s) = ReadyToReceive
                                   then UserReceive(StripNoOps(s))
                                   else StripNoOps(s),
    StripNoOps(ReceiveComplete(s)) = = if State(s) = Acking
                                       then ReceiveComplete(StripNoOps(s))
                                       else StripNoOps(s);

theorem StatesMatch, all s (    State(s) = State(StripNoOps(s))
                            and Sent(s) = Sent(StripNoOps(s))
                            and Received(s) = Received(StripNoOps(s))
                            and Buffer(s) = Buffer(StripNoOps(s)));
```

The definition of *StripNoOps* simply formalizes our intuition about events having no effect because they occur at an inappropriate time. For example, a *SendComplete* event after a *UserReceive* event can have no effect. The theorem *StatesMatch* says that the effects of a sequence of events are the same as the effects of a new sequence that has had the no-effect operations filtered out. This theorem was proved in ***Affirm***.

In the context of the above definitions, then, the following theorem says that the four operations, in the right order, add a message (and the correct one) to those received, no matter how many additional "rejected" operations may have been interleaved.

```
theorem ServiceProgress, all s1, s2, m
 (       StripNoOps(s2) = ReceiveComplete(UserReceive(SendComplete(UserSend(StripNoOps(s1), m))))
    and State(s1) = ReadyToSend
  imp    State(s2) = ReadyToSend
    and Sent(s2) = Sent(s1) Add m
    and Received(s2) = Received(s1) Add m);
```

This theorem has also been proved using ***Affirm***.

Finally we note that the system will progress around this cycle as long as each operation completes in finite time. This is an assumption at the service level, but of course it must be proved when we see how the protocol implements each operation.

3.7. Alternative Notations

Instead of *implicitly* representing the machine's state vector, we could have represented it *explicitly* by defining one constructor, say *Const*. *Const* takes a number of parameters (one per individual state variable), and creates one state vector out of them:

```
constructor Const(state, sent, received, buffer): SimpleMessageSystem;
```

The individual state variables are then defined as vector-extractors:

```
State(Const(state, sent, received, buffer)) == state,
Sent(Const(state, sent, received, buffer)) == sent,
Received(Const(state, sent, received, buffer)) == received,
Buffer(Const(state, sent, received, buffer)) == buffer;
```

and the state transition functions, nominally constructors, would become extenders:

```
UserSend(Const(state, sent, received, buffer), message)
  == if state = ReadyToSend
        then Const(Sending, sent Add message, received, buffer Add message)
        else {no change} Const(state, sent, received, buffer),

SendComplete(Const(state, sent, received, buffer))
  == if state = Sending
        then Const(ReadyToReceive, sent, received, buffer)
        else {no change} Const(state, sent, received, buffer),

UserReceive(Const(state, sent, received, buffer))
  == if state = ReadyToReceive
        then Const(Acking, sent, received Add Front(buffer), buffer)
        else {no change} Const(state, sent, received, buffer),

ReceiveComplete(Const(state, sent, received, buffer))
  == if state = Acking
        then Const(ReadyToSend, sent, received, Remove(buffer))
        else {no change} Const(state, sent, received, buffer),

InitializeService == Const(ReadyToSend, NewQueueOfMessage, NewQueueOfMessage, NewQueueOfMessage);
```

This notation often results in fewer axioms overall, but each axiom is usually much more complex than those of the notation we described above. This is especially true when one state has a large set of successor states. We have chosen the first notational method for expressing state vectors in ***Affirm*** because of its convenience. The axioms, with a bit of practice, are generally more understandable because each is relatively simple.

4. VERIFICATION ISSUES

As mentioned in Chapter 1, we would ideally like to verify three kinds of properties of a specification: *safety* (only correct things happen), *liveness* (eventually something happens), and *performance* (things happen promptly).

Safety properties are typically proved by structural induction, as was described in Section 1.2.2. Most of our work has focused on this concern.

Liveness properties may be handled by showing that the system terminates:

1. Some operation is always enabled, or the system has reached one of its final states; and
2. Each operation decreases some bounded measure function, which at some point (nominally, when it evaluates to zero) disables all operations (for example, by setting a special state variable to **false**; presumably all the operations are applicable only if the variable is **true**).

This issue is discussed at length in [2]. Temporal logic also provides convenient techniques for stating and proving liveness properties [20, 33]. We deal only briefly with liveness properties in this report.

Performance properties have traditionally been dealt with by other methods (e.g., queueing theory); we have not addressed this issue.

4.1. Verifying Properties of a Specification

As noted in Section 1.2, one of the main capabilities of ***Affirm*** is the ability to verify that a data type has certain desired properties. These properties are specified as theorems and are then proved using the interactive theorem prover of ***Affirm***.

Typically these theorems are invariants in the state transition model. That is, they are predicates on the state that are **true** in the initial state, and are preserved across all state transitions. In ***Affirm***, these theorems are proved from the axioms of the type being specified (and other predefined types) by structural induction. In the context of the simple message system of the preceding chapter, to prove a theorem P(*s*) for all states *s*, first prove the theorem P(*InitializeService*); then, assuming P(*s*) for some state *s*, prove P(*fcn*(*s*)) for each constructor *fcn* in the type. This suffices to show P(*s*) for all *s*.

It is also overkill. What is proved is that *any* order of occurrence of the events of the state transition machine is acceptable; the invariant still holds. Carrying out such a proof requires a ruggedized

machine that has extra tests to ensure that operations invoked at inappropriate times can do no harm: no state change occurs. Real protocols have (implicit) assumptions stating which operations can happen when. It is unlikely, for example, that a time-out can occur if there are no messages that have been sent but not yet acknowledged. Thus proving properties of a program that uses an abstract machine in a certain way may be easier (and allow a simpler machine specification) than proving properties of the machine for arbitrary programs.

4.2. Verifying the Protocol against the Service Specification

We must show that the detailed system (composed of stations interacting according to the protocol) does the same thing as the abstract system (specified by the service: see Section 1.3).

This brings us to the problem of what it means for one abstract machine (or set of machines) to *implement* another. There are two aspects of this relationship:

1. a static correspondence between each state of the higher level and the state(s) implementing it at the lower level, showing that every higher level state is in fact implemented; and

2. a dynamic correspondence between the transitions of the two levels, showing that the sequence of states reachable in the two levels are the same.

Point 1 is typically handled by giving a representation function *rep* from the state variables of the lower level to the state variables of the higher level. The function is specifically defined in this direction because there may be several lower level states that all represent the same higher level state (so the function has no inverse). Also, some lower level states may be intermediate states that do not represent any higher level state. As noted above, it must be shown that there is some lower level state to represent every higher level state.

To address point 2, the conventional approach involves specifying a fixed sequence of lower level operations implementing each higher level operation. Then it must be proved that if the two systems start in corresponding states, they will end up in corresponding states after corresponding operations.

Let S and s be higher and lower level states respectively. Let OP be a higher level operation and op be its lower level implementation, and let rep be a representation function (from s to S). Then this method attempts to show that for each OP

$$\forall S, s\ (S = rep(s) \supset OP(S) = rep(op(s)))$$

The difficulty of this approach in the protocol domain is that a higher level operation such as

sending a message may be accomplished by a *nondeterministic sequence* of lower level operations, including transmission, loss, time-outs, retransmissions, and receptions. Typically there will be a single low-level operation that starts the accomplishment of the higher level operation by "posting" some work to be done. This will then be followed by a nondeterministic series of lower level operations, invisible at the top level, that complete the results of the higher level operation in the unreliable low-level environment. These latter effects may be viewed as one or more spontaneous transitions of the higher level machine. Section 5 gives an example of this sort.

In this type of lower level specification, there are two sorts of operations: one set invoked directly by the users of the system (corresponding to the higher level operations), and a second set of internal operations.

Verification of this type of lower level specification is similar to the conventional situation discussed above, but must be augmented by a proof that the spontaneous higher level transitions (and *only* such transitions) are accomplished by the internal operations of the lower level. This additional proof is facilitated by defining the internal operations in a ruggedized fashion that includes tests in their definitions to force them to produce no changes if invoked at inappropriate times. The additional theorems to be proved take the following form: From any low-level state corresponding to a higher level state with spontaneous transitions, the next lower level state that "maps up" and can be reached by any sequence of internal lower level operations must correspond to the correct higher level state. We can define this recursively as follows.

$\forall$ *S* such that *S* has one or more spontaneous transitions
  ($\forall$ *s* such that *S* = *rep*(*s*)
    (SpontSucc(*S*) = *rep*(UpSuccessors(*s*, *S*))))

where *rep* is extended in the natural manner to sets
  SpontSucc(*S*) is the set of states reached from *S* by spontaneous transitions
  UpSuccessors(*s*, *S*) =
    {*s2*: Successor(*s*, *s2*) and MapsUp(*s2*) and *S* $\neq$ *rep*(*s2*)}
    $\bigcup$ UpSuccessors(*s3*, *S*)
    $\forall$*s3*: Successor(*s*, *s3*) and ~MapsUp(*s3*)
  Successor(*s1*, *s2*) = $\exists$ *internalOp* such that (*s2* = *internalOp*(*s1*))
  MapsUp(*s*) = **true** if *s* represents some high-level state

This general formulation often simplifies considerably, as shown in the example in Chapter 5.

4.3. Verifying a Program against the Protocol Specification

If we followed the pattern of the lower level (protocol) and higher level (service) specifications discussed above, each operation of the protocol specification would be implemented by a separate Pascal procedure. However, an actual implementation of a protocol is somewhat more constrained.

A state transition machine defines a global state and specifies how transitions change the state variables. Since the purpose of protocols is to provide for communication between disjoint processes, an actual implementation will be divided into cooperating stations (as described in Section 1.3); only the state variables describing the communications medium will be shared between stations.

Since losses are a spontaneous behavior of the medium, they are not implemented.

While it was convenient for our specification to allow operations to be invoked in any order, only certain sequences of operations are efficient. (For example, it makes little sense for the sender to retransmit without first checking for acknowledgments.) Therefore the programs typically exhibit only a subset of the allowable behavior. The intention is that only inefficient event sequences have been omitted.

Of course, many properties of states proved at higher levels may be transferred down to programs. However, the constraints introduced by the program may require additional proofs for liveness, e.g., the constraints do not introduce deadlock.

5. DETAILED EXAMPLE: THE ALTERNATING BIT PROTOCOL

We will continue the exposition of our methodology, using the Alternating Bit protocol as an example. First we will specify a protocol providing the simple data transfer service described earlier. We will then perform the various verification tasks.

5.1. A Brief Description of the Protocol

The Alternating Bit protocol [1, 5, 21, 20, 6] is intended to provide a simple but reliable message transfer service over an unreliable transmission medium. It attaches a one-bit sequence number to each message sent, and waits for an acknowledgment of the receipt of the message by the destination. The sequence number is complemented on each new message sent--hence the name of the protocol. If the acknowledgment is not received within a time-out period, the message is retransmitted (with the sequence number unchanged). The protocol guarantees correctly sequenced delivery of messages even if the medium loses messages and acknowledgments, but the medium cannot reorder messages.

To accomplish these functions, the sender and receiver stations maintain local sequence number counters. The sender uses its counter to remember the sequence number to attach to the next transmission. The receiver uses its counter to remember the sequence number of the next message it *expects* to receive, thus allowing for the removal of duplicate messages (which will be sent if an acknowledgment is lost).

The Alternating Bit protocol is a simple instance of a general class of data transfer protocols using *positive acknowledgments* and *retransmission on errors* [46, 44, 24]. This simple example allows only one unacknowledged message to be transmitted at a time. More complex protocols in this class use larger sequence numbers and allow multiple outstanding messages.

In Section 5.2 we provide an informal definition of a state transition machine for the Alternating Bit protocol, and in Section 5.3 this specification is translated into an ***Affirm*** representation. We then discuss the major verification step, showing that the protocol implements its service correctly. We will then discuss an important invariant of the protocol specification (independent of the service). Finally we give algorithms for the sender and receiver stations, and show that these algorithms properly implement the protocol.

5.2. A State Transition Machine for the Alternating Bit Protocol

The protocol machine described in this section closely parallels the service machine described in Chapter 3, with the addition of details concerning the internal operation of the protocol. The protocol is defined as a single machine rather than as separate sender and receiver components (see Section 7.1). Figure 5-1 illustrates the main data structures and operations of the protocol.

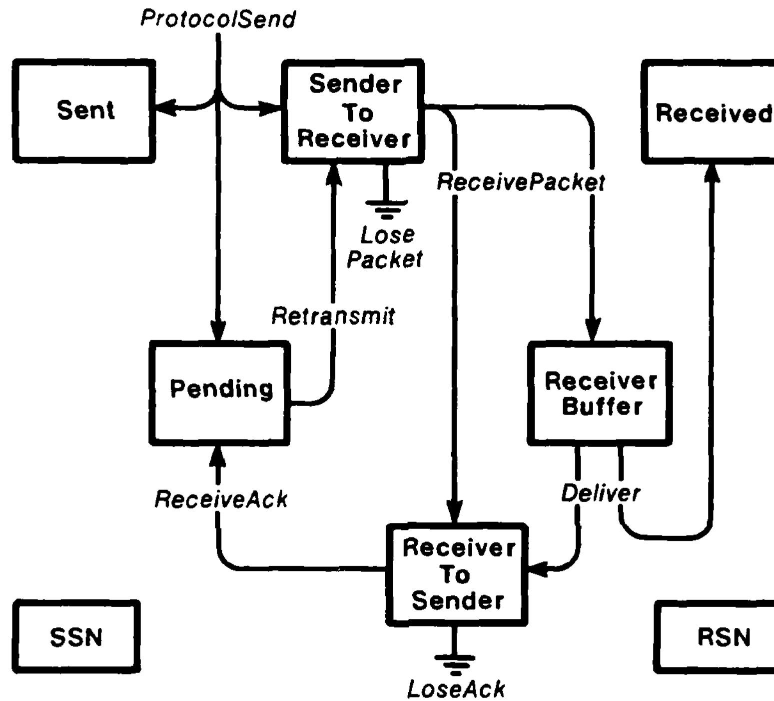


Figure 5-1: The protocol state transition machine

5.2.1. Data Types Used in the Specification

The protocol uses a few more data types than the service specification does. Their informal descriptions are gathered here for convenience.

Message
: As in the service specification, this type is a minimally defined data type that represents abstract contents.

Bit
: An enumerated type with two elements, arbitrarily called on and off. Functions include a "flip" operation that flips the value (from on to off or vice versa), represented by the unary not operator "~".

Packet

A record (or tuple) with two components: a value of type *Bit* (i.e., a sequence number) and a value of type *Message*.

Medium

Really a *QueueOfPacket* with the addition of operations to "lose" packets. Further enhancements (e.g., to allow the reordering of packets) might be desired in a more realistic medium. The *channels* of the protocol are of this type. The *Transmit* operation takes a value of type *Medium* and a value of type *Packet* and yields a value of type *Medium*. It thus corresponds to the *Add* operation of the *Queue* type. Similarly, *Receive* corresponds to the *Queue* operation *Remove*.

QueueOfPacket, *QueueOfMessage*, *SequenceOfMessage*

Standard data types from the ***Affirm*** Type Library.

5.2.2. State Variables

SenderToReceiver: *Medium*

The channel from the sender to the receiver.

ReceiverToSender: *Medium*

The channel from the receiver to the sender. For convenience, entire packets are returned as acknowledgments, rather than just the sequence numbers.

Pending: *QueueOfPacket*

The packet currently being transmitted, if any. *Pending* is either empty (i.e., *NewQueueOfPacket*), or contains exactly one packet. A *queue* type was used instead of a simple packet in order to avoid notions of a null packet, and to allow future extensions.

SSN: *Bit*

The sender's current sequence number (i.e., the next acknowledgment of interest).

RSN: *Bit*

The receiver's current sequence number (i.e., the number of the next packet expected).

ReceiverBuffer: *QueueOfPacket*

The packet received but not yet delivered to the user (if any). *ReceiverBuffer* is either empty, or has exactly one element. A *queue* type was used for convenience.

Sent: *SequenceOfMessage*

A sequence of all the messages sent but not necessarily acknowledged yet. (This variable would not be present in a real implementation; it is for specification purposes.)

Received: *SequenceOfMessage*

A sequence of all the messages successfully received. (This variable would not be present in a real implementation; it is for specification purposes.)

Of course, not all these data structures are visible or available to both stations (sender and receiver).

5.2.3. State Transition Functions

InitializeProtocol
Set the counters and the queues to their initial values.

ProtocolSend(m)
Given a message m, try to send the message as a packet. If no message is waiting to be acknowledged (*Pending* = *NewQueueOfPacket*) then accept the message m (by appending it to *Sent*) and transmit it (by constructing a packet with the current *SSN* and adding the packet to *SenderToReceiver*). Also remember that the packet is waiting to be acknowledged (by putting it in *Pending*).

ReceivePacket
Receive a packet, if one is available. If *SenderToReceiver* is nonempty, remove and examine the first packet. If it is the one expected (its sequence number matches *RSN*), then place it in *ReceiverBuffer* and flip *RSN*. If the packet has already been delivered, then send an acknowledgment by copying the packet to *ReceiverToSender*.

Deliver
Deliver a new message (if there is one to be delivered) to the user. If a message is available in *ReceiverBuffer*, append it to the *Received* queue, and acknowledge the message (by copying it to *ReceiverToSender*). Clear *ReceiverBuffer*.

ReceiveAck
Receive an acknowledgment, if any exists to be received. If *ReceiverToSender* is not empty, then remove the first packet. If the packet's sequence number doesn't match *SSN*, then ignore the packet. Otherwise, flip *SSN* and empty *Pending* (preparing for another *Send* operation).

Retransmit
Add the message in *Pending*, if any, to *SenderToReceiver*, i.e., re-send it.

LosePacket
Lose a packet by removing the front packet from *SenderToReceiver*, if it is not empty.

LoseAck
Lose an acknowledgment by removing the front of *ReceiverToSender*, if it is not empty.

As an example, a typical state of the system might be

ReceiveAck(Deliver(ReceivePacket(ProtocolSend(InitializeProtocol, m))))

This represents the sequence of operations (reversed from their functional representation)

InitializeProtocol; ProtocolSend(m); ReceivePacket; Deliver; ReceiveAck

5.3. The Affirm Representation

As was the case with the service specification, we simply turn state variables into selector functions of a data type; state transition functions (commands) become constructors. The definitions of the state transition functions become axioms. All the functions in the ***Affirm*** representation carry along an explicit parameter of the type being defined; it is a characterization of the current state.

What is displayed here is a stylized version of the axioms, omitting all axioms stating that some selector is not modified by some constructor. Appendix II contains the actual ***Affirm*** input.

```
state machine ABProtocol;

declare s: ABProtocol;
declare m: Message;

constructors
  InitializeProtocol, ProtocolSend(s,m), ReceivePacket(s), Deliver(s), ReceiveAck(s), Retransmit(s), LoseAck(s), LosePacket(s);
selectors InitialSequenceNumber, RSN(s), SSN(s): Bit;
selectors ReceiverToSender(s), SenderToReceiver(s): Medium;
selectors Received(s), Sent(s): QueueOfMessage;
selectors Pending(s), ReceiverBuffer(s): QueueOfPacket;

axioms {InitializeProtocol:}
    Pending(InitializeProtocol) == NewQueueOfPacket,
    Received(InitializeProtocol) == NewQueueOfMessage,
    ReceiverBuffer(InitializeProtocol) == NewQueueOfPacket,
    ReceiverToSender(InitializeProtocol) == InitializeMedium,
    RSN(InitializeProtocol) == InitialSequenceNumber,
    SenderToReceiver(InitializeProtocol) == InitializeMedium,
    Sent(InitializeProtocol) == NewQueueOfMessage,
    SSN(InitializeProtocol) == InitialSequenceNumber;

axioms {ProtocolSend:}
    Pending(ProtocolSend(s, m)) == if Pending(s) = NewQueueOfPacket
                                   then NewQueueOfPacket Add MakePacket(m, SSN(s))
                                   else Pending(s),
    SenderToReceiver(ProtocolSend(s, m)) == if Pending(s) = NewQueueOfPacket
                                            then Transmit(SenderToReceiver(s), MakePacket(m, SSN(s)))
                                            else SenderToReceiver(s),
    Sent(ProtocolSend(s, m)) == if Pending(s) = NewQueueOfPacket
                                then Sent(s) Add m
                                else Sent(s);
axioms {ReceivePacket:}
    ReceiverBuffer(ReceivePacket(s)) == if    Seq(Front(SenderToReceiver(s))) = RSN(s)
                                          and SenderToReceiver(s) ~= InitializeMedium
                                        then NewQueueOfPacket Add Front(SenderToReceiver(s))
                                        else ReceiverBuffer(s),
    ReceiverToSender(ReceivePacket(s)) == if    SenderToReceiver(s) ~= InitializeMedium
                                            and ReceiverBuffer(s) = NewQueueOfPacket
                                            and RSN(s) ~= Seq(Front(SenderToReceiver(s)))
                                          then Transmit(ReceiverToSender(s), Front(SenderToReceiver(s)))
                                          else ReceiverToSender(s),
    RSN(ReceivePacket(s)) == if Seq(Front(SenderToReceiver(s))) = RSN(s) and SenderToReceiver(s) ~= InitializeMedium
                             then ~RSN(s)
                             else RSN(s),
    SenderToReceiver(ReceivePacket(s)) == Receive(SenderToReceiver(s));
```

```
axioms {Deliver:}
   Received(Deliver(s)) = = if ReceiverBuffer(s) = NewQueueOfPacket
                            then Received(s)
                            else Received(s) Add Text(Front(ReceiverBuffer(s))),
   ReceiverBuffer(Deliver(s)) = = NewQueueOfPacket,
   ReceiverToSender(Deliver(s)) = = if ReceiverBuffer(s) = NewQueueOfPacket
                                    then ReceiverToSender(s)
                                    else Transmit(ReceiverToSender(s), Front(ReceiverBuffer(s)));
axioms {ReceiveAck:}
   Pending(ReceiveAck(s)) = = if Seq(Front(ReceiverToSender(s))) = SSN(s) and ReceiverToSender(s) ~= InitializeMedium
                              then NewQueueOfPacket
                              else Pending(s),
   ReceiverToSender(ReceiveAck(s)) = = Receive(ReceiverToSender(s)),
   SSN(ReceiveAck(s)) = = if Seq(Front(ReceiverToSender(s))) = SSN(s) and ReceiverToSender(s) ~= InitializeMedium
                          then ~SSN(s)
                          else SSN(s);
axiom {Retransmit:}
   SenderToReceiver(Retransmit(s)) = = if Pending(s) = NewQueueOfPacket
                                       then SenderToReceiver(s)
                                       else Transmit(SenderToReceiver(s), Front(Pending(s)));
axiom {LoseAck:}
   ReceiverToSender(LoseAck(s)) = = Receive(ReceiverToSender(s));

axiom {LosePacket:}
   SenderToReceiver(LosePacket(s)) = = Receive(SenderToReceiver(s));

end {ABProtocol};
```

5.4. Verifying the Protocol against the Service Specification

This section presents a detailed example of how to verify that a lower level state transition machine specification implements a higher level one. In this case the system in question is the Alternating Bit protocol, and the two levels are the service (higher) and protocol (lower) specifications.

5.4.1. Safety

The service specification (see Section 3.5) includes *UserSend* and *UserReceive* operations, and an *InitializeService* operation to initialize the system, all meant to be invoked by the users of the service. It also includes spontaneous transitions *SendComplete* and *ReceiveComplete*, modeling the completion of the *UserSend* and *UserReceive* operations within the distributed system providing the service. Hence there are four control states at the service level, as shown in Figure 1-1, with the two intermediate states explicitly displaying the delay between one user initiating an operation and the other user becoming aware of it. The state variables used at this level include a buffer *Buffer* for messages sent but not yet received (at most one is allowed), and queues *Sent* and *Received* that maintain histories of all messages sent and received (these are only used for specification purposes). There is also a control state variable *State* with four possible values.

The protocol level (see Section 5.3) has operations corresponding to each of the user operations at the service level:

InitializeService → *InitializeProtocol*
UserSend → *ProtocolSend*
UserReceive → *Deliver*

There is also a second set of protocol operations that collectively accomplish the spontaneous operations of the service level. These are *ReceivePacket*, *ReceiveAck*, *LosePacket*, *LoseAck*, and *Retransmit*. The service-level state variables *Sent* and *Received* are implemented transparently, while *Buffer* is implemented as the text of the first packet in the queue of packets called *Pending*. The service-level control states (*ReadyToSend*, *Sending*, *ReadyToReceive*, and *Acking*) correspond to four defined state classes at the protocol level (*S1*, *S2*, *S3*, and *S4*). Figure 5-2 summarizes these correspondences informally.

Service	Protocol
InitializeService	*InitializeProtocol*
Sent	*Sent*
Received	*Received*
Buffer	*Text(Front(Pending))*
State	
ReadyToSend	*S1*
Sending	*S2*
ReadyToReceive	*S3*
Acking	*S4*
UserSend	*ProtocolSend*
UserReceive	*Deliver*
SendComplete, *ReceiveComplete*	any sequence of the operations {*ReceivePacket*, *ReceiveAck*, *Retransmit*, *LosePacket*, *LoseAck*}

Figure 5-2: The correspondence between service and protocol-level state variables

Our method of proving that a protocol implements its service specification is to convert each of the service-level axioms into a theorem at the protocol level, and then to prove these theorems using the protocol specification. This follows the method of [17]. Appendix III.1 shows the formal

correspondence between functions at the two levels using a representation function *rep*, and Appendix III.2 defines the protocol-level state classes. The basic method is to replace each occurrence of the service machine state in the axioms of the service specification by the *rep* of its corresponding protocol states, and then to use the other rewrite rules displayed in Appendix III.1 until the expression is reduced to terms involving only protocol-level selectors and constructors.

This conversion is most conveniently discussed in three portions. The easiest axioms to convert are those defining the results of the user operations (*UserSend*, *UserReceive*, and *InitializeService*) on the state variables. Since each service-level operation is directly implemented by a single protocol-level operation, and the state variables also have a simple correspondence, the resulting theorems are easily obtained. Appendix III.3 shows how two service axioms are converted in detail, and Appendix III.4 gives all of the resulting theorems in this category.

The next group of theorems are those concerning the effects of the spontaneous service operations on the state variables. Here there is no fixed correspondence of one protocol operation for each service operation. Instead, we wish to show that any sequence of the five spontaneous protocol operations (*ReceivePacket*, *ReceiveAck*, *LosePacket*, *LoseAck*, and *Retransmit*) will have the specified effect. For state variables *Sent* and *Received* this is simple because the spontaneous operations are specified to have no effect on these variables. The first two theorems of Appendix III.5 state that each individual operation will have no effect, so we can also conclude that any sequence of these operations will have no effect. This may be viewed as a special case of structural induction, considering only the spontaneous operators, and attempting to show that *Sent* (or *Received*) is invariant.

The case for *Buffer* is more complex since there is a possible effect from the spontaneous operations. We must show that if the system is not in state *S4* (corresponding to *Acking* in the service specification), then there will be no effect, and if it is in state *S4*, then *Buffer* will become empty (i.e., *NewQueueOfMessage*). The first case is similar to the situation for *Sent* and *Received*, with the additional constraint that the system can never enter state *S4* from another state by spontaneous operations. The third theorem of Appendix III.5 shows that no single action can modify *Buffer* in this case, and therefore no sequence can, as above. For the *S4* case, the final theorem of Appendix III.5 states that the spontaneous operations either leave the system in state *S4* with *Buffer* unchanged, or set *Buffer* to *NewQueueOfMessage* and enter state *S1*. Once in state *S1*, we know from the previous theorem that there will be no further change to *Buffer*. We can then conclude that if the protocol progresses (to state *S1*), it behaves as specified in the service. This proves the safety of the protocol. A separate argument is necessary to prove liveness.

The final set of theorems, in Appendix III.6, covers the effects of the operations on the service-level state. For the user operations, we must show that the correct next state is generated by the corresponding protocol operation for each of the four states the system may be in. This is stated in the first and fourth group of theorems (the initial state was already covered). For the spontaneous operations, the situation is similar to *Buffer* above. We must show that any sequence of *ReceivePacket*, *ReceiveAck*, *LosePacket*, *LoseAck*, and *Retransmit* can cause only the transitions specified for *SendComplete* or *ReceiveComplete* (i.e., if the system progresses to a new state at all, it is the correct one). For the most part these theorems say that no state change takes place--only theorems *S1Succ1*, *S2Succ2*, and *S4Succ3* show actual progress (page 51). Once again, only safety is covered here.

All the theorems in Appendix III have been proved, showing that the protocol correctly implements the service. (The proofs of all theorems claimed to have been proved in this report are documented in [51].) The proofs require a number of definition invocations and substitutions that are tedious. They also require several lemmas concerning the relationship between *SSN* and the sequence numbers of the packets in the mediums. We cite just two as examples:

```
theorem PktsOldRP, all s, med (PktsOld(s, med) imp PktsOld(ReceivePacket(s), med));

theorem PktsOldPS,
 all s, m, med (     Pending(s) ~= NewQueueOfPacket
                 and PktsOld(s, med)
               imp PktsOld(ProtocolSend(s, m), med));
```

Theorem *PktsOldRP* says that if the packets in the medium *med* are old (i.e., with sequence number not equal to *SSN*(*s*)), then they are still old after a *ReceivePacket* event--the event's effects on the medium are limited to simply removing a packet. All the remaining packets are unaffected.

5.4.2. Liveness

In order to deal with liveness concerns, we must show that the implementation for each service-level operation terminates. This is trivial for the user operations, since each is directly implemented by a single protocol operation assumed to terminate. The difficulty comes with the so-called spontaneous operations. We must show that a finite sequence of internal protocol operations serves to accomplish the desired effect. Considering the *SendComplete* operation as an example, an argument of the following sort is necessary.

1. In (protocol) state *S2* (corresponding to service state *Sending*), the *Retransmit* operation is enabled and may place an arbitrary number of packets in the *SenderToReceiver* medium.

2. In state *S2*, if one of these packets reaches the receiver, the *ReceivePacket* operation will achieve the desired effects of *SendComplete* (i.e., change the state to *S3*, corresponding to *ReadyToReceive*).

3. If a large enough (but finite) number of packets are transmitted by the sender, one will reach the receiver.

These three points taken together imply that a finite number of protocol internal operations will accomplish the *SendComplete* service operation. A similar argument holds for the *ReceiveComplete* operation. Points 1 and 2 follow directly from the axioms for *Retransmit* and *ReceivePacket*. Point 3, however, requires an additional constraint on the simple medium: the number of packets that may be lost is bounded. As yet, there is no convenient method for expressing such *eventual delivery* constraints in ***Affirm***. Our liveness arguments must therefore remain informal. Berthomieu [2] and Hailpern [19, 20] deal with these concerns.

5.5. Protocol Properties and Invariants

As stated in Section 1.3.2, the essential verification of a protocol involves showing that it meets its service specification. However, it is also possible to prove properties of the protocol specification itself, independently of any service specification. In particular, a state invariant similar to the service requirements discussed in Section 3.6 is worth some discussion. Proving the invariant gives added confidence that the protocol specification is correct. The system invariant for the Alternating Bit protocol states that the protocol-level system is always in one of its four valid state classes:

theorem *MainSystemInvariant*, all s (InS1(s) or InS2(s) or InS3(s) or InS4(s));

We also note that by the definition of protocol state *S1* (in Section III.2),

InS1(s) ⊃ (Sent(s) = Received(s))

This is a protocol-level version of the service requirement.

The system invariant has been proved. The proof makes use of the theorems of Appendix III.6. Those theorems essentially detail how the state changes for each possible event. Most say that no change occurs. As with most abstract data types, much of the difficulty with this proof lies in developing a suitable invariant. We experimented with several versions of the protocol axioms and state class definitions before developing the present form.

5.6. Implementation

Having specified the Alternating Bit Protocol and proven that it has some desired properties, we must provide an implementation that meets these specifications. (See Section 4.3 for a general discussion.) Our implementation (in Appendix IV) has two stations:

- **Sender** contains procedures *ProtocolSend*, *SenderTimeout*, and *InitSender*; and

- **Receiver** contains *ReceivePacket*, *Deliver*, and *InitReceiver*.

They share the Medium variables *SenderToReceiver* and *ReceiverToSender*. Since both stations have local variables, we need two initialization routines. All other procedures correspond to the similarly-named events in the protocol specification, except for *SenderTimeout*. It combines the *Retransmit* and *ReceiveAck* events. Like the events of the specification, all procedures have no effect on the system if they are called at an inappropriate time.

Program variables correspond to state variables of the specification. Each procedure has an assertion of the form

VariablesMatch(*s*, ..vars..) imp VariablesMatch(event(*s*), ..new vars..)

In other words, for any state *s* that corresponds to the initial values of the program variables, the state resulting from the listed event will correspond to the variables after the routine finishes. (See Section IV.4.) For example, the assertion *DPost* (page 55) says "given any state *s* whose selectors *Sent*, *ReceiverBuffer*, and *ReceiverToSender* match the corresponding receiver variables, the new state resulting from a *Deliver*(*s*) event will have selectors that correspond to the values of the variables after *Deliver* is executed." *PSPost* (the assertion for *ProtocolSend*) adds one more stipulation: *ProtocolSend* sets a bit to inform its caller whether it had any effect.

The partial correctness of all these procedures has been proven using ***Affirm***. The proofs were quite straightforward, using only one lemma about the data type definitions and one lemma about the protocol specification. Theorem *PendingInvariant* states that *Pending* contains no more than one packet. It was easily proven from the axioms without reference to any other protocol invariants.

```
theorem SeqMatch(med, bit) imp (~Seqmatch(med, ~bit)) and med ~= NewQueueOfPacket;

theorem PendingInvariant, Remove(Pending(s)) = NewQueueOfPacket;
```

Since the implementation is in keeping with the specification, its safety follows from the earlier proof (in Section 5.4). Liveness has not been formally proven for either level. Any liveness proof must consider that the implementation does not exercise the full range of event sequences possible under the specification. (For example, *Retransmit* is always preceded by *ReceiveAck*.) Informally, it may be seen that only ineffective sequences have been excluded, so progress will not be impeded.

6. FURTHER APPLICATIONS

This chapter briefly mentions some further work we have accomplished in applying our methodology to several more complex protocols.

6.1. Stenning's Data Transfer Protocol

The protocol described in [44] ignores the aspects involved in connection establishment, and instead emphasizes the data transfer aspects. It is designed to operate correctly even though the channel may lose, duplicate, or re-order packets in transit. It is a generalization of the Alternating Bit protocol as discussed in Section 5.1, since it allows several messages to be in transit at once.

Stenning defined two processes: a *transmitter* and a *receiver.* The transmitter sends messages from a given sequence of messages to the receiver, using a communication line. The receiver in turn accepts messages from the line, stores them in an output sequence, and acknowledges their receipt by sending a message to the transmitter via another communication line. The communication lines are unreliable; messages traveling in either direction can be lost, reordered, corrupted, or duplicated. Given such an environment, the protocol is supposed to ensure correct delivery of the messages.

The protocol uses a conventional positive-acknowledgment, retransmission-on-time-out technique, and the receiver and transmitter both maintain windows of messages. The transmitter's window contains messages sent but not yet acknowledged. Similarly, the receiver can buffer-ahead messages received out of order (up to some limit), awaiting receipt of the next expected message.

The ***Affirm*** specification of the Data Transfer Protocol, as well as a proposed safety invariant and documentation of its partial proof, are included in [49].

6.2. Transport Service

The transport service represents a protocol layer allowing <u>many</u> users to exchange data. Users are identified by *port addresses*. In order to exchange messages, users must first establish a connection between themselves by appropriate requests to the system; once this is done, users may exchange data in both directions independently.

The exchange itself functions as in the data transfer protocol above, but is controlled by the receiving end (in each direction), through the use of explicit *credits*, i.e., permission to send one or more messages. Once users are done communicating, they ask the system to disconnect the established connection.

We have specified a transport service (but not the protocol implementing the service), and proved several properties about the specification. The specification is done in two levels. The lower level describes one half-duplex connection that knows about the connection status at both ends. The upper level uses two such half-duplex connections, one for each direction, with a shared connection status, thus modeling a full-duplex connection between each pair of users. This division permits the separation of addressing properties from data transfer properties.

Properties proved about this specification show that normal sequences of connection setup and data transfer commands will have their anticipated effects. An interesting detail discovered during these proofs was that the specification precluded a user from establishing a connection with itself.

Complete details of the specification and proven properties may be found in [41].

6.3. Selective Repeat Transport Protocol

A transport protocol similar to Stenning's is specified in [3]. It involves the transfer of messages between a sender and a receiver over an unreliable medium (it may lose messages, but not reorder them). The sender has a window of messages that have been sent but not yet acknowledged. If the acknowledgment does not arrive within a certain (fixed but arbitrary) time, the message is considered to have been lost and is retransmitted. This protocol is proven to be partially correct with respect to the property of "correctly transferring data across the medium."

Progress properties and their characterization in **Affirm** are examined in [2]. In particular, an extension of Floyd's "well-founded set" method [8] is used to show the termination of a data transfer protocol.

6.4. Connection Establishment Protocol

A protocol to provide the kind of connection-establishment service described in Section 6.2 has been specified in [43]. The protocol modeled in that paper is the *three-way handshake* used in the ARPANET TCP algorithm. Although the protocol has not been verified against a complete service specification, several interesting properties have been proved. Work is continuing.

7. PROBLEMS AND EXTENSIONS

While we feel that we have had considerable success in handling protocols with ***Affirm***, several areas need further work. In this section we briefly discuss problems encountered and possible extensions.

7.1. Composition of Specifications

Given that a protocol layer is composed of several interacting stations, it is reasonable to specify the behavior of each station separately, i.e., by presenting its local view of the rest of the system [42]. In a second step, these several local views could be combined to specify the overall behavior of the layer.

At present, the techniques described in the previous sections do not allow the straightforward composition of such specifications; all specifications thus far have described systems from a global reference point.

7.2. Concurrency

A protocol layer supports several users, and may receive simultaneous requests for service from them (e.g., one side is sending a long message while the other acknowledges a previous message). A fully adequate specification method should allow for concurrent operations for both service specifications and protocol specifications. Furthermore, since the stations composing the layer operate independently, the verification method must be able to analyze systems with concurrently executing components.

A basic assumption of most state transition models is that the transitions are atomic, serial operations. This assumption is carried over to the ***Affirm*** specifications where the axioms define the effects of each atomic operation (constructor function). However, this limitation is not as serious as it might at first appear, because by defining operations with a small enough grain the assumption of atomicity is reasonable. For systems with several independent components, the effect of concurrency can be approximated by considering all possible interleavings of the operations of each component.

7.3. Exceptions

The main purpose of a protocol specification is to define *allowed* or *normal* sequences of operations and their effects. Unfortunately, it is a fact of life in the protocol world that users occasionally issue invalid commands, and even protocol stations send inappropriate messages to each other. Thus it is inadequate to state merely that the protocol behavior is undefined for invalid inputs, or that some unspecified party is responsible for guaranteeing that inputs are valid. A richer vocabulary for specifying the handling of such *exceptional conditions* should be supported, including:

1. ignore invalid inputs (i.e., they have no effect),

2. reject them (i.e., they have no effect, but an error indication is returned to the requesting party), and

3. enter an error-recovery portion of the protocol.

Axiomatic specification methods have difficulties with (2) and (3), and the example protocol specifications prepared in ***Affirm*** to date have been limited to ignoring invalid inputs, or simply not defining the results. Several methods to extend axiomatic techniques to handle exceptions have been proposed, but we have not yet determined the best way to proceed in ***Affirm***.

7.4. Specification and Verification of Systems with More than Two Interacting Entities

So far, we have considered only protocols that involve essentially two interacting entities over a transmission medium. This covers a large number of protocols in current use. Nevertheless, there are protocols involving more than two interacting entities (e.g., routing in packet-switching networks). It appears that the techniques discussed in this report can be applied to the specification of these protocols as well, but we have not done it.

As one would expect, there is a combinatorial explosion on the number of possible states of the system. It is at this point that the ability to decompose the overall system description into the description of its components becomes crucial, since it allows the analysis of the behavior of the system through the analysis of the behavior of its components. We are investigating extensions of our techniques to handle such situations.

7.5. Higher Level Protocols

The main application of formal specification methods to protocols has been at the data transfer level, where the first concerns are overcoming message loss, damage, and reordering. Much less work has been done on formally specifying higher level protocols that focus more on translation into and out of canonical forms (e.g., a virtual terminal or file). Furthermore, the operations to be specified are more specialized to the area of concern of the protocol (e.g., graphics, terminal handling, speech compression) than to general data transfer. It remains to be seen whether the same methods are applicable at these higher levels, or whether a new set of abstractions (e.g., involving canonical forms) will be more suitable.

8. CONCLUSION

We have chosen to combine the state transition model and abstract data type approaches for several reasons. First, we have a strong methodology and a rapidly evolving, powerful supporting tool: ***Affirm***. A natural question is whether such a methodology can accommodate a diverse set of formalisms and modeling methods.

This question first arose in conjunction with a toy Security Kernel [29], where we were presented with a state transition specification of an operating system kernel with operations such as *SwapProcesses* and *RaiseBlockLevel*. It was quite natural to represent the specification as a data type and then do an induction proof of an important invariant about relative block and process levels.

We then applied the same method to protocols and have, on the whole, been quite satisfied. Its limitations are touched upon in Chapter 7, but within these limits we have conducted a broad exploration of several protocol issues.

All methods have limitations. Some of the limitations of other methods are handled nicely in our approach. For example, we have no problem with unbounded objects, which cause difficulties for finite-state modeling approaches. However, we lack the decision ability of algorithms based on finite-state exploration and its ability to simply reveal errors.

Another advantage of our approach is the capability to execute specifications: axioms have a natural rewriting rule representation that we exploit. That is, we can take a set of axioms, plug in special values, and see where the rewriting leads. The determinism and executability of axioms is an aid in evaluating the accuracy of specifications, independent of their ability to support proofs. This advantage has been exploited in [43].

Our method also leads naturally from specification to verification, using the standard data type induction methods. No further mechanisms were needed to adjust ***Affirm*** to state transition specifications, although a "front-end" to handle our stylized type specifications would be useful.

In conclusion, a basis has been laid for further steps toward practical specification and verification of not just protocols, but also of any system expressible as a state transition machine. Experience indicates that real protocols can be handled [42]. The major remaining task is to consolidate techniques for proving progress and liveness.

APPENDIX I
DATA TRANSFER SERVICE SPECIFICATION

The service specification uses three auxiliary data types: *ControlState*, a simple enumerated type with four constants (specified in this appendix), *Message*, a type about which we make no assumptions (except the standard one: there is an equality operation on the type), and *QueueOfMessage*, an instantiation of the generic *QueueOfElemType* type from the ***Affirm*** type library [50;Vol.III].

The following text is in exactly the form in which it would be submitted to the system, except for the use of multiple fonts. In particular, the "no change" axioms deleted from the axiom sets of the stylized state machine description on page 15 are included here.

```
type SimpleMessageSystem;

needs types Message, QueueOfMessage, ControlState;
declare s: SimpleMessageSystem;
declare m: Message;
interface State(s): ControlState;
interfaces
    Sent(s), Received(s), Buffer(s): QueueOfMessage;
interfaces
    InitializeService, UserSend(s, m), SendComplete(s), UserReceive(s), ReceiveComplete(s): SimpleMessageSystem;
interface Induction(s): Boolean;

axioms
    State(UserSend(s, m)) = = if State(s) = ReadyToSend
                               then Sending
                               else State(s),
    State(SendComplete(s)) = = if State(s) = Sending
                                then ReadyToReceive
                                else State(s),
    State(UserReceive(s)) = = if State(s) = ReadyToReceive
                               then Acking
                               else State(s),
    State(ReceiveComplete(s)) = = if State(s) = Acking
                                   then ReadyToSend
                                   else State(s),
    State(InitializeService) = = ReadyToSend;

axioms
    Sent(UserSend(s, m)) = = if State(s) = ReadyToSend
                              then Sent(s) Add m
                              else Sent(s),
    Sent(SendComplete(s)) = = Sent(s),
    Sent(UserReceive(s)) = = Sent(s),
    Sent(ReceiveComplete(s)) = = Sent(s),
    Sent(InitializeService) = = NewQueueOfMessage;
```

```
axioms
    Received(UserSend(s, m)) = = Received(s),
    Received(SendComplete(s)) = = Received(s),
    Received(UserReceive(s)) = = if State(s) = ReadyToReceive
                                 then Received(s) Add Front(Buffer(s))
                                 else Received(s),
    Received(ReceiveComplete(s)) = = Received(s),
    Received(InitializeService) = = NewQueueOfMessage;

axioms
    Buffer(UserSend(s, m)) = = if State(s) = ReadyToSend
                               then Buffer(s) Add m
                               else Buffer(s),
    Buffer(SendComplete(s)) = = Buffer(s),
    Buffer(UserReceive(s)) = = Buffer(s),
    Buffer(ReceiveComplete(s)) = = if State(s) = Acking
                                   then Remove(Buffer(s))
                                   else Buffer(s),
    Buffer(InitializeService) = = NewQueueOfMessage;

schema Induction(s)
    = = cases(Prop(InitializeService),
              all s, m (IH(s) imp Prop(UserSend(s, m))),
              all s (IH(s) imp Prop(SendComplete(s))),
              all s (IH(s) imp Prop(UserReceive(s))),
              all s (IH(s) imp Prop(ReceiveComplete(s))));
end {SimpleMessageSystem} ;

type ControlState; {An enumerated type, with four distinct constants.}
declare cs: ControlState;
interfaces
    ReadyToSend, Sending, ReadyToReceive, Acking: ControlState;
axioms {These axioms state that all the constants of this type are distinct.}
    cs = cs = = TRUE,
    ReadyToSend = Sending = = FALSE,
    ReadyToSend = ReadyToReceive = = FALSE,
    ReadyToSend = Acking = = FALSE,
    Sending = ReadyToSend = = FALSE,
    Sending = ReadyToReceive = = FALSE,
    Sending = Acking = = FALSE,
    ReadyToReceive = ReadyToSend = = FALSE,
    ReadyToReceive = Sending = = FALSE,
    ReadyToReceive = Acking = = FALSE,
    Acking = ReadyToSend = = FALSE,
    Acking = Sending = = FALSE,
    Acking = ReadyToReceive = = FALSE;
end {ControlState} ;

type Message; {A type about which we make the absolutely minimal assumptions: there is an equality relation.}
declare m: Message;
axiom m = m = = TRUE;
end {Message} ;
```

APPENDIX II
THE PROTOCOL REPRESENTATION

These axioms are in exactly the form in which they would be submitted to ***Affirm***; see Appendix I for an explanation of how their form differs from the earlier, stylized, presentation. The auxiliary types *Message*, *Packet*, *Medium*, *Bit*, and *QueueOfMessage* are also listed here.

```
type ABProtocol;

needs types Message, Packet, QueueOfMessage, QueueOfPacket, Medium, Bit;

declare s, ss: ABProtocol;
declare m, mm: Message;
declare med: Medium;
declare packetq: QueueOfPacket;
declare pkt: Packet;

interfaces
    Sent(s), Received(s), Text(packetq): QueueOfMessage;

interfaces
    SenderToReceiver(s), ReceiverToSender(s): Medium;

interfaces
    ReceiverBuffer(s), Pending(s): QueueOfPacket;

interfaces
    InitialSequenceNumber, SSN(s), RSN(s): Bit;

interfaces
    InitializeProtocol, Deliver(s), ProtocolSend(s, m), ReceivePacket(s),
    ReceiveAck(s), Retransmit(s), LosePacket(s), LoseAck(s): ABProtocol;

interfaces
    NormalForm(s), Induction(s): Boolean;

axiom s = s = = TRUE;

axioms
    Sent(ProtocolSend(s, m)) = = if Pending(s) = NewQueueOfPacket
                                  then Sent(s) Add m
                                  else Sent(s),
    Sent(ReceivePacket(s)) = = Sent(s),
    Sent(ReceiveAck(s)) = = Sent(s),
    Sent(Deliver(s)) = = Sent(s),
    Sent(Retransmit(s)) = = Sent(s),
    Sent(LosePacket(s)) = = Sent(s),
    Sent(LoseAck(s)) = = Sent(s),
    Sent(InitializeProtocol) = = NewQueueOfMessage;
```

```
axioms
    Received(ProtocolSend(s, m)) = = Received(s),
    Received(ReceivePacket(s)) = = Received(s),
    Received(ReceiveAck(s)) = = Received(s),
    Received(Deliver(s)) = = if ReceiverBuffer(s) = NewQueueOfPacket
                             then Received(s)
                             else Received(s) Add Text(Front(ReceiverBuffer(s))),
    Received(Retransmit(s)) = = Received(s),
    Received(LosePacket(s)) = = Received(s),
    Received(LoseAck(s)) = = Received(s),
    Received(InitializeProtocol) = = NewQueueOfMessage;

axioms
    Text(NewQueueOfPacket) = = NewQueueOfMessage,
    Text(packetq Add pkt) = = Text(packetq) Add Text(pkt);

axioms
    SenderToReceiver(ProtocolSend(s, m)) = = if Pending(s) = NewQueueOfPacket
                                             then Transmit(SenderToReceiver(s), MakePacket(m, SSN(s)))
                                             else SenderToReceiver(s),
    SenderToReceiver(ReceivePacket(s)) = = Receive(SenderToReceiver(s)),
    SenderToReceiver(ReceiveAck(s)) = = SenderToReceiver(s),
    SenderToReceiver(Deliver(s)) = = SenderToReceiver(s),
    SenderToReceiver(Retransmit(s)) = = if Pending(s) = NewQueueOfPacket
                                        then SenderToReceiver(s)
                                        else Transmit(SenderToReceiver(s), Front(Pending(s))),
    SenderToReceiver(LosePacket(s)) = = Receive(SenderToReceiver(s)),
    SenderToReceiver(LoseAck(s)) = = SenderToReceiver(s),
    SenderToReceiver(InitializeProtocol) = = InitializeMedium;

axioms
    ReceiverToSender(ProtocolSend(s, m)) = = ReceiverToSender(s),
    ReceiverToSender(ReceivePacket(s)) = = if    SenderToReceiver(s) ~ = InitializeMedium
                                              and ReceiverBuffer(s) = NewQueueOfPacket
                                              and RSN(s) ~ = Seq(Front(SenderToReceiver(s)))
                                           then Transmit(ReceiverToSender(s), Front(SenderToReceiver(s)))
                                           else ReceiverToSender(s),
    ReceiverToSender(ReceiveAck(s)) = = Receive(ReceiverToSender(s)),
    ReceiverToSender(Deliver(s)) = = if ReceiverBuffer(s) = NewQueueOfPacket
                                     then ReceiverToSender(s)
                                     else Transmit(ReceiverToSender(s), Front(ReceiverBuffer(s))),
    ReceiverToSender(Retransmit(s)) = = ReceiverToSender(s),
    ReceiverToSender(LosePacket(s)) = = ReceiverToSender(s),
    ReceiverToSender(LoseAck(s)) = = Receive(ReceiverToSender(s)),
    ReceiverToSender(InitializeProtocol) = = InitializeMedium;

axioms
    ReceiverBuffer(ProtocolSend(s, m)) = = ReceiverBuffer(s),
    ReceiverBuffer(ReceivePacket(s)) = = if    Seq(Front(SenderToReceiver(s))) = RSN(s)
                                            and SenderToReceiver(s) ~ = InitializeMedium
                                         then NewQueueOfPacket Add Front(SenderToReceiver(s))
                                         else ReceiverBuffer(s),
    ReceiverBuffer(ReceiveAck(s)) = = ReceiverBuffer(s),
    ReceiverBuffer(Deliver(s)) = = NewQueueOfPacket,
    ReceiverBuffer(Retransmit(s)) = = ReceiverBuffer(s),
    ReceiverBuffer(LosePacket(s)) = = ReceiverBuffer(s),
    ReceiverBuffer(LoseAck(s)) = = ReceiverBuffer(s),
    ReceiverBuffer(InitializeProtocol) = = NewQueueOfPacket;
```

```
axioms
    Pending(ProtocolSend(s, m)) = = if Pending(s) = NewQueueOfPacket
                                    then NewQueueOfPacket Add MakePacket(m, SSN(s))
                                    else Pending(s),
    Pending(ReceivePacket(s)) = = Pending(s),
    Pending(ReceiveAck(s)) = = if Seq(Front(ReceiverToSender(s))) = SSN(s) and ReceiverToSender(s) ~ = InitializeMedium
                               then NewQueueOfPacket
                               else Pending(s),
    Pending(Deliver(s)) = = Pending(s),
    Pending(Retransmit(s)) = = Pending(s),
    Pending(LosePacket(s)) = = Pending(s),
    Pending(LoseAck(s)) = = Pending(s),
    Pending(InitializeProtocol) = = NewQueueOfPacket;

axioms
    SSN(ProtocolSend(s, m)) = = SSN(s),
    SSN(ReceivePacket(s)) = = SSN(s),
    SSN(ReceiveAck(s)) = = if Seq(Front(ReceiverToSender(s))) = SSN(s) and ReceiverToSender(s) ~ = InitializeMedium
                           then ~SSN(s)
                           else SSN(s),
    SSN(Deliver(s)) = = SSN(s),
    SSN(Retransmit(s)) = = SSN(s),
    SSN(LosePacket(s)) = = SSN(s),
    SSN(LoseAck(s)) = = SSN(s),
    SSN(InitializeProtocol) = = InitialSequenceNumber;

axioms
    RSN(ProtocolSend(s, m)) = = RSN(s),
    RSN(ReceivePacket(s)) = = if Seq(Front(SenderToReceiver(s))) = RSN(s) and SenderToReceiver(s) ~ = InitializeMedium
                              then ~RSN(s)
                              else RSN(s),
    RSN(ReceiveAck(s)) = = RSN(s),
    RSN(Deliver(s)) = = RSN(s),
    RSN(Retransmit(s)) = = RSN(s),
    RSN(LosePacket(s)) = = RSN(s),
    RSN(LoseAck(s)) = = RSN(s),
    RSN(InitializeProtocol) = = InitialSequenceNumber;

schema
    NormalForm(s) = = cases(Prop(InitializeProtocol),
                            all ss, mm (Prop(ProtocolSend(ss, mm))),
                            all ss (Prop(ReceivePacket(ss))),
                            all ss (Prop(ReceiveAck(ss))),
                            all ss (Prop(Deliver(ss))),
                            all ss (Prop(Retransmit(ss))),
                            all ss (Prop(LosePacket(ss))),
                            all ss (Prop(LoseAck(ss)))),
    Induction(s) = = cases(Prop(InitializeProtocol),
                           all ss, mm (IH(ss) imp Prop(ProtocolSend(ss, mm))),
                           all ss (IH(ss) imp Prop(ReceivePacket(ss))),
                           all ss (IH(ss) imp Prop(ReceiveAck(ss))),
                           all ss (IH(ss) imp Prop(Deliver(ss))),
                           all ss (IH(ss) imp Prop(Retransmit(ss))),
                           all ss (IH(ss) imp Prop(LosePacket(ss))),
                           all ss (IH(ss) imp Prop(LoseAck(ss))));
end {ABProtocol};
```

```
type Medium;

needs type Packet;

declare m, m1, m2: Medium;
declare pkt, pkt1, pkt2: Packet;

interfaces
    InitializeMedium, Transmit(m, pkt), Receive(m), Lose(m): Medium;

interface Front(m): Packet;

interfaces
    Empty(m), pkt in m, Induction(m): Boolean;

infix in;

axioms
    m = m = = TRUE,
    Transmit(m, pkt) = InitializeMedium = = FALSE,
    InitializeMedium = Transmit(m, pkt) = = FALSE,
    Transmit(m1, pkt1) = Transmit(m2, pkt2) = = ((m1 = m2) and (pkt1 = pkt2));

axioms
    Receive(InitializeMedium) = = InitializeMedium,
    Receive(Transmit(m, pkt)) = = if m = InitializeMedium
                                  then InitializeMedium
                                  else Transmit(Receive(m), pkt);

axiom Lose(m) = = Receive(m);

axiom Front(Transmit(m, pkt)) = = if m = InitializeMedium
                                  then pkt
                                  else Front(m);

axiom Empty(m) = = (m = InitializeMedium);

axioms
    pkt in InitializeMedium = = FALSE,
    pkt in Transmit(m, pkt1) = = ((pkt = pkt1) or pkt in m);

schema Induction(m)
      = = cases(Prop(InitializeMedium),
              all m, pkt (IH(m) imp Prop(Transmit(m, pkt))));

end {Medium};
```

```
type Bit;

declare b, b1, b2: Bit;

interfaces
    on, off, ~b1: Bit;

interface NormalForm(b): Boolean;

axiom ~~b = = b;

axioms
    b=b = = TRUE,
    on = off = = FALSE,
    off = on = = FALSE,
    b1 = ~b2 = = b1 ~= b2,
    ~b1 = b2 = = b1 ~= b2;

schema NormalForm(b) = = cases(Prop(on), Prop(off));

end {Bit} ;


type Message; {A type about which we make the absolutely minimal assumptions: there is an equality relation.}

declare m: Message;

axiom m=m = = TRUE;

end {Message} ;


type Packet;

needs types Message, Bit;

declare pkt: Packet;
declare b, b1, b2: Bit;
declare m, m1, m2: Message;

interface MakePacket(m, b): Packet;

interface Seq(pkt): Bit;

interface Text(pkt): Message;

axioms
    pkt=pkt = = TRUE,
    MakePacket(m1, b1) = MakePacket(m2, b2) = = ((m1 = m2) and (b1 = b2));

axiom Seq(MakePacket(m, b)) = = b;

axiom Text(MakePacket(m, b)) = = m;

end {Packet} ;
```

```
type QueueOfMessage;

needs type Message;

declare q, q1, q2, qq: QueueOfMessage;
declare i, i1, i2, ii: Message;

interfaces
    NewQueueOfMessage, q Add i, Remove(q), Append(q1, q2), que(i): QueueOfMessage;
infix Add;

interfaces
    Front(q), Back(q): Message;

interfaces
    NormalForm(q), Induction(q), i in q: Boolean;
infix in;

axioms
    q = q = = TRUE,
    q Add i = NewQueueOfMessage = = FALSE,
    NewQueueOfMessage = q Add i = = FALSE,
    q1 Add i1 = q2 Add i2 = = ((q1 = q2) and (i1 = i2));

axioms
    Remove(NewQueueOfMessage) = = NewQueueOfMessage,
    Remove(q Add i) = = if q = NewQueueOfMessage
                        then q
                        else Remove(q) Add i;
axioms
    Append(q, NewQueueOfMessage) = = q,
    Append(q, q1 Add i1) = = Append(q, q1) Add i1;

axiom que(i) = = NewQueueOfMessage Add i;

axiom Front(q Add i) = = if q = NewQueueOfMessage
                         then i
                         else Front(q);

axiom Back(q Add i) = = i;

axioms
    i in NewQueueOfMessage = = FALSE,
    i in (q Add i1) = = (i in q or (i = i1));

rulelemma Append(NewQueueOfMessage, q) = = q;

schema
    NormalForm(q) = = cases(Prop(NewQueueOfMessage),
                            all qq, ii (Prop(qq Add ii))),

    Induction(q) = = cases(Prop(NewQueueOfMessage),
                           all qq, ii (IH(qq) imp Prop(qq Add ii)));

end {QueueOfMessage};
```

APPENDIX III
SERVICE AXIOMS → PROTOCOL THEOREMS

This appendix contains the correspondence between the service and protocol specifications of the Alternating Bit protocol, and lists the theorems generated as part of the job of proving that the protocol implements the service. These theorems have been proved using ***Affirm***. The proofs are documented in [51].

III.1. The Correspondence between the Service and the Protocol

declare s: ABProtocol;
declare m: Message;

interface *rep*(s): ABProtService;

1. InitializeService = = *rep*(InitializeProtocol)
2. $\text{Sent}_{\text{service}}$(*rep*(s)) = = $\text{Sent}_{\text{protocol}}$(s)
3. $\text{Received}_{\text{service}}$(*rep*(s)) = = $\text{Received}_{\text{protocol}}$(s)
4. Buffer(*rep*(s)) = = Text(Front(Pending(s)))
5. State(*rep*(s)) = = if InS1(s)
                then ReadyToSend
                else if InS2(s)
                    then Sending
                    else if InS3(s)
                        then ReadyToReceive
                        else Acking
6. UserSend(*rep*(s), m) = = *rep*(ProtocolSend(s, m))
7. Receive(*rep*(s)) = = *rep*(Deliver(s))
8. SendComplete(*rep*(s))
   = = *rep*({LosePacket LoseAck Retransmit ReceivePacket ReceiveAck}* (s))
9. ReceiveComplete(*rep*(s))
   = = *rep*({LosePacket LoseAck Retransmit ReceivePacket, ReceiveAck}* (s))

III.2. Correspondence of States Between Service and Protocol

The four states in the service specification, *ReadyToSend*, *Sending*, *ReadyToReceive*, and *Acking*, correspond to four states in the protocol specification, labeled *S1*, *S2*, *S3*, and *S4*. The predicates in the protocol specification defining these states are defined in ***Affirm*** as follows.

```
InS1(s) {ReadyToSend}
    = = (  Pending(s) = NewQueueOfPacket
        and ReceiverBuffer(s) = NewQueueOfPacket
        and Sent(s) = Received(s)
        and PktsOld(s, SenderToReceiver(s))
        and PktsOld(s, ReceiverToSender(s))
        and RSN(s) = SSN(s))
```

```
InS2(s) {Sending}
    = = (  Pending(s) ~= NewQueueOfPacket
       and ReceiverBuffer(s) = NewQueueOfPacket
       and Sent(s) = Received(s) Add Text(Front(Pending(s)))
       and PktsCurrentOrOld(s, SenderToReceiver(s))
       and PktsOld(s, ReceiverToSender(s))
       and RSN(s) = SSN(s))

InS3(s) {ReadyToReceive}
    = = (  Pending(s) ~= NewQueueOfPacket
       and ReceiverBuffer(s) = Pending(s)
       and Sent(s) = Received(s) Add Text(Front(Pending(s)))
       and PktsCurrent(s, SenderToReceiver(s))
       and PktsOld(s, ReceiverToSender(s))
       and RSN(s) ~= SSN(s))

InS4(s) {Acking}
    = = (  Pending(s) ~= NewQueueOfPacket
       and ReceiverBuffer(s) = NewQueueOfPacket
       and Sent(s) = Received(s)
       and PktsCurrent(s, SenderToReceiver(s))
       and PktsCurrentOrOld(s, ReceiverToSender(s))
       and RSN(s) ~= SSN(s))
```

III.3. Example: Mapping Two Service Axioms into Protocol Theorems

Service axiom

$\text{Received}_{service}(\text{UserSend}(S, m)) == \text{Received}_{service}(S)$

use S = *rep*(s)

$\text{Received}_{service}(\text{UserSend}(rep(s), m)) = \text{Received}_{service}(rep(s))$

use 6

$\text{Received}_{service}(rep(\text{ProtocolSend}(s, m))) = \text{Received}_{service}(rep(s))$

use 3

$\text{Received}_{protocol}(\text{ProtocolSend}(s, m)) = \text{Received}_{protocol}(s)$

Service axiom

$\text{Sent}_{service}(\text{UserSend}(S, m)) == \text{if State}(S) = \text{ReadyToSend then Sent}_{service}(S) \text{ Add m else Sent}_{service}(S)$

use S = *rep*(s)

$\text{Sent}_{service}(\text{UserSend}(rep(s), m)) == \text{if State}(rep(s)) = \text{ReadyToSend then Sent}_{service}(rep(s)) \text{ Add m else Sent}_{service}(rep(s))$

use 6

$Sent_{service}(rep(ProtocolSend(s, m))) == $ if $State(rep(s)) = ReadyToSend$
then $Sent_{service}(rep(s))$ Add m
else $Sent_{service}(rep(s))$

use 2

$Sent_{protocol}(ProtocolSend(s, m)) == $ if $State(rep(s)) = ReadyToSend$
then $Sent_{protocol}(s)$ Add m
else $Sent_{protocol}(s)$

use 5

$Sent_{protocol}(ProtocolSend(s, m)) == $ if InS1(s)
then $Sent_{protocol}(s)$ Add m
else $Sent_{protocol}(s)$

III.4. Effects on State Variables by User Operations

{for the *Send* operation:}

theorem *SS*, Sent(ProtocolSend(s, m)) = if InS1(s)
then Sent(s) Add m
else Sent(s);

theorem *RS*, Received(ProtocolSend(s, m)) = Received(s);

theorem *BS*, Text(Pending(ProtocolSend(s, m))) = if InS1(s)
then Text(Pending(s)) Add m
else Text(Pending(s));

{for the *Receive* operation:}

theorem *SR*, Sent(Deliver(s)) = Sent(s);

theorem *RR*, Received(Deliver(s)) = if InS3(s)
then Received(s) Add Front(Text(Pending(s)))
else Received(s);

theorem *BR*, Text(Pending(Deliver(s))) = Text(Pending(s));

{for the *InitializeProtocol* operation:}

theorem *SI*, Sent(InitializeProtocol) = NewQueueOfMessage;

theorem *RI*, Received(InitializeProtocol) = NewQueueOfMessage;

theorem *BI*, Text(Pending(InitializeProtocol)) = NewQueueOfMessage;

theorem *TI*, InS1(InitializeProtocol);

III.5. Effects on State Variables by Spontaneous Operations

{for the *Sent* state variable:}
theorem *SentSpont*,
Sent(ReceiveAck(s)) = Sent(s)
and Sent(ReceivePacket(s)) = Sent(s)
and Sent(Retransmit(s)) = Sent(s)
and Sent(LosePacket(s)) = Sent(s)
and Sent(LoseAck(s)) = Sent(s);

{for the *Received* state variable:}
theorem *ReceivedSpont*,
Received(ReceiveAck(s)) = Received(s)
and Received(ReceivePacket(s)) = Received(s)
and Received(Retransmit(s)) = Received(s)
and Received(LosePacket(s)) = Received(s)
and Received(LoseAck(s)) = Received(s);

{for the *Buffer* state variable:}
theorem *BufferSpont1*,
~InS4(s)
imp Pending(ReceiveAck(s)) = Pending(s) and ~InS4(ReceiveAck(s))
and Pending(ReceivePacket(s)) = Pending(s) and ~InS4(ReceivePacket(s))
and Pending(Retransmit(s)) = Pending(s) and ~InS4(Retransmit(s))
and Pending(LosePacket(s)) = Pending(s) and ~InS4(LosePacket(s))
and Pending(LoseAck(s)) = Pending(s) and ~InS4(LoseAck(s));

theorem *BufferSpont2*,
InS4(s)
imp (Pending(ReceiveAck(s)) = Pending(s) and InS4(ReceiveAck(s))
or InS1(ReceiveAck(s)))
and Pending(ReceivePacket(s)) = Pending(s) and InS4(ReceivePacket(s))
and Pending(Retransmit(s)) = Pending(s) and InS4(Retransmit(s))
and Pending(LosePacket(s)) = Pending(s) and InS4(LosePacket(s))
and Pending(LoseAck(s)) = Pending(s) and InS4(LoseAck(s));

III.6. The next-state Transitions for all Operations

{for the *ProtocolSend* operation:}
theorem *S1Succ1*, {*Move from state S1 to state S2*}
InS1(s) imp InS2(ProtocolSend(s, m));

theorem *S2Succ1*, {*No change*} InS2(s) imp InS2(ProtocolSend(s, m));
theorem *S3Succ1*, {*No change*} InS3(s) imp InS3(ProtocolSend(s, m));
theorem *S4Succ1*, {*No change*} InS4(s) imp InS4(ProtocolSend(s, m));

{for the *ReceivePacket* operation:}
theorem *S1Succ2*, {*No change*} InS1(s) imp InS1(ReceivePacket(s));

theorem *S2Succ2*, {*No change, or move from state S2 to state S3*}
InS2(s) imp InS2(ReceivePacket(s)) or InS3(ReceivePacket(s));

theorem *S3Succ2*, {*No change*} InS3(s) imp InS3(ReceivePacket(s));
theorem *S4Succ2*, {*No change*} InS4(s) imp InS4(ReceivePacket(s));

{for the *ReceiveAck* operation:}
theorem *S1Succ3*, {*No change*} InS1(s) imp InS1(ReceiveAck(s));
theorem *S2Succ3*, {*No change*} InS2(s) imp InS2(ReceiveAck(s));
theorem *S3Succ3*, {*No change*} InS3(s) imp InS3(ReceiveAck(s));

theorem *S4Succ3*, {*No change, or move from state S4 to state S1*}
InS4(s) imp InS4(ReceiveAck(s)) or InS1(ReceiveAck(s));

{for the *Deliver* operation:}
theorem *S1Succ4*, {*No change*} InS1(s) imp InS1(Deliver(s));
theorem *S2Succ4*, {*No change*} InS2(s) imp InS2(Deliver(s));
theorem *S3Succ4*, {*No change*} InS3(s) imp InS4(Deliver(s));
theorem *S4Succ4*, {*No change*} InS4(s) imp InS4(Deliver(s));

{for the *Retransmit* operation:}
theorem *S1Succ5*, {*No change*} InS1(s) imp InS1(Retransmit(s));
theorem *S2Succ5*, {*No change*} InS2(s) imp InS2(Retransmit(s));
theorem *S3Succ5*, {*No change*} InS3(s) imp InS3(Retransmit(s));
theorem *S4Succ5*, {*No change*} InS4(s) imp InS4(Retransmit(s));

{for the *LoseAck* operation:}
theorem *S1Succ6*, {*No change*} InS1(s) imp InS1(LoseAck(s));
theorem *S2Succ6*, {*No change*} InS2(s) imp InS2(LoseAck(s));
theorem *S3Succ6*, {*No change*} InS3(s) imp InS3(LoseAck(s));
theorem *S4Succ6*, {*No change*} InS4(s) imp InS4(LoseAck(s));

{for the *LosePacket* operation:}
theorem *S1Succ7*, {*No change*} InS1(s) imp InS1(LosePacket(s));
theorem *S2Succ7*, {*No change*} InS2(s) imp InS2(LosePacket(s));
theorem *S3Succ7*, {*No change*} InS3(s) imp InS3(LosePacket(s));
theorem *S4Succ7*, {*No change*} InS4(s) imp InS4(LosePacket(s));

APPENDIX IV
IMPLEMENTING PROCEDURES AND ASSERTIONS

IV.1. Asserted Procedures for the Sender

```
procedure Sender(var SenderToReceiver,ReceiverToSender: Medium);
{This is an environment for the send operations ProtocolSend and SenderTimeout.
        It has no body and no assertions.}

var Pending: QueueOfPacket; var Sent:QueueOfMessage;  var SSN:Bit;

procedure ProtocolSend(m:Message; var success:Boolean)
imports(var Sent:QueueOfMessage; var SenderToReceiver: Medium; var Pending: QueueOfPacket; SSN: Bit);
post  PSPost(m, success, Sent, Sent', SenderToReceiver, SenderToReceiver', Pending, Pending', SSN);

{does a ProtocolSend(p,m); sets success bit if we did something. Note
        that ReceiverToSender is not imported.}
begin {ProtocolSend}
if Empty(Pending) then
        begin;
        Pending:= que(MakePacket(m,SSN));
        SenderToReceiver:= Transmit(SenderToReceiver,Front(Pending));
        success:= TRUE;
        Sent:= Add(Sent, m);
        end
else success:= FALSE;
end {ProtocolSend};


procedure SenderTimeout
imports(var SenderToReceiver,ReceiverToSender: Medium;
     var Pending: QueueOfPacket; var SSN: Bit);
post  STPost(SenderToReceiver, SenderToReceiver', ReceiverToSender,
        ReceiverToSender', Pending, Pending', SSN, SSN');

{Performs a Retransmit(ReceiveAck(p))}
begin {SenderTimeout}
        if SeqMatch(ReceiverToSender,SSN) {includes test for Empty}
          then {get a valid Ack}
                begin;
                Pending:=Remove(Pending);
                SSN:= ~SSN;
                end;
        ReceiverToSender := Receive(ReceiverToSender);
        if ~Empty(Pending) then
                SenderToReceiver := Transmit(SenderToReceiver,Front(Pending));
end {SenderTimeout};

procedure InitSender
imports(var Pending: QueueOfPacket; var SSN:Bit);
post  ISPost(Sent,Pending,SSN);
begin
        Sent:= NewQueueOfMessage;
        Pending:= NewQueueOfPacket;
        SSN:= InitialSequenceNumber;
end {InitSender};

begin {Sender has no body}; end;
```

IV.2. Asserted Procedures for the Receiver

```
procedure Receiver(var SenderToReceiver, ReceiverToSender: Medium);

var Out: QueueOfMessage;
var RSN:Bit;
var ReceiverBuffer: QueueOfPacket;

procedure ReceivePacket
imports(var SenderToReceiver,ReceiverToSender: Medium; var RSN: Bit; var ReceiverBuffer: QueueOfPacket);
post  RPPost( ReceiverBuffer, ReceiverBuffer', SenderToReceiver, SenderToReceiver',
      ReceiverToSender, ReceiverToSender', RSN, RSN');

{Doesn't deliver, just places in ReceiverBuffer. Only Acks after delivery}
begin
if SeqMatch(SenderToReceiver,RSN) then
        begin
        {Something we were waiting for. Accept, prepare to deliver.
         Won't Ack until delivered}
        RSN := ~RSN;
        ReceiverBuffer := que(Front(SenderToReceiver));
        end
else if SeqMatch(SenderToReceiver,~RSN) and Empty(ReceiverBuffer) then
        {Having delivered, we ACK when requested for the last packet}
        ReceiverToSender := Transmit(ReceiverToSender,
                                Front(SenderToReceiver))
end; {that's all for ReceivePacket}


procedure Deliver
imports(var Out: QueueOfMessage; var ReceiverToSender: Medium;
      var ReceiverBuffer: QueueOfPacket);
post  DPost(Out, Out', ReceiverBuffer, ReceiverBuffer', ReceiverToSender,
      ReceiverToSender');
begin
if ~Empty(ReceiverBuffer)  then
        begin
        Out := Out Add Text(Front(ReceiverBuffer));
        ReceiverToSender:= Transmit(ReceiverToSender, Front(ReceiverBuffer));
        ReceiverBuffer:= NewQueueOfPacket;
        end;
end; {end of Deliver}

procedure InitReceiver
imports(var RSN:Bit; var ReceiverBuffer: QueueOfPacket);
post  IRPost(Out,ReceiverBuffer,RSN);
begin
        Out:= NewQueueOfMessage;
        RSN:= InitialSequenceNumber;
        ReceiverBuffer:= NewQueueOfPacket;
end {InitReceiver};

begin: {Receiver has no body} end;
```

IV.3. Definitions for the Assertions

```
define      DPost(sent',sent,rbuf',rbuf,rs',rs) ==
        all s(  ReceiverVarsMatch(s,sent,rbuf,SenderToReceiver(s),rs,RSN(s))
                imp
                ReceiverVarsMatch(Deliver(s),sent',rbuf',SenderToReceiver(s),
                        rs',RSN(s)));

define      RPPost(rbuf',rbuf,sr',sr,rs',rs,rsn',rsn) ==
        all s(  ReceiverVarsMatch(s,Received(s),rbuf,
                        sr,rs,rsn)
                imp
                ReceiverVarsMatch(ReceivePacket(s),Received(s),rbuf',
                        sr',rs',rsn'));


define      IRPost(out,rbuf,rsn) ==
        some sr,rs(     ReceiverVarsMatch(InitializeProtocol,
                                out,rbuf,sr,rs,rsn));

define      STPost(sr',sr,rs',rs,pend',pend,ssn',ssn) ==
        all s(  SenderVarsMatch(s,Sent(s),pend,sr,rs,ssn)
                imp
                SenderVarsMatch(Retransmit(ReceiveAck(s)),
                        Sent(s),pend',sr',rs',ssn'));

define      PSPost(msg,succ,sent',sent,sr',sr,pend',pend,ssn) ==
        all s(  SenderVarsMatch(s,sent,pend,sr,ReceiverToSender(s),ssn)
                imp
                SenderVarsMatch(ProtocolSend(s,msg),
                        sent',pend',sr',ReceiverToSender(s),ssn))
        and     succ = Empty(pend);

define      ISPost(sent,pend,ssn) ==
        some sr,rs(
                SenderVarsMatch(InitializeProtocol,sent,pend,sr,rs,ssn));
```

IV.4. Context in Which the Assertions are Defined

```
type ABContext;

declare s,s',s1,s2                   :ABProt5;
declare msg                          :Message;
declare sent,sent',out,out'          :QueueOfMessage;
declare qp,pend,pend',rbuf,rbuf'     :QueueOfPacket;
declare sr,sr',rs,rs',s2r,r2s        :Medium;
declare rsn,ssn,rsn',ssn'            :Bit;
declare b,succ,succ'                 :Boolean;

interface      SenderVarsMatch(s,sent,pend,sr,rs,ssn),
               ReceiverVarsMatch(s,out,rbuf,sr,rs,rsn),
               SelectorsMatch(s,sent,out,pend,rbuf,sr,rs,ssn,rsn),
               SeqMatch(sr,bit),
               Empty(qp): Boolean;

define     SenderVarsMatch(s,sent,pend,sr,rs,ssn) ==
               SelectorsMatch(s,sent,Received(s),pend,ReceiverBuffer(s),sr,rs,ssn,RSN(s)),

           ReceiverVarsMatch(s,out,rbuf,sr,rs,rsn) ==
               SelectorsMatch(s,Sent(s),out,Pending(s),rbuf,sr,rs,SSN(s),rsn);

define      SelectorsMatch(s,sent,out,pend,rbuf,s2r,r2s,ssn,rsn) ==
               sent = Sent(s) and
               out = Received(s) and
               pend = Pending(s) and
               rbuf= ReceiverBuffer(s) and
               s2r = SenderToReceiver(s) and
               r2s = ReceiverToSender(s) and
               ssn = SSN(s) and
               rsn = RSN(s);

axiom       SeqMatch(pktbuf,i) ==
               not Empty(pktbuf) and Seq(Front(pktbuf)) = i;

define     Empty(qp) == qp=NewQueueOfPacket;

interface      PSPost(msg,succ,sent',sent,sr',sr,pend',pend,ssn),
               STPost(sr',sr,rs',rs,pend',pend,ssn',ssn),
               ISPost(sent,pend,ssn),
               RPPost(rbuf',rbuf,sr',sr,rs',rs,rsn',rsn),
               DPost(sent',sent,rbuf',rbuf,rs',rs),
               IRPost(out,rbuf,rsn)                    :Boolean;

note the assertion definitions go here;

end {ABContext};
```

REFERENCES

1. Bartlett, K. A., R. A. Scantlebury, and P. T. A. Wilkinson, "Note on reliable full duplex transmission over half duplex links," *Communications of the ACM* 12 (5), May 1969, 260-261.

2. Berthomieu, B., *Proving Progress Properties of Communication Protocols in **Affirm***, USC/Information Sciences Institute, Program Verification Project, Affirm Memo 35, September 1980.

3. Berthomieu, B., *Selective Repeat Protocol: Axiomatization and Proofs*, USC/Information Sciences Institute, Program Verification Project, Affirm Memo 36, September 1980.

4. Berzins, V. A., *Abstract Model Specifications for Data Abstractions*, Massachusetts Institute of Technology, Technical Report MIT/LCS/TR-221, July 1979.

5. Bochmann, G. V., and J. Gecsei, "A unified method for the specification and verification of protocols," in *Proceedings of the IFIP Congress*, pp. 229-234, Toronto, Canada, August 1977.

6. Brand, D., and W. H. Joyner, "Verification of protocols using symbolic execution," *Computer Networks* 2 (4-5), September/October 1978.

7. Flon, L., and J. Misra, "A unified approach to the specification and verification of abstract data types," in *Proceedings of the Conference on Specification of Reliable Software*, pp. 162-169, IEEE Computer Society, April 1979.

8. Floyd, R. W., "Assigning meanings to programs," in J. T. Schwartz (ed.), *Proceedings of Symposia in Applied Mathematics*, pp. 19-32, American Mathematical Society, 1967.

9. Gerhart, S. L., et al., "An overview of ***Affirm***: a specification and verification system," in *Proceedings IFIP 80*, pp. 343-348, Australia, October 1980.

10. Goguen, J. A., J. W. Thatcher, and E. G. Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types," in R. T. Yeh (ed.), *Current Trends in Programming Methodology*, pp. 80-149, Prentice-Hall, 1978.

11. Goguen, J. A., J. W. Thatcher, and E. G. Wagner, "Abstract data types as initial algebras and the correctness of data representations," in R. T. Yeh (ed.), *Current Trends in Programming Methodology, Volume IV*, Prentice-Hall, 1978.

12. Goguen, J. A., and J. J. Tardo, "An introduction to OBJ: a language for writing and testing algebraic program specifications," in *Proceedings of the Specifications of Reliable Software Conference*, pp. 170-189, IEEE Computer Society, Technical Committee on Software Engineering, April 1979.

13. Good, D. I., R. M. Cohen, and J. Keeton-Williams, "Principles of proving concurrent programs in GYPSY," in *Proceedings of 6th ACM Symposium on Principles of Programming Languages*, pp. 42-52, ACM SIGPLAN, 1979.

14. Guttag, J. V., *The Specification and Application to Programming of Abstract Data Types*, Ph.D. thesis, University of Toronto, Department of Computer Science, October 1975.

15. Guttag, J. V., and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica* 10, 1978, 27-52.

16. Guttag, J. V., E. Horowitz, and D. R. Musser, "The design of data type specifications," in R. T. Yeh (ed.), *Current Trends in Programming Methodology*, pp. 60-79, Prentice-Hall, 1978. (An expanded version of a paper which appeared in *Proceedings of the Second International Conference on Software Engineering*, October 1976.)

17. Guttag, J. V., E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *CACM* 21, December 1978, 1048-1064. (Also USC/Information Sciences Institute RR-76-48, August 1976.)

18. Guttag, J. V., "Notes on type abstraction," *IEEE Transactions on Software Engineering* SE-6 (1), January 1980, 13-23.

19. Hailpern, B. T., *Verifying Concurrent Processes Using Temporal Logic*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, August 1980. Technical Report No. 195.

20. Hailpern, B., and S. Owicki, "Verifying network protocols using temporal logic," in *Proceedings of the 1980 Trends and Applications Symposium: Computer Network Protocols*, National Bureau of Standards, Gaithersburg, Maryland, May 1980.

21. Hajek, J., "Automatically verified data transfer protocols," in *Proceedings of the International Conference on Computer Communication*, pp. 749-756, International Council for Computer Communication, 1978.

22. Hoare, C. A. R., "Proof of correctness of data representations," *Acta Informatica* 1 (4), 1972, 271-281.

23. *Reference Model of Open Systems Architecture*. Publication number TC97/SC16/N537, International Organization for Standardization.

24. Krogdahl, S., "Verification of a class of link-level protocols," *BIT* 18, 1978, 436-448.

25. Liskov, B., and S. N. Zilles, "Specification techniques for data abstractions," *IEEE Transactions on Software Engineering* SE-1 (1), March 1975, 7-19.

26. Liskov, B., and V. Berzins, "An appraisal of program specifications," in P. Wegner (ed.), *Research Directions in Software Technology*, MIT Press, 1979.

27. Locasso, R., J. Scheid, D. V. Schorre, and P. Eggert, *The Ina Jo Specification Language Reference Manual*, System Development Corporation, Technical Report TM-(L)-6021/001/00, June 1980.

28. Loeckx, J., *Algorithmic Specifications of Abstract Data Types*, Universität des Saarlandes (Saarbrücken), Technical Report, 1980.

29. Millen, J. K., *Operating System Security Verification*, The MITRE Corporation, Technical Report M79-223, September 1979.

30. Musser, D. R., "A data type verification system based on rewrite rules," in *Proceedings of the Sixth Texas Conference on Computing Systems*, pp. 1A22-1A31, Austin, Texas, November 1977.

31. Musser, D. R., "Abstract data type specification in the ***Affirm*** system," *IEEE Transactions on Software Engineering* SE-6 (1), January 1980, 24-32.

32. Overman, W. T., *Formal verification of GMBs*, University of California, Los Angeles, Computer Science Department, Internal Memorandum 176, 1977.

33. Owicki, S., and L. Lamport, *Proving Liveness Properties of Concurrent Programs*, Stanford University, Technical Report S&L 1 (Op. 57), October 1980.

34. Parnas, D. L., "A technique for software module specification with examples," *Communications of the ACM* 15 (5), May 1972, 330-336.

35. Parnas, D., "The use of precise specifications in the development of software," in *Proceedings of the IFIP Congress 1977*, pp. 861-868, IFIP, 1977.

36. Parnas, D., "Designing software for ease of extension and contraction," in *Proceedings of the Third International Conference on Software Engineering*, pp. 264-277, IEEE-ACM, May 1978.

37. Principato, R. N., Jr., *A Formalization of the State Machine Specification Technique*, Massachusetts Institute of Technology, Technical Report MIT/LCS/TR-202, May 1978.

38. Razouk, R. R., and G. Estrin, "Validation of the X.21 interface specification using SARA," in *Proceedings of the 1980 Trends and Applications Symposium: Computer Network Protocols*, pp. 155-167, National Bureau of Standards, Gaithersburg, Maryland, May 1980.

39. Robinson, L., and O. Roubine, *SPECIAL - A Specification and Assertion Language*, Stanford Research Institute, Technical Report CSL-46, 1977.

40. Robinson, L., and K. Levitt, "Proof techniques for hierarchically structured programs," *Communications of the ACM* 20 (4), April 1977, 271-283.

41. Schwabe, D., *Transport Protocol Specification in* ***Affirm***, USC/Information Sciences Institute, Program Verification Project, Affirm Memo 19, March 1980.

42. Schwabe, D., *Formal Techniques for Specification and Verification of Protocols*, Ph.D. thesis, University of California, Los Angeles, Computer Science Department, March 1981. (Also UCLA Technical Report ENG 8209.)

43. Schwabe, D., *Formal Specification and Verification of a Connection Establishment Protocol*, USC/Information Sciences Institute, ISI/RR-81-91, March 1981.

44. Stenning, N. V., "A data transfer protocol," *Computer Networks* 1, 1976, 99-110.

45. Subrahmanyan, P. A., *On Proving the Correctness of Data Type Implementations*, University of Utah, Department of Computer Science, Technical Report, September 1979.

46. Sunshine, C. A., *Interprocess Communication Protocols for Computer Networks*, Ph.D. thesis, Stanford University, 1975.

47. Sunshine, C. A., "Formal methods for protocol specification and verification," *Computer* 12 (9), September 1979, 20-27.

48. Sunshine, C. A., *Formal Modeling of Communication Protocols*, USC/Information Sciences Institute, RR-81-89, March 1981.

49. Thompson, D. H., *A Behavioral Axiomatization of the Stenning Data Transfer Protocol*, USC/Information Sciences Institute, Program Verification Project, Affirm Memo 16, June 1980.

50. Thompson, D. H., S. L. Gerhart, R. W. Erickson, S. Lee, and R. L. Bates, eds., *The **Affirm** Reference Library*, USC/Information Sciences Institute, 1981. (Five volumes: Reference Manual, User's Guide, Type Library, Annotated Transcripts, and Collected Papers, 450 pages.)

51. Thompson, D. H., and R. W. Erickson, *Documentation of the Proofs for the **Affirm**-Protocol Paper*, USC/Information Sciences Institute, Program Verification Project, Affirm Memo 39, January 1981.

52. Wulf, W. A., R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Transactions on Software Engineering* SE-2 (4), December 1976, 253-265.

DA

FIL

6