FORMAL SPECIFICATION AND VERIFICATION OF A CONNECTION ESTABLISHMENT PROTOCOL

Daniel Schwabe*

Departamento de Informática
PUC/RJ

Abstract: This paper presents an exercise in the verification of a connection establishment protocol. A specification language named SPEX, tailored for 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 are shown to be correct.

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 non-trivial.

This paper presents an exercise in verifying that a particular algoritm to realize an important function in computer networks, namely *connection establishment*, does indee 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 being used in practice, given in the language introduced earlier. Section 3 then discusses particular properties of this protocol and shows their verification.

1.1 Connection Establishment Protocol - This section presents the motivation for connection establishment protocols in general and for the threeway handshake used in the ARPANET in particular*.

Consider a distirbuted system with several interconnected nodes. The nodes are connected by an unreliable transmisson 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 this 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 this will allow the detection of 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 maintaim communication between processes is quite significant, it is desirable for the nodes to be able to keep these resources allocated only while the exchange is taking place, thus increasing their utilization.

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 connections is open between two processes, i.e., a particular frame of references 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 and/or duplicate packets that flow between the two nodes. Since connections can open and close, it is

*This research was conducted at the Information Sciences Institute-USC, and was supported by the Defense Advanced Research Projects Agency under contract number DAHC15 72 C 0308. The author was partially supprted by CAPES-Brazilian Government under contract 1247/76. Views and conclusions contained in this paper are the author's.

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

possible for packets from old incarnations to be in the medium, and obviously they should not be mistaken for packets belonging to a newly opened connection.

Since packets may be lost, a positive acknowledgement-retransmission on timeout scheme is used. In other words, a copy of each packet sent is kept by the sender until an acknowledgement of its reception by the receiver is received. If. after some predefined amount of time, no acknowledgment themselves are not acknowledged.

An important fact to notice is 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 establisment; each node will consider the connection to be open upon sending and receiving this packet . It is clear that the node receiving this paket 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. A problem equivalent to this is discussed in [2].

In many systems,connections are opened and closed quite frequently. In view of the fact that 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 as a current one, thereby initiating a connection with the wrong frame of reference[4].

A problem still remains as to how to identify packets from previous incarnations as being old. The sequence numbers chosen to stablish the frame of reference of a new connection must prevent that Reference[18] discusses this issue in more detail.

A protocol has been proposed to handle the connection establishment problems as discussed in the previous paragraphs. It is called the three-way handshake[14,19]. The particular version used here is taken from TCP[TCP80], the second generation transport level protocol being used in the ARPA internet sytem.

This protocol derives its name from the sequence of steps a node goes through in order to establish a connection. Suppose node A wishes to communicate with node B, and that node A taskes the initiative. Then, they 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 acknowledgmenent,together called SYNACK(for SYNchronize and ACKnowledge).
3. Node A receives the SYNACK packet, verifies that the ACK portion does indeed acknowledges 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 is own previous SYN, and then considers the connection to be opened.

There are two basic modes in which to open 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 [16]. 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 layer 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 non-deterministic state transition system, with some specialized features to facilitate protocol specification. SPEX is discussed at greater length in [12].

A layer is regarded as consisting 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 layer 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 layer, it is necessary to describe the behavior of each 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* aprt), 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 also 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[4,6,8,9]. 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 as a result of this occurrence.

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 layer or other layers using the layer in which the node is defined.

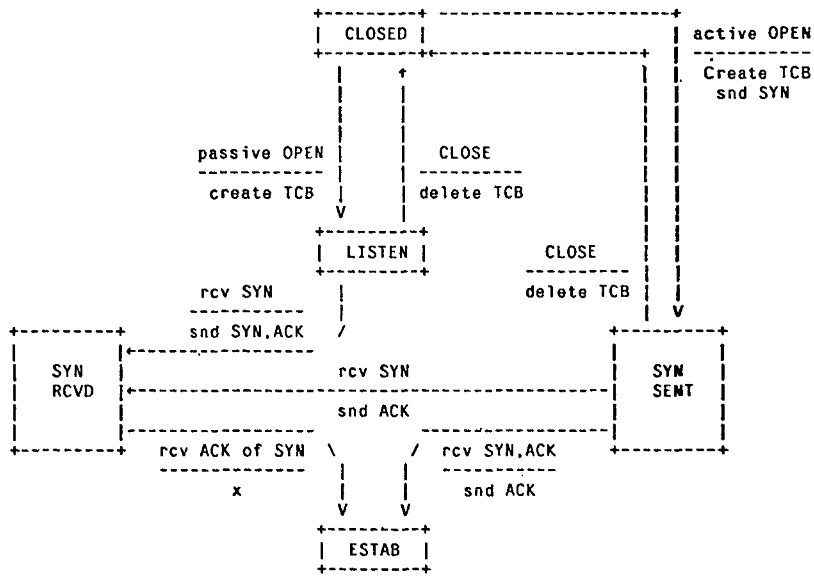

Figure 1-1: Three-Way Handshake State Transition Diagram

Accordingly, the interface variables at each node are divided into two kinds: Those that are *exported* to other layers and those that are connected to other nodes in the same layer. 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, this means data in that variable flows in both directions.

The actual behavior of a node is given by discribing how a node reacts to the occurrence of certain specified events. Each event known at a node has a *pre-condition* associated with it; this pre-condition is a predicate involving state and interface variables at that node. As long as a pre conditions 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, whenever a variable is *not* mentioned on the left hand side of any event effects statement it means that 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 *e1* at some node N1 to change the value of some interface variable at another node, say N2. In fact, *e1* 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 value of any variables at system creation time. The most general way to specify this is by giving 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 layer.

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 intuite understanding of the system.

Assumed properties are used to *define* certain operations in a non-computational fashion by giving inputputput relationships between arguments and returned values.

SPEXifications** 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**. Reference[12] discusses this translation in detail.

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]. Addtional 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.

Following the algebraic style of specifications [5,6,7,8,9], a data type specified by first defining three sets of functions:

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

2. *Extenders* (or *Modifiers*). These functions also have the data type being specified as their range, but in constrat 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 such a specification is to view values of the type in terms of the constructors wich 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 ((NewQueueOfInteger Add 1) Add 2) Add 3

This 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*. The second part of a data type specification

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

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

Figure 1-2: Signature of type *QueueOfInteger*

provides semantics for the operations whose domain and information was give in the first part. Extenders and selectors are defined by equational axioms of the form *lhs == rhs* relating how each function behaves when applied to each of the constructors. Constructor functions are treated as primitive, unspecified operations.

Example of axioms taken from a specification of the type *QueueOfInteger* are given in Figure 1-3.

```
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 *QueueOfInteger*

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

#Elements(Append(q,q')) = #Elements(q)+#Elements(q')

* "SPEXification" will be used to mean SPEXspecification.

** I.e., whether the specification captures the designer's intuitive understanding of the system

Properties of a data type are proved using a method called *structural induction* [7,13] which is based on the notion that all values of the data type can be produced by repeated applications of the *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(NewQueue)*).
2. Assuming P is true for some value q, then it is also true for all values obtained by a applying constructors to q(e.g., for all q,i $P(q)$ implies *P(q Add i)*).

There much more to specifying a data types 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. Also, the axioms should facilitate simple proofs. In other words, the type has an associated *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 schema part of a type provides the proof structure.

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:

try proposition Set up *proposition* as the current goal.

employ Induction(v) Induction is a user-defined schema for the the type of induction desired and v is the variable to be induced upon. The proof structure is modified to show the induction.

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

suppose proposition - Break the current goal into two subgoals, one with the additive hypothesis *proposition* and the other with ~ *proposition*.

split Break up the proposition at a designated spot into subgoals, e.g., the proposition H imp(C_1 and C_2) can be split into the two propositions H imp C_1 and (H and C_1) impC_2.

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

invoke defn Invoke a definition *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 AFFIRM is represented by a named node in a directed acyclic graph called the *proof forest*. The proof of a theorem comprises a tree, whose named arcs represent AFFIRM commands and thus deductive steps. AFFIRM checks for circularity within the current tree.

An example of an 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 expression, 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. Also, many suffer from lack of a solid theory and/or of automated tools for verification. Reference[15] provides a survey of this work.

Although the underlying model of SPEX is not new, it is beleived to be the first language allowing the formal specification of non-deterministic 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 combinational explosion when analyzing more complex protocols. Reference [12] contains an example in which a complex protocol, involving an arbitrary number of nodes, is specified, but where the complexity of the proof is independent on 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.

After giving the state variables, interfaces, initial state, and events for one station, 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 next packet coming in.
TimeoutBuffer - is a queue of packets containing copie of packets which have been sent but not yet ackonowledged*.

The exported interface to using layers contains two variables.
Command - is a command buffer through which the user indicates what type of open request is desired.
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 layer, namely:
InPort - is a queue 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 *Timeout Buffer* be empty and the sequence numbers and incarnation number of incoming packes be zero**

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 whenthe 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 fiel is *rst*(reset). This is control packet used to indicate the discovery of an anomalous situation.
ReceiveAck - which is caused when an acknowledgement packet arrives.
ReceiveSyn - which is caused when a packet arrives whose control fiel is *syn*(synchronize). This is a connection request.
ReceiveSynAck - which is caused when a packet which is both an acknowledgement and a connection request arrives.

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 queue, and the *receive* operation is modeled a *Remove* from the queue, with the packet delivered obtained by *from* 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 acknowledgement field.
Ctl - is the control field of the packet.

As an illustration of the effects of an event , consider the *ActiveOpen* event. Its pre-condition 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 simplication of the one given in TCP. 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 pairs 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 number 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 a passive one, 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 simple.

For the purpose of verifying properties of the three-way handshake, the SPEXification has been manually translated into an algebraic data type

* 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.

**Zero is used an arbitrary initial value.

specification that can be understood by the AFFIRM system. Appendix II contains the generated axioms and auxiliary data type definitions (e.g., Packet, QueueOfPacket, etc.) in AFFIRM system

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 point in time at which the claim of functional correcness 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 considering only one node's point of view*.

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

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

This property is proved to be invariant using inductive proof methods which are used for abstract data types. After working with this specification, it became apparent 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 simulataneos active open commands are issued at both nodes, as follows: one side may be in the SynSent state and may have already received an acknowledgement 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 thar OldUnack (the oldest unacknowledged sequence number) is not ISS anymore. Since this side has received an acknowledgement for its SYN, it can be sure that the other side knows its Seq#ToSend and its Incarnation#Out. Hence the statement of functional correctness must be strengthened(for one side only) as follows:

```
Theorem FC:
((StateOf=Established)or((StateOf=SynSent)and
  OldUnack~=ISS)@Right

        imp

  Seq#ToReceive@Left = Seq#ToSend@Right
and Incarnation#In@Left= Incarnation#Out@Right;
```

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 amounts to, given a *goal* state(e.g., Established , examining how each event can move the system *into* that state(e.g., *Receive Ack* 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 and/or followed by some *replace*, *case* 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 Received@Right occurs(←< 1)**. The relevant case to consider has the node at right in SynSent or in SynReceived , and the incoming acknowledgement 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 ←< 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 ←<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.

To see that the sequence numbers correspond,it suffices to see that, if the state of a node is not Listen or Close , 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 ←<4), and that all SYN packets carry ISS as their sequence numbers. Since the Seq#ToReceive is taken from the SYN packet, it must

* The notation *P*@*n* means *P* is to be evaluated in node *n*.

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

**Indicators of the from ←< n are used to point to the corresponding places in the proof tree perforce be ISS+1(see theorem Seq#ToReceiveVals, applied at ↞<5.). Therefore Seq#ToReceive@Left= Seq#ToSend@Right.

The next relevant case is when a ReceiveSyn@ Left occurs(↞<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 ~ = ISS (theorem SynchNoLorCorSS, applied at ↞<7). Therefore this situation really cannot occur.

The other relevant cases are when a Receive SynAck 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 we may want to show 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 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 bot 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* and/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 in the number of occurrences of the *LoseMessage* event, as well as on the size of the medium.

Accordingly, the number of ocurrences 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 acknowledgement 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 pre-condition 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:
forall S,i
        [~ PreCond(S,ReceiveXX) and
         ~ PreCond(S,Timeout)
     and ~PreCond(S,LoseMessage) and
           StateOf~=Closed]@i
     and ~(StateOf@i= Listen and
           StateOf@OppositeSide(i)=Listen)
  imp      (StateOf=Established)@Left and
           (StateOf=Established)@Right;
```

where XX={Ack,Syn,SynAck,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 happerns, 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 protocol simplification introduced that a node always returns to Close state when a RST packet arrives. Note that this is *not* the scenario describe above.

An interesting observation is that, 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 in which it went into 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.

Reference[1] discusses the verification of other types of liveness properties in algebraically described state transition systems.

4. CONCLUSIONS

This paper has presented an exercise in the verification of properties of a connection establishment protocol. A specification language tailored for the need of communications protocols has been proposed, and is 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

Node A		Node B
CLOSED		CLOSED
act. Open	<SEQ=200><CTL=SYN> -> ...	
(delayed)	<- <SEQ=300><CTL=SYN>	*act. OPEN*
SYNSENT		SYNSENT
		... rcv SYN
SYNSENT		SYNRECEIVED
	<- <SEQ=301><ACK=201><CTL=ACK>	*snd ACK*
rcv ACK		
SYNSENT		SYNRECEIVED
	<- <SEQ=100><CTL=SYN> ***old duplicate !!***	
rcv SYN		
ESTABLISHED		SYNRECEIVED
snd ACK	<SEQ=201><ACK=101><CTL=ACK> -> ***bad ACK !!***	
	*** *bad data might be accepted here* ***	
	e.g. <- <SEQ=101><DATA>	
		rcv ACK
ESTABLISHED		SYNRECEIVED
	<- <SEQ=101><CTL=RST>	*snd RST*
rcv RST		
CLOSED		SYNRECEIVED
	<- <SEQ=300><CTL=SYN> *original delayed syn*	
rcv SYN		
CLOSED		SYNRECEIVED
snd RST	<SEQ=0><ACK=301><CTL=RST> ->	
		rcv RST
		discard- bad ACK#
rcv ACK	<- <SEQ=301><ACK=201><CTL=ACK>	*snd ACK*
CLOSED		SYNRECEIVED
snd RST	<SEQ=201><CTL=RST> ->	*rcv RST*
CLOSED		CLOSED

Figure 3-3: Example of a liveness error in the three-way handshake

techniques; further work is described in [12,18]. 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.

Acknowledgements - I wish to thank Carl Sunshine for his constructive criticism of the work presented in this paper and careful review of the paper itself; Jon Postel for bearing with me and taking the time to answer all those stupid 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. Finally, I thank Danny Cohen, whose support made it possible for me to work at ISI.

The presentation of this paper was made possible in part by a grant from IBM-Brasil and in part by the Brazilian Government.

References

1 - Berthomieu,B., Proving Progress Properies of Communication Protocols in AFFIRM, Information Sciences Institute, Program Verification Project, Affirm Memo 35, September - 1980.

2 - Cohen,D. and Yemini,Y., "Protocols for Dating Coordination", in Proceedings of the 4th Berkeley Conference on Distributed Data Management and Computer Networks, pp.179-188, Lawrence Berkeley, California,August 28-30 1979. Also in The Oceanview Tales, ISI/RR-79-83, USC/Information Sciences Institute, Marina Del Rey, California.

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

4 - Gerhart,S.L., et al.,"An overview of AFFIRM: a specification and verification system", in Proceedings IFIP80 , pp. 343-348, Australia, October 1980.

5 - Goguen,J.A., Thatcher,J.W., and Wagner,E.G., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in Yeh,R.T.(ed.), Current Trends in Programming Methodology, pp.80-149, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1978

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

7 - Guttag,J.V., Horowitz,E., and Musser,D.R., "Abstract Data Types and Software Validation", Communications of the ACM 21, December 1978, 1048-1064. (Also USC Information Sciences Institute RR-76/48, August 1976).

8 - Guttag, J.V., and Horning, J.J., "The Algebraic Specification of Abstract Data Types", Acta Informatica 10, 1978 , 27-52

9 - Liskov,B.H. and Zilles,S.N., "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering SE-1,(1), March 1975, 7-19

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

11 - Owicki,S.S. and Gries,D., "Verifying Properties of Parallel Programs: An Axiomatic Approach", Communications of the ACM 19,(5), May 1976

12 - Schwabe,D., Formal Techniques for Specification and Verification of Protocols, Ph.D. thesis, Report CSD 810401, Computer Science Department, University of California at Los Angeles, 1981.

13 - Spitzen,J. and Wegbreit,B.,"The verification and synthesis of data structures", Acta Informatica 4,1975, 127-144.

14 - Sunshine,C.A. and Dalal,Y.K.,"Connection Management in Transport Protocols, "Computer Network 2,(6), December 1978.

15 - Sunshine,C.A., Formal Modelling of Communication Protocols, USC Information Sciences Institute, Technical Report ISI/RR-81-89, February 1981

16 - J.B. Postel, Editor, DoD Standard Transmission Control Protocol-January 1980. Prepared by University of Southern California-Information Sciences Institute for DARPA-IPTO. Also in ACM SIGCOMM Quartely Review October 1980

17 - Thompson,D.H.S.L. Gerhart,R.W. Erickson .,S. Lee, and R.L.Bates, eds., The AFFIRM Reference Library, USC Information Sciences Institute, 1981. 5 vols: Reference Manual, User's Guide, Type Library, Annotated Transcripts, and Collected Papers; 500 pages

18 - Thompson,D.H.C.A. Sunshine,R.W.Erickson,S.L. Gerhart, and D.Schwabe, Specification and Verification of Communication Protocols in AFFIRM using State Transition Models, USC Information Sciences Institute, Technical Report ISI/RR-81-88, February 1981. (Also submitted for publication)

19 - Tomlinson,R.S. "Selecting Sequence Numbers", in Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, pp 11-23, ACM, Santa Monica, California, March 1975. Also IFIP TC6.1(INWG) Protocol Note Nº 2, August 1974.

```
theorem Synchronized,    StateOf(S,Right) = Established
                      or StateOf(S,Right) = SynSent and OldUnack(S,Right) ~= ISS(Right)
                   imp synchronized(S);
Synchronized uses EorSSimpEorSR%, SynchNoLorCorSS%, AcksAndSyns%, FrontInQ%,
Seq#ToSendVals%, and Seq#ToReceiveVal%.

proof tree:
 Synchronized
       apply  EorSSimpEorSR {proved by Schwabe using AFFIRM 120 on  4-Feb-81 in
                                transcript <SCHWABE>AFFIRMTRANSCRIPT.3-FEB-81.2}
 54    put  S'=S
 55    employ  Induction(S)
  Empty:
     Immediate
  apr:
     56    employ  NormalForm(ii$)
      ActiveOpen:
         57    cases
         65    invoke  IH
         66    replace
         67    invoke  synchronized  | all  |
         (proven!)
      PassiveOpen:{Synchronized, apr:}
         58    cases
         125   invoke  IH
         126   invoke  synchronized  | all  |
         (proven!)
      LoseMessage:{Synchronized, apr:}
         59    invoke  IH
         128   invoke  synchronized  | all  |
         (proven!)
      Timeout:{Synchronized, apr:}
         60    invoke  IH
         130   invoke  synchronized  | all  |
         (proven!)
      ReceiveRst:{Synchronized, apr:}
         61    cases
         131   employ  NormalForm(i')
          Left:
             132   invoke  IH
             134   invoke  synchronized  | all  |
             (proven!)
          Right:
             133   invoke  IH
             136   invoke  synchronized  | all  |
             (proven!)
      ReceiveAck:{Synchronized, apr:}
         62    cases
         69    employ  NormalForm(i')
          Left:
             70    invoke  IH
             72    invoke  synchronized  | all  |
             (proven!)
          Right:                                                  ←←<1
             71    invoke  IH
             74    replace
             75    invoke  synchronized  | all  |
             76    apply  AcksAndSyns                             ←←<2
             77    put       pk = Front(Medium(ss', Left))
                         and S=ss'
             78    apply  FrontInQ
             79    put  Q = Medium(ss', Left)
             80    replace
             81    invoke  PreCond  | -4 : -3  |
             82    apply  Seq#ToSendVals                          ←←<4
             83    put  S=ss'
             84    invoke  IncomingAck#Valid  | all  |
             85    invoke  HasSyn                                 ←←<3
             86    replace
             87    apply  Seq#ToReceiveVal                        ←←<5
             88    put  S=ss'
             (proven!)
      ReceiveSyn:{Synchronized, apr:}
         63    cases
         90    invoke  IH                                         ←←<6
         91    replace
         92    invoke  synchronized  | all  |
         93    cases
         94    replace
         95    apply  SynchNoLorCorSS                             ←←<7
         97    put  S=ss'
         98    replace
         (proven!)
      ReceiveSynAck:{Synchronized, apr:}
         64    cases
         99    employ  NormalForm(i')
          Left:
             100   invoke  IH
             102   invoke  synchronized  | all  |
             103   cases
             104   replace
             105   apply  SynchNoLorCorSS
             106   put  S=ss'
             (proven!)
```

Figure 3-1: Proof tree for the functional correctness of the three way handshake

```
Right:{Synchronized, apr:, ReceiveSynAck:}
    101   invoke  IH
    108   invoke  synchronized  | all  |
    109   apply  AcksAndSyns
    110   put        S=ss'
              and pk = Front(Medium(ss', Left))
    111   apply  FrontInQ
    112   put  Q = Medium(ss', Left)
    113   replace
    114   invoke  IncomingAck#Valid  | last  | ,    PreCond  | 1  |
    115   replace
    116   invoke  HasSyn
    117   invoke  PreCond
    118   replace
    119   apply  Seq#ToSendVals
    120   put  S=ss'
    121   replace
    122   apply  Seq#ToReceiveVal
    123   put  S=ss'
    124   replace
    (proven!)
```

Figure 3-1: Proof tree continued

```
theorem Synchronized,      StateOf(S, Right) = Established
             or    StateOf(S, Right) = SynSent
               and OldUnack(S, Right) ~ = ISS(Right)
          imp synchronized(S);

theorem AcksAndSyns,       pk in Medium(S, Left)
             and StateOf(S, Left) ~ = Listen
             and StateOf(S, Left) ~ = Closed
             and Inc # Ack(pk) = Incarnation # Out(S, Right)
             and (Control(pk) = ack) or (Control(pk) = synack)
          imp HasSyn(S, pk);

theorem FrontInQ, Q~ = NewQueuOfPacket imp Front(Q) in Q;

theorem Seq # ToSendVals,       StateOf(S, Right) ~ = Closed
               and StateOf(S, Right) ~ = Listen
            imp Seq # ToSend(S, Right) = 1 + ISS(Right);

theorem Seq # ToReceiveVal,       StateOf(S, Right) ~ = Closed
                and StateOf(S, Right) ~ = Listen
                and    StateOf(S, Left) = SynReceived
                   or StateOf(S, Left) = Established
                and  Incarnation # Out(S, Right) = Incarnation # In(S, Left)
             imp Seq # ToReceive(S, Left) = 1 + ISS(Right);

theorem EorSSimpEorSR,       StateOf(S, Right) = Established
              or    StateOf(S, Right) = SynSent
                and OldUnack(S, Right) ~ = ISS(Right)
           imp    StateOf(S, Left) = Established
              or StateOf(S, Left) = SynReceived;

theorem SynchNoLorCorSS,       StateOf(S, Right) = Established
                or    StateOf(S, Right) = SynSent
                  and OldUnack(S, Right) ~ = ISS(Right)
             imp    StateOf(S, Left) ~ = Listen
                  and StateOf(S, Left) ~ = Closed
                  and StateOf(S, Left) ~ = SynSent;
define   synchronized(S)
     = = (   Seq # ToReceive(S, Left) = Seq # ToSend(S, Right)
         and Incarnation # In(S, Left) = Incarnation # Out(S, Right)),

      HasSyn(S, pk)
       = = some SS, SS', pk'
          (   SS join SS' = S
           and pk in Medium(SS, Right)
           and Inc # Seq(pk') = Inc # Ack(pk)
           and Inc # Seq(pk') = Incarnation # In(S, Left)
           and if Control(pk) = synack
                then Control(pk') = syn
                else (Control(pk') = syn or Control(pk') = synack));
```

Figure 3-2: Theorems and definitions used in the proof of the three way handshake

I. SPEXification of the Three Way Handshake

```
Node(Station)[
    State Variables
    [
        ISS,                                        | Initial Send Sequence #
        Incarnation#In,                             | Incarnation # of incoming packets
        Incarnation#Out,                            | Incarnation # of outgoing packets
        OldUnack,                                   | Oldest unacknowledged seq. #
        Seq#ToSend,                                 | Seq # to put in the next outgoing packet
        Seq#ToReceive                               | Next expected seq #
        :Nat,                                       | Nat stands for Natural

        TimeoutBuffer : QueueOfPackets,             | buffer with packets sent and not acknowledged
    ]

    Interfaces
    [
      Exported::
        Command : Command,                          | One of {Active,Passive,Null}
        StateOf : SysState,                         | State of this side of the connection
      Internal::
        InPort ,                                    | msgs coming in
        OutPort                                     | msgs going out
        :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 pre-conditions
      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
    ]

    Behavior
    [
                                                    |first we define some auxiliary predicate and
                                                    |functions to improve readability of the specifica

      define IncomingAck#Valid ==

        ( AckNumber(Front(InPort)) = +OldUnack) and
        Ack#Inc(Front(InPort)) = Incarnation#Out;
                                                    | Acknowledgement for X has Ack = X + 1

      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);

ReceiveAck::
  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 = SynReceived
          then if IncomingAck#Valid and IncomingSeq#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,

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          | i can be one of {Left,Right}
        ( ~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 ;
    ]
```

NOTE: Due to space limitations, only a representative set of the axioms generated from the SPEXification of the three-way handshake are included. The full set can be found in [12].

II. Axioms generated from the SPEXification of the Three Way Handshake

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

interface ISS(i):Integer;

interface
    TimeoutBuffer(S,i),
    Medium(S,i)
   :QueueOfPackets;

interface
    StateOf(S,i)
   :SysState;

interface
    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,i),
      IncomingSeq#Valid(S,i)
    : 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,LoseMessage(i)) == Medium(S,i) ~= NewQueueOfPackets,

    PreCond(S,ReceiveRst(i)) ==
    ( Medium(S,OppositeSide(i)) ~= NewQueueOfPackets) and
     Control(Front(Medium(S,OppositeSide(i)))) = rst,

    PreCond(S,ReceiveAck(i)) ==
    ( Medium(S,OppositeSide(i)) ~= NewQueueOfPackets) and

    Control(Front(Medium(S,OppositeSide(i)))) = ack,

    PreCond(S,ReceiveSyn(i)) ==
    ( Medium(S,OppositeSide(i)) ~= NewQueueOfPackets) and
     Control(Front(Medium(S,OppositeSide(i)))) = syn,

    PreCond(S,ReceiveSynAck(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 {ReceiveAck}

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

    Incarnation#In(S apr ReceiveAck(i),j) == Incarnation#In(S,j),
```

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

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

Seq # ToReceive(S apr ReceiveAck(i),j) = = Seq # ToReceive(S,j),

StateOf(S apr ReceiveAck(i),j) = =
 if i = j and PreCond(S,ReceiveAck(i))
  then if StateOf(S,i) = SynReceived
       then if IncomingAck # Valid(S,i) and IncomingSeq # Valid(S,i)
            then Established
            else SynReceived
       else StateOf(S,i)
  else StateOf(S,j),

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

   Medium(S apr ReceiveAck(i),j) = =
   if PreCond(S,ReceiveAck(i)) then
    if i = j
    then if StateOf(S,i) = Closed or StateOf(S,i) = Listen
            or ((StateOf(S,i) = SynSent) and ~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 if StateOf(S,i) = SynReceived
                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 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 {LoseMessage}

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

    Incarnation # In(S apr LoseMessage(i),j) = = Incarnation # In(S,j),

    OldUnack(S apr LoseMessage(i),j) = = OldUnack(S,j),

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

    Seq # ToReceive(S apr LoseMessage(i),j) = = Seq # ToReceive(S,j),

    StateOf(S apr LoseMessage(i),j) = = StateOf(S,j),

    Medium(S apr LoseMessage(i),j) = =
     if i = j and PreCond(S,LoseMessage(i))
     then Remove(Medium(S,i))
     else Medium(S,j),

    TimeoutBuffer(S apr LoseMessage(i),j) = = TimeoutBuffer(S,j);
```

Auxiliary Data Type Definitions

```
type Packet;

needs types Integer, ControlField;

declare   dummy, pk: Packet;
declare   seq #, ack #, inc # s, inc # a: Integer;
declare   cf: ControlField;

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

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

interface Control(pk): ControlField;

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 #, inc # s, ack #, inc # a, cf)) = = seq #;

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

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

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

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

end {Packet} ;
type QueueOfPacket;

needs  type Packet;

declare dummy, q, q1, q2, qq: QueueOfPacket;
declare i, i1, i2, ii: Packet;

interfaces
    NewQueueOfPacket, q Add i, Remove(q),
    Append(q1, q2), que(i): QueueOfPacket;

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));

rulelemma
    Append(NewQueueOfPacket, q) = = q;

schemas NormalForm(q) = = cases(Prop(NewQueueOfPacket),
                    all qq, ii (Prop(qq Add ii))),

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

end {QueueOfPacket} ;
```