INTEGRATED METHODS FOR PROTOCOL SPECIFICATION AND VERIFICATION

Benedetto L. DiVito

Technical Report 24 March 1982

Institute for Computing Science
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

A methodology for the formal specification and deductive verification of protocols is presented. Protocol modules are modeled as sequential processes and abstract behavioral specifications are given for each. In concurrent systems such processes are connected by buffers and communicate by passing messages, as in the Gypsy model of concurrency. Methods similar to those of Gypsy for verifying safety properties are discussed and illustrated with an example, the data transfer protocol introduced by Stenning. The methodology is geared toward the use of automated verification technology. A fully mechanical proof of the example in this paper was obtained using the Boyer-Moore theorem prover.

1 Introduction

Recently, the importance of applying formal methods to the problem of communication protocol design has been recognized. A number of approaches to protocol specification and verification have been put forth, using a large variety of models and techniques. Surveys of this work are readily available [Sunshine79, Bochmann80]. Several distinct lines of research have subsequently emerged, although it is safe to say that we have not yet heard the final word.

In this paper, we report on a new methodology that has been developed to address this problem. It is based on techniques for verifying systems of concurrent processes that communicate by message passing. Only verification of safety properties are considered at this time; we hope to consider liveness properties in the future. Included is a novel method for stating abstract behavioral specifications of protocol modules. It avoids the highly procedural forms of specification that are in common use today. Also, a significant aspect of the methodology is the use of mechanical theorem proving tools to actually carry out the proofs. We view this as important not so much for the obvious advantages of automation but rather for the elimination of errors that can result from doing tedious proofs by hand.

Overall we would characterize the methodology as being an integration of many tools and techniques from a diversity of sources. Among them are verification of concurrent and sequential processes, functional language design, state transition models, decision table techniques, deductive theory building and mechanical theorem proving. Particularly important is the integration of techniques for modeling concurrent processes and state transition systems into a unified framework. The manner in which these ideas have been forged together reflect our belief that an effective methodology is the result of a delicate balance between theoretical and practical considerations.

Much of this work is based on concepts found in Gypsy [Good77-2a, Good78-2]. The Gypsy methodology, developed at the University of Texas at Austin, is a highly successful methodology for specifying, verifying and implementing concurrent programs. An initial attempt at protocol verification was performed as a straightforward application of the Gypsy methods and automated tools [DiVito81]. With additional help from the AFFIRM verification system [Musser80, Gerhart80], the experiment was successfully completed. Nevertheless, several aspects of this approach were somewhat less than satisfying, most notably the procedural form of specifying protocol behavior. This situation led to the search for a new methodology that was more suited to the special needs of protocol work. Full details of the subsequent research are contained in [DiVito82-2]. Only a simplified form of the model is reported in this paper.

Essentially what was done was to start with some of the better ideas of Gypsy and combine them with a few additional ones to create a new integrated methodology. Unfortunately, this meant that we could not make direct use of Gypsy’s automated tools. We did, however, take the opportunity to try other theorem provers and found that we could make good use of the Boyer-Moore theorem prover [Boyer79]. At the present time we are using fully mechanical theorem proving tools and a partially automated verification condition generator. Other work is currently done by hand.

The methodology has successfully been applied to a procotol first introduced and analyzed by Stenning [Stenning76]. Fully mechanical proofs of the verification conditions have been obtained. To illustrate our concepts we use this protocol as an example throughout the rest of this paper. Hailpern and Owicki have also analyzed this protocol, presenting proofs of both safety and liveness properties [Hailpern81]. Our methods for verifying safety properties are similar to theirs. There is also some similarity between our proofs of concurrency and those reported in [Misra81]. Although it may be less apparent, our technique for stating behavioral specifications bears some resemblance to the protocol work being done with AFFIRM [Schwabe81, Thompson81].

2 Process Model

We begin by introducing the concurrent process model that underlies our basic methods. It is based on the process model of Gypsy. Both our model and the Gypsy model represent pure message passing systems.

Active computing entities are known as *sequential processes*. Each such process is assumed to be resident on a single processor and communicates with its environment (other processes) via message buffers. A message buffer is a finite size queue that connects two sequential processes, one sending to it and the other receiving from it. No other means of data sharing is allowed; all variables within a process are strictly local. Note that a buffer may be of size zero, in which case the message passing regime would be similar to that of Communicating Sequential Process (CSP) models [Hoare78].

We can build hierarchical process models using *concurrent processes*. A concurrent process consists of several subprocesses plus a number of local buffers to interconnect them. The subprocesses may themselves be either sequential or concurrent. Thus a concurrent process may be viewed as a tree structured object, where the leaves represent sequential processes (executing entities) and the nonleaf nodes represent concurrent processes.

In order to formally specify and verify process models, we provide a specification language. Within this language it is possible to give precise and yet reasonably abstract specifications of sequential and concurrent processes. The language has two distinct subcomponents: a set of language structures for stating process definitions and a functional sublanguage for performing operations on data objects. Actually, the functional language component is needed to express both process definitions and assertions for verification.

We illustrate these concepts with an example protocol. It is essentially the protocol described by Stenning in [Stenning76]. Readers unfamiliar with it need not be concerned; it will be presented in piecewise fashion throughout the rest of this paper. The purpose of this protocol is to provide a unidirectional data transfer service between a pair of user processes. A similar, though bidirectional, data transfer function would normally be part of a comprehensive transport protocol. Such a transport protocol would include connection management services as well. However, for the purposes of this paper we will restrict our attention to the simple data transfer function.

```
                +-----------------+
                |                 |
+-----+         |    Transport    |        +-----+
|     |-------->|     service     |------->|     |
+-----+         |                 |        +-----+
source          +-----------------+          sink
```

Figure 1: Transport service, external view.

We model the simple transport service by a single concurrent process as shown in Fig. 1. This process represents what is sometimes referred to as a protocol machine. In this case it has one input buffer named "source" and one output buffer named "sink." The transport service simply moves messages (in sequence) from the source to the sink.

Next we examine the components of this concurrent process. Fig. 2 shows that it is made up of a sender, a receiver and two medium processes. The sender and receiver are sequential processes (the transport stations) whereas the transmission media are modeled as (potentially) concurrent processes. For our purposes the medium processes can be considered independent datagram networks, although in practice a single datagram network would be employed. Datagram networks have the property that delivery is unreliable to the extent that packets may be lost, reordered, duplicated or corrupted. It is the responsibility of the transport protocol to overcome these sources of unreliability. We make the usual assumption that packets received with checksum errors are filtered out at some lower level and would appear to be lost by the network. Therefore, the medium processes in our model do not corrupt packets.

There are four internal buffers used to interconnect the subprocesses of the transport service process. The upper path from sender to receiver is for the transmission of packets and the lower path is for returning acknowledgments. A packet contains both a user message and a sequence number. An acknowledgment consists of a sequence number only. Throughout this paper, sequence numbers are

```
           +------------------------------------+
           |              +------+              |
           |              |      |              |
           |   sndr_pkt-->|medium|-->rcvr_pkt   |
           |      ^       |      |       |      |
           |      |       +------+       |      |
           |      |                      v      |
           |   +------+              +------+   |
+-----+    |   |      |              |  re- |   |    +-----+
|     |--->|-->|sender|              |ceiver|-->|--->|     |
+-----+    |   |      |              |      |   |    +-----+
source     |   +------+              +------+   |      sink
           |      ^                      |      |
           |      |       +------+       |      |
           |      |       |      |       v      |
           |   sndr_ack<--|medium|<--rcvr_ack   |
           |              |      |              |
           |              +------+              |
           +------------------------------------+
                    transport_service
```

Figure 2: Transport service, internal view.

```
process transport_service (input  source: message;
                           output sink: message) =
begin
  buffers (sndr_pkt, rcvr_pkt: packet;
           sndr_ack, rcvr_ack: natural);

  cobegin
    sender (source, sndr_ack, sndr_pkt);
    receiver (rcvr_pkt, sink, rcvr_ack);
    medium (sndr_pkt, rcvr_pkt);
    medium (rcvr_ack, sndr_ack);
  end
end
```

Figure 3: Transport service process definition.

modeled as unbounded natural numbers.

The transport service process can be given a concurrent process definition, as shown in Fig. 3. This is the realization of the structure shown in Fig. 2. There are three major parts to a concurrent process definition.

1. The process header identifies the input and output buffers and the type of objects that flow through them.
2. The internal buffers are declared together with their element data types.
3. The subprocess structure is introduced by means of a "cobegin" statement. Each subprocess is "called" by passing buffers as its parameters. These may be either internal or external buffers.

Note that two separate instances of the medium process are included. For simplicity in the process model and later in the proof methods, buffers are restricted to the single input, single output case. In other words, a buffer connects exactly one process output to exactly one process input.

The medium process may be partially defined by

process medium (**input** in_buf: T;
output out_buf: T) = **pending**

The declaration above contains only a process header and states that the actual definition is pending. This expresses the fact that we are unconcerned with the internal structure of the medium. All that we care about at this point is that it has an input buffer and an output buffer of some type T.

Observe from Figures 2 and 3 that there is nothing that distinguishes the form of the sender and receiver processes from that of the medium processes. We intend the sender and receiver to be sequential processes but at the external process interface this information is hidden. It is precisely this characteristic of the model that allows us to easily build hierarchical process structures. We do, however, provide a way to define sequential processes; it will be described in a later section along with the details of the sender and the receiver.

We have chosen to use message passing as the sole means of interprocess communication. Calls on protocol services are being modeled by send and receive primitives. This is a useful abstraction for service primitives that are actually realized in a multitude of ways within implementations of real protocol modules. More importantly, the behavior of protocol modules can fruitfully be specified in an event driven manner, where an event corresponds to the reception of a message from an input buffer. The simplicity and elegance of this technique is largely due to the use of a straightforward message passing discipline.

3 Specification Language Concepts

Formal specification requires the use of a language for referring to data objects and functions of data objects. This is the functional language component mentioned earlier. Naturally, a formally defined semantics is a prerequisite for a language that is to be used in conjunction with automated verification tools. It is beyond the scope of this paper, however, to give a rigorous definition of our specification language. Instead, we briefly and informally describe the highlights of the functional language component. This language serves a dual purpose: it is used both for expressing the definition of a protocol as well as for expresing assertions about it. We will discuss the basic concepts in this section and subsequently introduce specific operators as needed.

Many of the basic concepts about data types are drawn from Gypsy. Gypsy, in turn, has incorporated many ideas from Pascal. We adopt much of the syntactic and semantic flavor of these languages here, although we emphasize that we are *not* defining a programming language. Our goal is to develop a language in which it is possible to express operations on *mathematical objects* in a *nonprocedural* way and to express the kinds of relations among objects that are encountered in assertions.

A primary motivation for the development of this language was the desire to express generic concepts and properties. This characteristic is viewed as essential for constructing a reuseable theory of a given problem domain. A sufficiently general concept or property can be used repeatedly for different problems, or can be instantiated in different ways for the same problem. To this end, two main generic mechanisms have been incorporated into the language: parameterized data types and functional arguments, that is, passing names of functions as parameters in a function call. In this paper we will not concern ourselves much with the former but we will be dealing with the latter insofar as we will introduce several predefined functions that have functional arguments.

Our universe of discourse consists of typed data objects and functions on those data objects. A data type may be a simple data type or a structured data type. The simple data types include boolean, integer (and its subtype natural) and Pascal-like enumeration types. The structured data types are record, sequence and mapping. Records coincide with the Pascal notion of record; sequences and mappings are similar to the corresponding types in Gypsy. We will discuss the concepts of sequence and mapping in some detail, but we rely on the reader’s familiarity with Pascal for an understanding of the other types.

A sequence type may be defined by a type declaration such as

type S = sequence **of** T

where T is the element type. An object of type S is a finite sequence of elements of type T. The basic operators for constructing sequences are as follows:

[e1, ..., en]   explicit construction
[]   or   null   empty sequence
S <: e   or   S apr e   append element right
e :> S   or   e apl S   append element left
S1 @ S2   or   S1 join S2   append sequence

We also have several extraction functions on sequences.

first(S) / last(S)   first/last element of S
nonfirst(S) / nonlast(S)   tail/ head of S

There are additionally some relational operators on sequences, which will be introduced as needed. The sequence data type plays a central role in the verification methods, as will be seen later.

The mapping data type may be thought of as a generalization of an array with integer subscripts. A mapping data object consists of a set of ordered pairs that define a correspondence between certain integers and elements of some type T. Actually, it is convenient to use a sequence representation for mappings. An object of type

type M = mapping **of** T

is represented by an object with the structure

sequence **of** record (dom: natural; rng: T)

Such an object consists of a finite sequence of pairs, each having a *domain* value and a *range* value. This sequence is further constrained to be ordered by the domain values, which are unique natural numbers. An example of a mapping value would be

M = [[1,a], [2,b], [4,c], [9,d]].

Note that gaps in the domain values may exist.

Since mappings are just a special type of sequence we can make use of the sequence functions to operate on mappings as sequences of domain-range pairs. In addition, we have provided operators to perform mapping specific functions. The basic constuctor function for mappings is one which adds a new domain-range pair to a mapping. This function is given the following syntactic form:

M with ([i] := e)

The value of this construction is a new mapping containing the pair [i,e] at the appropriate place. If a pair with domain value i already exists, it is replaced by the new one. For the sample mapping shown above we have

M with ([5] := x) = [[1,a], [2,b], [4,c], [5,x], [9,d]]
M with ([2] := x) = [[1,a], [2,x], [4,c], [9,d]].

To extract objects from a mapping there also are several functions.

M[i]   range value for domain value i
domain(M)   sequence of domain values
range(M)   sequence of range values

Below are some examples of the use of these functions.

M[1] = a   M[4] = c
domain(M) = [1, 2, 4, 9]
range(M) = [a, b, c, d]

Because of the definition of mapping, the domain function necessarily returns a monotonically increasing sequence of natural numbers. Additional operators will be discussed in later sections.

Earlier we mentioned the desire to make use of functional arguments to functions. The intention is to provide certain predefined functions that take both normal data objects and names of functions as parameters. These functions will typically take a sequence argument and a unary function argument and apply the function to the elements of the sequence to produce a new object. The simplest example of this is a function we call "apply," whose effect is

apply (f, S) = [f(S_1), ..., f(S_n)].

This function produces a new sequence from S by applying the function f to each element of S. It is similar to the MAPCAR function of LISP. By providing functions of this sort, we hope to free the user from having to define his own recursive functions.

All functions described in this paper are predefined operators in our language. The generic features of these functions are intended to make them reuseable for other problems. Space limitations preclude our discussing these and other important concepts here. A mechanism for defining functions is also part of the language. Rules for passing parameters are a necessary part of that mechanism. In the presence of functional arguments it is particularly important to ensure that user defined functions cannot cause the resulting theory to be inconsistent. These and other issues are taken up in [DiVito82-2].

We close this section with an example of some of the foregoing concepts drawn from the Stenning protocol. Referring back to Fig. 2, recall that the objects flowing from the source and those flowing to the sink are "messages," the exact type of which is not of concern to us. However, the objects transmitted from the sender to the receiver are "packets," which are defined to have the following type.

type packet = record (mssg: message; seqno: natural)

A packet is a record of two fields: a message field and a sequence number field. A sequence of packets would then have the type

type pkt_seq = sequence **of** packet.

It is useful to be able to extract the message fields from a sequence of packets. This is easily done with the apply function. If S is an object of type pkt_seq, then

apply (".mssg", S)

will produce the sequence of messages extracted from S. Here we use the notation ".mssg" to denote the function of extracting the message field from a record of type packet.

4 Protocol Specifications

Let us now turn our attention to the problem of specifiying protocol behavior. This is a challenging problem from the standpoint of devising a method that achieves a suitable level of abstraction while still allowing us to capture the essential details that define a protocol. An often used technique for protocol specification is known as the "abstract program approach." This involves giving a procedural specification by exhibiting a set of procedures in a conventional high level programming language that implement the protocol modules. We take the position that this technique produces an overly detailed form of specification. For this reason we have sought a method that yields an intrinsically higher level of abstraction. The resulting method, termed *abstract behavioral specification*, can fulfill the needs of precise definition and analysis, while still serving as a basis for implementation.

Consider a sequential process of the kind described in Section 2. It is possible to give specifications for such a process by regarding its behavior as being *event driven*. By this we mean that the life of a process consists of a continuous series of event processing cycles. Each cycle begins with the occurrence of an event, then is followed by the processing of that event, and ends when the process goes back to wait for the next event. Two kinds of events are recognized: the reception of a message from one of the input buffers and an internally generated timeout event. The quiescent state for a process is to be waiting for a message from one or more of its input buffers.

The event driven view of process behavior leads to the adoption of a state transition paradigm for the specification of a protocol module. Local state information is assumed to be maintained by the process; this is collectively referred to as the *state vector*. Processing of an event causes a transition to occur, which updates the state vector. Similar models of protocol systems are in common use today. However, there are some important distinguishing features of our model.

First of all, the state vector is strictly local to a single process; there is no notion of a global state vector. This preserves the modular character of the specification and proof methods. Since a process communicates with its external environment through explicit output buffers, we must account for this interface in the behavioral specification. Accordingly, the actions of event processing are divided into only two categories.

1. *Response*. Messages may be sent to the output buffers; this constitutes a modification of the external environment.
2. *Transition*. The state vector is updated; this constitutes a modification of the internal environment.

The exact mechanism used to state a detailed specification, which is based on the use of decision tables, is the other major distinguishing feature of our method. Before introducing it, we must elaborate on the overall structure of the sequential process model.

```
process sender (inputs source: message;
                       ack_in: natural;
                output pkt_out: packet) =
begin
  state vector (unack: natural;
                next: natural;
                queue: mapping of packet;
                timing: boolean;
                to_time: natural)
    initially (0, 0, null, false, 0);

  events
    next - unack < send_window =>
      on receipt of mess from source
        handle by source_hdlr;
    true =>
      on receipt of ack from ack_in
        handle by ack_hdlr;
    timing =>
      after to_time handle by timeout_hdlr;
  end;
end;
```

Figure 4: Sender process.

4.1 Sequential process definitions

Figures 4 and 5 contain process definitions for the sender and receiver processes of the Stenning protocol example. These represent fairly abstract process definitions, having a common form which may be regarded as a process schema. There are three major parts to a sequential process definition.

1. The process header identifies the input and output buffers and the types of objects that flow through them. This is completely analogous to the concurrent process case.

```
process receiver (input   pkt_in: packet;
                  outputs sink: message;
                          ack_out: natural) =
begin
  state vector (next: natural;
                queue: mapping of packet)
    initially (0, null);

  events
    true => on receipt of pkt from pkt_in
          handle by pkt_hdlr;
  end;
end;
```

Figure 5: Receiver process.

2. The nature of the state vector is revealed by declaring it as a record-like abstract data object. Its components are named and typed using the abstract types provided in the specification language. Its initial value is also declared.

3. A list of event processing statements is provided. Each statement corresponds to a particular class of event, either a message reception or a timeout. Each statement refers to an *event handler*, which is a separate language structure refined at the next lower level.

Thus the process schema contains declarations for both data and control. The list of event processing statements may be thought of as a variation of the guarded command construction, which is embedded within a nonterminating loop. Each statement begins with a boolean guard that determines whether the given event is eligible for selection on a given cycle. In this way, it is possible to select a subset of the input buffers upon which to wait for the next event. Messages arriving to a buffer with a corresponding false guard will be queued but not received until the guard becomes true. Similarly, a timeout event with a false guard means that the timer is currently turned off for that event.

From Fig. 4 we see that the sender has a state vector with five components. The interpretation of these variables is as follows.

unack - sequence number of oldest outstanding message
next - next unused sequence number
queue - retransmission queue
timing - indicates when any messages are outstanding
to_time - absolute timeout time

The quantity (next - unack) is the number of outstanding messages and is used to control whether any new messages are accepted from the source. The send and receive windows are given by the constants send_window and rcv_window.

The receiver has two state vector components.

next - sequence number of next expected message
queue - receive queue

The receiver is driven entirely by incoming packets; it does not set itself any timeouts.

4.2 Event handlers

Next we describe how the details of event handling are specified. Each handler will in fact be represented by a special form of decision table. The decision table will indicate the appropriate response and transition actions to be performed under various conditions pertinent to the given event. This technique offers a way to state the specification using a precise and concise notation, while still allowing it to be done in a functional, nonprocedural manner. It also provides the advantages of structuring the specification in such a way that a limited analysis for consistency and completeness may be performed, as well as yielding a very straightforward procedure to generate verification conditions.

```
  source_hdlr           1 2
+======================+=====+
| timing               | F T |
+======================+=====+
| pkt_out              | A A |
+----------------------+-----+
| unack                | - - |
| next                 | B B |
| queue                | C C |
| timing               | T - |
| to_time              | D - |
+======================+=====+

Where
  A = [[mess, next]]
  B = next + 1
  C = queue with ([next] := [mess, next])
  D = time + delta_t
```

```
  ack_hdlr           1 2
+==================+=====+
| ack > unack      | T T |
| ack = next       | F T |
| timing           | T T |
+==================+=====+
| pkt_out          | - - |
+------------------+-----+
| unack            | A A |
| next             | - - |
| queue            | B C |
| timing           | - F |
| to_time          | - - |
+==================+=====+

Where
  A = ack
  B = upper (queue, ack)
  C = null
```

```
  timeout_hdlr    1
+===============+===+
| ---           |   |
+===============+===+
| pkt_out       | A |
+---------------+---+
| unack         | - |
| next          | - |
| queue         | - |
| timing        | - |
| to_time       | B |
+===============+===+

Where
  A = range (queue)
  B = time + delta_t
```

Figure 6: Sender event handlers.

```
  pkt_hdlr                          1 2 3 4
+================================+=========+
| pkt.seqno = next               | F - T T |
| pkt.seqno > next               | F T - - |
| pkt.seqno - next < rcv_window  | - T - - |
| pkt.seqno in domain(queue)     | - F - - |
| (next+1) in domain(queue)      | - - F T |
+================================+=========+
| sink                           | - - B D |
| ack_out                        | A - C E |
+--------------------------------+---------+
| next                           | - - H J |
| queue                          | - G - K |
+================================+=========+

Where
  A = [next]
  B = [pkt.mssg]
  C = [next+1]
  D = pkt.mssg :> apply (".mssg", range (consec (queue)))
  E = [reach(queue)]
  G = queue with ([pkt.seqno] := pkt)
  H = next + 1
  J = reach(queue)
  K = upper(queue, reach(queue))
```

Figure 7: Receiver event handler.

Event handler specifications for the sender and receiver processes can be found in Figures 6 and 7. Each decision table is in a special kind of extended entry format, where expressions are abbreviated by single letter entries which are defined below the table. For the benefit of the reader who may be unfamiliar with decision tables, we will discuss the semantics in some detail. Further information on decision tables can be found in [Metzner77].

```
                         Stub              Rules
              +=====================+============+
              |                     |            |
  Condition   |    Boolean          | T, F, or   |
    part      |    expressions      | don't care |
              |                     |            |
              +=====================+============+
              |    Output           |            |
              |    buffers          | Response   |
   Action     +---------------------+------------+
    part      |    State            |            |
              |    vector           | Transition |
              |    components       |            |
              +=====================+============+
```

Figure 8: General form of event handler decision table.

The general form for our protocol decision tables is depicted in Fig. 8. The rules (columns) of a table represent alternatives in the way an event is handled. Based on conditions that hold when an event occurs, one of the rules is selected for execution. The selected rule then dictates the set of actions that are performed in response to the event.

The conditions under which rules are selected are encoded in the upper part of the table. These rows contain boolean expressions that refer to the values of the state vector components and the value of the message just received from the corresponding input buffer (if applicable). A rule is eligible for selection if the conjuction of conditions satisfies the truth value entries given in the column. If more than one rule is so eligible, then one is selected nondeterministically. If none is eligible, then no actions are performed; in other words, the event is ignored. Sometimes this is referred to as an implicit ELSE rule.

Actions to be performed by the selected rule are encoded in the lower part of the table. This part is split into two subsections: the response part and the transition part. In the response part, one row is provided for each output buffer of the process. An entry for one of these rows is an expression which evaluates to a *sequence* of messages to be sent to the coresponding output buffer. A null entry denotes the null sequence or no messages sent toward that buffer. In the transition part, there is one row for each state vector component. An entry here denotes the new value to be assumed by a state vector component. A null entry indicates that the value remains unchanged. Note that there is no significance to the relative ordering of rows and columns within a table. It is important to realize that the expressions refer to the old values of state vector elements; the total effect of a transition is that of a simultaneous assignment.

We have now outlined the basic technique for specifying the behavior of a sequential process. This method obviates the need to state such specifications in a highly procedural way. A simple example protocol does not do justice, however, to the notational power of the decision table. The advantage in conciseness is much more apparent when the event processing that must be specified is very complex.

Some of the functions in Figures 6 and 7 have yet to be described so we take up that task now. The function upper(M, i) returns the submapping of M in which the domain values are all greater than or equal to i. The function lower(M, i) is defined in an analogous manner. Another function for extracting a submapping is consec(M). It returns the maximal, initial submapping of M such that all of its domain values are consecutive integers. The next domain value in this sequence (the first missing value) is obtained from the function reach(M).

An informal explanation of the event processing for the sender and receiver of Figures 6 and 7 would no doubt help clarify the behavioral specifications. When the sender receives a message from the source, it immediately forms a packet out of it, sends the packet out and saves it on the retransmission queue. It then updates next and starts a timeout if necessary. The current value of time is obtained from the implicit variable "time." When a valid acknowledgment arrives, the sender updates unack and deletes the portion of the queue which is thereby acknowledged. When a timeout occurs, the entire queue is resent and a new timeout is started. For simplicity, only a single timeout is set at any given time, rather than one for every outstanding message.

Upon receipt of an incoming packet, the receiver must distinguish several cases based on where the packet lies in the sequence space. If its sequence number is less than next, then it is an old duplicate and the current value of next is sent back. If its sequence number is greater than next, but still within the window, then it is queued if a copy has not already been saved. When a sequence number equal to next arrives, then as many messages as can legitimately be delivered are sent to the sink. A new value of next is derived and sent as an acknowledgment, and the queue is updated to delete the messages just delivered.

4.3 Example of event processing

The foregoing concepts are illustrated below with an example from the incoming packet handler of Fig. 7. Suppose that the receiver process is currently waiting for message number 6 and has several other messages already in its receive queue.

next = 6
queue = [[7,[b,7]], [8,[c,8]], [10,[e,10]]]
domain(queue) = [7, 8, 10]

Now suppose that the packet [a,6] arrives through the receiver's pkt_in buffer. This causes rule 4 of the decision table to be selected. Some of the functions that must be evaluated then include

consec(queue) = [[7,[b,7]], [8,[c,8]]]
reach(queue) = 9
upper(queue, 9) = [[10,[e,10]]]
range(consec(queue)) = [[b,7], [c,8]]
apply(".mssg", range(consec(queue))) = [b, c]

Thus the response to this incoming packet will be

1. Send [a, b, c] to sink
2. Send [9] to ack_out

and the transition will be

1. Set next to 9
2. Set queue to [[10,[e,10]]]

5 Service Specifications

Up to this point, we have not introduced any concepts explicitly for the purpose of doing verification. In fact, the techniques described thus far would be useful for protocol specification regardless of whether we intended to carry out any verification. Nevertheless, we would like very much to extend our formal machinery to be able to prove properties about the protocols we model. The main property of interest is some formal statement of the services provided by a protocol.

Our method for expressing a service specification is based on the use of an assertion about the input-output characteristics of a concurrent process. Such an assertion takes an external view of a process; it only makes use of information that can be gleaned from external observations of the process. This type of assertion is to be held invariant by a process. Hence it is referred to as an *external invariant*.

In order to express these assertions we make use of a special mechanism known as a *history*. This is an instance of the more general class of *auxiliary variables*, a well established tool for the verification of concurrent programs. Our notion of history follows that of [Howard76] and [Good79]. A message history (or buffer history) records all of the messages received from an input buffer or those sent to an output buffer during the life of a process. The history itself is a sequence data object. For example, the transport service process of Fig. 1 would have an input history associated with the buffer "source" and an output history associated with the buffer "sink." Each of these has the data structure

type msg_history = sequence **of** message.

Notationally, we follow the simple convention of giving a history the same name as the buffer to which it applies (as long as there is no ambiguity).

So we are now in a position to state the service specification for the Stenning protocol as a safety property of the transport process. In words, it is the usual property that the output history is an initial subsequence of the input history. This is expressed quite simply as

sink initial source

where "initial" is an infix relational operator and sink and source are the history names. In general, service specifications are stated as a conjunction of relations between histories or expressions involving histories.

This type of assertion has been referred to as an invariant, but of course it cannot be true at all instants of time. It is only required to hold when a process is idle, waiting for input. A concurrent process is idle when all of its subprocesses are idle. Hence the assertion actually holds at a restricted set of points in time. Note that this notion of invariant assertion is different from both the Gypsy concept of blockage assertion, which holds when a process is blocked waiting on inputs *or* outputs, as well as other forms of invariants that must hold after every separate I/O operation. An advantage of this form of assertion is that stronger relations between histories can generally be stated since the assertions need not hold at so many intermediate states.

Stating service specifications in this manner has the advantage that any data delivery property we wish to prove about a protocol can be expressed directly as its service specification. In contrast, if a state machine form of specification is used, it is necessary to derive the delivery properties from the specification as a separate step. On the other hand, connection management features are more difficult to capture using a history-based specification. With the restricted histories described in this paper it would not even be possible. However, the more general process model provides a way to specify the relative order of messages in different histories. This gives us the needed capability, but it remains to be seen how well it will work in practice.

6 Verification Methods

We have thus far presented a hierarchical model of concurrent processes, techniques for specifying the behavior of protocol modules, and a method for stating formal service specifications. Now we tie all of these things together by explaining the verification methods. These are based in part on the verification techniques developed for Gypsy, where the concepts for modular verification of concurrent processes were first introduced [Good79]. Other sources of methods include the traditional principles of Floyd-Hoare inductive assertion techniques.

```
              Transport service
             [external invariant]
                ^      ^      ^
                |      |      |
       +----------+    |     +----------+
       |               |                |
     sender          medium           receiver
    [external       [external        [external
     invariant]      invariant]       invariant]
       ^            (assumed)           ^
       |                                |
     sender                           receiver
    [internal                        [internal
     invariant]                       invariant]
       ^                                ^
       |                                |
     sender                           receiver
  [behavioral                      [behavioral
   specification]                   specification]
```

Figure 9: Proof structure for Stenning protocol.

6.1 Verifying concurrent processes

In Section 5 the concept of the external invariant of a process was discussed. The only data objects that may be referenced by such an assertion are the histories of the input/output buffers of the process in question. No knowledge of the internal structure of a process is used. This is the technique that enables a modular proof structure to be obtained. When verifying a concurrent process, we only make use of the external invariants of its subprocesses. The conjunction of these invariants, plus knowledge of the topology of process interconnection through message buffers, must together imply the external invariant of the parent process. One level of the concurrent process hierarchy is thereby verified by this procedure. The overall proof structure is depicted in Fig. 9 for the Stenning protocol example.

```
+---------+                                           +---------+
|         | out_hist   +--------+  in_hist  |         |
| Process |----------->| buffer |---------->| Process |
|    P    |            +--------+           |    Q    |
+---------+                                           +---------+

                 in_hist initial out_hist
```

Figure 10: The buffer axiom.

Interconnection topology gives us the "glue" with which to bind the separate subprocess invariants during a proof. Specifically, we must be able to relate the input history and the output history for each interconnecting buffer. The designations input history and output history are with respect to the *processes* (Fig. 10). The dual convention would be just as valid, but we find it more natural to take a process-centric view of the world. The basic axiom for buffers is that the input history for a buffer is an initial subsequence of the corresponding output history. Often this relation can be strengthened to equality, depending on the behavior of the receiving process. If a process always waits on a particular buffer when it is idle, then whenever the external invariant holds the buffer must be empty, and therefore these histories must be equal. Extensions to account for the waiting behavior of processes are included in the more general model [DiVito82-2].

For the Stenning protocol example, the transport service process would be verified by proving the verification condition (VC) shown in Fig. 11. External invariants are represented by functions with names of the form "p_ext." Arguments to these functions consist of the appropriate histories for the input and output buffers of the processes. For interconnecting buffers, input and output histories are designated by "b_in" and "b_out" for buffer b. Notice that four instances of the buffer axiom are included in this VC. If an automatic proof system were cognizant of this special relationship between histories, they could be left out of the VC entirely. In that case, these relationships would be implicitly supplied when they were needed in the proof. Actually, these relations can all be strengthened to equality for this example, but we wanted to display the general form. The invariants themselves will be presented in Section 6.3.

```
    sender_ext (source, sndr_ack_in, sndr_pkt_out)
  & receiver_ext (rcvr_pkt_in, sink, rcvr_ack_out)
  & medium_ext (sndr_pkt_in, rcvr_pkt_out)
  & medium_ext (rcvr_ack_in, sndr_ack_out)
  & sndr_pkt_in initial sndr_pkt_out
  & rcvr_pkt_in initial rcvr_pkt_out
  & sndr_ack_in initial sndr_ack_out
  & rcvr_ack_in initial rcvr_ack_out
-> transport_service_ext (source, sink)
```

Figure 11: Concurrent process verification condition.

6.2 Verifying sequential processes

Both concurrent and sequential processes are given an external invariant. As just noted, we prove that the external invariant of a concurrent process holds by examining its internal structure, namely the subprocesses that compose it. An analogous procedure is needed for proving that the external invariant of a sequential process holds. An intermediate assertion is used for this purpose, the *internal invariant*. It differs from the external invariant only in the data objects that it may reference. An internal invariant may refer to the state vector components of a sequential process as well as to its buffer histories. Otherwise, it is required to hold at precisely the same points in time as the external invariant. This assertion may be regarded as the analog of the loop invariant in the inductive assertion method for sequential program verification.

The internal invariant now acts as a bridge between the behavioral specification and the external invariant, as shown in Fig. 9. This structure gives rise to three kinds of verification conditions.

1. The internal invariant must be shown to imply the external invariant.
2. The internal invariant must be shown to hold with the initial value of the state vector and null buffer histories.
3. It must be shown that the internal invariant is maintained after an event is processed. This involves proving that for each event class, and for each decision table rule within that class, the actions performed will preserve the invariance of the assertion. This must also be shown for the ELSE rules of the decision tables.

Because of the inherent simplicity of these proof methods, it is easy to design an algorithm that will generate the proper verification conditions from a suitable representation of the behavioral specifications. A rough prototype has in fact been implemented.

As an example of the first kind of VC, we have for the receiver process

```
  receiver_int (pkt_in, sink, ack_out, next, queue)
-> receiver_ext (pkt_in, sink, ack_out)
```

where receiver_int stands for the internal invariant. Similarly,

```
receiver_int (null, null, null, 0, null)
```

represents the initialization VC for the receiver.

The third kind of verification condition is where most of the work lies. As an illustration of one of these VCs, consider Fig. 12. This VC corresponds to rule 4 of the receiver's incoming packet handler, containing the most complicated processing of this protocol. It has the general form

$$
\begin{aligned}
&\text{invariant } (H_1, \ldots, H_m, V_1, \ldots, V_n) \\
&\ \& \text{ conditions under which event and rule are selected} \\
&\text{-> invariant } (H_1\text{<:}p,\ H_2@R_2, \ldots, H_m@R_m,\ T_1, \ldots, T_n)
\end{aligned}
$$

where the $\{H_i\}$ are histories, the $\{V_i\}$ are state vector components, the $\{R_i\}$ are response expressions, the $\{T_i\}$ are transition expressions and p is the value of the incoming packet. Of course, those histories belonging to the other input buffers do not change, since only one message can be received during a single event.

```
    receiver_int (pkt_in, sink, ack_out, next, queue)
  & pkt.seqno = next
  & (next + 1) in domain(queue)
-> receiver_int (pkt_in <: pkt,
                 sink @ (   pkt.mssg
                        :> apply(".mssg",
                                 range(consec(queue)))),
                 ack_out @ [reach(queue)],
                 reach(queue),
                 upper (queue, reach(queue)) )
```

Figure 12: One of the receiver's verification conditions.

6.3 Invariants for the example protocol

Returning to our example protocol, we will present a set of invariants for the various processes. Let us begin with the external invariants of the medium, sender and receiver processes. The medium process can be characterized by a very simple assertion. Since it delivers messages with a certain degree of unreliability, the most that can be said about it is that any message appearing at its output must have been received from its input. However, nothing can be said about the ordering of messages or the number of occurrences of a given message that appear at the output. This relationship between histories is expressed by

```
msg_out follows msg_in.
```

Here "follows" is a relational operator on sequences like initial, except that it is a much weaker relation. The predicate x follows y holds if every element of x appears somewhere in y. If x and y were reduced to sets by removing duplicates, then the follows relation would be equivalent to the subset relation. When applied to sequences, though, follows may hold even if x is larger than y.

The external invariant of the sender process is given by

```
  consistent (".seqno", "=", pkt_out)
& source = apply (".mssg",
          range (latest (".seqno", pkt_out))).
```

Here a couple of new functions are encountered that need to be explained. The predicate "consistent" is used in this case to express the fact that any two packets in the pkt_out history with the same sequence number field also have the same message field. This fact is needed in the proof of the receiver since it allows the receiver to treat all packets with the same sequence number interchangeably. More generally, if f is a unary function of objects of type T, g is a binary predicate on T, and S is a sequence of T, then consistent(f, g, s) means

```
for all x, y in S, f(x) = f(y) -> g(x, y).
```

The function "latest" is used to create a mapping object from a sequence. Let S be sequence of T and f be a unary, integer valued function on T. A call on latest(f, S) first applies f to each element e of S to create the domain-range pairs [f(e), e] and then creates a mapping out of these by retaining the rightmost (latest) pair of those with duplicate domain values. For example,

```
pkt_out = [[a,0], [b,1], [a,0], [c,2]]
latest (".seqno", pkt_out) =
    [[0,[a,0]], [1,[b,1]], [2,[c,2]]].
```

In this way, we can create a value which is an ordered packet stream with all the duplicate entries removed. By extracting the message fields we recover the original sequence of messages received from the source.

The external invariant for the receiver is somewhat more complicated.

```
   consistent (".seqno", "=", pkt_in)
-> if 0 in domain (latest (".seqno", pkt_in))
     then         sink
          initial apply (".mssg",
                        range(consec(latest(".seqno",
                                            pkt_in))))
     else sink = null
   fi.
```

The implication reflects the fact that the receiver can only deliver the correct messages if the incoming packets have the consistency property. Its conclusion gives the relation between input and output histories, as was done for the sender’s invariant. In this case, the additional complexity results from the fact that an incoming packet may have arrived at a time when its sequence number fell beyond the receive window. The receiver would have ignored it but it still would have appeared in the input history. Hence the messages actually delivered to the sink may lag those that could have been delivered based on purely logical considerations. Observe that in all of these invariants there is no mention of the acknowledgment path from receiver to sender. Due to the unboundedness of sequence numbers in our model of this protocol, sequence numbers are not being reused and hence the proof is independent of the acknowledgment handling.

```
  consistent (".seqno", "=", pkt_out)
& range(queue) follows pkt_out
& source = apply (".mssg",
                range (latest (".seqno", pkt_out)))
& if next = 0
    then   ptk_out = null
         & queue = null
    else   pkt_out ne null
         &   last (latest (".seqno", pkt_out)).dom
           = next - 1
         &   highest (apply (".seqno", pkt_out))
           = next - 1
         & (   queue ne null
            -> last(queue).dom = next - 1)
  fi
```

Figure 13: Sender’s internal invariant.

We conclude by presenting the internal invariants for the sender and receiver processes of Figures 13 and 14. We make no attempt at annotation; they are shown only to satisfy the reader’s curiosity.

```
    consistent (".seqno", "=", pkt_in)
 -> (         queue
      follows upper (latest (".seqno", pkt_in), next + 1)
    & next le reach (latest (".seqno", pkt_in))
    & if 0 in domain (latest (".seqno", pkt_in))
      then   next > 0
           &   sink
             = apply (".mssg",
                    range (lower (latest (".seqno",
                                    pkt_in),
                             next - 1)))
      else   next = 0
           & sink = null
      fi )
```

Figure 14: Receiver's internal invariant.

7 Mechanical Proof

One of the major goals for this research was to attempt to apply mechanical theorem proving tools to the problem of protocol verification. Moreover, this goal has directly influenced the development of parts of the methodology, primarily the design of the specification language. Although the technology of automatic theorem proving is still far from being truly automatic, it is felt that the coming years will bring steady progress so that mechanical proving will become reasonably practical. In the meantime, there is no denying the benefits of a machine proof in terms of offering an extremely high degree of proof integrity.

In our present work, we have done a fair amount of experimentation with the Boyer-Moore theorem prover [Boyer79]. A fully mechanical proof of the protocol example in this paper was carried out with this theorem prover. The prover has been used extensively on program verification problems of various kinds. It runs as an Interlisp program on DEC 10 and DEC 20 systems. Formulas are expressed in a notation similar to LISP S-expressions. The prover's two most important features for our work are its ability to carry out induction proofs and its ability to search a database of lemmas during a proof. During a proof there is no direct interaction with the user. A user does interact, albeit indirectly, by causing an adequate set of lemmas to be proved beforehand.

If we were to try to prove a set of verification conditions from scratch our chances of success would be very slim indeed. Effective use of the prover requires that we build a theory of our basic concepts that is sufficiently rich to allow proofs to be of manageable size. The strategy employed to achieve this aim is outlined below.

- The basic data types and functions of our specification language are realized by shells and recursive functions in the Boyer-Moore theory. Shells are a form of data typing mechanism in this theory.
- A set of fundamental properties of these functions is postulated and proved. Most of these are properties of the sequence and mapping operators, which are usually proved by induction on the structure of one of the sequence variables. A body of generally useful lemmas is thereby built up.
- Proofs of the verification conditions are attempted without resorting to the use of induction. We would like the prover to rely only on the previously proved lemmas and its simplification heuristics.
- Study of failed proof attempts leads to the conception of additional lemmas which are then proved and added to the knowledge base. This process continues until all of the verification conditions have been proved.

This strategy was successfully applied in our proof of the Stenning procotol. All told, there were 14 nontrivial verification conditions to be proved. Their proofs took a total of about 28 minutes of CPU time on a DEC 2060 system with a KL processor. At the time, a database of some 300 lemmas had been constructed. About 200 of these were left activated for automatic search by the theorem prover.

Having developed this sizeable body of lemmas, we now have the rudiments of a reuseable theory for the verification of transport protocols. Most of these lemmas have been expressed as fairly general facts about the predefined operators and consequently may be brought to bear on similar verification problems. The next protocol we verify will undoubtedly benefit from the existence of this database and a great deal of duplicate effort can be avoided.

We would like to give the reader a feel for the kind of proofs we are talking about and demonstrate how the lemmas interact to achieve them. We will take one of the VCs and give an outline of its proof. The VC we have chosen is the one for the concurrent process, shown in Fig. 11. It states that the external invariants of the sender, receiver and medium processes together imply the external invariant of the transport process.

Rather that working with the original VC, let us simplify it somewhat, eliminate irrelevant information and factor out the trivial cases. This will leave us with the following simpler and more interesting conjecture.

```
      consistent (".seqno", "=", pkt_out)
    & pkt_in follows pkt_out
    & 0 in domain (latest (".seqno", pkt_in))
    &       sink
      initial apply (".mssg",
                   range (consec (latest (".seqno",
                                          pkt_in))))
 ->       sink
    initial apply (".mssg",
                 range (latest (".seqno", pkt_out)))
```

We have also done some renaming of the history variables to improve readability.

This conjecture can be proved by the application of five lemmas from our database. These lemmas are in the form of conditional rewrite rules. When applying such lemmas, the theorem prover makes use of a technique known as "backchaining." In this procedure, the prover finds a lemma whose conclusion it would like to use, instantiates the variables of the lemma, and then tries to establish each of the hypotheses of the lemma. While trying to establish the hypotheses, additional applications of lemmas may be attempted, requiring even more hypotheses be established. If the backchaining terminates successfully, then the original lemma may be used to rewrite the conjecture.

Proceeding with our proof, we would first like to try to use the transitivity lemma

```
   x initial y  &  y initial z
 -> x initial z
```

under the assignment

```
 x = sink
 z = apply (".mssg",
           range (latest (".seqno", pkt_out)))
```

Since the last hypothesis of our conjecture will satisfy the first hypothesis of the lemma, assuming we properly instantiate the free variable y, our conjecture will be proved if we can establish

```
       apply (".mssg",
             range (consec (latest(".seqno", pkt_in))))
 initial apply (".mssg",
             range (latest (".seqno", pkt_out)))
```

Now we may backchain with the lemmas

```
   x initial y
-> apply (f, x) initial apply (f, y)

   x initial y
-> range(x) initial range(y)
```

to yield the new goal

```
     consec (latest (".seqno", pkt_in))
initial latest (".seqno", pkt_out)
```

Backchaining again with

```
   x follows y  &  0 in domain(x)
-> consec(x) initial y
```

means that all we need to do is establish

```
     latest (".seqno", pkt_in)
follows latest (".seqno", ptk_out)
```

However, by applying

```
   x follows y  &  consistent (f, "=", y)
-> latest (f, x) follows latest (f, y)
```

we will have satisfied all our backchained hypotheses and therefore the original conjecture will have been proved. To keep the above presentation simple, we ignored the type constraints on the variables of the lemmas. In actuality, this is additional information that must be established when applying lemmas. The real theorem prover takes about 30 seconds to find the proof of the original verification condition from which this example was extracted.

8 Conclusion

We have described a methodology for formally specifying and verifying communication protocols. It is based on the use of concurrent process models and abstract techniques for specifying protocol behavior. Techniques for verifying safety properties have been developed based in part on the methods of Gypsy. The origin of a deductive theory of formal protocol concepts has evolved. The methodology has been used successfully in a trial application, yielding a fully mechanical proof of the Stenning protocol. Although the emphasis has been on protocols, it is clear that the methods could be applied to other problems of distributed systems.

Future work will attempt proofs of more complicated protocols. A prime candidate is the data transfer function of the DoD’s Transmission Control Protocol (TCP) [Postel80]. This would be primarily an extension of the example protocol in this paper. Also desirable would be attempts to model connection management features of protocols. Future development of methods should include several types of extensions. A way to deal with bounded sequence numbers is an extension that would make the protocol models more realistic. This would involve placing limits on the degree to which a medium can reorder packets. Of particular importance is the need to extend the methods to handle verification of liveness properties. A natural choice would be the use of temporal logic, but this may present a problem for mechanical proof. Finally, developing a specialized protocol verification system would be a helpful aid in managing verification activities.

I. Collected Specifications for Example

The protocol specifications for the example used in this paper have been collected and reproduced below.

II. Process definitions

```
process transport_service (input  source: message;
                           output sink: message) =
begin
  buffers (sndr_pkt, rcvr_pkt: packet;
           sndr_ack, rcvr_ack: natural);

  cobegin
    sender (source, sndr_ack, sndr_pkt);
    receiver (rcvr_pkt, sink, rcvr_ack);
    medium (sndr_pkt, rcvr_pkt);
    medium (rcvr_ack, sndr_ack);
  end
end


process medium (input  in_buf:  T;
                output out_buf: T) = pending


process sender (inputs source: message;
                       ack_in: natural;
                output pkt_out: packet) =
begin
  state vector (unack: natural;
                next: natural;
                queue: mapping of packet;
                timing: boolean;
                to_time: natural)
    initially (0, 0, null, false, 0);

  events
    next - unack < send_window =>
      on receipt of mess from source
        handle by source_hdlr;
    true =>
      on receipt of ack from ack_in
        handle by ack_hdlr;
    timing =>
      after to_time handle by timeout_hdlr;
  end;
end;
```

```
  source_hdlr           1 2
+=====================+=====+
| timing              | F T |
+=====================+=====+
| pkt_out             | A A |
+---------------------+-----+
| unack               | - - |
| next                | B B |
| queue               | C C |
| timing              | T - |
| to_time             | D - |
+=====================+=====+

Where
  A = [[mess, next]]
  B = next + 1
  C = queue with ([next] := [mess, next])
  D = time + delta_t
```

```
  ack_hdlr           1 2
+==================+=====+
| ack > unack      | T T |
| ack = next       | F T |
| timing           | T T |
+==================+=====+
| pkt_out          | - - |
+------------------+-----+
| unack            | A A |
| next             | - - |
| queue            | B C |
| timing           | - F |
| to_time          | - - |
+==================+=====+

Where
  A = ack
  B = upper (queue, ack)
  C = null
```

```
  timeout_hdlr    1
+===============+===+
| ---           |   |
+===============+===+
| pkt_out       | A |
+---------------+---+
| unack         | - |
| next          | - |
| queue         | - |
| timing        | - |
| to_time       | B |
+===============+===+

Where
  A = range (queue)
  B = time + delta_t
```

process receiver (**input** pkt_in: packet;
outputs sink: message;
ack_out: natural) =
begin
state vector (next: natural;
queue: mapping of packet)
initially (0, null);

events
true => **on receipt of** pkt **from** pkt_in
handle by pkt_hdlr;
end;
end;

```
  pkt_hdlr                          1 2 3 4
```

```
+================================+=========+
| pkt.seqno = next               | F - T T |
| pkt.seqno > next               | F T - - |
| pkt.seqno - next < rcv_window  | - T - - |
| pkt.seqno in domain(queue)     | - F - - |
| (next+1) in domain(queue)      | - - F T |
+================================+=========+
| sink                           | - - B D |
| ack_out                        | A - C E |
+--------------------------------+---------+
| next                           | - - H J |
| queue                          | - G - K |
+================================+=========+

Where
  A = [next]
  B = [pkt.mssg]
  C = [next+1]
  D = pkt.mssg :> apply (".mssg", range (consec (queue)))
  E = [reach(queue)]
  G = queue with ([pkt.seqno] := pkt)
  H = next + 1
  J = reach(queue)
  K = upper(queue, reach(queue))
```

III. External invariants

```
transport_service:

    sink initial source



medium:
s    msg_out follows msg_in



sender:

      consistent (".seqno", "=", pkt_out)
    & source = apply (".mssg",
                      range (latest (".seqno", pkt_out)))



receiver:

       consistent (".seqno", "=", pkt_in)
    -> if 0 in domain (latest (".seqno", pkt_in))
         then         sink
              initial apply (".mssg",
                             range(consec(latest(".seqno",
                                                 pkt_in))))
         else sink = null
       fi
```

IV. Internal invariants

```
sender:

      consistent (".seqno", "=", pkt_out)
    & range(queue) follows pkt_out
    & source = apply (".mssg",
                      range (latest (".seqno", pkt_out)))
    & if next = 0
        then   ptk_out = null
             & queue = null
        else   pkt_out ne null
             &   last (latest (".seqno", pkt_out)).dom
               = next - 1
             &   highest (apply (".seqno", pkt_out))
               = next - 1
             & (   queue ne null
                -> last(queue).dom = next - 1)
      fi



receiver:

       consistent (".seqno", "=", pkt_in)
    -> (          queue
          follows upper (latest (".seqno", pkt_in), next + 1)
        & next le reach (latest (".seqno", pkt_in))
        & if 0 in domain (latest (".seqno", pkt_in))
            then   next > 0
                 &   sink
                   = apply (".mssg",
                            range (lower (latest (".seqno",
                                                  pkt_in),
                                          next - 1)))
            else   next = 0
                 & sink = null
          fi )
```

V. Summary of Predefined Functions

A glossary of the predefined functions used in this paper is given below.

e apl S	Also written e :> S. Appends element e to the left end of sequence S.
apply(f,S)	Applies the function f to each element of sequence S, collecting the result into a new sequence.
S apr e	Also written S <: e. Appends element e to the right end of sequence S.
consec(M)	Extracts the maximal initial submapping of M such that all its domain values are consecutive integers.
consistent(f,g,S)	A predicate that is true if for every two elements x, y of S, f(x) = f(y) -> g(x,y).
domain(M)	Returns the sequence of domain values of mapping M.
first(S)	Evaluates to the first element of sequence S, or to a default value if S is null.
S1 follows S2	A relational operator on sequences that holds if every element of S1 is also an element of S2.
e in S	A predicate that is true if e is an element of sequence S.
S1 initial S2	A relational operator on sequences that holds if S1 is an initial segment of S2.
S1 join S2	Also written S1 @ S2. Returns the concatenation of sequences S1 and S2.
last(S)	Evaluates to the last element of S, returning a default value if S is null.
latest(f,S)	Creates a mapping result from the sequence S by applying function f to each element e, forming pairs [f(e),e] and collecting the rightmost pair of those with identical domain values.
lower(M,i)	Extracts the submapping of M in which all domain values are less than or equal to i.
nonfirst(S)	Returns the sequence obtained from S by removing the first element.
nonlast(S)	Returns the sequence obtained from S by removing the last element.
range(M)	Extracts the sequence of range values from mapping M.
reach(M)	Computes the integer that is one plus the highest domain value of consec(M).
upper(M,i)	Extracts the submapping of M in which all domain values are greater than or equal to i.
M with ([i]:=e)	Produces a new mapping from M by inserting the pair [i,e] at the appropriate place, replacing any existing pair having the same domain value.

References

Table of Contents

List of Figures