Daniel Schwabe

Formal Specification and Verification of a Connection-Establishment Protocol
## Formal Specification and Verification of a Connection-Establishment Protocol

### Abstract

This paper presents an exercise in the verification of a connection-establishment protocol. A specification language named SPEX, tailored to the needs of communications protocols, is proposed, and its relation to a semi-automated verification system, Affirm, is discussed. This language is then used to specify a connection protocol currently being used. Certain errors are uncovered by analysis using the verification system. However, the major portion of the protocol's operation is shown to be correct.
Daniel Schwabe

Formal Specification and Verification of a Connection-Establishment Protocol

UNIVERSITY OF SOUTHERN CALIFORNIA

This research is supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0008. Views and conclusions contained in this report are the author's 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

ACKNOWLEDGMENTS .................................................................................................................. iv  

1. INTRODUCTION .................................................................................................................. 1  
   1.1 Connection-Establishment Protocols ............................................................................. 1  
   1.2 Overview of SPEX ......................................................................................................... 3  
   1.3 Overview of Algebraic Specification of Data Types and of Affirm ............................... 6  
   1.4 Relation to Other Work ............................................................................................... 9  

2. SPECIFICATION OF THE THREE-WAY HANDSHAKE IN SPEX ........................................ 11  

3. VERIFICATION .................................................................................................................... 15  
   3.1 Introduction .................................................................................................................. 15  
   3.2 Functional Correctness ............................................................................................... 15  
   3.3 Liveness ..................................................................................................................... 20  

4. CONCLUSIONS ..................................................................................................................... 23  

1. SPEXIFICATION OF THE THREE-WAY HANDSHAKE ....................................................... 25  

II. AXIOMS GENERATED FROM THE SPEXIFICATION OF THE THREE-WAY HANDSHAKE .......... 33  
   II.1 Three-Way Handshake ............................................................................................... 33  
   II.2 Auxiliary Data Type Definitions ............................................................................... 40  

REFERENCES .......................................................................................................................... 43  

# FIGURES

Figure 1-1: Three-way handshake state-transition diagram ..................................................... 4  
Figure 1-2: Signature of type QueueOfInteger ......................................................................... 7  
Figure 1-3: Some axioms for type QueueOfInteger .................................................................. 8  
Figure 3-1: Proof tree for the functional correctness of the three-way handshake ..................... 17  
Figure 3-1: Proof tree (continued) ....................................................................................... 18  
Figure 3-2: Theorems and definitions used in the proof of the three-way handshake .............. 19  
Figure 3-3: Example of a liveness error in the three-way handshake ....................................... 22
ACKNOWLEDGMENTS

I wish to thank Carl Sunshine for his constructive criticism of the work presented in this report and careful review of the report itself; Jon Postel for taking the time to answer all those naive questions about TCP; the members of the Program Verification group at ISI, in particular Susan Gerhart, David Thompson and Rod Erickson for the discussions while the work was being developed and for making Affirm such a convenient tool to use. I also thank Danny Cohen, whose support made it possible for me to work at ISI. My graduate studies at UCLA were supported by CAPES-Brazilian Government under contract 1247/76.
1. INTRODUCTION

Computer networks are becoming increasingly widespread; their use already permeates our everyday life. As a consequence, their correct functioning becomes paramount. Given that computer networks are extremely complex systems, the task of certifying that they behave properly is nontrivial.

This report presents an exercise in verifying that a particular algorithm to realize an important function in computer networks, namely connection establishment, does indeed behave properly. The methods discussed are applicable for analyzing a wide range of other network functions as well.

The remainder of this section gives background material. Section 1.1 discusses the nature and need for connection establishment in computer networks; Section 1.2 then presents a new language suitable for the specification of protocols; and Section 1.3 describes a system in which properties of such specifications can be proved.

Section 2 presents a specification of a connection protocol currently in use, given in the language introduced earlier. Section 3 then discusses particular properties of this protocol and shows their verification.

1.1 Connection-Establishment Protocols

This section presents the motivation for connection-establishment protocols in general and for the three-way handshake used in the ARPANET in particular. Consider a distributed system with several interconnected nodes. The nodes are connected by an unreliable transmission medium in which messages may be lost or duplicated, and each node has several processes. Imagine now that two processes wish to communicate; a common method to overcome the possible loss of data is to attach a sequence number to each data packet that flows, in either direction, between them. If the two nodes can agree on a starting number to be used, again in each direction, then they will be able to detect packets arriving out of order or being duplicated.

Suppose now that the system, when it is created, initializes the nodes to have agreed-upon sequence numbers, thus allowing the data transfer to take place immediately. Unfortunately, such systems are impractical, for a number of reasons.

First, since the system is intended to be distributed, a failure at one node would require the whole system to be re-initialized. Second, although there is a potential for communication between any two processes in the system, only a few pairs will actually be engaged in data exchange at any one time. Since the resources needed to maintain communication between processes are quite significant, the nodes should be able to keep these resources allocated only while the exchange is taking place, thus increasing their utilization.

The reader familiar with the three-way handshake may skip this section.

1
These considerations lead to the notion of connections. When two processes wish to communicate, the corresponding nodes will cooperate among themselves to establish a common frame of reference, e.g., sequence numbers for data flowing in each direction, for the exchange of data; when the exchange is complete, the connection is closed, freeing the resources for use by other processes. The period of time that a particular connection is open between two processes, i.e., the period of time a particular frame of reference is in effect, is called an incarnation of that connection.

It is clear that for the exchange of data to be successful, the two nodes must agree on the state of the connection. A further problem is introduced by the fact that the transmission medium may delay or duplicate packets that flow between the two nodes. Since connections can open and close, it is possible for packets from old incarnations to be in the medium; when they are present, they should not be mistaken for packets belonging to a newly opened connection.

Since packets may be lost, a positive-acknowledgment retransmission-on-timeout scheme is used. In other words, the sender keeps a copy of each packet sent until the receiver acknowledges that the packet has been received. If no acknowledgment arrives after some predefined amount of time, it is assumed that the packet or its acknowledgment was lost and it is retransmitted. Acknowledgments themselves are not acknowledged.

It is important to note that if there is a positive probability (no matter how small) that a packet is lost, then it is actually impossible to completely separate the connection-establishment from the data transfer itself. To see why, consider the last (synchronization) packet exchanged during the connection establishment; each node will consider the connection to be open upon sending and receiving this packet. It is clear that the node receiving this packet can be sure that the other node has a compatible view of the connection. The sender, however, cannot be so sure, given the possibility that this last packet may be lost; only when the first data packet arrives (in the reverse direction) will it be sure that the other node actually received it. Therefore, the sender node must maintain both the data exchange and the connection-establishment information for that period of time. An equivalent problem is discussed in [2].

In many systems, connections are opened and closed quite frequently. Since the medium may duplicate packets, it is possible for a connection-request packet from a previous incarnation to appear at one node at such a time as to be mistaken for a current one, thereby initiating a connection with the wrong frame of reference (see [15]).

A problem still remains as to how to identify packets from previous incarnations as being old. The sequence numbers chosen to establish the frame of reference of a new connection must prevent that. Sunshine, in [15], discusses this issue in more detail.

A protocol has been proposed to handle the connection-establishment problems discussed so far. It is called the three-way handshake [19, 15]. The particular version used here is taken from TCP [12], the second-generation transport-level protocol being used in the ARPA internet system.
This protocol derives its name from the sequence of steps a node goes through in order to establish a connection. Suppose that node A wishes to communicate with node B and that node A takes the initiative. The two nodes then go through the following steps:

1. Node A sends node B a connection request, called SYN (for SYNchronize).

2. Node B receives the SYN packet, and responds with a SYN of its own together with an acknowledgment, together called SYNACK (for SYNchronize and ACKnowledge).

3. Node A receives the SYNACK packet, verifies that the ACK portion does indeed acknowledge its own previous SYN, and sends an ACK packet acknowledging node B's SYN. At this point, node A considers the connection to be opened.

4. Node B receives the ACK packet, verifies that it does acknowledge its own previous SYN, and then considers the connection to be opened.

There are two basic modes for opening a connection: an active mode, in which the issuing node takes the initiative; and a passive mode, in which the issuing node merely listens for incoming connection requests, and accepts the first to come in. The basic protocol described above can be modified to handle the case when both nodes do an active open simultaneously.

If at any point an incorrect packet arrives, then a RST (reset) packet is sent back to abort the connection-opening procedure.

Figure 1-1 contains a state-transition diagram taken from [12]. It does not show transitions caused by RST or incorrect packets.

1.2 Overview of SPEX

We present here an overview of a language, called SPEX, to be used for the specification of a distributed system in general and computer networks in particular. This language will be used later to describe the three-way handshake protocol. As will be evident from the details given below, the underlying model in SPEX is that of a nondeterministic state-transition system, with some specialized features to facilitate protocol specification. SPEX is discussed at greater length in [13].

A system is regarded as consisting of a set of interconnected Nodes. In the case of the example presented here, a Node can be a Station or a Medium. The pattern of interactions of the nodes constitutes the layer's definition. A particular pattern of behavior characterizes a node's type. A system may in general be composed of several distinct types of nodes, each with its own behavior, and may have several instances of each type of node as well.

Thus, in order to completely characterize a system, it is necessary to describe the behavior of each type of node (given in the Node Behavior part of the specification), the set of instances of each node type and the way
the instances are interconnected (given in the Topology part), and the desired properties of the interactions between the instances (given in the Properties part). In addition, the specification of any data types used in specifying a node’s behavior must be included.

A node is some entity that has some internal State Variables and some externally visible Interface Variables; these variables may be of arbitrarily complex data types (which may be defined using algebraic data type specification methods [9, 7, 8, 5]). A node reacts to a set of specified Events. When one such event occurs, some state variables and some interface variables may have their values changed.

State variables can be accessed only locally at each node. Interface variables, on the other hand, can be accessed from the outside—this is how a node communicates with the outside world, i.e., other nodes in the same system or other systems using the system in which the node is defined. Accordingly, the interface variables at each node are divided into two kinds: those that are exported to other systems, and those that are
A CONNECTION-ESTABLISHMENT PROTOCOL

connected to other nodes in the same system. In addition, each interface variable may have a direction of data flow associated with it, meaning that data in that variable flows into or out of a node; if no direction is specified, data in that variable flows in both directions.

The actual behavior of a node is given by describing how a node reacts to the occurrence of certain specified events. Each event known at a node has a precondition associated with it; this precondition is a predicate involving state and interface variables at that node. As long as a precondition is true, its associated event is said to be enabled; enabled events may fire at any time.

The node's behavior is given in terms of the new values of all its variables when each of the possible events occurs. All changes for an event are considered to happen simultaneously, i.e., the events are considered atomic. This means that if any variable $X$ is used to compute the new value of some variable, the value used in the computation is the value $X$ had before the event happened. For brevity's sake, if a variable is not mentioned on the left-hand side of any event-effects statement, then its value is not changed by the occurrence of that event.

Since state variables are not visible externally, they can be regarded as history variables [11] which accumulate information about the computation.

Since interface variables are externally visible, it is possible for an event $eI$ at some node NI to change the value of some interface variable at another node, say N2. In fact, $eI$ may actually enable some event at N2; this is effectively how nodes exchange data and synchronize their activity.

The last item necessary to completely describe a node's behavior is its Initial State, specifying the values of any variables at system-creation time. The most general way to specify the initial state is to give predicates which must be true in the initial state; it may not be necessary or even possible to give actual values to the variables.

All of the above must be specified for each node type that exists in the system.

The overall system behavior specified is defined as the set of all valid sequences of events. A valid sequence is formed by starting from an initial state (i.e., a state satisfying the initial state predicates) and successively firing enabled events; it may be of infinite length. If it is of finite length, then the final state arrived at by executing the sequence has no enabled events.

Once all node types have been specified, it is necessary to describe how the several nodes are connected. This is achieved by allowing interface variables at each node to be connected to interface variables at other nodes; the intended semantics is that these are in fact shared variables between the corresponding nodes.

The Topology part then specifies how the interface variables of each node in the system (i.e., each instance of each type of node) are connected to interface variables of the other nodes.
The *Properties* section states two kinds of properties of the protocol, *Assumed* and *Asserted* properties. Asserted properties are those that must be proved true by the specifier and serve as an additional check of the accuracy of the specification. In other words, proving these properties increases the confidence of the specifier that the specification corresponds to her/his intuitive understanding of the system.

Assumed properties are used to *define* certain operations in a noncomputational fashion by giving input-output relationships between arguments and returned values.

*SPEX*ifications\(^2\) can be conveniently translated into algebraic style data type specifications of the kind that are supported by the *Affirm* system (see Section 1.3). This capability can be exploited to prove properties of the protocol using analysis methods from the abstract data type specification domain or to perform a limited form of symbolic execution of the specification, which helps in determining the accuracy of the specification.\(^3\) This translation is discussed in detail in [13].

An overview of algebraic specification of data types and of *Affirm* is given in the next section.

### 1.3 Overview of Algebraic Specification of Data Types and of Affirm

The material presented in this section has been abridged from [4, 17].

*Affirm* [10] is an experimental system for the algebraic specification of and the verification of properties of user-defined abstract data types. The heart of the system is a natural deduction theorem prover for the interactive proof of these properties, which are stated in the predicate calculus extended with data types. Programs, written in a variant of Pascal extended with user-defined abstract data types, may be verified using the inductive assertion method [3]. Additional features include tools for the analysis of algebraic specifications, a library of useful data types, and user interface facilities. Experience with *Affirm* 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 security kernels.

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

Like other specification and verification systems, *Affirm* follows its own particular theoretical and programming paradigm--abstract data types specified algebraically and properties verified by rewriting rule techniques. A brief description of the algebraic style of data type specifications and of the theorem-proving portions of *Affirm* follows.

---

\(^2\) *SPEX*ification: will be used to mean *SPEX* specification.

\(^3\) i.e. whether the specification captures the designer's intuitive understanding of the system.
Following the algebraic style of data type specifications [9, 7, 8, 6, 5], 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 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—they are derived operators. These functions can be defined in terms of the constructors.

3. **Selectors.** These functions yield values of types other than the one being specified. The general term for these functions is selector, but functions yielding values of type Boolean are often termed predicates. These functions are defined in terms of the parameters of the constructors.

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). Observe that these extender functions can be defined in terms of the constructors NewQueue and Add. Example selector functions are Front, #Elements and in (a predicate). These are definable in terms of the parameters to Add.

The effect of such a specification is to view values of the type in terms of the constructors which can build them. Hence, 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

\[(\text{NewQueueOfInteger Add 1) Add 2) Add 3}\]

The first part of a specification gives the **signature** of all operations, i.e., their domains and their ranges. Figure 1-2 shows an example for the type QueueOfInteger.

```
declare q,q':QueueOfInteger;
declare i:integer;

interface NewQueueOfInteger, q Add i : QueueOfInteger;
interface Remove(q), Append(q,a') : QueueOfInteger;
interface #Elements(q), Front(q) : Integer;
interface i in q : Boolean;
```

**Figure 1-2: Signature of type QueueOfInteger**

The second part of a data type specification provides semantics for the operations whose domain and range information was given in the first part. Extenders and selectors are defined by equational axioms of the form
FORMAL SPECIFICATION AND VERIFICATION OF

\( lhs = rhs \) relating how each function behaves when applied to each of the constructors. Constructor functions are treated as primitive, unspecified operations.

Examples of axioms taken from a specification of the type \( \text{QueueOfInteger} \) are given in Figure 1-3.

\begin{verbatim}
axioms
    Remove(NewQueueOfInteger) = NewQueueOfInteger.
    Remove(q Add i) = if q = NewQueueOfInteger then q
                        else Remove(q) Add i.
    #Elements(NewQueueOfInteger) = 0,
    #Elements(q Add i) = #Elements(q) + 1;
    Append(q, NewQueueOfInteger) = q,
    Append(q1, q2 Add i) = Append(q1, q2) Add i.

Figure 1-3: Some axioms for type \( \text{QueueOfInteger} \)
\end{verbatim}

Data types in general have properties that the specifier may wish to prove. For example, "the number of elements in the concatenation of two queues is the sum of the number of elements in each queue." Formally, this property is stated as

\[
# \text{Elements(Append}(q,q')) = # \text{Elements}(q) + # \text{Elements}(q')
\]

Properties of a data type are proved using a method called \textit{structural induction} \cite{14, 7} which is based on the notion that all values of the data type can be produced by repeated applications of the \textit{constructor} functions. To prove a property \( P \) of all elements of a data type, it suffices to show that

1. It is true for the "base" cases—the constructors that produce values of the type without taking values of the type as arguments (e.g., \( P(\text{NewQueue}) \)).
2. Assuming \( P \) is true for some value \( q \), then it is also true for all values obtained by applying constructors to \( q \) (e.g., for all \( q, i \ P(q) \) implies \( P(q \ Add \ i) \)).

There is much more to specifying a data type specification than just giving a set of axioms. A good data type specification should provide the desired set of operations. These operations should have the expected (intuitive) properties. The axioms should also facilitate simple proofs. In other words, the type has an associated \textit{theory} that expresses properties derived from the axioms. (Building these theories is a mathematical art.) The main method of proof of such properties is induction, for which the \textit{schema} part of a type provides the proof structure.

\textit{Affirm} is not exactly a proof checker, nor is it a proof finder. The responsibility for finding and executing a proof strategy rests solely with the user. At each proof step, modifications are made to a system-maintained
proof structure. Then the rewriting rules of the data types of the program, together with the rules of
propositional logic, are applied to simplify the proposition currently being worked upon. In general, the user
is attempting to reduce a formula to a set of subgoals so simple that their proofs are immediate, i.e., can be
obtained by the system without further direction. Some example commands for carrying out proofs and their
effects are:

\textbf{try \textit{proposition}} Set up \textit{proposition} as the current goal.

\textbf{employ \textit{Induction}(v)}

\textit{Induction} is a user-defined schema for the type of induction desired and \textit{v} is the variable to
be induced upon. The proof structure is modified to show the various cases of the
induction.

\textbf{apply \textit{proposition}} Use \textit{proposition} as a lemma in the proof (\textit{proposition} must be proved or assumed
separately). A separate \textit{put} command instantiates the variables in the lemma to the proper
values in the current goal.

\textbf{suppose \textit{proposition}}

Break the current goal into two subgoals, one with the additional hypothesis \textit{proposition}
and the other with \textit{\neg \textit{proposition}}.

\textbf{split}

Break up the proposition at a designated spot into subgoals, e.g., the proposition \textit{H imp (C}_1
\textit{and C}_2\textit{)} can be split into the two propositions \textit{H imp C}_1\textit{ and (H and C}_1\textit{) imp C}_2\textit{.}

\textbf{replace}

Replace subexpressions with other subexpressions according to designated equalities in the
current proposition.

\textbf{invoke \textit{defn}}

Invoke a definition \textit{defn} that the user has made at some time.

The user can explore various avenues of proof until the proof is complete or until the conjecture is found to
be unprovable, at which point the proof of the corrected conjecture must be restarted or the bad proof steps
corrected.

Each theorem or intermediate proposition in \textit{Affirm} is represented by a named node in a directed acyclic
graph called the \textit{proof forest}. The proof of a theorem comprises a tree, whose named arcs represent \textit{Affirm}
commands and thus deductive steps. \textit{Affirm} checks for circularity within the current tree.

An example of an \textit{Affirm} proof is discussed in Section 3.

1.4 Relation to Other Work

There is a large body of work regarding techniques for specifying protocols. These include Petri nets (and
related graph models), formal languages, sequencing expressions, and (parallel) programming languages.
Much of this work is limited in expressive power, in the sense that specifications grow unproportionally large
as the complexity of the protocol being specified increases. Many also suffer from lack of a solid theory or of automated tools for verification. Sunshine [16] provides a survey of this work.

Although the underlying model of SPEX is not new, it is believed to be the first language allowing the formal specification of nondeterministic state transition systems in a modular, hierarchical fashion, and for which semi-automated verification tools exist. An important advantage of the modularization and the symbolic nature of the specification is that there is no combinatorial explosion when analyzing more complex protocols. Schwabe [13] provides an example in which a complex protocol, involving an arbitrary number of nodes, is specified and verified, but where the complexity of the proof is independent of the number of nodes.
2. SPECIFICATION OF THE THREE-WAY HANDSHAKE IN SPEX

This section examines a SPEXification of the three-way handshake protocol described informally in Section 1.1. Appendix I contains the actual text of the SPEXification.

First the state variables, interfaces, initial state, and events for one station are given; the main portion of the specification shows the behavior of the station for each event. A small specification for the medium is also given, stating that the medium is essentially a queue with an added LoseMessage event. In the sequel, a brief explanation of the SPEXification is given.

The three-way handshake protocol involves two nodes with identical behavior. The corresponding node type is Station.

Each station needs the following state variables:

ISS is some constant to be used as Initial Send Sequence number.

Incarnation#In
is an incarnation identification for the packets coming in from the other node.

Incarnation#Out
is an incarnation identification for the packets leaving this node.

OldUnack
is the sequence number of the oldest sent packet which has not yet been acknowledged.

Seq#ToSend
is the sequence number that should be attached to the next data packet to be sent.

Seq#ToReceive
is the expected sequence number of the next packet coming in.

TimeoutBuffer
is a queue of packets containing copies of packets which have been sent but not yet acknowledged.\(^4\)

The exported interface to using systems contains two variables:

Command
is a command buffer through which the user indicates what type of open request is desired.

\(^4\)Strictly speaking, TimeoutBuffer does not have to be a queue, but just a collection, of packets. Modeling it as a queue results in simpler axioms in this situation.
StateOf
is a variable that remembers the state of the station, i.e., somehow remembers the recent history of messages that have been exchanged. Its value can be one of \{Closed, Listen, SynSent, SynReceived, Established\}.

Each station has two interface variables which are internal to the system, namely:

InPort
is a queue of incoming packets, with possible loss.

OutPort
is a queue of outgoing packets, with possible loss.

The initial state of each station requires that the State of the station be Closed, the TimeoutBuffer be empty and the sequence numbers and incarnation number of incoming packets be zero.\(^5\)

The events to which a station can react are:

ActiveOpen
which is caused when the user issues an active open command. This means that a connection request will be sent to the other party.

PassiveOpen
which is caused when the user issues a passive open command. This means that the station will listen for incoming connection requests and accept the first one that comes.

Timeout
which is caused when a timeout occurs, i.e., when a certain amount of time has elapsed without a packet being acknowledged.

ReceiveRst
which is caused when a packet arrives whose control field is rst (reset). This is a control packet used to indicate the discovery of an anomalous situation.

ReceiveAck
which is caused when an acknowledgment packet arrives.

ReceiveSyn
which is caused when a packet arrives whose control field is syn (synchronize). This is a connection request.

ReceiveSynAck
which is caused when a packet which is both an acknowledgment and a connection request arrives.

\(^5\)Zero is used as an arbitrary initial value.
The node type representing the medium has only an interface variable, Buffer, which is a queue of packets. There is only one event that can happen, LoseMessage, which models the medium being faulty. Note that the transmit operation of the medium is modeled as an Add to the queue, and the receive operation is modeled as a Remove from the queue, with the packet delivered obtained by Front of the queue (before the Remove).

The definition of the data type Packet can be found in Appendix II. A brief description is given here.

The fields of a packet are the following:

**SeqNumber**

is the sequence number of the packet.

**Seq# Inc**

is the incarnation number associated with the sequence number.

**AckNumber**

is the sequence number that the packet is acknowledging.

**Ack# Inc**

is the incarnation number of the acknowledgment field.

**Cil**

is the control field of the packet.

As an illustration of the effects of an event, consider the ActiveOpen event (see page 26). Its precondition states that it can fire only if the StateOf the node is Closed, and the user issued an active open command by placing the value Active in the Command buffer. When this event fires, the effects specified state, for instance, that a SYN packet is sent to the other side by appending it to the OutPort interface variable. It is also specified that the StateOf state variable becomes SynSent.

Finally, the Topology section states that there are two stations, Left and Right, connected by a medium in each direction (i.e., OutPort@Left, Buffer@LeftToRight, and InPort@Right are all a single shared queue).

The Properties section states properties concerning the correct operation of the system that will be discussed in Section 3.

The SPEXification given in Appendix I is a simplification of the one given in TCP [12]. The main differences are:

- TCP allows connections between arbitrary pairs of addresses within a large address space. As in TCP, the SPEXification assumes this addressing function is performed by a higher (sub) level, so that only fixed pair of nodes need be considered.

- TCP uses a sequence number and an initial send sequence number selection algorithm to handle the problems of distinguishing incarnations. TCP sequence numbers correspond roughly to a
concatenation of incarnation and sequence numbers in our specification. TCP sequence numbers are of finite size, whereas they are of infinite size in the SPEXification.

- The SPEXification concerns itself only with the connection-opening phase of the protocol; it does not allow closing of the connection in the middle of an opening. Likewise, it does not allow data to be sent while a connection is being opened.

- When a RST packet arrives at a node that is in SYNSENT state, the TCP remembers whether the connection started via an active or via a passive open. If the open was passive, the station returns to the LISTEN state rather than closing the connection. The SPEXification always closes the connection after a reset. This modification does not affect the functional correctness of the protocol, but makes the corresponding SPEXification simpler.

For the purposes of verifying properties of the three-way handshake, the SPEXification has been manually translated into an algebraic data type specification that can be understood by the Affirm system. Appendix II contains the generated axioms and auxiliary data type definitions (e.g., Packet, QueueOfPacket) in Affirm syntax.
3. VERIFICATION

3.1 Introduction

This section discusses the verification of properties concerning functional correctness and liveness. The discussion is presented in terms of the algebraic style data type specification as understood by Affirm.

As was discussed in Section 1.1, the functional correctness of a connection protocol cannot be completely separated from the succeeding data transfer phase. This introduces a problem as to the instant at which the claim of functional correctness should be made. Ideally, functional correctness should state that

At the end of the connection phase, both stations are in the Established state and are synchronized, which means that "old" data will not be accepted, but "new" data will be.

Therefore, it would be necessary to describe at least part of the data transfer protocol as well.

Because the data transfer has been omitted from the specification, a modified version of this property must be used. The following sections describe this in more detail.

3.2 Functional Correctness

Consider now the functional correctness of the protocol, as stated above, but from only one node's point of view:6

(StateOf = Established)@Right
imp Seq # ToReceive@Left = Seq # ToSend@Right and
Incarnation # In@Left = Incarnation # Out@Right ;

In English, this says that if the station on the Right side is in the Established state, then the connection is synchronized for data flowing out of this node.

This property is proved to be invariant by inductive proof methods which are used for abstract data types. Work with this specification showed that this theorem was not strong enough to be used in an inductive proof, for the following reason. Careful study of the protocol shows that it is possible for the above properties to hold in the SynSent state also, when simultaneous active open commands are issued at both nodes, as follows: one side may be in the SynSent state and may already have received an acknowledgment for its SYN packet; this side would not enter the Established state until it receives the SYN packet from the other side. This situation is characterized by the fact that OldUnack (the oldest unacknowledged sequence number) is not ISS anymore. Since this side has received an acknowledgment for its SYN, it can be sure that the other side knows its Seq # ToSend and its Incarnation # Out.

---

6The notation P@n means P is to be evaluated in node n.
Hence the statement of functional correctness must be strengthened (for one side only) as follows:

**Theorem FC:**

\[
\begin{align*}
&((\text{StateOf} = \text{Established}) \text{ or } ((\text{StateOf} = \text{SynSent}) \text{ and } \text{OldUnack} = \text{ISS})) @ \text{Right} \\
&\quad \text{imp} \\
&\quad \text{Seq} # \text{ToReceive}@ \text{Left} = \text{Seq} # \text{ToSend}@ \text{Right} \text{ and} \\
&\quad \text{Incarnation} # \text{In}@ \text{Left} = \text{Incarnation} # \text{Out}@ \text{Right};
\end{align*}
\]

This need to strengthen or generalize a theorem in order to prove its invariance is typical of inductive proof methods used for abstract data types.

Notice that this strengthened statement implies the weaker one, so that proving the stronger one proves the weaker one as well.

Figure 3-1 contains a proof tree for this theorem produced by the *Affirm* system; the lemmas and definitions used are given in Figure 3-2 (these figures contain axioms and theorems stated using *Affirm* syntax; the correspondence to *SPEX* syntax should be obvious). The proof follows an inductive argument over all possible events in the system. Broadly speaking, this can be expressed as the following: given a goal state (e.g., Established), examine how each event can move the system into that state (e.g., ReceiveAck event in SynReceived state). In general, there are many states from which the system may move into the goal state. Considering now each of those states, one uses the inductive hypothesis to try to prove the theorem.

After some examination of the proof tree, it is possible to see that most cases follow directly from the inductive hypotheses; this can be seen in the proof tree by looking at the branches and noticing where only an *invoke IH* command (possibly preceded or followed by some *replace, cases* and *invoke* commands) was given. Now the cases are examined which do not follow directly from the inductive hypotheses, i.e., involve the application of some lemmas.

Consider what happens when a ReceiveAck@Right occurs (\(\leftrightarrow 1\)). The relevant case to consider has the node at right in SynSent or in SynReceived, and the incoming acknowledgment has the current incarnation number (since otherwise the packet would be discarded as old). In other words, the incarnation number in the packet is equal to Incarnation#Out@Right. (See hypotheses of theorem AcksAndSyns in Fig. 3-2, applied at \(\leftrightarrow 2\).) But if the incarnation number is current, then there must have been a SYN packet in the past which this current packet acknowledges (see definition of HasSyn, invoked at \(\leftrightarrow 3\)). Thus, the current ACK carries the same incarnation number that the SYN carried, which means that the station at left has its Incarnation#In set to the incarnation number of that SYN packet. Therefore, we can conclude that Incarnation#Out@Right = Incarnation#In@Left.

Numbers on the left should be ignored; they result from bookkeeping in *Affirm*.

Indicators of the form \(\leftrightarrow n\) are used to point to the corresponding places in the proof tree.
A CONNECTION-ESTABLISHMENT PROTOCOL

Theorem Synchronized. \[ \text{StateOf}(S, \text{Right}) = \text{Established} \]
or \[ \text{StateOf}(S, \text{Right}) = \text{SynSent} \text{ and } \text{OldUnack}(S, \text{Right}) = \text{ISS}(\text{Right}) \]
implies \text{Synchronized}(S).

Synchronized uses \text{EorSSimpEorSR}, \text{SynchNoLoCorSS}, \text{AcksAndSyns}, \text{FrontInQ}, \text{Seq#ToSendVals}, \text{and Seq#ReceiveVal}.

Proof tree:

66: \text{Synchronized}
   \text{apply EorSSimpEorSR} \text{ proved by Schwabe using Affirm 120 on 4-Feb-81 in transcript \langle SCHWABE Affirmtranscript.3-FEB-81.2\rangle}
70: \text{put } S' = S
74: \text{employ \text{Induction}(S)}
   \text{Empty: Immediate}
76: \text{apr: }
56 \text{employ NormalForm(i1S)}
78: \text{ActiveOpen:}
57 \text{cases}
80: \text{66 invoke IH}
82: \text{66 replace}
84: \text{67 invoke synchronized [all [proven!]
221: \text{PassiveOpen: \{Synchronized, apr:}
58 \text{cases}
223: \text{126 invoke IH}
225: \text{126 invoke synchronized [all [proven!]
225: \text{(proven!)}
227: \text{LossMessage: \{Synchronized, apr:}
59 \text{invoke IH}
229: \text{128 invoke synchronized [all [proven!]
231: \text{Timeout: \{Synchronized, apr:}
60 \text{invoke IH}
233: \text{130 invoke synchronized [all [proven!]
233: \text{(proven!)}
235: \text{ReceiveRst: \{Synchronized, apr:}
61 \text{cases}
237: \text{131 employ NormalForm(i'1')}
239: \text{Left:}
132 \text{invoke IH}
241: \text{134 invoke synchronized [all [proven!]
243: \text{Right:}
133 \text{invoke IH}
253: \text{136 invoke synchronized [all [proven!]
253: \text{(proven!)}
88: \text{ReceiveAck: \{Synchronized, apr:}
62 \text{cases}
90: \text{69 employ NormalForm(i'1')}
92: \text{Left:}
70 \text{invoke IH}
94: \text{72 invoke synchronized [all [proven!]
94: \text{(proven!)}
96: \text{Right:}
71 \text{invoke IH}
98: \text{replace}
105: \text{76 invoke synchronized [all [proven!}
107: \text{76 apply AcksAndSyns}
109: \text{77 put pk = Front(Medium(ss', Left))}
111: \text{78 apply FrontInQ}
113: \text{79 put Q = Medium(ss', Left)}
116: \text{80 replace}
121: \text{81 invoke PreCond [all [proven!}
123: \text{82 apply Seq#ToSendVals [all [proven!}
125: \text{83 put S = ss'}
127: \text{84 invoke IncomingAck#Valid [all [proven!}
129: \text{85 invoke HasSyn [all [proven!}
131: \text{86 replace}
133: \text{87 apply Seq#ReceiveVal [all [proven!}
135: \text{88 put S = ss'}
138: \text{(proven!)}

Figure 3-1: Proof tree for the functional correctness of the three-way handshake.
137: \textbf{ReceiveSyn}:\{\textbf{Synchronized, apr:}\}  \hfill \leftarrow \llap{以前}\rightarrow 6

139: \hfill 90 \text{ invoke IH}

141: \hfill 91 \text{ replace}

143: \hfill 92 \text{ invoke synchronized } | \text{ all } |

145: \hfill 93 \text{ cases}

149: \hfill 94 \text{ replace}

151: \hfill 95 \text{ apply } \text{SynchNoLorCorSS}  \hfill \leftarrow \llap{以前}\rightarrow 7

153: \hfill 96 \text{ put } S=ss'

155: \hfill 98 \text{ replace } \\
\text{(proven!)}

167: \textbf{ReceiveSynAck}:\{\textbf{Synchronized, apr:}\} \\
\hfill 64 \text{ cases}

169: \hfill 99 \text{ employ NormalForm('')}

171: \hfill \textbf{Left:}

173: \hfill 100 \text{ invoke IH}

175: \hfill 102 \text{ invoke synchronized } | \text{ all } |

177: \hfill 103 \text{ cases}

179: \hfill 104 \text{ replace}

181: \hfill 105 \text{ apply } \text{SynchNoLorCorSS}

183: \hfill 106 \text{ put } S=ss'

185: \hfill \text{(proven!)}

187: \hfill \textbf{Right:}\{\textbf{Synchronized, apr:}, \textbf{ReceiveSynAck:}\} \\
\hfill 101 \text{ invoke IH}

185: \hfill 103 \text{ invoke synchronized } | \text{ all } |

187: \hfill 109 \text{ apply } \text{AcksAndSyns}

189: \hfill 110 \text{ put } S=ss'

191: \hfill 111 \text{ and } \text{pk} = \text{Front(Medium(ss', Left))}

193: \hfill 112 \text{ put } Q = \text{Medium(ss', Left)}

195: \hfill 113 \text{ replace}

197: \hfill 114 \text{ invoke } \text{IncomingAck#Valid} \ | \text{ last } | \text{ PreCond } | \text{ 1 } |

201: \hfill 116 \text{ invoke HasSyn}

203: \hfill 117 \text{ invoke PreCond}

205: \hfill 118 \text{ replace}

207: \hfill 119 \text{ apply } \text{Seq#ToSendVals}

211: \hfill 120 \text{ put } S=ss'

213: \hfill 121 \text{ replace}

215: \hfill 122 \text{ apply } \text{Seq#ToReceiveVal}

217: \hfill 123 \text{ put } S=ss'

219: \hfill 124 \text{ replace } \\
\text{(proven!)}

\textbf{Figure 3-1: Proof tree (continued)\hfill}
A CONNECTION-ESTABLISHMENT PROTOCOL

Figure 3-2: Theorems and definitions used in the proof of the three-way handshake
For the sequence numbers to correspond, it suffices to see that, if the state of a node is not Listen or Closed, then its Seq#ToSend is always equal to ISS + 1 (Seq#ToSend will not change until data is sent -- see theorem Seq#ToSendVals, applied at \( \Leftarrow \leftrightarrow 4 \)), and that all SYN packets carry ISS as their sequence numbers. Since the Seq#ToSend is taken from the SYN packet, it must perforce be ISS + 1 (see theorem Seq#ToReceiveVals, applied at \( \Leftarrow \leftrightarrow 5 \)). Therefore Seq#ToReceive@Left = Seq#ToSend@Right.

The next relevant case is when a ReceiveSyn@Left occurs (\( \Leftarrow \leftrightarrow 6 \)). This can be correct only if the node at left is in either Listen or SynSent: all other cases either cause an error or ignore the packet. But a careful examination of the state machine shows that it is not possible to have the station at one side in either Listen or SynSent, and the other in either Established or in SynReceived with OldUnack \( \neq \) ISS (theorem SynchNoLorCorSS. applied at \( \Leftarrow \leftrightarrow 7 \)). Therefore this situation really cannot occur.

The other relevant cases are when a ReceiveSynAck occurs at either node. If it happens at the node at right, then the proof follows the same argument as the case for the ReceiveAck@Right. If it happens at the node at left, then the proof follows the reasoning for the case ReceiveSyn@Left.

### 3.3 Liveness

Another useful property that this protocol possesses is Liveness, which states that either some event in the system is enabled or the system is in its final state. Since open events are user generated, these events are ignored, and we assume that the system starts in a state where neither side is in the Closed state and both sides are not passively listening. In this case, it is expected that the correct protocol will complete the connection-establishment and reach a final global state in which both sides have reached the Established state.

In order to prove such a property, however, it is necessary to prevent certain sequences from actually being valid for the system. These are sequences composed entirely of LoseMessage or Timeout events. Such sequences reflect fairness assumptions on the medium, as well as finite capacity. Thus, restrictions must be made in the specification to insure the fairness of the medium. These restrictions are incorporated by including a limit on the number of occurrences of the LoseMessage event, as well as on the size of the medium.

Accordingly, the number of occurrences of the LoseMessage event is limited by having an extra auxiliary counter such that LoseMessage can be enabled only when the counter is positive, and each time LoseMessage fires it decreases the counter by one. It is set to some constant value each time a message or an acknowledgment is received. This constant value must be finite, but can be arbitrarily large.

The capacity of the medium can be taken into consideration by augmenting the precondition of all events that put something into the medium with a test to see if the length of the corresponding queue is less than a certain constant, which again must be finite but arbitrarily large. This rules out behaviors in which a node times out over and over, without anything else happening in the system.
With these modifications introduced, an attempt was made to prove that this protocol is alive, i.e., it satisfies

**Theorem Liveness:**

For all $S_i$

\[
\neg \text{PreCond}(S, \text{Receive}XX) \wedge \neg \text{PreCond}(S, \text{Timeout}) \wedge \\
\neg \text{PreCond}(S, \text{LoseMessage}) \wedge \text{StateOf} = \text{Closed} \cap_i \\
\neg (\text{StateOf} \cap_i = \text{Listen} \wedge \text{StateOf} \cap_{\text{Opposite Side}}(i) = \text{Listen}) \implies \\
\text{StateOf} \cap_{\text{Left}} \text{Established} \wedge \text{StateOf} \cap_{\text{Right}} \text{Established}
\]

where $XX = \{\text{Ack}, \text{Syn}, \text{SynAck}, \text{Rst}\}$.

An inductive proof goes through for all cases except for ReceiveRst. After some investigation, it was found that there is a scenario in which it is possible for the two nodes to end in the Closed state, which is a contradiction of the theorem! Figure 3-3 shows this scenario (with SEQ treated as a single item representing both the sequence number and the incarnation number).

This situation is considered an error because old duplicate packets in the medium prevent a connection from being established. Note that this is a liveness error, not a safety error, since nothing bad happens, i.e., no incorrect synchronization or data transfer takes place, but the intended progress does not occur.

Another situation in which there is no progress may occur because of the introduced protocol simplification that a node always returns to Closed state when a RST packet arrives. Note that this is not the scenario described above.

Interestingly, if data packets are allowed to be sent, this scenario can be continued in such a way that it actually accepts data incorrectly. It is sufficient for the appropriate old data packets to arrive at Node A at the point Node A entered the Established state and before any RST packets were sent by Node B; this is indicated in Figure 3-3. However, it should be noted that this situation depends on an extremely unlikely timing of message exchanges, which is not expected to be of practical significance.

This incorrect data can be avoided with a small change in the protocol. Work is under way to verify that a corrected version of the three-way handshake avoids it.

In [1], Berthomieu discusses the verification of other types of liveness properties in algebraically described state transition systems.
Figure 3-3: Example of a liveness error in the three-way handshake
4. CONCLUSIONS

This report has presented an exercise in the verification of properties of a connection-establishment protocol. A specification language tailored to the need of communications protocols has been proposed, and its relation to a semi-automated verification system discussed. This language was then used to specify a connection protocol currently being used, and certain errors were uncovered using the verification system, although the major portion of the protocol's operation was shown to be correct.

This work is part of an ongoing project to develop better protocol specification and analysis techniques; further work is described in [13, 18]. Our preliminary experience indicates that the combination of state transition and abstract data type specification methods being pursued provides a reasonably convenient and powerful approach to these problems.
I. SPECIFICATION OF THE THREE-WAY HANDSHAKE

Node(Station):

State Variables
- ISS,
- Incarnation # In,
- Incarnation # Out,
- OldUnack,
- Seq # ToSend,
- Seq # ToReceive
: Nat,

| TimeoutBuffer : QueueOfPackets, |
| Initial Send Sequence # |
| Incarnation # of incoming packets |
| Incarnation # of outgoing packets |
| Oldest unacknowledged seq. # |
| Seq # to put in the next outgoing packet |
| Next expected seq. # |
| Nat stands for Natural |
| buffer with packets sent and not acknowledged |

Interfaces

Exported:
- Command : Command,
- StateOf : SysState,

Internal:
- InPort,
- OutPort
 :QueueOfPackets;

Initial State

| Incarnation # Out = Maxval(InPort Append OutPort) and Maxval produces a unique value |
| see Properties section |
| Incarnation # In = 0 and |
| Seq # ToSend = 0 and |
| Seq # ToReceive = 0 and |
| StateOf = Closed and |
| OldUnack = 0, |
| TimeoutBuffer = NewQueueOfPackets |

Events

| Events and their preconditions |
| ActiveOpen : PreCond is StateOf = Closed and Command = Active, |
| PassiveOpen : PreCond is StateOf = Closed and Command = Passive, |
| Timeout : PreCond is TimeoutBuffer = NewQueueOfPackets, |
| ReceiveRst : PreCond is InPort = NewQueueOfPackets and Control(Front(InPort)) = rst, |
| ReceiveAck : PreCond is InPort = NewQueueOfPackets and Control(Front(InPort)) = ack, |
| ReceiveSyn : PreCond is InPort = NewQueueOfPackets and Control(Front(InPort)) = syn, |
| ReceiveSynAck : PreCond is InPort = NewQueueOfPackets and Control(Front(InPort)) = synack |

| One of (Active, Passive, Null) |
| State of this side of the connection |
| msgs coming in |
| msgs going out |
Behavior

[First we define some auxiliary predicates and
t-functions to improve readability of the specification]

define IncomingAck # Valid =

( AckNumber(Front(InPort)) = + OldUnack) and
Ack # Inc(Front(InPort)) = Incarnation # Out;

define IncomingSeq # Valid =

( SeqNumber(Front(InPort)) = Seq # ToReceive) and
Seq # Inc(Front(InPort)) = Incarnation # In;

ActiveOpen::
Command + Null,
Incarnation # Out = Maxval(InPort Append OutPort),
OldUnack = ISS,
Seq # ToSend = + ISS
StateOf = SynSent
TimeoutBuffer =

NewQueueOfPackets Add pkt(ISS,Maxval(InPort Append Outport),AnyNat,AnyNat,syn)
OutPort =
Outport Add pkt(ISS,Maxval(InPort Append Outport),AnyNat,AnyNat,syn)

PassiveOpen::
Command = Null,
StateOf = Listen,
TimeoutBuffer = NewQueueOfPackets;

ReceiveRst::
StateOf =
if StateOf = SynSent and IncomingAck # Valid
then Closed
else if StateOf = Listen
then Listen
else if IncomingSeq # Valid
then Closed
else StateOf,
TimeoutBuffer ←
if StateOf = SynSent and IncomingAck # Valid
  then NewQueueOfPackets
else if IncomingSeq # Valid
  then NewQueueOfPackets
else TimeoutBuffer,

InPort ← Remove(InPort),

Receive Ack::
OldUnack ←
if StateOf = SynSent
  then if IncomingAck # Valid
    then + OldUnack
    else OldUnack
  else if StateOf = SynReceived
    then if incomingAck # Valid and IncomingSeq # Valid
      then + OldUnack
      else OldUnack
    else OldUnack,

StateOf ←
if StateOf = SynReceived
  then if IncomingAck # Valid and IncomingSeq # Valid
    then Established
    else SynReceived
  else StateOf,

TimeoutBuffer ←
if StateOf = Closed or StateOf = Listen
  then NewQueueOfPackets
else if StateOf = SynSent
  then if IncomingAck # Valid
    then DeletePacket(TimeoutBuffer, Seq # ToSend)
    else TimeoutBuffer
  else if StateOf = SynSent
    then if IncomingAck # Valid
      then DeletePacket(TimeoutBuffer, Seq # ToSend)
      else TimeoutBuffer
  else TimeoutBuffer
else TimeoutBuffer,
OutPort ←
  if StateOf = Closed or StateOf = Listen
  or ((StateOf = SynSent) and ¬ IncomingAck # Valid)
  then OutPort
    Add pkt(AckNumber(Front(InPort)),
           Ack # Inc(Front(InPort)),
           AnyNat, AnyNat, rst)
  else if StateOf = SynReceived
    then if ¬ IncomingSeq # Valid
      then OutPort
        Add pkt(Seq # ToSend, Incarnation # Out,
                Seq # ToReceive, Incarnation # In,
                ack)
      else if ¬ IncomingAck # Valid
        then OutPort
          Add pkt(AckNumber(Front(InPort)),
                   Ack # Inc(Front(InPort)),
                   AnyNat, AnyNat, rst)
      else OutPort
  else OutPort,
InPort := Remove(InPort);
ReceiveSyn::
  Incarnation # Out ←
    if StateOf = Listen
      then MaxVal(InPort Append OutPort)
    else Incarnation # Out,
Incarnation # In ←
  if ((StateOf = Listen) or StateOf = SynSent)
    then Seq # Inc(Front(InPort))
    else Incarnation # In,
OldUnack ←
  if StateOf = Listen
    then ISS
    else OldUnack,
Seq # ToSend ←
  if StateOf = Listen
    then + ISS
    else Seq # ToSend,
Seq # ToReceive ←
  if StateOf = Listen or StateOf = SynSent
    then + SeqNumber(Front(InPort))
    else Seq # ToReceive,
StateOf +
  if StateOf = Listen
   then SynReceived
  else if StateOf = SynSent
   then if OldUnack = ISS
   then SynReceived
   else Established
  else StateOf,

TimeoutBuffer +
  if StateOf = Listen
   then NewQueueOfPackets
      Add pkt(ISS,Maxval(InPort Append OutPort),
      + SeqNumber(Front(InPort)),
      ,Seq # Inc(Front(InPort)),
      synack)
  else if StateOf = Closed
   then NewQueueOfPackets
  else TimeoutBuffer,

OutPort +
  if StateOf = SynSent
   then OutPort
      Add pkt(Seq # ToSend,Incarnation # Out,
      + SeqNumber(Front(InPort)),
      ,Seq # Inc(Front(InPort)),
      ack)
  else if StateOf = SynReceived or StateOf = Established
   then if IncomingSeq # Valid
   then OutPort
  else OutPort
      Add pkt(Seq # ToSend,
      Incarnation # Out,
      Seq # ToReceive,
      Incarnation # In,
      ack)
  else if StateOf = Listen
   then OutPort
      Add pkt(ISS,Maxval(InPort Append OutPort),
      + SeqNumber(Front(InPort)),
      ,Seq # Inc(Front(InPort)),
      synack)
  else OutPort
      Add pkt(0',Incarnation # Out,
      + SeqNumber(Front(InPort)),
      ,Seq # Inc(Front(InPort)),
      rst),

InPort + Remove(InPort) ;
ReceiveSynAck::
  Incarnation # In +
  if (StateOf = SynSent) and IncomingAck # Valid
    then Seq # Inc(Front(InPort))
    else Incarnation # In,

OldUnack +
  if StateOf = SynSent
    then if IncomingAck # Valid
       then + OldUnack
       else OldUnack
    else if StateOf = SynReceived or StateOf = Established
       then if IncomingAck # Valid and IncomingSeq # Valid
          then + OldUnack
          else OldUnack
       else OldUnack,

Seq # ToReceive +
  if StateOf = SynSent
    then if IncomingAck # Valid
       then + SeqNumber(Front(InPort))
       else Seq # ToReceive
    else Seq # ToReceive,

StateOf +
  if StateOf = SynSent and IncomingAck # Valid
    then Established
    else StateOf,

TimeoutBuffer +
  if StateOf = Closed or StateOf = Listen
    then NewQueueOfPackets
  else if StateOf = SynSent
    then if IncomingAck # Valid
       then DeletePacket(TimeoutBuffer,OldUnack)
       else NewQueueOfPackets
       else TimeoutBuffer,
OutPort +
if StateOf = Closed or StateOf = Listen
then OutPort
   Add pkt(AckNumber(Front(InPort)),
   Ack # Inc(Front(InPort)),
   AnyNat, AnyNat,
   rst)
else if StateOf = SynSent
then if IncomingAck # Valid
   then OutPort
      Add pkt(Seq # ToSend, Incarnation # Out,
      + SeqNumber(Front(InPort)),
      Seq # Inc(Front(InPort)),
      ack)
   else OutPort
      Add pkt(AckNumber(Front(InPort)),
      Ack # Inc(Front(InPort)),
      AnyNat, AnyNat,
      rst)
else if StateOf = Established
then if IncomingSeq # Valid
then OutPort
else OutPort
   Add pkt(Seq # ToSend, Incarnation # Out,
   Seq # ToReceive,
   Incarnation # In,
   ack)
else if StateOf = SynReceived
then if ~IncomingSeq # Valid
then OutPort
   Add pkt(Seq # ToSend, Incarnation # Out,
   Seq # ToReceive, Incarnation # In,
   ack)
else if ~IncomingAck # Valid
then OutPort
   Add pkt(AckNumber(Front(InPort)),
   Ack # Inc(Front(InPort)),
   AnyNat, AnyNat,
   rst)
else OutPort,

InPort + Remove(InPort);

Timeout::
   OutPort + OutPort Append TimeoutBuffer ;

[Node Station]
Node(Medium)

State Variables[] No state variables []

Interfaces
[
    Exported:
    Buffer : QueueOfPacket ;
]

Initial State
[
    Buffer = NewQueueOfPacket ;
]

Events[ LoseMessage : PreCond is Buffer ~ = NewQueueOfPacket ;]

Behavior
[
    LoseMessage::
    Buffer + Remove(Buffer);
]

Node Medium[]

Topology
[
    There is a medium RightToLeft and a medium LeftToRight
    There are two instances of node type Station: Left and Right

Instances::
    RightToLeft, LeftToRight : Medium,
    Left, Right : Station;

Connections::
    InPort@Left, OutPort@Right <-> Buffer@RightToLeft,
    OutPort@Left, InPort@Right <-> Buffer@LeftToRight;
]

Properties
[
    assume Maxval(Q),
    forall pk(
        pk in Q imp (Maxval(Q) > Seq # Inc(pk) and Maxval(Q) > Ack # Inc(pk))
    ),

    assert CorrectSynch,
    ((StateOf = Established) or StateOf = SynSent and OldUnack ~ = ISS)@Right imp
    Seq # ToSend@Right = Seq # ToReceive@Left and
    Incarnation # Out@Right = Incarnation # In@Left;

    assert Liveness,
    For all i
        ( ~PreCond(ReceiveAck) and ~PreCond(ReceiveSyn) and
        ~PreCond(ReceiveSynAck) and ~PreCond(ReceiveRst) and
        ~PreCond(Timeout) and ~PreCond(LoseMessage) and StateOf~ = Closed)@i
        and ~ (StateOf@i = Listen and StateOf@OppositeSide(i) = Listen)
        imp (StateOf = Established)@Left and (StateOf = Established)@Right ;
]
II. AXIOMS GENERATED FROM THE SPEXIFICATION OF THE THREE-WAY HANDSHAKE

The axioms that follow contain the translation of the SPEXification of the three-way handshake given in Appendix 1. The medium interface variables (InPort, OutPort and Buffer) have been collapsed into the variable Medium. The precondition of event $e$ is called $PreCond(S, e)$. The Command interface variable has been eliminated, since it is not really necessary when analyzing the properties of the system specified.

II.1 Three-Way Handshake

```plaintext
type ThreeWay;
needs types Event,SequenceOfEvent,Packet,QueueOfPackets,SysState,Side;
declare Q,q,q':QueueOfPackets;
declare seq #,seq #,ack #,snd #:Integer;
declare cl:ControlField;
declare S:SS,SS':SequenceOfEvent;
declare pe:Event;
declare pk,pk':Packet;
declare i,i,j:Side;

interface ISS(i):Integer;
interface
  TimeoutBuffer(S,i),
  Medium(S,i):QueueOfPackets;
interface
  StateOf(S,i):SysState;
  Maxval(q),
  Incarnation # in(S,i),
  Incarnation # Out(S,i),
  OldUnack(S,i),
  Seq # ToSend(S,i),
  Seq # ToReceive(S,i):Integer;

interface Induction(S):Boolean;

{auxiliary functions to help in the readability of the axioms}

interface PreCond(S,pe),
  IncomingAck # Valid(S,j),
  IncomingSeq # Valid(S,j):Boolean;
define (auxiliary function definitions)
  PreCond(S,ActiveOpen(i)) = StateOf(S,i) = Closed,
  PreCond(S,PassiveOpen(i)) = StateOf(S,i) = Closed,
  PreCond(S,Timeout(i)) = TimeoutBuffer(S,i) = NewQueueOfPackets,
  PreCond(S,LostMessage(i)) = Medium(S,i) = NewQueueOfPackets,
  PreCond(S,ReceiveRst(i)) = (Medium(S,OppositeSide(i)) = NewQueueOfPackets) and
```
Control(Front(Medium(S.OppositeSide(i)))) = rsl.

PreCond(S.ReceiveAck(i)) =
( Medium(S.OppositeSide(i)) = NewQueueOfPackets) and
Control(Front(Medium(S.OppositeSide(i)))) = ack.

PreCond(S.ReceiveSync(i)) =
( Medium(S.OppositeSide(i)) = NewQueueOfPackets) and
Control(Front(Medium(S.OppositeSide(i)))) = syn.

PreCond(S.ReceiveSyncAck(i)) =
( Medium(S.OppositeSide(i)) = NewQueueOfPackets) and
Control(Front(Medium(S.OppositeSide(i)))) = synack.

IncomingAck#Valid(S,i) = =
( AckNumber(Front(Medium(S.OppositeSide(i)))) = 1 + OldUnack(S,i)) and
Inc # Ack(Front(Medium(S.OppositeSide(i)))) = incarnation # Out(S,i).

IncomingSeq#Valid(S,i) = =
( SeqNumber(Front(Medium(S.OppositeSide(i)))) = Seq # ToReceive(S,i))
and Inc # Seq(Front(Medium(S.OppositeSide(i)))) = incarnation # In(S,i).

axioms {Initial State}

\[ \text{Incarnation # Out(Empty,i)} = \text{Maxval(Medium(Empty,Left) Append Medium(Empty,Right))}, \]
\[ \text{Incarnation # In(Empty,i)} = 0, \]
\[ \text{OldUnack(Empty,i)} = 0, \]
\[ \text{Seq # ToSend(Empty,i)} = 0, \]
\[ \text{Seq # ToReceive(Empty,i)} = 0, \]
\[ \text{Medium(Empty,i)} = \text{NewQueueOfPackets}, \]
\[ \text{StateOf(Empty,i)} = \text{Closed}, \]
\[ \text{TimeoutBuffer(Empty,i)} = \text{NewQueueOfPackets}; \]

axioms {Active Open}

\[ \text{Incarnation # Out(S apr ActiveOpen(i),i)} = \]
\[ \text{if } i = j \text{ and PreCond(S,ActiveOpen(i))}
\text{ then Maxval(Medium(S,Left) Append Medium(S,Right))}
\text{ else incarnation # Out(S,i).} \]
\[ \text{Incarnation # In(S apr ActiveOpen(i),i)} = \text{incarnation # In(S,i)}. \]
\[ \text{OldUnack(S apr ActiveOpen(i),i)} = \]
\[ \text{if } i = j \text{ and PreCond(S,ActiveOpen(i))}
\text{ then ISS(i)}
\text{ else OldUnack(S,i).} \]
\[ \text{Seq # ToSend(S apr ActiveOpen(i),i)} = \]
\[ \text{if } i = j \text{ and PreCond(S,ActiveOpen(i))}
\text{ then 1 + ISS(i)}
\text{ else Seq # ToSend(S,i).} \]
\[ \text{Seq # ToReceive(S apr ActiveOpen(i),i)} = \text{Seq # ToReceive(S,i).} \]
\[ \text{StateOf(S apr ActiveOpen(i),i)} = \]
\[ \text{if } i = j \text{ and PreCond(S,ActiveOpen(i))}
\text{ then SynSent}
\text{ else StateOf(S,i).} \]
A CONNECTION-ESTABLISHMENT PROTOCOL

TimeoutBuffer(S apr ActiveOpen(i),j) =
  if i = j and PreCond(S,ActiveOpen(i))
  then NewQueueOfPackets
      Add pkt(ISS(i),Maxval(Medium(S,Left) Append Medium(S,Right)),
            AnyNat,AnyNat,syn)
  else TimeoutBuffer(S,j).

Medium(S apr ActiveOpen(i),j) =
  if i = j and PreCond(S,ActiveOpen(i))
  then Medium(S,i)
      Add pkt(ISS(i),Maxval(Medium(S,Left) Append Medium(S,Right)),
            AnyNat,AnyNat,syn)
  else Medium(S,j).

axioms {PassiveOpen}

  Incarnation#Out(S apr PassiveOpen(i),j) = Incarnation#Out(S,j).
  Incarnation#In(S apr PassiveOpen(i),j) = Incarnation#In(S,j).
  OldUnack(S apr PassiveOpen(i),j) = OldUnack(S,j).
  Seq#ToSend(S apr PassiveOpen(i),j) = Seq#ToSend(S,j).
  Seq#ToReceive(S apr PassiveOpen(i),j) = Seq#ToReceive(S,j).

  StateOf(S apr PassiveOpen(i),j) =
    if i = j and PreCond(S,PassiveOpen(i))
    then Listen
    else StateOf(S,j).
  TimeoutBuffer(S apr PassiveOpen(j),i) = NewQueueOfPackets.
  Medium(S apr PassiveOpen(j),i) = Medium(S,j).

axioms {ReceiveRst}

  Incarnation#Out(S apr ReceiveRst(i),j) = Incarnation#Out(S,j).
  Incarnation#In(S apr ReceiveRst(i),j) = Incarnation#In(S,j).
  OldUnack(S apr ReceiveRst(i),j) = OldUnack(S,j).
  Seq#ToSend(S apr ReceiveRst(i),j) = Seq#ToSend(S,j).
  Seq#ToReceive(S apr ReceiveRst(i),j) = Seq#ToReceive(S,j).

  StateOf(S apr ReceiveRst(i),j) =
    if i = j and PreCond(S,ReceiveRst(i))
    then if StateOf(S,i) = SynSent and IncomingAck#Valid(S,i)
         then Closed
         else if StateOf(S,i) = Listen
                then Listen
                else if IncomingSeq#Valid(S,i)
                       then Closed
                       else StateOf(S,i)
                else StateOf(S,i).
  TimeoutBuffer(S apr ReceiveRst(i),j) =
    if i = j and PreCond(S,ReceiveRst(i))
    then if StateOf(S,j) = SynSent and IncomingAck#Valid(S,j)
         then NewQueueOfPackets
         else if IncomingSeq#Valid(S,j)
                then NewQueueOfPackets
                else TimeoutBuffer(S,i)
         else TimeoutBuffer(S,j).
  Medium(S apr ReceiveRst(i),j) =

FORMAL SPECIFICATION AND VERIFICATION OF

if $i = j$
then Medium($S, i$)
else if PreCond($S, \text{ReceiveAck}(i))$ and $j = \text{OppositeSide}(i)$
then Remove(Medium($S, j$))
else Medium($S, j$);

axioms {ReceiveAck)}

Incarnation #$\text{Out}(S \text{ apr ReceiveAck}(i), j) = = \text{Incarnation} #$\text{Out}(S, j)$.

Incarnation #$\text{In}(S \text{ apr ReceiveAck}(i), j) = = \text{Incarnation} #$\text{In}(S, j)$.

OldUnack($S \text{ apr ReceiveAck}(i), j) = =$
if $i = j$ and PreCond($S, \text{ReceiveAck}(i))$
then if StateOf($S, i$) = SynSent
then if IncomingAck #$\text{Valid}(S, i)$
then $1 + \text{OldUnack}(S, j)$
else \text{OldUnack}(S, j)
else if StateOf($S, i$) = SynReceived
then if IncomingAck #$\text{Valid}(S, i)$ and IncomingSeq #$\text{Valid}(S, i)$
then $1 + \text{OldUnack}(S, j)$
else \text{OldUnack}(S, j)
else \text{OldUnack}(S, j)
else \text{OldUnack}(S, j).

Seq #$\text{ToSend}(S \text{ apr ReceiveAck}(i), j) = = \text{Seq} #$\text{ToSend}(S, j)$.

Seq #$\text{ToReceive}(S \text{ apr ReceiveAck}(i), j) = = \text{Seq} #$\text{ToReceive}(S, j)$.

StateOf($S \text{ apr ReceiveAck}(i), j) = =$
if $i = j$ and PreCond($S, \text{ReceiveAck}(i))$
then if StateOf($S, i$) = SynSent
then if IncomingAck #$\text{Valid}(S, i)$ and IncomingSeq #$\text{Valid}(S, i)$
then Established
else SynReceived
else StateOf($S, i$)
else StateOf($S, j$).

TimeoutBuffer($S \text{ apr ReceiveAck}(i), j) = =$
if $i = j$ and PreCond($S, \text{ReceiveAck}(i))$
then if StateOf($S, j$) = Closed or StateOf($S, i$) = Listen
then NewQueueOfPackets
else if StateOf($S, i$) = SynSent
then if IncomingAck #$\text{Valid}(S, i)$ and IncomingSeq #$\text{Valid}(S, i)$
then DeletePacket(TimeoutBuffer($S, j$), Seq #$\text{ToSend}(S, j)$)
else TimeoutBuffer($S, j$)
else if StateOf($S, i$) = SynSent
then if AckNumber(Front(Medium($S, \text{OppositeSide}(i)$))) = $1 + \text{OldUnack}(S, i)$
then DeletePacket(TimeoutBuffer($S, j$), Seq #$\text{ToSend}(S, j)$)
else TimeoutBuffer($S, j$)
else TimeoutBuffer($S, j$)
else TimeoutBuffer($S, j$).

Medium($S \text{ apr ReceiveAck}(i), j) = =$
if PreCond($S, \text{ReceiveAck}(i))$
then
if $i = j$
then Medium($S, j$)
Add pkt(AckNumber(Front(Medium($S, \text{OppositeSide}(i)$))),
inc #$\text{Ack}(\text{Front(Medium($S, \text{OppositeSide}(i)$)))}$,
AnyNat, AnyNat, $\text{rat}$)
else if StateOf($S, i$) = SynReceived
then if $\text{IncomingSeq} #$\text{Valid}(S, i)$
then Medium($S, j$)
Add pkt(Seq #$\text{ToSend}(S, j)$),
Incarnation #$\text{Out}(S, j)$,
Seq #$\text{ToReceive}(S, j)$,
Incarnation #$\text{In}(S, j)$,
A CONNECTION-ESTABLISHMENT PROTOCOL

ack)
else if ~incomingAck # Valid(S,i) then
    Medium(S,i)
    Add pkt(AckNumber(Front(Medium(S,OppositeSide(i))))),
    Inc # Ack(Front(Medium(S,OppositeSide(i)))),
    AnyNat,AnyNat,
    rst)
    else Medium(S,j)
    else Medium(S,i)
    else Medium(S,j);

axioms {ReceiveSyn}

Incarnation # Out(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if StateOf(S,i) = Listen
      then Maxval(Medium(S,Left)) Append Medium(S,Right))
    else Incarnation # Out(S,i)
    else Incarnation # Out(S,j).

Incarnation # In(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if (StateOf(S,j) = Listen or StateOf(S,j) = SynSent)
      then Inc # Seq(Front(Medium(S,OppositeSide(i))))
    else Incarnation # In(S,i)
    else Incarnation # In(S,j).

OldUnack(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if StateOf(S,i) = Listen
      then ISS(i)
    else OldUnack(S,i)
    else OldUnack(S,j).

Seq # TOSEND(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if StateOf(S,i) = Listen
      then 1 + ISS(i)
    else Seq # TOSEND(S,j)
    else Seq # TOSEND(S,j).

Seq # TOreceive(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if StateOf(S,i) = Listen or StateOf(S,i) = SynSent
      then 1 + SeqNumber(Front(Medium(S,OppositeSide(i))))
    else Seq # TOreceive(S,i)
    else Seq # TOreceive(S,j).

StateOf(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if StateOf(S,i) = Listen
      then SynReceived
        else if StateOf(S,i) = SynSent
          then if OldUnack(S,i) = ISS(i)
            then SynReceived
              else Established
          else SynReceived
        else StateOf(S,i)
        else StateOf(S,j),

TimeoutBuffer(S apr ReceiveSyn(i),j) = =
if i = j and PreCond(S.ReceiveSyn(i)) then
    if StateOf(S,i) = Listen
      then NewQueueOfPackets
        else if StateOf(S,i) = Listen
          then NewQueueOfPackets
            else NewQueueOfPackets
        else NewQueueOfPackets
          else NewQueueOfPackets
        else NewQueueOfPackets
          else NewQueueOfPackets
        else NewQueueOfPackets
          else NewQueueOfPackets
          .synack)
else if StateOf(S,j) = Closed
  then NewQueueOfPackets(S,j)
else TimeoutBuffer(S,j),

Medium(S apr ReceiveSyn(i,j)) = =
if PreCond(S,ReceiveSyn(i)) then
  if i = j then if StateOf(S,i) = SynSent
    then Medium(S,i)
    Add pkt(# ToSend(S,i).Incarnation # Out(S,i),
      1 + SeqNumber(Front(Medium(S,OppositeSide(i))))
        .Inc # Seq(Front(Medium(S,OppositeSide(i)))),
        ack)
else if StateOf(S,i) = SynReceived or StateOf(S,i) = Established
  then if incomingSeq # Valid(S,i)
    then Medium(S,i)
  else Add Medium(S,j)
    Add pkt(# ToSend(S,j).Incarnation # Out(S,j),
      Seq # ToReceive(S,j).Incarnation # In(S,j),
      ack)
else if StateOf(S,j) = Listen
  then Medium(S,j)
  Add pkt(SS(i),Maxval(Medium(S,Left)) Append
    Medium(S,Right)),
      1 + SeqNumber(Front(Medium(S,OppositeSide(i))))
        .Inc # Seq(Front(Medium(S,OppositeSide(i)))),
        synack)
else Add Medium(S,j)
  Add pkt(0,Maxval(Medium(S,Left)) Append
    Medium(S,Right)),
      1 + SeqNumber(Front(Medium(S,OppositeSide(i))))
        .Inc # Seq(Front(Medium(S,OppositeSide(i)))),
        rst)
else if j = OppositeSide(i)
  then Remove(Medium(S,j))
else Medium(S,j);

axioms {ReceiveSynAck)

  Incarnation # Out(S apr ReceiveSynAck(i,j)) = = Incarnation # Out(S,j),

  Incarnation # In(S apr ReceiveSynAck(i,j)) = =
  if i = j and PreCond(S,ReceiveSynAck(i)) then
    if (StateOf(S,i) = SynSent and IncomingAck # Valid(S,i)
      then Inc # Seq(Front(Medium(S,OppositeSide(i))))
        else Incarnation # In(S,j)
        else Incarnation # In(S,j),

  Seq # ToSend(S apr ReceiveSynAck(i,j)) = = Seq # ToSend(S,j),

  OldUnack(S apr ReceiveSynAck(i,j)) = =
  if i = j and PreCond(S,ReceiveSynAck(i)) then
    if StateOf(S,i) = SynSent
      then if IncomingAck # Valid(S,i)
        then 1 + OldUnack(S,i)
        else OldUnack(S,j)
      else OldUnack(S,j)
      else if StateOf(S,j) = SynReceived or StateOf(S,j) = Established
        then if IncomingAck # Valid(S,j) and IncomingSeq # Valid(S,j)
          then 1 + OldUnack(S,j)
          else OldUnack(S,j)
      else OldUnack(S,j)
      else OldUnack(S,j)

  Seq # ToReceive(S apr ReceiveSynAck(i,j)) = =
  if i = j and PreCond(S,ReceiveSynAck(i)) then
    if StateOf(S,i) = SynSent
      then if IncomingAck # Valid(S,i)
        then 1 + SeqNumber(Front(Medium(S,OppositeSide(i))))
        else Seq # ToReceive(S,j)
else Seq # ToReceive(S.i)
else Seq # ToReceive(S.j).

StateOf(S apr ReceiveSynAck(i),j) =
if i = j and PreCond(S.ReceiveSynAck(i))
then if StateOf(S,i) = SynSent and incomingAck # Valid(S.i)
   then Established
   else StateOf(S,i)
else StateOf(S,j),

TimeoutBuffer(S apr ReceiveSynAck(i),j) =
if i = j and PreCond(S.ReceiveSynAck(i))
then if StateOf(S,i) = Closed or StateOf(S,i) = Listen
then NewQueueOfPackets
else if StateOf(S,i) = SynSent
   then DeletePacket(TimeoutBuffer(S.i),OldUnack(S.j))
else NewQueueOfPackets
else TimeoutBuffer(S.j),

Medium(S apr ReceiveSynAck(i),j) =
if PreCond(S.ReceiveSynAck(i)) then
   if i = j
   then if StateOf(S,i) = Closed or StateOf(S,i) = Listen
      then Medium(S.i)
      Add pkt(AckNumber(Front(Medium(S.OppositeSide(i)))),
            Inc # Ack(Front(Medium(S.OppositeSide(i)))),
            AnyNat, AnyNat,
            rst)
   else if StateOf(S,i) = SynSent
      then if incomingAck # Valid(S,i)
         then Medium(S.i)
         Add pkt(Seq # ToSend(S.i),Incarnation # Out(S.i),
                1 + SeqNumber(Front(Medium(S.OppositeSide(i)))),
                ,Inc # Seq(Front(Medium(S.OppositeSide(i)))),
                ack)
         else Medium(S.i)
         Add pkt(AckNumber(Front(Medium(S.OppositeSide(i)))),
                 Inc # Ack(Front(Medium(S.OppositeSide(i)))),
                 AnyNat, AnyNat,
                 rst)
   else StateOf(S,i) = Established
   then if -IncomingSeq # Valid(S,i)
      then Medium(S.i)
      Add pkt(Seq # ToSend(S.i),
            Incarnation # Out(S.i),
            Seq # ToReceive(S.i),
            Incarnation # In(S.i),
            ack)
      else StateOf(S,i) = SynReceived
      then if -IncomingAck # Valid(S,i)
         then Medium(S.i)
         Add pkt(Seq # ToSend(S.i),
                Incarnation # Out(S.i),
                Seq # ToReceive(S.i),
                Incarnation # In(S.i),
                ack)
      else if -incomingAck # Valid(S,i) then
         Medium(S.i)
         Add pkt(AckNumber(Front(Medium(S.OppositeSide(i)))),
                 Inc # Ack(Front(Medium(S.OppositeSide(i)))),
                 AnyNat, AnyNat,
                 rst)
   else Medium(S.i)
else Medium(S.i)
else if j = OppositeSide(i)
then Remove(Medium(S.j))
else Medium(S.j)
else Medium(S.j),
axioms {Timeout}

\[ \text{Incarnation} \# \text{Out}(S \text{ apr } \text{Timeout}(i),j) = \text{Incarnation} \# \text{Out}(S,j). \]

\[ \text{Incarnation} \# \text{In}(S \text{ apr } \text{Timeout}(i),j) = \text{Incarnation} \# \text{In}(S,j). \]

\[ \text{OldUnack}(S \text{ apr } \text{Timeout}(i),i) = \text{OldUnack}(S,j). \]

\[ \text{Seq} \# \text{ToReceive}(S \text{ apr } \text{Timeout}(i),j) = \text{Seq} \# \text{ToReceive}(S,j). \]

\[ \text{StateOf}(S \text{ apr } \text{Timeout}(i),i) = \text{StateOf}(S,j). \]

\[ \text{Medium}(S \text{ apr } \text{Timeout}(i),j) = \]
\[ \begin{cases} 
\text{Append}(\text{Medium}(S,j), \text{TimeoutBuffer}(S,j)) & \text{if } i = j \text{ and } \text{PreCond}(S,\text{Timeout}(i)) \\
\text{Medium}(S,j) & \text{else} 
\end{cases} \]

\[ \text{TimeoutBuffer}(S \text{ apr } \text{Timeout}(i),j) = \text{TimeoutBuffer}(S,j). \]

axioms {LoseMessage}

\[ \text{Incarnation} \# \text{Out}(S \text{ apr } \text{LoseMessage}(i),j) = \text{Incarnation} \# \text{Out}(S,j). \]

\[ \text{Incarnation} \# \text{In}(S \text{ apr } \text{LoseMessage}(i),j) = \text{Incarnation} \# \text{In}(S,j). \]

\[ \text{OldUnack}(S \text{ apr } \text{LoseMessage}(i),i) = \text{OldUnack}(S,j). \]

\[ \text{Seq} \# \text{ToReceive}(S \text{ apr } \text{LoseMessage}(i),j) = \text{Seq} \# \text{ToReceive}(S,j). \]

\[ \text{StateOf}(S \text{ apr } \text{LoseMessage}(i),i) = \text{StateOf}(S,j). \]

\[ \text{Medium}(S \text{ apr } \text{LoseMessage}(i),j) = \]
\[ \begin{cases} 
\text{Remove}(\text{Medium}(S,j)) & \text{if } i = j \text{ and } \text{PreCond}(S,\text{LoseMessage}(i)) \\
\text{Medium}(S,j) & \text{else} 
\end{cases} \]

\[ \text{TimeoutBuffer}(S \text{ apr } \text{LoseMessage}(i),j) = \text{TimeoutBuffer}(S,j). \]

end;

II.2 Auxiliary Data Type Definitions

type Packet;

needs types Integer, ControlField;

declare dummy, pk: Packet;

declare seq #, ack #, inc # a, inc # a: Integer;

declare cf: ControlField;

interface pkt(seq #, ack #, inc # a, inc # a, cf): Packet;

interfaces SeqNumber(pk), AckNumber(pk), Inc # Seq(pk), Inc # Ack(pk): Integer;

interface Control(pk): ControlField;
A CONNECTION-ESTABLISHMENT PROTOCOL

axiom dummy = pk

  = = ( (SeqNumber(dummy) = SeqNumber(pk) and AckNumber(dummy

  ) = AckNumber(pk)

and Control(dummy) = Control(pk)

and Inc # Ack(dummy) = Inc # Ack(pk)

and Inc # Seq(dummy) = Inc # Seq(pk));

axiom SeqNumber(pkt(seq # s, ack #, inc # a, cf)) = seq #;

axiom AckNumber(pkt(seq # s, ack #, inc # a, cf)) = ack #;

axiom Inc # Seq(pkt(seq # s, ack #, inc # a, cf)) = inc # s;

axiom Inc # Ack(pkt(seq # s, ack #, inc # a, cf)) = inc # a;

axiom Control(pkt(seq # s, ack #, inc # a, cf)) = cf;

end {Packet};

type QueueOf Packet;

needs type Packet

declare dummy, q, q1, q2, qq: QueueOf Packet;

declare i, i1, i2, ii: Packet;

interfaces

NewQueueOf Packet, Add i, Remove(q),

Append(q1, q2), que(i): QueueOf Packet;

infix Add;

interfaces

Front(q), Back(q): Packet;

interfaces

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

infix in;

axioms dummy = dummy = = TRUE,

q Add i = NewQueueOfPacket = = FALSE,

NewQueueOfPacket = q Add i = = FALSE,

q1 Add i1 = q2 Add i2 = = ((q1 = q2) and (i1 = i2)).

Remove(NewQueueOfPacket) = NewQueueOfPacket,

Remove(q Add i) = = if q = NewQueueOfPacket

then q

else Remove(q Add i).

Append(q, NewQueueOfPacket) = = q,

Append(q, q1 Add i1) = = Append(q, q1) Add i1,

que(i) = = NewQueueOfPacket Add i,

Front(q Add i) = = if q = NewQueueOfPacket

then i

else Front(q).

Back(q Add i) = = i,

i in NewQueueOfPacket = = FALSE,

i in (q Add i1) = = (i in q or (i = i1));

ruleslemma

Append(NewQueueOfPacket, q) = = q;

schemas NormalForm(q) = = cases(Prop(NewQueueOfPacket),

all qq, ii (Prop(qq Add iii)));
\[ \text{induction}(q) = \text{cases}(\text{Prop}(\text{NewQueueOfPacket}), \]
\[ \text{all } q, \, \exists (\text{PH}(qq) \implies \text{Prop}(qq \implies \text{Add } ii))); \]
\[ \text{end } \{\text{QueueOfPacket}\}; \]
REFERENCES


