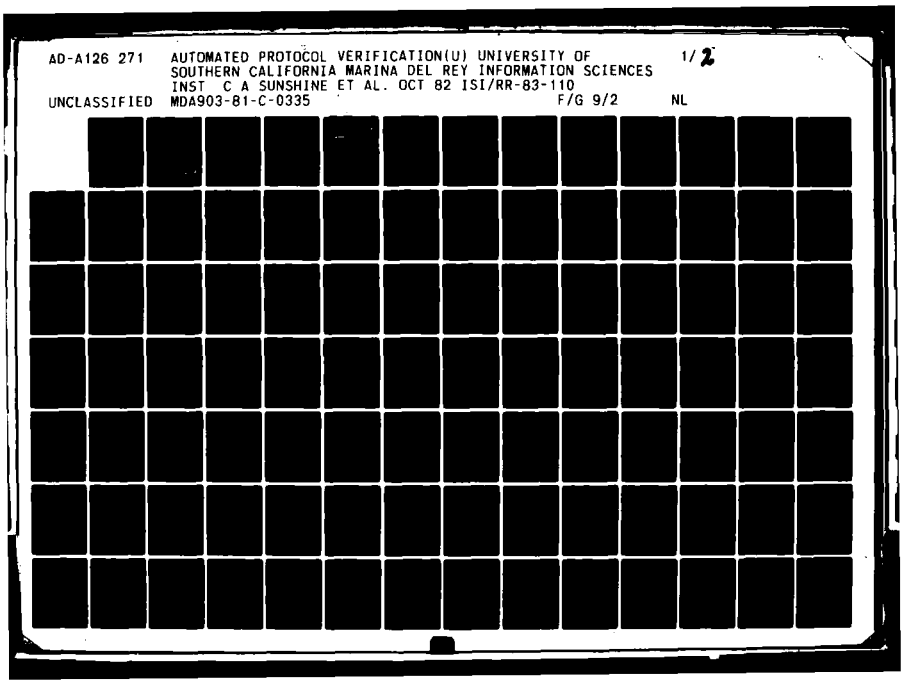
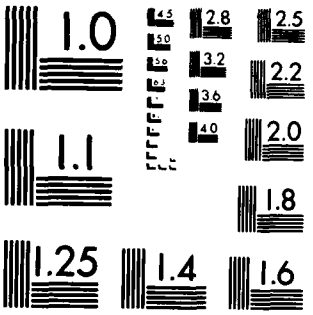

AD-A126 271
AUTOMATED PROTOCOL VERIFICATION(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST C A SUNSHINE ET AL. OCT 82 ISI/RR-83-110
1/2
UNCLASSIFIED
MDA903-81-C-0335
F/G 9/2
NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A126271

ISI/RR-83-110
October 1982

University of Southern California

Carl A. Sunshine
David A. Smallberg

Automated Protocol Verification

DTIC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC
ELECTE
APR 1 1983
A

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

83 04 01 019

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-83-110	2. GOVT ACCESSION NO. AD-A126	3. RECIPIENT'S CATALOG NUMBER 271
4. TITLE (and Subtitle) Automated Protocol Verification		5. TYPE OF REPORT & PERIOD COVERED Research Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Carl A. Sunshine and David A. Smallberg		8. CONTRACT OR GRANT NUMBER(s) MDA903 81 C 0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE October 1982 13. NUMBER OF PAGES 105
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

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

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

.....

18. SUPPLEMENTARY NOTES

Carl Sunshine's current address is: Sytek
19401 S. Vermont Ave., Suite J-104
Torrance, California 90502

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

abstract machine, Affirm, automated verification, computer network, Concurrent State Deltas, Formal Development Methodology, formal modeling, Gypsy, Ina Jo, induction, protocol, specification, state transition, symbolic execution, theorem prover, verification

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

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

20. ABSTRACT

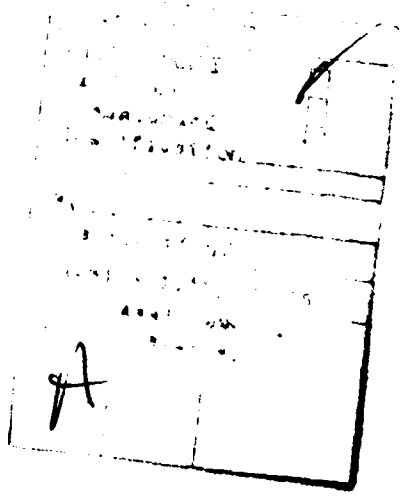
Four general-purpose automated verification systems (Affirm, Formal Development Methodology (Ina Jo), Gypsy, and Concurrent State Deltas) were applied to computer network protocols in order to evaluate the ability of such systems to provide significant results in formal protocol specification and verification. Each system had a particular strength: Affirm was most polished and flexible; FDM supported abstract machine specifications directly; Gypsy supported "stateless" I/O history type specifications; CSD supported more automatic proof and timing properties. However, none of the sytems had all necessary features or was powerful enough to fully handle complex protocols. The relative strengths of the systems are compared, detailed examples of their use are presented, and suggestions for further work are given.

ISI/RR-83-110
October 1982

University of Southern California

Carl A. Sunshine
David A. Smallberg

Automated Protocol Verification



INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

ACKNOWLEDGMENTS

We would like to thank the developers of the Affirm, FDM (Ina Jo), Gypsy, and State Delta systems for making their systems available to us. Special thanks go to Susan Gerhart, Dave Thompson, Dan Schwabe, Chris Landauer, Sue Landauer, Ben DiVito, and Bill Overman for their collaborative work and help rendered while we were trying to use the systems.

CONTENTS

FIGURES

1. INTRODUCTION

For the past two years, a major element of our protocol research at USC/Information Sciences Institute has been an effort to apply existing automated verification systems to communication protocols. Initially our work focused on the Affirm system [Gerh 80, Suns 81a]. More recently we have experimented with the Formal Development Methodology (FDM, also known as Ina Jo), Gypsy, and State Delta systems. This report presents the detailed results of our work with the latter three systems. A shorter preliminary report on this material may be found in [Suns 82a].

The work described here represents the conclusion of our protocol verification work. A summary of all work done in this area and a complete list of reports produced may be found in [Suns 82b].

The four automated verification systems we studied were chosen for a combination of factors including initial estimates of quality, significance, representation of important approaches, and availability to us. Other systems that we (perhaps unfairly) excluded from further consideration were HDM [HDM 79] and the associated Boyer-Moore theorem prover [BoMo 79], SPV [SVG 79], Ordinary [Gogu 81], Approver [Haje 77], Perturbation Analysis [West 82], Ogive/Ovide [Diaz 82], Cesar [QuSi 81], and Sara [RaEs 80]. Overviews and comparisons of some of these systems may be found in other comparative evaluation efforts [Suns 81c], [Diaz 82], [Crai 81].

This report is organized into sections, one on each verification system. Each section begins with some general background on the system and then presents some comments on our experience applying that system to protocols. Conclusions relevant to the individual system appear at the end of each section, while overall conclusions and comparisons are in a separate final section.

Our major interest throughout this work has been on design verification rather than code or implementation verification. Hence we have attempted to develop "abstract" specifications for the services and entities of a given protocol layer and to prove that the combined operation of the entities plus the lower layer service has certain properties or meets some service specification. We have been less interested in the problem of verifying that a specific program or code correctly implements a protocol entity.

A uniform set of example protocols were employed with each system. These were the well-known Alternating Bit protocol (in a form including arbitrary message loss and retransmission) and the "three-way handshake" connection-establishment protocol from the DARPA TCP [Post 81]. The former served to test capabilities of the systems to handle "data transfer" type functions, while the latter served to test control functions.

Our goal was largely methodological in this work: to evaluate the ability of existing automated verification systems to provide useful results in the domain of communication protocols. We did not expect to discover errors, since the protocols used as examples were quite mature, although our work with Affirm did reveal an obscure bug with the three-way handshake [Schw 81]. The bug has since been corrected.

2. FORMAL DEVELOPMENT METHODOLOGY (FDM)

Our main interest in FDM was its explicit support for abstract machine models. Thus we expected it would be easy to specify our example protocols, already formulated in terms of an abstract machine model. We also hoped to use the explicit mapping constructs in FDM to do hierarchical proofs showing that a protocol implemented its service. Finally, we wished to experiment with the automated tools associated with FDM, particularly the interactive theorem prover (ITP), to assess their capabilities.

2.1 BACKGROUND

As in Affirm and Gypsy, FDM includes a specification language (Ina Jo[1]), a language processor, a verification condition (VC) generator, and an ITP. The system is specifically intended to model hierarchies of abstract machines, with mappings from higher to lower layers defined. The language is an extension of predicate calculus with some built-in data types (integers, booleans, enumerated sets, lists, records), and it allows the definition of new subtypes and combinations of these types [Loca 80].

The basic unit of specification is an event (called "transform") for which the effect on all state variables must be defined. "No change" is assumed for all variables not mentioned. Properties to be proved about the highest level specification may be conventional invariants (called "criteria") and may also include "constraints" relating the values of state variables before and after an event. (We have not found a need for constraints in any of our protocol examples.)

The Ina Jo language processor converts specifications (including properties to be proved) into theorems to be proved, one for the initial conditions and one for each transform stating that the properties are maintained. All proof is by contradiction, so these theorems are in a form such that a contradiction must be shown between the hypotheses and all disjuncts of the conclusions (see below). Thus the ITP is more specialized than in Affirm, with induction (over the transforms) and proof by contradiction built in. The prover also automatically generates a number of "corollaries" to each expression resulting from a proof step, based on a large set of built-in facts about the basic types and operators of the language. Either the direct result or any of these corollaries may be used in further proof steps.

FDM was developed by the System Development Corporation. It runs on an IBM 4331 VM/370 system and is based on a LISP-like proprietary compiler writing system called CWIC.

[1] Ina Jo is a registered trademark of the System Development Corporation.

2.2 SPECIFICATION

Our initial work with Ina Jo involved redoing a small example involving several restaurant patrons ordering and paying their bills [Suns 81b]. This provided some useful comparisons with Affirm and helped us develop a method for translating between Affirm and Ina Jo specifications.

Some preliminary work had already been done in cooperation with Chris Landauer at SDC to specify the Alternating Bit protocol and service. These specifications were indeed very easy to produce from the corresponding Affirm specifications. The translation strategy we developed was quite straightforward, with Affirm "selector functions" becoming Ina Jo state variables, axioms with constructor functions becoming transforms, and theorems becoming criteria (or invariants). The Affirm axioms for the "initial" event required special treatment and translation into initial value assignments for the state variables in Ina Jo. The induction schema from Affirm were unnecessary, since induction is built into the ITP (see above). Lack of data type definition facilities in Ina Jo could have been a problem, but the built-in data types proved adequate for our examples.

The resulting specification for the Alternating Bit service and protocol are shown in Figures 2-1 and 2-2. A major feature to be noticed is the terse operator syntax chosen for Ina Jo. This certainly reduces understandability for the casual user, but it also makes for less typing and shorter listings and is thought by some experienced users to be an advantage. In addition to the convention that state variables not mentioned at all in a transform remain unchanged, the "No Change" operator is a useful abbreviation for the frequent {new value of X = old value of X} construct.

Because of limitations in the methodology (discussed in Section 2.3), our main efforts with Ina Jo were directed to the three-way handshake example. Once again, it was fairly easy to develop a specification in Ina Jo from our Affirm work. Time limitations required some simplifications. A reliable medium was assumed, so retransmission was eliminated. However, nondeterministic choice of either active, passive, or no open request was allowed so that simultaneous connection attempts and resets were possibilities. Another shortcut was to specify only one node and its properties, eliminating the "mirror image" portion of each proof.

The resulting specification is shown in Figure 2-3. As noted above, transforms for only one node are given (left), although state variables for both sides are defined. The "user command" events (Active Open and Passive Open) are treated uniformly with the receipt of messages from the medium. Each has its effects defined in a transform that takes the form {if preconditions are true, **then** state changes, **else** nothing happens (except consumption of the input)}. We could have chosen to

```
Key:    NC" means no change to     N" means new value of
        T" means type              ;; means concatenation of two lists
        .n means list element n    .x means structure element x
        :n means rest of list starting from element n
        ;. means concatenation of list and element
        x => y <> z means if x then y else z

specification AB_service
level service

type message
type QofMessage = list of message
type States = (IDLE, BUSY)
variable Sent, Received, Buf : QofMessage
variable ServState : States

criterion
((Buf=NIL <-> ServState=IDLE)
 & (Buf:2 = NIL)
 & (Sent = Received ;; Buf ))

initial
Sent=NIL & Received=NIL & Buf=NIL & ServState=IDLE

transform Send(m:Message) external
  effect
    (ServState=IDLE =>
      N"Sent = Sent;.m & N"Buf = Buf;.m & N"ServState = BUSY
    <> NC"(Sent, Buf, ServState) )

transform Rcv external
 effect
    (ServState=BUSY =>
      N"Received = Received;.Buf.1 & N"Buf = Buf:2 & N"ServState = IDLE
    <> NC"(Received, Buf, ServState) )

end service
end AB_service
```

Figure 2-1: Alternating Bit protocol service in Ina Jo

produce an explicit response (OK or error) in an additional "user interface" state variable as part of each user command's effects, but this was largely extraneous to the correct functioning of the protocol we wished to investigate.

2.3 VERIFICATION

As noted above, our first experience with FDM's ITP was a simple example (about restaurant diners) that served to identify differences and similarities with Affirm [Suns 81b]. This experience highlighted the fact that induction is built into the Ina Jo language processor, which produces

```
specification AB_Protocol
level protocol

type message
type seqnum=T"i:integer (i >= 0)
type packet = structure of
  (seq = seqnum,
  text = message)
type queueofmessage = list of message
type queueofpacket = list of packet

variable Sent, Rcvd: queueofmessage
variable PBuf, ABuf, Pending: queueofpacket
variable SSN, RSN: seqnum

define seqmatch(p: queueofpacket, s:seqnum): Boolean ==
   p ~= nil & (p.1).seq = s
define extract_text(p:queueofpacket): queueofmessage ==
  (p = nil => nil <> extract_text(p:2) ; (p.1).text)

criterion
  Sent = Rcvd ;; extract_text(Pending)

initial
  Sent=nil & Rcvd=nil & PBuf=nil & ABuf=nil & Pending=nil & SSN=0 & RSN=0

transform Send(M:message) external effect
  (Pending = nil => N"Pending = Pending ;. (SSN,M) & N"PBuf = PBuf ;. (SSN,m)
                 <> NC"(PBuf, Pending)

transform Rcv external effect
  ( PBuf ~= nil => N"PBuf = PBuf:2 & N"ABuf = ABuf ;. PBuf.1
                <> NC"(PBuf, ABuf))
  & (seqmatch(PBuf,RSN) => N"RSN = RSN+1 & N"Rcvd = Rcvd ;. (PBuf.1).text
                        <> NC"(RSN, Rcvd))

transform Update external effect
  (ABuf ~= nil => N"ABuf = ABuf:2 <> NC"(ABuf))
  & (seqmatch(ABuf,SSN) => N"Sent = Sent ;. (Pending.1).text
                           & N"Pending = nil & N"SSN = SSN+1
                        <> NC"(Sent, Pending, SSN))

transform LosePkt external effect
  (PBuf =nil => N"PBuf = PBuf:2 <> NC"PBuf)

transform LoseAck external effect
  (ABuf =nil => N"ABuf = ABuf:2 <> NC"ABuf)

transform Timeout external effect
  N"PBuf = (Pending = nil => PBuf ;. Pending.1 <> PBuf)

end protocol
end AB_Protocol
```

Figure 2-2: Alternating Bit protocol in Ina Jo

```
specification TCP3
level mechanism

type SeqNum = T"i:Integer(i >= 0)
type Message
type PacketOp = (Syn,SynAck,Ack,Reset)
type Packet = structure of
    (Seq = Integer,
     Ack = SeqNum,
     Op = PacketOp )
type States = (CLOSED,LISTEN,SYNSENT,SYNRECEIVED,ESTABLISHED)
type Channel = list of Packet

variable LState:States
variable LSeqToSend,LSeqToRcv,LOldUnack:SeqNum
variable LtoR:Channel
variable RState:States
variable RSeqToSend,RSeqToRcv,ROldUnack:SeqNum
variable RtoL:Channel

criterion
LState=ESTABLISHED -> LSeqToSend=RSeqToRcv & RSeqToSend=LSeqToRcv

initial
LState=CLOSED & RState=CLOSED & RtoL=NIL & LtoR=NIL
 & LSeqToSend=0 & LSeqToRcv=0 & LOldUnack=0
 & RSeqToSend=0 & RSeqToRcv=0 & ROldUnack=0

define LAckTest:Boolean == (RtoL.1).Ack = LOldUnack+1
define LSeqTest:Boolean == (RtoL.1).Seq = LSeqToRcv
define MakePkt(s1:SeqNum, s2:SeqNum, o:PacketOp):Packet == (s1,s2,o)

transform LActiveOpen External
effect (LState = CLOSED
=> N"LState=SYNSENT &
   N"LSeqToSend = LSeqToSend+2 &
   N"LtoR = LtoR;.MakePkt(0,0,Syn) &
   N"LOldUnack = LSeqToSend+1
<> NC"(LState,LSeqToSend,LtoR,LOldUnack))

transform LPassiveOpen external
effect (LState = CLOSED
=> N"LState=LISTEN
<> NC"(LState))

transform LRcvReset external
effect (RtoL~=NIL & (RtoL.1).Op=Reset
=> N"RtoL = RtoL:2
   & (LState=SYNSENT & LAckTest
     |  LState~=SYNSENT & LState~=LISTEN & LSeqTest
   => N"LState = CLOSED
   <> NC"(LState))
<> NC"(RtoL,LState))
```

Figure 2-3: Simple three-way handshake protocol in Ina Jo

```
transform LRcvAck external
effect (RtoL~=NIL & (RtoL.1).Op=Ack
=> N"RtoL = RtoL:2
   & ( LState=SYNRECEIVED & LAckTest & LSeqTest
     => (N"LOldUnack=LOldUnack+1 & N"LState=ESTABLISHED)
     <> NC"(LOldUnack,LState)  )
   & N"LtoR=(LState=CLOSED | LState=LISTEN | LState=SYNSENT & ~LAckTest
       | LState=SYNRECEIVED & LSeqTest & ~LAckTest
     => LtoR;.MakePkt((RtoL.1).Ack,0,Reset)
     <> (LState=SYNRECEIVED & ~LSeqTest
        => LtoR;.MakePkt(LSeqToSend,LSeqToRcv,Ack)
        <> LtoR ))
<> NC"(RtoL,LOldUnack,LState,LtoR))

transform RcvSyn external
effect (RtoL~=NIL & (RtoL.1).Op=Syn
=> N"RtoL = RtoL:2
   & N"LSeqToSend = (LState=LISTEN => LSeqToSend+2 <> LSeqToSend)
   & N"LOldUnack = (LState=LISTEN => LSeqToSend+1 <> LOldUnack)
   & N"LSeqToRcv = (LState=LISTEN | LState=SYNSENT
     => (RtoL.1).Seq + 1 <> LSeqToRcv)
   & N"LState = (LState=LISTEN | LState=SYNSENT
     => SYNRECEIVED
     <> LState )
   & N"LtoR = (LState=SYNSENT
     => LtoR;.MakePkt(LSeqToSend,(RtoL.1).Seq+1,Ack)
     <> (LState=LISTEN
         => LtoR;.MakePkt(LSeqToSend+1,(RtoL.1).Seq+1,SynAck)
         <> (LState=CLOSED
             => LtoR;.MakePkt(0,(RtoL.1).Seq+1,Reset)
             <> LtoR)))
<> NC"(RtoL,LSeqToSend,LOldUnack,LSeqToRcv,LState,LtoR))

transform LRcvSynAck external
effect (RtoL~=NIL & (RtoL.1).Op=SynAck
=> N"RtoL = RtoL:2
   & ( LState=SYNSENT & LAckTest
      => (N"LOldUnack=LOldUnack+1 & N"LState=ESTABLISHED
          & N"LSeqToRcv=(RtoL.1).Seq+1)
      <> NC"(LOldUnack, LState, LSeqToRcv) )
   & ( (LState=CLOSED | LState=LISTEN | LState=SYNSENT & ~LAckTest
       | LState=SYNRECEIVED & LSeqTest & ~LAckTest)
      => N"LtoR = LtoR;.MakePkt((RtoL.1).Ack,0,Reset)
      <> ( (LState=SYNRECEIVED | LState=ESTABLISHED) & ~LSeqTest
         | LState=SYNSENT & LAckTest
         => N"LtoR = LtoR;.MakePkt(LSeqToSend,N"LSeqToRcv,Ack)
         <> NC"(LtoR) ))
<> NC"(RtoL,LOldUnack,LState,LSeqToRcv,LtoR))

end mechanism
end TCP3
```

Figure 2-3: Simple three-way handshake protocol in Ina Jo (continued)

theorems to be proved from specifications, and that proof by contradiction is built into the ITP. It also familiarized us with the automatic generation of corollaries (some useful, most not) and the inability to introduce lemmas (without proving them) during the course of a proof. These and other points about ITP are illustrated in the following discussion.

We particularly wanted to use Ina Jo's mapping constructs to show that a lower level specification (the protocol) properly implements a higher level specification (the service). Ina Jo provides constructs to define how each higher level state variable and operator is implemented in the lower level. Unfortunately, these constructs only support a fixed mapping of events: One higher level event may be defined to be implemented as a single or fixed sequence of lower level events. For the Alternating Bit protocol, the Send service event is accomplished at the protocol level by a nondeterministic series of send, message loss, resend, acknowledge, Ack loss, and Ack receive events. Such nondeterministic sequences cannot be expressed in Ina Jo, and so we could not make use of the mapping facilities or perform any hierarchical proofs.

We were able fairly easily to complete a proof that the top (service) level specification of the Alternating Bit protocol met its criterion. Our main discovery here was that a number of lemmas about lists not known to the ITP had to be supplied. The ITP has built in a large number of lemmas that are automatically supplied as corollaries whenever applicable. The user may supply additional lemmas at the beginning of a proof (in "library" mode) when they may be assumed true (temporarily). But if the proof progresses to a point where a new lemma is discovered to be necessary, either the lemma must be proved at that point before proceeding or the proof must be restarted with that lemma introduced at the beginning so it may be assumed and applied later. This makes any but the most trivial proof a somewhat frustrating and repetitive process.

2.3.1 Proof of the Three-Way Handshake

Our major proof efforts were directed to the three-way handshake. The main correctness property (criterion) for this protocol (from Figure 2-3) states that sequence numbers in the two nodes are properly synchronized when a connection is established:

```
LState=ESTABLISHED -> LSeqToSend=RSeqToRcv & RSeqToSend=LSeqToRcv
```

When we tried to prove this property, it became clear that a number of lemmas would be necessary. Because of the difficulty just mentioned of introducing lemmas into an ongoing proof within the ITP, we decided to produce an equivalent specification in Affirm and to develop the proof there, where lemmas could be introduced when needed.

A lengthy trial-and-error process then ensued as we attempted to discover an adequate set of lemmas within the Affirm system. This is typical in developing a new data type. One must choose between formulating a lemma that looks correct and using it to continue the main proof, or immediately proving the lemma. In the first case it may turn out that the lemma was not correct or was not strong enough to be proved on its own. In the second case, after the effort of proving the lemma, it may turn out to be not quite what was needed in the main proof after all. This discovery process would have been an order of magnitude more difficult if carried out with the FDM ITP.

Figure 2-4 gives an example of the successive refinements that were needed for one lemma. The lemma states essentially that for every state t of the system with a Syn packet in the left-to-right medium, every other packet p in the medium after the Syn packet has a higher sequence number. The first version of the lemma, which stated this only for a Syn packet at the front of the medium, proved too weak. The second version generalized this to a packet p1 anywhere in the medium (between queues q1 and q2, each possibly empty). This version proved too unwieldy, so version three introduced a definition of a "splice" operation sp(q,p,r) for the relation "sequence with packet p between two subsequences q and r". Version 4 corrected a final error by eliminating Reset type packets from consideration (they have different sequence number generation rules).

Our proof efforts with Affirm used each of these versions in trying to prove the main theorem. When the lemma was discovered to be unsatisfactory, we were able to define a new version, return to the places in the main proof where the lemma was used, replace it with the new version, and redo only those portions of the proof affected. This ability to "random access" and replace portions of a proof tree in Affirm was very important in developing successful proofs of complex systems.

To indicate the number and type of lemmas needed to prove the main correctness property given above, Figure 2-5 shows a hierarchy of the lemmas that were developed in the course of the proof. The main theorem (Sync1) needed three major lemmas, each of which in turn needed some difficult lemmas as well as numerous specialized properties about queues, sequences, and the "splice" operation sp mentioned above. It is rather depressing that for even such a relatively lock-step example, such a rich structure of lemmas had to be developed about the system. The only redeeming hope is that once this structure is developed, more complex variations of the system (e.g., with retransmission) could be handled with relatively minor modifications. Unfortunately, we have not had time to test this hypothesis extensively.

Another aspect of the Affirm work is worth mentioning. We originally modeled the system as a state transition system with an overall state variable as a parameter to each function [Suns 81a]. To prove

```
theorem SynLowSeq1,
all t,p (Control(Front(LtoR(t)))=Syn and p in Remove(LtoR(t))
imp Seq(p) > Seq(Front(LtoR(t))) );


theorem SynLowSeq2,
all t,q1,q2,p1,p2 (LtoR(t)=((q1 Add p1) Append q2) and
      Control(p1)=Syn and p2 in q2
imp Seq(p2) > Seq(p1) );


interface sp(q,p,r):SequenceOfElemType;

define sp(q,p,r) == (q apr p) join r;

theorem SynLowSeq3,
all t,q,r,l,m ( LtoR(t)=sp(q,l,r) and Control(l)=Syn and m in r
imp Seq(m) > Seq(l) );


theorem SynLowSeq4,
all t,q,r,l,m ( LtoR(t)=sp(q,l,r) and Control(l)=Syn and m in r
      and Control(m)~=Reset
imp Seq(m) > Seq(l) );
```

Figure 2-4: Refinement of a lemma (in Affirm)

some of the lemmas, it became apparent that historical reference would be necessary--that is, we had to formulate properties that explicitly referred to both current and previous states of the system. In our previous work with Affirm we had developed a method for using event sequences as the state parameter of the system to support such historical reference [Schw 81]. Theorem WasSyn3 is of this sort, stating that if the current state has a Syn packet in the medium, there must have been an earlier state when an Active Open was done to generate the Syn packet. Ina Jo directly supports statements relating values before and after a single transform with the "constraint" construct. The same approach of using an event sequence as major state variable could be used in FDM to deal with more distant historical reference, but this seems foreign to its explicit abstract machine orientation.

The proof effort that finally succeeded had several components. At the lowest level, an explicit event sequence specification in Affirm was used to prove lemma WasSyn3 and several other lemmas relating the sequence numbers of Syn and Ack packets in transit to one another and to the nodes. These lemmas were then used to prove GoodAck and GoodSynAck lemmas, which did not require historical reference and hence could be defined in terms of the specification without event sequence state variables. This definition directly corresponds to the Ina Jo model. For comparison, the main

theorem (and one other simple property) were then proved in both Affirm and ITP from these lemmas. Our final remarks on verification will be a comparison of these proofs.

2.3.2 Proof Examples

First we shall look at a simple property, which states that when a node is in states Synsent or SynReceived, its Sequence-Number-To-Send state variable is one greater than its Oldest-Unacked state variable (i.e., it has sent a Syn packet with sequence number X, which has yet to be acknowledged, and the next packet will carry sequence number X + 1).

The FDM ITP transcript for this criterion is shown in Appendix I. Note that Ina Jo creates a separate theorem for each transform, plus one for the initial conditions. User commands follow the "." prompt at the start of a line. We have used the list command to show the initial theorems (in the contradiction form) produced by ITP (all lines numbered X.1-Y). The form of the theorem for the initial conditions is

```
(initial conditions)
and not (criterion to be proved)
```

which is just the contradiction of `(initial conditions) imply (criterion)`. The form for each transform is

```
(criterion before transform)
and (effects of transform)
and not (criterion after transform)
```

Since each theorem is already in contradiction form, it must be shown false, not true.

For this simple property, all the proof commands to ITP involve selection of cases, substitutions to perform, and simplifications that must be requested explicitly (the "prove" command allows a component of some previous line, such as Boolean 1 (the true side of an implication), to be selected as a subcase). No lemmas are used, and about 25 commands are needed. In addition to the transcript, ITP produces a file of the commands used, shown (edited) at the end of Appendix I as a summary of the proof.

Note that only those lines affected by a proof step are output as new results by ITP. Previous results remain available by reference to their line numbers. Corollaries are also automatically generated whenever possible (all lines with a "-" in their number). The explanation for each result appears at the end of the line along with references to any lines from which it was derived. Each new subcase (or theorem) requested adds one more decimal point to the line numbers until it is proved.

```
Level   Theorems

1       Sync1

2       LSTS, GoodAck, GoodSyn

3       RSeqToRcvBig, RSeqSame, S2

4       SynLowSeq, SeqsGrow, SeqToSendBig

5       WasSyn, NextPkt, SeqToSendGrows


declare t,u: TCP5;
declare p,l: Packet;
declare q,r: QueueOfPacket;
declare e: Event

theorem Sync1,
   LState(t)=ESTABLISHED
   imp LSeqToSend(t)=RSeqToRcv(t) and LSeqToRcv(t)=RSeqToSend(t);

theorem GoodAck,
   p in RtoL(t) and Control(p)=Ack and Ack(p)=LOldUnack(t)+1
        and Seq(p)=LSeqToRcv(t) and LState(t)=SYNRECEIVED
   imp Seq(p)=RSeqToSend(t) and Ack(p)=RSeqToRcv(t);

theorem GoodSynAck,
   p in RtoL(t) and Control(p)=SynAck and Ack(p)=LOldUnack(t)+1
        and LState(t)=SYNSENT
   imp Seq(p)+1=RSeqToSend(t) and Ack(p)=RSeqToRcv(t);

theorem RSeqToRcvBig,
   p in RtoL(t) and Control(p)=Ack
   imp Seq(p) <= RSeqToSend(t) and Ack(p) <= RSeqToRcv(t);

theorem S2, RSeqToRcv(t) <= LSeqToSend(t);

theorem RSeqSame,
   LState(t) = SYNSENT and p in RtoL(t) and Control(p) = Ack
        and Ack(p) = LSeqToSend(t) and Seq(p) = Seq(Front(RtoL(t)))+1
        and Control(Front(RtoL(t))) = Syn
   imp Seq(p) = RSeqToSend(t);

theorem SynLowSeq,
   LtoR(t) = sp(q,l,r) and Control(l) = Syn and m in r
      and Control(m) ~= Reset
   imp Seq(m) > Seq(l);
```

Figure 2-5: Lemma hierarchy for three-way handshake proof (in Affirm)

```
theorem SeqToSendGrows,
   LSeqToSend(t apr e) >= LSeqToSend(t));

theorem SeqsGrow1, all u,q,l,r,m (LtoR(u) = sp(q, l, r)
                        and Control(l) ~= Reset
                        and m in r and (Control(m) ~= Reset)
                    imp Seq(m) >= Seq(l) );

theorem WasSyn3,
   LtoR(t) = sp(q,l,r) and Control(l) = Syn
   imp C(t,q,l));

   define C(t,q,l)==
   some u,v (t=sp(u,LActiveOpen,v) and LState(u)=CLOSED and LtoR(u)=q
      and LSeqToSend(u)=Seq(l)-1 and LSeqToSend(u apr LActiveOpen)=Seq(l)+1);

theorem NextPkt4,
   LtoR(t apr e) = LtoR(t) or D(t,e));

   define D(t,e) ==
      some m (LtoR(t apr e) = LtoR(t) apr m
              and (Control(m) = Reset
                  or (Seq(m) <= LSeqToSend(t apr e)
                     and LSeqToSend(t) <= Seq(m)));

theorem SeqToSendBig,
   m in LtoR(t)
   imp Control(m)=Reset or Seq(m) <= LSeqToSend(t);
```

Figure 2-5: Lemma hierarchy for three-way handshake proof (in Affirm) (continued)

The edited Affirm proof transcript for this same property is shown in Appendix II. Several automatic simplification features have been turned on, and then an initial command to use induction is given (line 81). (User commands are on the sequentially numbered lines after the U: prompt.) The system then completes the proof largely on its own, generating a case for each event in the induction schema for the type and simplifying the resulting expressions to TRUE in all but one case (where the user must request expansion of a definition). Then a proof summary is requested, showing the sequence of steps used in the proof, including those generated automatically. In the Affirm transcript, the complete theorem is rewritten after each proof step, not just the affected lines as in ITP. This can produce a lengthier and less relevant transcript, although production of all corollaries in ITP seems to even things out again. Specific expressions to operate on must be identified in Affirm by name (e.g., the invoke command) and perhaps by instance(s) rather than by line number.

Appendix III shows the proof of the main synchronization property in ITP. The first step is to read in the four lemmas needed from a file in "library mode" and to defer their proofs. Then each theorem to be proved is listed and appropriate substitutions, subcases, simplifications, and lemma instantiations are directed. The resulting proof is over 500 lines long, and it should be emphasized that this is a "clean" version, with all necessary lemmas already discovered and false paths eliminated.

The equivalent proof in Affirm is shown in Appendix IV. After the property to be proved and the necessary lemmas are read in, induction is requested. Much as in the ITP proof, this process yields a case for each event. However, Affirm is able to complete the first four cases automatically, yielding a shorter transcript. (In other proofs, ITP might proceed more automatically.) Note that the lemmas required in the two substantial cases (Ack and SynAck) are the same in both proofs, although the order of application varies. The end of the transcript shows the proof tree in summary form.

2.4 SUMMARY

It was indeed convenient to write state transition type specifications in Ina Jo, so long as only relatively simple data types were needed. The translation from Affirm specifications to Ina Jo or vice versa is quite straightforward for the type of specifications used. The No Change operator and default were useful abbreviations. Terse operator syntax reduced understandability by relatively new users but was felt by some experienced users to be advantageous.

Efforts to construct formal mappings from service level to protocol level for the Alternating Bit protocol were unsuccessful because a nondeterministic mapping was required to represent the faulty medium. Ina Jo supports only the fixed mapping of a higher level transform into lower level transforms. However, this is a common weakness of most systems supporting formal mapping. In other kinds of systems the mapping constructs (where they are applicable) have proved quite useful.

The proof process in Ina Jo is more specialized; induction and proof by contradiction are built into the specification processor and ITP, which produce theorems to be proved false. Our experience with Affirm shows that in some cases invariants can be usefully simplified before induction is employed, eliminating identical steps in each induction case. In the ITP, the proof-by-contradiction method proved a convenient way to break down proofs into components. The subcase selection commands were also well developed. Automatic generation of corollaries was a mixed blessing, since many were not used, but those that were used came for "free."

A serious shortcoming in Ina Jo is the requirement that all lemmas to be used in a proof be introduced before the proof is started or proved at their point of introduction. This makes the kind of proof development process that is inevitably necessary in a complex system highly frustrating and tedious. This is all the more true since there is no effective way to "replay" the commands of a previous proof effort other than by retyping them.

The FDM ITP is less polished than Affirm's, with occasional abends, unhelpful error messages, and not-so-pretty printing of results. In our experience, simplification of results is not as automatic as in Affirm, typically requiring more explicit substitute- and simplify-type commands to complete a proof. However, it is not clear that this would be true for other examples. The user is obliged to specify explicit line numbers of previous results to be used in a proof step, necessitating a hardcopy output device for effective use of the system. There is no capability to jump around the proof tree, trying different branches for a while (proof must proceed linearly), and backup and history listing facilities are rather crude.

3. GYPSY

Our main interest in Gypsy was its orientation toward buffer history type specifications with no explicit internal state variables. We also hoped to exploit the modular proof capabilities of Gypsy so that only those portions of a proof affected by changes would have to be redone.

3.1 BACKGROUND

Gypsy is a Pascal-based language with extensions supporting concurrent processing and program verification [Good 78, GoDi 81]. The language encourages program modularity by forbidding global variables. All interprocedural communication must take place through parameters. A procedure may start up the parallel execution of other procedures. These processes may communicate only through shared message queues, called buffers. A process may send a message to or receive a message from a buffer and will block if the buffer is full or empty, respectively.

To allow verification that a program performs the task it is supposed to perform, assertions may be attached to each procedure. An entry assertion must hold whenever the procedure is invoked; an exit assertion must hold whenever the procedure terminates; and a blockage assertion must hold whenever execution of the procedure is blocked waiting to send to or receive from a buffer. Gypsy enforces modular specifications by requiring that these external assertions not refer to internal variables of the procedure; instead, they must be expressed in terms of the procedure's parameters.

A procedure is proved to meet its external specification by the standard inductive assertion method. Every loop must contain an assertion which holds every time it is encountered during program execution. A verification condition (VC) generator follows every path through the program from one assertion to another, generating VCs which, if true, ensure that the procedure meets its specification. If the VC generator encounters a procedure call along some path, that procedure's entry condition must be checked and its exit assertion may then be assumed. When an operation is encountered which could cause the procedure to block, a VC is generated saying that if the operation blocks, then the procedure's blockage assertion holds.

The VC generator can prove only trivial VCs by itself. Others are left for the human user to prove with the assistance of the Gypsy theorem prover. The prover performs expression simplification automatically, but most other tasks (such as substitution of equalities, case splitting, and chaining on implications) are best done with human guidance. (There is a command which instructs the prover to try these techniques on its own, but the command doesn't usually work effectively until the last few steps of the proof.) The user may introduce lemmas at any point in the proof.

Specifications for Gypsy programs involving concurrent processes usually contain assertions about the sequences of messages that the processes send to or receive from their message buffers. Gypsy supports the user in making and proving statements about these buffer histories. There is no support, however, for histories of another sort--those of program states--so in general it is impossible to make assertions about liveness properties in Gypsy. For this reason, we considered only safety properties in our specifications of the Alternating Bit and three-way handshake protocols.

Unlike the other systems we worked with, which require nonprocedural formulations of specifications, Gypsy encourages the user to write specifications as programs, at least for concurrent systems. This is because the way to specify a multiprocess system in Gypsy is to define an overall procedure with the individual processes in a Cobegin statement and their interconnecting buffers specified as parameters (see below). Then the VC generator automatically manages the details of building theorems about a parallel program from the assertions describing the behavior of each component process.

Gypsy was developed at the University of Texas and runs on a DEC TOPS-20[2] system under ELISP.

3.2 ALTERNATING BIT PROTOCOL

Our specification of the Alternating Bit protocol followed that of DiVito [Divi 81] and will be only briefly described here. We wrote it to practice using the Gypsy system but did not prove the protocol completely. The detailed specification appears in Appendix V. The program texts of DiVito's and our specifications are quite similar; we differ in the way our assertions are stated. DiVito defined predicates which abstracted the notions of proper transmission and proper reception, whereas we expressed these notions as conjunctions of several properties which were more concrete. Gypsy seemed equally easy to use with either approach for a protocol as simple as this one. (For the three-way handshake protocol, though, we found that defining predicates to represent complex boolean expressions not only made the specification easier to read but improved the performance of the VC generator.)

We defined a packet to contain a one-bit sequence number and a message field. The main procedure starts the concurrent execution of the sender process, the receiver process, the sender-to-receiver medium process for message traffic, the receiver-to-sender process for acknowledgments, and a timer process. The transmission media are allowed to lose packets, but they may not reorder or duplicate them.

---

[2]DEC and TOPS-20 are trademarks of Digital Equipment Corporation.

The protocol was modeled as a nonterminating program--new messages could always appear in the sender's input queue. For this reason, we stated the protocol service property as a blockage condition. The property says that whenever a process is blocked, all messages taken from the sender's input queue appear in the same order in the receiver's output queue, with the possible exception of the last message, which might still be in transit.

Attempting to prove this property taught us our first lessons about Gypsy. We found that we could not write some of the blockage assertions for the sender and receiver procedures without first writing their program text, because until we knew when the processes could block, we didn't know what had to be true at those times. We also learned that Gypsy does not support proof by induction; we discuss this further below. Otherwise, we found Gypsy quite adequate for specifying and proving this simple data transfer protocol.

3.3 THREE-WAY HANDSHAKE

3.3.1 The Specification

The three-way handshake, being more state-oriented than the Alternating Bit protocol, presented us with difficult specification and proof problems. Because of our limited time, we simplified the protocol in several ways:

- Only one "connection" between a fixed pair of users is considered--there is no addressing.
- All packets in the system belong to the current incarnation.
- Nodes never retransmit packets.
- Each of the two nodes receives exactly one active or passive open command before any messages are sent. Note that we still allow the case where both nodes receive active opens.
- Sequence numbers are monotonically increasing, unbounded integers.

In this section we summarize our specification; the full specification appears in Appendix VI.

The main procedure Protocol sets up the two nodes, the two communications media (one for packet traffic in each direction), and the buffers linking them as shown in Figure 3-1. (Ignore the exit assertion and associated node parameters shown in Appendix VI for now.)

Each node is started with an arbitrary initial sequence number for packets to be sent and a command to perform (an active or a passive open).

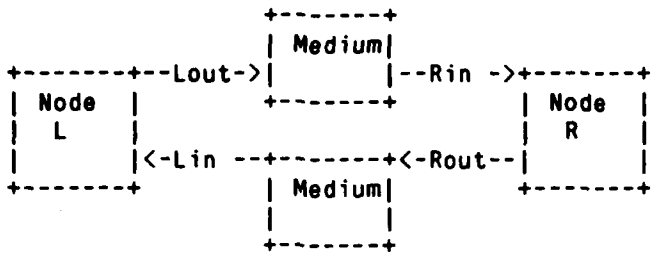


Figure 3-1: Diagram of the Protocol procedure

The underlying medium is assumed to be defined in a lower protocol layer. We use the property that any message delivered is one that was sent, with no guarantee that packets sent are delivered or that they are delivered in order with no duplications. Since we are not considering liveness, we need no other assumptions about the medium.

```
procedure Medium(var packets_in : packetbuff<input>;
                 var packets_out : packetbuff<output> ) =
begin
 exit all p : packet,
    p in outto(packets_out,myid)  ->  p in infrom(packets_in,myid);
 pending;
end; { Medium }
```

The Node procedure initially acts on the active or passive open with which it was called. It then repeatedly receives and acts on packets until it enters the established state. To keep the proof manageable (see the section on Complexity Limitation below), the details of packet processing in each different state are given in a separate procedure.

3.3.2 Complications

Exit vs. Block

When we attempted to state the protocol service property we wanted to prove, we found that we had to introduce some extra variables and statements into the program. We wished to show that when the nodes are in the established state, the sequence number of the next packet each node will send is the sequence number the other node expects to receive.

We first had to decide whether this property should be stated as an exit assertion or a blockage assertion of the Protocol procedure. As a blockage assertion, it would have to hold every time one of the medium or node processes blocked. Because the program contains many sends to buffers, there

are many ways the processes could block, but in only a few would both nodes be in the established state. This would cause many VCs to be generated, VCs which, while simple for us to prove, would not be recognized as trivial by the Gypsy VC generator. We felt that since there are many ways to block, but only one way to exit, casting the service property as an exit assertion would be more convenient. To do this, we had to force the medium processes to terminate once the nodes reached the established state. An extra boolean field, Vislast, was added to each packet. (The prefix V indicates the field is there for verification purposes only.) This field is normally false; but once a node is ready to exit, it sends an extra packet with the field true to the medium that receives its messages. When the medium receives this packet, it too exits.

Extra Variables

When we tried to state the service property, which relates the sequence numbers the nodes expect to send and receive, we encountered another problem. Because Gypsy forbids the exit assertion of the Protocol procedure from talking about variables defined in that procedure, we found it necessary to introduce the additional parameters VLseqno, VLackno, VRseqno, and VRackno to carry the necessary information. Just before the nodes exit, they set these parameters.

Complexity Limitation

In the loop in the Node procedure, where it is decided what action to take based on the current state and the incoming packet, we originally wrote one large case statement discriminating on the state, with each arm being a case statement on the input packet. We felt that the specification was more readable with the decision logic all in one place. With this organization, though, there were about 25 paths through the loop. We discovered that the VC generator could not handle this many paths, so we were forced to put the code in separate procedures (e.g., dosynsent). This meant restating the actions to be taken in the algebraic form required for exit assertions. Since Gypsy, unlike Ina Jo, has no abbreviated notation for "no change to these variables," the exit assertions were in fact more complex than the original program text.

3.3.3 Verification

Our biggest problem in the verification of the protocol was that since the protocol was originally presented as a state transition machine, we kept looking at proof techniques which modeled state exploration methods. At first, we were stumped because there was no way outside of the Node procedure to refer to the state variables of a node. Eventually we realized that we could derive the values of those state variables from the buffer histories of the node's input and output. We still thought that in our proof we would have to consider the states of both nodes together, exploring the large space formed by the cross product of each node's states. Fortunately, we found we could

formulate two properties (described below) that enabled us to prove the correctness of the nodes' interaction, leaving us with the now simplified task of proving a node's behavior correct independent of other processes.

Correctness of the Nodes' Interaction

The VC generated for the Protocol procedure is (after a little simplification using a lemma about buffer properties) two instances of the following lemma, one for each direction of packet flow:

```
lemma mainlemma(oseq1, iseq1, oseq2, iseq2 : packetseq) =
   [ estab = mystateof(oseq1,iseq1)  and
     estab = mystateof(oseq2,iseq2)  and
     seqnotosendprop(oseq1,iseq1)  and
     seqnotorcvprop(oseq2,iseq2)  and
     [ all p : packet,  p in iseq2  ->  p in oseq1 ]
   ]  ->
   seqnotosendof(oseq1,iseq1) = seqnotorcvof(oseq2,iseq2);
```

The function mystateof(oseq,iseq) yields the major state (e.g., estab) of a node whose input buffer history is iseq and whose output history is oseq. The functions seqnotosendof and seqnotorcvof similarly give the next sequence number the node would send and the next sequence number it expects to receive. Seqnotosendprop and seqnotorcvprop are the two properties mentioned above. Their exact specification is given at the end of Appendix VI.

The property seqnotosendprop states that if the node is in a state where it has sent a syn packet, then all syn packets it has sent have the same sequence number and the next sequence number to send is one greater than that number.

The property seqnotorcvprop says that if the node is in a state where a good syn has been received, then for some syn packet which has been received, the next sequence number to expect is one more than the number of that packet.

The proof of the main lemma appears in Appendix VII as a demonstration of the Gypsy prover.

Correctness of a Single Node

The exit property we wished to prove about the Node procedure says that the node is in the established state and the auxiliary parameters Vseqno and Vackno are set to the next sequence number to send and the next one to receive, respectively. These assertions must be made in terms of the histories of the node's input and output buffers. To prove this property, we insert an inductive assertion into the main loop of the procedure. The assertion states that the values of the internal variables are the values that could be deduced from the buffer histories.

```
procedure Node( ... ) =
begin
 exit
   estab = mystateof(outto(outbuff,myid),infrom(inbuff,myid))  and
   Vseqno = seqnotosendof(outto(outbuff,myid),infrom(inbuff,myid)) and
   Vackno = seqnotorcvof(outto(outbuff,myid),infrom(inbuff,myid));

     { initialization }
 ...

     { main loop }
 loop
  assert
   mystate = mystateof(outto(outbuff,myid),infrom(inbuff,myid))  and
   seqnotosend = seqnotosendof(outto(outbuff,myid),infrom(inbuff,myid)) and
   oldestunack = oldestunackof(outto(outbuff,myid),infrom(inbuff,myid)) and
   seqnotorcv = seqnotorcvof(outto(outbuff,myid),infrom(inbuff,myid));
  if mystate = estab then
    leave
  end;
  receive pin from inbuff;
  { make a state transition }
  ...
 end; { loop }

 Vseqno := seqnotosend;
 Vackno := seqnotorcv;
end; { Node }
```

This formulation requires us to prove approximately 100 lemmas about the state transitions (4 conjuncts each for about 25 transitions).

As an example, one of the lemmas is

```
  synrcvd = mystateof(oseq,iseq)  and
  pin.op = ack  and
  pin.seqno = seqnotorcvof(oseq,iseq)  and
  pin.ackno = 1 + oldestunackof(oseq,iseq)
-> estab = mystateof(oseq, iseq <: pin)    { "<:" means concatenate }
```

To prove this lemma within the Gypsy system, we must define the function mystateof precisely. The easiest way to do this seems to be to restate the transition function:

```
function mystateof(oseq, iseq : packetseq) : nodestate =
begin
 exit (assume
  ... and
  [ mystate(oseq,nonlast(iseq)) = synrcvd  and
    last(iseq).op = ack  and
    last(iseq).seqno = seqnotorcv(oseq,nonlast(iseq))  and
    last(iseq).ackno = 1 + oldestunackof(oseq,nonlast(iseq))
   -> result = estab
  ] and
  ...
 );
end;
```

Therefore, proving the node behaves properly means proving that this functional representation is equivalent to the program text representation of the state transition definition. We did not have time to do this part of the proof, but we foresee no problems here other than coping with tedium.

Correctness of seqnotosendprop and seqnotorcvprop

It appeared to us that these properties would require a proof by induction on sequences of state transitions. The Gypsy prover does not understand about induction, but the VC generator makes implicit use of induction when generating VCs for programs with loops. We devised a scheme which lets us use the VC generator to structure a proof by induction.

As an example, suppose we wish to prove that the size of any sequence is nonnegative:

```
lemma size_lemma(s : packetseq) =
  size(s) ge 0;
```

(At one time the Gypsy theorem prover did not have this fact built in.) We would reformulate this lemma as a function whose body is a program containing a loop:

```
lemma size_lemma(s : packetseq) =
 sizeprop(s);

function sizeprop(s : packetseq) : boolean =
begin
 exit result = [ size(s) ge 0 ];
 var t : packetseq := s;
 loop
  assert size(s) ge size(t);
  if t = null(packetseq) then
    leave
  end;
  t := nonlast(t);
 end;
end;
```

The VC generated by performing one iteration of the loop is

```
 size(s) ge size(t)  and  t ne null(packetseq)
-> size(s) ge size(nonlast(t))
```

which is the induction step needed to prove the lemma.

In the case of the properties seqnotosendprop and seqnotorcvprop, the loop in the Node procedure itself was used for the induction, since that loop goes through all possible state transitions.

Relaxing the Simplifications

Our specification assumed that each node received exactly one open command before any packets were received. If we removed this restriction so that open commands could arrive at any time, we would have to introduce incarnation numbers into packets in order to tell which connection requests they corresponded to. Statements involving sequence numbers would have to take these incarnation numbers into account. To make assertions about a node's behavior, we would have to be able to talk about the order in which open commands and packets arrived; Gypsy has ways of expressing the merger of histories of more than one buffer. We expect that a proof of this protocol would still use the properties seqnotosendprop and seqnotorcvprop, with incarnation numbers considered along with sequence numbers. We do not believe any other major properties would need to be discovered to complete the proof.

3.4 COMMENTS ON THE GYPSY THEOREM PROVER

Once we had some practice, we found the Gypsy theorem prover moderately easy to use. We were able to invent lemmas in the middle of a proof, with the choice of proving them immediately or deferring their proof until after the main theorem was proved. The prover, when instructed to substitute equalities, usually guessed the correct direction of the substitution; when it was wrong, we could easily tell it to substitute the other way. Techniques such as proof by contradiction and case splitting were built in.

We did run into a number of annoying deficiencies, however. The prover did not produce a proof tree at the end of a proof; we had to wade through transcripts to recall what steps we had taken, weeding out the false ones by hand. Proofs had to be completed in one sitting; there was no way to save the state of the proof, log off, and come back later to complete it. We were also continually frustrated at the gaps in the prover's understanding of built-in types; some properties of sequences, for example, were automatically used by the simplifier, while others had to be explicitly introduced and proved.

3.5 SUMMARY

By its restrictions on interprocedural communication, Gypsy encourages its users to develop modular specifications. At times, however, we felt that Gypsy imposed a little too much modularity. The prohibition against global variables sometimes required the introduction of extra variables in order to state properties to be verified. For protocols with many states, we found that the requirement that our exit assertions be written in terms of buffer histories forced us to restate in functional form the state transitions which had already been expressed as program text.

The implementation of Gypsy is continually being improved. We noticed that some of its limitations vanished during the course of our research. At this writing, those remaining include the absence of a recorded proof tree, the inability to interrupt and resume a proof, and the inconvenience of performing inductive proofs.

The major limitation of Gypsy for protocol specification and verification is that liveness properties cannot even be stated, much less verified.

4. STATE DELTA

Our major interest in the State Delta system was its ability to perform symbolic execution to accomplish proofs without user guidance. This was very attractive after our often tedious experience with interactive provers. We were also interested in testing whether the explicit time bounds supported in state deltas would facilitate proofs and allow the handling of progress properties as well as safety.

Because the State Delta system was in an early prototype state of development and hence was rather cumbersome and difficult to use, our experiments were quite limited. First we give some background; then we discuss the results of these limited efforts.

4.1 BACKGROUND

The State Delta system developed at USC/Information Sciences Institute includes a specification language and a symbolic execution system and simplifier for carrying out proofs [Croc 77]. The initial system covered only a single sequential process, but recent extensions to "concurrent state deltas" (CSDs) have been made by Overman [Over 81, OvCr 82]. The basic unit of specification is a CSD which gives a precondition, a postcondition, a read list, a mod list, and time bounds. The meaning of this CSD is that if the precondition ever becomes true, then at some future time within the specified time bounds the postcondition will be true, and in the interim only the variables on the read list will be referenced and only those on the mod list will be modified. There is also a Wait construct, which specifies a delay until either a given condition is satisfied or some maximum time transpires, with postconditions for either case.

Higher level specifications are themselves CSDs. Proof proceeds by determining which lower level CSDs are enabled (have true preconditions) from the preconditions of the high-level CSD and then symbolically executing all the low-level CSDs, keeping track of time and generating all possible interleavings of CSDs that are simultaneously completed in different processors. Any conflict in the use of shared variables is noted, and the proof succeeds if the symbolic execution necessarily leads to a state satisfying the high-level CSD's postconditions (and read list, mod list, and time bounds).

Unlike the previous proof systems, which are interactive, the symbolic execution is completely automatic and requires no user aid. In practice, however, system resources are exhausted for specifications of any complexity, and the user must provide some appropriate intermediate CSDs to force pruning of the proof tree (identical states reached on different branches are not recognized

unless explicitly entered as intermediate CSDs). Induction is not directly supported and must also be introduced explicitly if needed, for example, via a loop CSD.

The CSD system was developed at USC/Information Sciences Institute and runs on a DEC TOPS-20 system under Interlisp.

4.2 ALTERNATING BIT PROTOCOL

Overman has already done some preliminary work on a simplified version of the Alternating Bit protocol. Figures 4-1 through 4-3 (from [Over 81]) show the state variables needed and the CSDs for Sender and Receiver processors. The "PC" terms in pre- and postconditions refer to the "major state" (or program counter) of each process. Note that the medium has been incorporated as CSD SB in the Sender and SB in the Receiver (with delay range between MINDELAY and MAXDELAY) to reduce the number of separate processes. In these CSDs, message loss is limited to MAXLOSS consecutive times to guarantee progress (otherwise the execution tree would clearly be infinite, including an unbounded number of message loss and retransmission events).

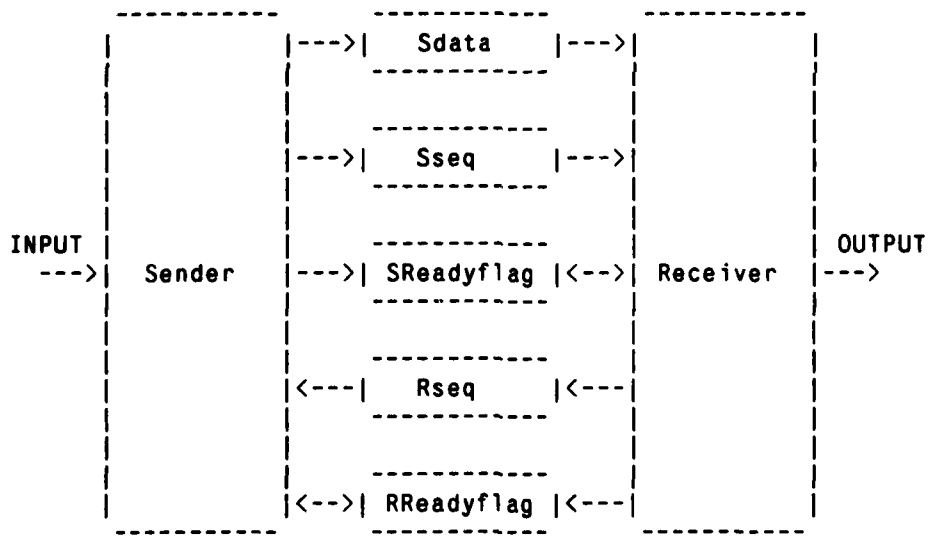


Figure 4-1: CSD Alternating Bit protocol block diagram

All CSDs other than SB are given time zero except the Sender's retransmission interval (SC), which is a Wait construct with maximum time TIMEOUT. The "io" clause is a special construct introduced to model interaction with the external environment or users.

```
SA    [CSD  pre:    (.SenderPC=A)
            read:   (INPUT)
            mod:    (SMessage SenderPC)
            procs:  (Sender)
            io:     (INPUT)
            post:   (#SMessage=.INPUT #SenderPC=B)
            time:   0]

SB    [CSD  pre:    (.SenderPC=B)
            read:   (Slost)
            mod:    (SenderPC)
            procs:  (Sender)
            post:   ((if .UNDEFINED=TRUE or .Slost ge MAXLOSS then #SenderPC=Bt
                       else #SenderPC=Bf))
            time:   (RANGE MINDELAY MAXDELAY)]

SBt   [CSD  pre:    (.SenderPC=Bt)
            read:   (SMessage SendSeqNo)
            mod:    (Sdata Sseq SReadyflag Slost SenderPC)
            procs:  (Sender)
            post:   (#Sdata=.SMessage #Sseq=.SendSeqNo #SReadyflag=TRUE #Slost=0
                      #SenderPC=C)
            time:   0]

SBf   [CSD  pre:    (.SenderPC=Bf)
            read:   (Slost)
            mod:    (Slost SenderPC)
            procs:  (Sender)
            post:   (#Slost=.Slost+1 #SenderPC=C)
            time:   0]

SC    [WAIT pre:    (.SenderPC=C)
            exp:    .RReadyflag=TRUE
            mod:    (SenderPC)
            procs:  (Sender)
            thenpost: (#SenderPC=Ct)
            elsepost: (#SenderPC=B)
            read:   (RReadyflag)
            time:   TIMEOUT]

SCt   [CSD  pre:    (.SenderPC=Ct)
            read:   (Rseq SendSeqNo)
            mod:    (RReadyflag SendSeqNo SenderPC)
            procs:  (Sender)
            post:   (#RReadyflag=FALSE
                    (if .Rseq=.SendSeqNo then #SendSeqNo=~.SendSeqNo
                         and #SenderPC=A
                     else #SendSeqNo=.SendSeqNo and #SenderPC=B))
            time:   0]
```

Figure 4-2: CSDs for the Sender

```
RA   [WAIT pre:    (.ReceiverPC=A)
           exp:    .SReadyflag=TRUE
           mod:    (ReceiverPC SReadyflag ReceivedSeqNo)
           procs: (Receiver)
           thenpost: (#SReadyflag=FALSE #ReceivedSeqNo=.Sseq
                      (if .ExpectedSeqNo=.Sseq then #ReceiverPC=At
                       else #ReceiverPC=B))
           read:   (SReadyflag Sseq ExpectedSeqNo)]

RAt  [CSD  pre:    (.ReceiverPC=At)
           read:   (Sdata ExpectedSeqNo)
           mod:    (OUTPUT ExpectedSeqNo ReceiverPC)
           procs: (Receiver)
           io:     (OUTPUT)
           post:   (#OUTPUT=.Sdata #ExpectedSeqNo=~.ExpectedSeqNo #ReceiverPC=B)
           time:   0]

RB   [CSD  pre:    (.ReceiverPC=B)
           read:   (Rlost)
           mod:    (ReceiverPC)
           procs: (Receiver)
           post:   ((if .UNDEFINED=TRUE or .Rlost ge MAXLOSS then #ReceiverPC=Bt
                      else #ReceiverPC=Bf))
           time:   (RANGE MINDELAY MAXDELAY)]

RBt  [CSD  pre:    (.ReceiverPC=Bt)
           read:   (ReceivedSeqNo)
           mod:    (Rseq RReadyflag Rlost ReceiverPC)
           procs: (Receiver)
           post:   (#Rseq=.ReceivedSeqNo #RReadyflag=TRUE #Rlost=0 #ReceiverPC=A)
           time:   0]

RBf  [CSD  pre:    (.ReceiverPC=Bf)
           read:   (Rlost)
           mod:    (Rlost ReceiverPC)
           procs: (Receiver)
           post:   (#Rlost=.Rlost+1 #ReceiverPC=A)
           time:   0]
```

Figure 4-3: CSDs for the Receiver

For a simple case, MAXLOSS = 1 and TIMEOUT > MAXDELAY were assumed, so that at most one message was ever active in the system. The correctness property for this simple case states that if the system starts with properly synchronized state variables, it will return to the same state but with one message forwarded (indicated by the "io" part of the CSD). (A "." before a variable means its value when the CSD starts, and a " # " means its value when it ends.)

```
[CSD SPEC1
      pre:   (.SenderPC=A .ReceiverPC=A .SReadyflag=FALSE .RReadyflag=FALSE
              .Slost=0 .Rlost=0 .SendSeqNo=.ExpectedSeqNo)
      read:  (SenderPC ReceiverPC SReadyflag RReadyflag ReceivedSeqNo Sdata
              ExpectedSeqNo INPUT Slost Rlost SMessage Sseq Rseq SendSeqNo)
      mod:   (SenderPC ReceiverPC SReadyflag RReadyflag ReceivedSeqNo Sdata
              ExpectedSeqNo OUTPUT Slost Rlost SMessage Sseq Rseq SendSeqNo)
      procs: (Sender Receiver)
      io:    (INPUT OUTPUT)
      post:  (#SenderPC=A #ReceiverPC=A #SReadyflag=FALSE #RReadyflag=FALSE
              #Slost=0 #Rlost=0 #SendSeqNo=#ExpectedSeqNo)
      time:  (RANGE 2*MINDELAY 5*MAXDELAY+3*TIMEOUT)]
```

The proof for this case (shown in Appendix VIII) was accomplished automatically in a few minutes of CPU time and is discussed further in [Over 81].

The assumptions were then relaxed, first to allow an arbitrary value of MAXLOSS. This value required us to develop an induction CSD for the Sender and Receiver, stating that either transmission succeed or the loss counter be decremented by one each time an attempt was made to transmit a message or acknowledgment. The time limit on retransmission was then relaxed to allow retransmission while another message might still be under way. This significantly increased the asynchrony of the system and again required development of induction CSDs as well as several other intermediate states of the system. The correct formulation of these intermediate-level CSDs was a difficult process, requiring much ingenuity to see what was necessary and tedious refinement to get the formulation just right. Details are given in [Over 81].

4.3 THREE-WAY HANDSHAKE

Our main efforts in the State Delta system were with the three-way handshake protocol. An automatic reachability analysis seemed particularly attractive here because of similar work (done manually) by Sunshine in his Ph.D. dissertation [Suns 75].

One difficulty in specification stemmed from the requirement that exactly one state delta be active at any time. Hence simultaneous user and network inputs were not allowed, so it was necessary to start each side with a user command (active or passive open) and then allow only network inputs.

Our initial specification was an attempt to directly translate an abstract machine model into CSDs. There were four processors: S and its mirror image, T, for the TCP nodes, and a medium in each direction. Each CSD corresponded to a transition in an abstract machine model and hence was defined to take zero time (except for the medium). There was one Wait construct, which each

processor executed while awaiting input. The medium was assumed perfect, so no CSD was needed for retransmission in the nodes.

The symbolic execution of this specification ran for over four hours of CPU time without completing. Despite the lack of message loss or retransmission, there are many branches in the execution tree resulting from different interleavings of processors in the simultaneous Active Open case: The execution trace showed a depth of 14 interleavings before the final state was reached on one path!

Our first improvement was to eliminate the explicit processors for the media and to put the message transmission wait into each TCP node, as for the Alternating Bit protocol. The resulting specification (for one side) is shown in Appendix IX. (Note that this is the output of a "pretty printer" for the CSDs, which must actually be input as LISP S expressions.) Thus to send messages, a Wait construct (smed) was added to the node, causing it to wait until the other node had an empty input buffer. This limitation reduced the number of possible messages outstanding to one in each direction, which was adequate for this example and greatly reduced the possible interleavings, so that a complete proof tree was successfully generated in about one hour of CPU time (with a maximum depth of seven).

Note that the property to be proved is itself a CSD, stating that if both TCP nodes start in the CLOSED state with user Open commands in their input buffers, and the media are empty, then the system will reach the ESTABLISHED state with properly synchronized sequence numbers:

```
[CSD SPEC
   pre:  (.SState=Closed .TState=Closed
          .SIn.type=ActiveOpen and .TIn.type=PassiveOpen or
          .SIn.type=ActiveOpen and .TIn.type=ActiveOpen
          .SSendFlag=False .TSendFlag=False .MstBuf.type=Empty
          .MtsBuf.type=Empty)
   procs: (S T)
   post:  (#SState=Established #TState=Established
           #SSeqToReceive=#TSeqToSend #TSeqToReceive=#SSeqToSend)]
```

Even one hour of CPU time was still too much, so our next refinement was to identify an intermediate state that appeared on many execution branches and to break the overall CSD into two corresponding halves. This intermediate state (for the simultaneous Active Open case) has both nodes in the SynReceived state with properly sequenced Ack messages about to be sent:

```
(SState=SynReceived SBuf.type=Ack SSendFlag=TRUE SIn.type=Empty
 SBuf.seq=SSeqToSend=SMaxval+1=TSeqToReceive SOldUnack=SMaxval
 SBuf.ack=SSeqToReceive TBuf.ack=TSeqToReceive
 TState=SynReceived TBuf.type=Ack TSendFlag=TRUE TIn.type=Empty
 TBuf.seq=TSeqToSend=TMaxval+1=SSeqToReceive TOldUnack=TMaxval)
```

This refinement effectively eliminated duplicate paths in the lower part of the tree so that the proof could be completed in a total of 14 minutes (8 minutes for the first half up to this intermediate state, whose proof is shown in Appendix X, and 6 minutes for the second half).

Our next step was to use the queue data type (which had just been added to the system) for the media and to introduce the possibility for message loss (all sending CSDs) and retransmission (CSD sz). The resulting specification is shown in Appendix XI. Predicates have been introduced as abbreviations for the more complex expressions now required in the CSDs. Message loss is limited to MAXLOSS consecutive times, as in the Alternating Bit protocol.

An initial proof attempt of this new specification with MAXLOSS equal to 1 and retransmission timeout greater than all other CSD times ran for over 2 hours without completing even the simple Active/Passive opening case. With MAXLOSS equal to 0 (perfect media) and retransmissions limited to one per message sent, a proof completed in 90 minutes. Once again we identified an intermediate state of the system and broke the proof into two halves, as follows:

```
(SState=SynSent SIn.type=(ENQUEUE NULLQUEUE SynAck)
 SIn.seq=(ENQUEUE NULLQUEUE TMaxval) SIn.ack=(ENQUEUE NULLQUEUE SMaxval+1)
 SSeqToSend=SMaxval+1=TSeqToReceive SOldUnack=SMaxval
 SPending.type=Syn SPending.seq=SMaxval SPending.ack=0
 TState=SynReceived TSeqToSend=TMaxval+1 TOldUnack=TMaxval
 TPending.type=SynAck SPending.seq=TMaxval TPending.ack=TSeqToReceive
 (TIn.type=NULLQUEUE TIn.seq=NULLQUEUE TIn.ack=NULLQUEUE)
 or (TIn.type=(ENQUEUE NULLQUEUE Syn)STimeoutFlag=FALSE TTimeoutFlag=FALSE
    SIn.seq=(ENQUEUE NULLQUEUE SMaxval) SIn.ack=(ENQUEUE NULLQUEUE 0))
```

Using this intermediate state, the proof for the simple Active/Passive case was accomplished in seven plus seven minutes. Unfortunately, when we tried to reintroduce message loss, the proof again became unworkable, and lack of time precluded further refinements.

4.4 DISCUSSION

The CSD system is the only one to include time bounds and hence to be able to deal with progress concerns simultaneously with safety. If a proof succeeds, it shows that the goal is reached in the specifed time (if any), not merely that the goal may be reached. The time bounds may also be used effectively to eliminate paths from the execution tree that would otherwise have to be considered (e.g., specifying that retransmission interval is greater than transmission delay). However, the inclusion of time bounds also complicates the symbolic execution and so is not always practical.

The basic flavor of CSD specifications is quite different from an abstract machine with atomic events separated by long periods when nothing happens. With state deltas, the events have a definite duration and the atomic points in time are their completion/commencement. State variables may change values during a CSD, since only their values at its completion are specified. Since the symbolic execution traces give the sequence of CSD completions, it is difficult to compare them with the state reachability graphs of conventional abstract machine models.

Another limitation of CSDs is the requirement that exactly one CSD be firable at any moment within a single processor. This causes difficulties in modeling nondeterministic behavior such as the condition when both a user input and a message from the network are queued for processing by an entity.

The CSD system is in an early stage of development and hence is still rather clumsy to use. There is little documentation, and only the system's implementors are really capable of using it. The specification language is simply LISP expressions with a particular form required, so input of specifications is rather painful. If a small portion of the specification is changed, there is no capability to determine which portions of previous proofs might be unaffected and thus avoid repeating them.

For simple cases, where the CSD system can complete a proof automatically, it is clearly superior to interactive provers. This is particularly relevant for the state reachability type of properties important in the three-way handshake. The difficulty of proving the simple (no loss or retransmission) case with CSDs was much less than with the systems based on invariant properties. However, when behavior becomes more complex, a great deal of human ingenuity is still required to formulate successful intermediate-level CSDs, many having the form of invariant properties that must be proved by induction. In this case, the kind of insight needed and the difficulty with all the proof systems becomes similar.

5. CONCLUSIONS

Two results of our experience with automated verification systems are clear: None of the systems has all the features desired, and none of them is ready for routine or mechanical application to real-world protocols. All of the interactive systems lack ability to complete seemingly simple or similar portions of a proof themselves, while symbolic execution in the CSD system shows promise in this dimension. All of the systems except Gypsy are missing the capability to omit redoing portions of a proof unaffected by small changes in the specifications or theorems. Affirm omits proof by contradiction, while FDM insists on it. Only Affirm supports induction directly. FDM lacks the ability to introduce lemmas on the fly, as they are needed in the course of a proof.

With the exception of the CSD system, none of the systems is able to handle progress or liveness properties very well. And despite much effort to provide support, hierarchical verification (e.g., of a formal service specification rather than just a set of plausible properties) remains quite difficult for protocols because of their nondeterministic error-recovery mechanisms. Certainly none of the systems has integrated the kind of performance or even probabilistic concerns [Yemi 82] necessary to go beyond functional correctness of protocols.

If we tried to rank the systems by their ease of use and maturity, Affirm would be a clear first, with Gypsy and FDM in a second category and CSD a distant third.

It should be noted that our experience and hence conclusions were colored by our emphasis on verification of protocol designs (i.e., specifications) rather than code. Affirm is particularly strong in this area, while Gypsy and FDM are oriented more toward development and proof of operational code and include features for these purposes that were not fully exercised by our experiments.

Our experience confirms the fact known to verification experts, but not widely appreciated by others, that the major contribution of automated verification systems is NOT to reduce the amount of human ingenuity required to accomplish a proof but rather to increase the certainty of correctness. If the user has the ingenuity to formulate the problem in a tractable fashion and the stamina to follow through all the tedium, the formally verified conclusion does seem to be far more reliably correct than that of hand proofs. Thus some useful results, albeit at high cost, can be obtained from current automated verification systems in analyzing features of real-world protocols.

A number of useful improvements identified in the course of this research have already been incorporated into several of the systems. We feel that the field as a whole shows sufficient promise

that more widespread and routine formal verification of protocol designs may be feasible within a few years if research into automated verification continues to be supported. Whether the best features of different systems can be combined into one more successful system remains a tantalizing question.

I. FDM ITP PROOF TRANSCRIPT (EDITED) FOR SIMPLE LEMMA

Notes:

Commands follow the "." prompt and end with the "!" prompt:
- List (line number range)
- Simplify (line)
- Substitute (into line from lines)
- Instantiate (line with expression)
- Prove (subcomponent of line)
- Theorem (expression)
- Contradiction (line numbers)

"*" as line number signifies the most recent line.

```
SDC'S INTERACTIVE THEOREM PROVER, RELEASE 11.35I
ENTER LIBRARY MODE
.leave!

LEAVE LIBRARY MODE

THEOREM FROM LEVEL MECHANISM FOR: INITIAL CONDITIONS
READY FOR INPUT 48.1-12

.l *-1 *-12!
 48.1-1  LState = CLOSED  (48.1)  'AND SPLIT'
 48.1-2  RState = CLOSED  (48.1)  'AND SPLIT'
 48.1-3  LSeqToSend = 0  (48.1)  'AND SPLIT'
 48.1-4  LSeqToRcv = 0  (48.1)  'AND SPLIT'
 48.1-5  LOldUnack = 0  (48.1)  'AND SPLIT'
 48.1-6  RSeqToSend = 0  (48.1)  'AND SPLIT'
 48.1-7  RSeqToRcv = 0  (48.1)  'AND SPLIT'
 48.1-8  ROldUnack = 0  (48.1)  'AND SPLIT'
 48.1-9  RtoL = NIL  (48.1)  'AND SPLIT'
 48.1-10  LtoR = NIL  (48.1)  'AND SPLIT'
 48.1-11  LState = SYNSENT | LState = SYNRECEIVED  (48.1)  'AND SPLIT'
 48.1-12  LSeqToSend ~= LOldUnack+1  (48.1)  'AND SPLIT'

.subst *-11 *-1.1!
 48.2  CLOSED = SYNSENT | CLOSED = SYNRECEIVED  (48.1-11 48.1-1)
                                                SUBSTITUTION(48.1-1L)
 48.2-1  False  (48.2)  SIMPLIFICATION
48  NOT PRINTED  'Q.E.D.'

THEOREM FROM LEVEL MECHANISM FOR: LACTIVEOPEN
READY FOR INPUT 49.1-10

.l *-1 *-10!
 49.1-1  LState = SYNSENT | LState = SYNRECEIVED -> LSeqToSend = LOldUnack+1
                                                    (49.1)  'AND SPLIT'
 49.1-2  (LState = CLOSED
     =>  N" LState = SYNSENT & N" LSeqToSend = LSeqToSend+2
```

```
          & N" LtoR = LtoR;.MakePkt(0, 0, Syn) & N" LOldUnack = LSeqToSend+1
     <> N" LState = LState & N" LSeqToSend = LSeqToSend & N" LtoR = LtoR
          & N" LOldUnack = LOldUnack)  (49.1)  'AND SPLIT'
 49.1-3  N" RtoL = RtoL  (49.1)  'AND SPLIT'
 49.1-4  N" ROldUnack = ROldUnack  (49.1)  'AND SPLIT'
 49.1-5  N" RSeqToRcv = RSeqToRcv  (49.1)  'AND SPLIT'
 49.1-6  N" RSeqToSend = RSeqToSend  (49.1)  'AND SPLIT'
 49.1-7  N" RState = RState  (49.1)  'AND SPLIT'
 49.1-8  N" LSeqToRcv = LSeqToRcv  (49.1)  'AND SPLIT'
 49.1-9  N" LState = SYNSENT | N" LState = SYNRECEIVED  (49.1)  'AND SPLIT'
 49.1-10  N" LSeqToSend ~= N" LOldUnack+1  (49.1)  'AND SPLIT'

.p *-2 B1!
  49.2.1  LState ~= CLOSED  (9 2)  ASSUME
  49.2.2  AND NOT PRINTED  (49.2.1 49.1-8 49.1-7 49.1-6 49.1-5 49.1-4 49.1-3
                                                    49.1-2)  'SIMPLIFIED EFFECT'
  49.2.2-1  N" LState = LState  (49.2.2)  'AND SPLIT'
  49.2.2-2  N" LSeqToSend = LSeqToSend  (49.2.2)  'AND SPLIT'
  49.2.2-3  N" LtoR = LtoR  (49.2.2)  'AND SPLIT'
  49.2.2-4  N" LOldUnack = LOldUnack  (49.2.2)  'AND SPLIT'
  49.2.3  False  (49.1-1 49.2.2-4 49.2.2-2 49.2.2-1 49.1-9 49.1-10)
                                                    'SIMPLIFIED NEW CRITERION'
 49.2  LState = CLOSED  (49.2.3)  'Q.E.D.'
 49.3  AND NOT PRINTED  (49.2 49.1-8 49.1-7 49.1-6 49.1-5 49.1-4 49.1-3 49.1-2)
                                                        'SIMPLIFIED EFFECT'
 49.3-1  N" LState = SYNSENT  (49.3)  'AND SPLIT'
 49.3-2  N" LSeqToSend = LSeqToSend+2  (49.3)  'AND SPLIT'
 49.3-3  N" LtoR = LtoR;.MakePkt(0, 0, Syn)  (49.3)  'AND SPLIT'
 49.3-4  N" LOldUnack = LSeqToSend+1  (49.3)  'AND SPLIT'

.subst .1-10 *-2.1,*-4.1!
 49.4  LSeqToSend+2 ~= LSeqToSend+1+1  (49.1-10 49.3-2 49.3-4)
                                                    SUBSTITUTION(49.3-2L, 49.3-4L)
 49.4-1  False  (49.4)  SIMPLIFICATION
49  NOT PRINTED  'Q.E.D.'

THEOREM FROM LEVEL MECHANISM FOR: LPASSIVEOPEN
READY FOR INPUT 50.1-14

.1 *-12 *-14!
 50.1-12  N" LState = SYNSENT | N" LState = SYNRECEIVED  (50.1)  'AND SPLIT'
 50.1-13  N" LSeqToSend ~= N" LOldUnack+1  (50.1)  'AND SPLIT'
 50.1-14  LSeqToSend ~= LOldUnack+1 (50.1-9 50.1-11 50.1-13) 'NEW SUBSTITUTION'

.p *-2 B1!
  50.2.1  LState ~= CLOSED  (9 2)  ASSUME
  50.2.2  N" LState = LState  (50.2.1 50.1-11 50.1-10 50.1-9 50.1-8 50.1-7
              50.1-6 50.1-5 50.1-4 50.1-3 50.1-2)  'SIMPLIFIED EFFECT'
  50.2.3  LState = SYNSENT | LState = SYNRECEIVED  (50.1-14 50.1-9 50.1-11
                      50.2.2 50.1-12 50.1-13)  'SIMPLIFIED NEW CRITERION'
.si .1-1!
   50.2.4  False  (50.1-14 50.2.3 50.1-1)  SIMPLIFICATION
 50.2  LState = CLOSED  (50.2.4)  'Q.E.D.'
 50.3  N" LState = LISTEN  (50.2 50.1-11 50.1-10 50.1-9 50.1-8 50.1-7 50.1-6
                      50.1-5 50.1-4 50.1-3 50.1-2)  'SIMPLIFIED EFFECT'
```

```
.subst .1-12 *.1!
 50.4  LISTEN = SYNSENT | LISTEN=SYNRECEIVED (50.1-12 50.3)SUBSTITUTION(50.3L)
 50.4-1  False  (50.4)  SIMPLIFICATION
50  NOT PRINTED  'Q.E.D.'

THEOREM FROM LEVEL MECHANISM FOR: LRCVRESET
READY FOR INPUT 51.1-13

.1 *-11 *-13!
 51.1-11  N" LState = SYNSENT | N" LState = SYNRECEIVED  (51.1)  'AND SPLIT'
 51.1-12  N" LSeqToSend ~= N" LOldUnack+1  (51.1)  'AND SPLIT'
 51.1-13  LSeqToSend ~= LOldUnack+1 (51.1-8 51.1-10 51.1-12) 'NEW SUBSTITUTION'

.p *-2 B1!
  51.2.1  RtoL = NIL | RtoL.1.Op ~= Reset  (18 1)  ASSUME
  51.2.2  AND NOT PRINTED  (51.2.1 51.1-10 51.1-9 51.1-8 51.1-7 51.1-6 51.1-5
                                51.1-4 51.1-3 51.1-2)  'SIMPLIFIED EFFECT'
  51.2.2-1  N" RtoL = RtoL  (51.2.2)  'AND SPLIT'
  51.2.2-2  N" LState = LState  (51.2.2)  'AND SPLIT'
  51.2.3  LState = SYNSENT | LState = SYNRECEIVED  (51.1-13 51.1-8 51.1-10
              51.2.2-2 51.1-11 51.1-12)  'SIMPLIFIED NEW CRITERION'
.si .1-1!
  51.2.4  False  (51.1-13 51.2.3 51.1-1)  SIMPLIFICATION
 51.2  AND NOT PRINTED  (51.2.4)  'Q.E.D.'
 51.2-1  RtoL ~= NIL  (51.2)  'AND SPLIT'
 51.2-2  RtoL.1.Op = Reset  (51.2)  'AND SPLIT'
 51.3  AND NOT PRINTED  (51.2-2 51.2-1 51.1-10 51.1-9 51.1-8 51.1-7 51.1-6
                       51.1-5 51.1-4 51.1-3 51.1-2)  'SIMPLIFIED EFFECT'
 51.3-1  N" RtoL = RtoL:2  (51.3)  'AND SPLIT'
 51.3-2  (LState = SYNSENT & LAckTest | LState ~= SYNSENT & LState ~= LISTEN &
                                                              LSeqTest
        =>  N" LState = CLOSED <> N" LState = LState)  (51.3)  'AND SPLIT'

.p *-2 B1!
  51.4.1  AND NOT PRINTED  (9 2 36 2 38)  ASSUME
  51.4.1-1  LState ~= SYNSENT | ~LAckTest  (51.4.1)  'AND SPLIT'
  51.4.1-2  LState = SYNSENT | LState = LISTEN | ~LSeqTest (51.4.1)'AND SPLIT'
  51.4.2  N" LState = LState (51.4.1-2 51.4.1-1 51.3-1 51.3)'SIMPLIFIED EFFECT'
  51.4.3  LState = SYNSENT | LState = SYNRECEIVED  (51.1-13 51.1-8 51.1-10
                       51.4.2 51.1-11 51.1-12)  'SIMPLIFIED NEW CRITERION'
.si .1-1!
  51.4.4  False  (51.1-13 51.4.3 51.1-1)  SIMPLIFICATION
 51.4  LState = SYNSENT & LAckTest | LState ~= SYNSENT & LState ~= LISTEN
                                    & LSeqTest  (51.4.4)  'Q.E.D.'
 51.5  N" LState = CLOSED  (51.4 51.3-1 51.3)  'SIMPLIFIED EFFECT'

.su .1-11 *.1!
 51.6  CLOSED = SYNSENT | CLOSED = SYNRECEIVED (51.1-11 51.5)SUBSTITUTION(51.5L)
  51.6-1  False  (51.6)  SIMPLIFICATION
51  NOT PRINTED  'Q.E.D.'

THEOREM FROM LEVEL MECHANISM FOR: LRCVACK
READY FOR INPUT 52.1-11

.1 *-8 *-11!
```

```
 52.1-8  N" LSeqToSend = LSeqToSend  (52.1)  'AND SPLIT'
 52.1-9  N" LState = SYNSENT | N" LState = SYNRECEIVED  (52.1)  'AND SPLIT'
 52.1-10  N" LSeqToSend ~= N" LOldUnack+1  (52.1)  'AND SPLIT'
 52.1-11  LSeqToSend ~= N" LOldUnack+1  (52.1-8 52.1-10)  'NEW SUBSTITUTION'

.p *-2 B1!
  52.2.1  RtoL = NIL | RtoL.1.Op ~= Ack  (18 1)  ASSUME
  52.2.2  AND NOT PRINTED  (52.2.1 52.1-8 52.1-7 52.1-6 52.1-5 52.1-4 52.1-3
                                                    52.1-2)  'SIMPLIFIED EFFECT'
  52.2.2-1  N" RtoL = RtoL  (52.2.2)  'AND SPLIT'
  52.2.2-2  N" LOldUnack = LOldUnack  (52.2.2)  'AND SPLIT'
  52.2.2-3  N" LState = LState  (52.2.2)  'AND SPLIT'
  52.2.2-4  N" LtoR = LtoR  (52.2.2)  'AND SPLIT'
  52.2.3  False  (52.1-1 52.2.2-2 52.1-8 52.2.2-3 52.1-9 52.1-10)  'SIMPLIFIED
                                                                    NEW CRITERION'
 52.2  AND NOT PRINTED  (52.2.3)  'Q.E.D.'
 52.2-1  RtoL ~= NIL  (52.2)  'AND SPLIT'
 52.2-2  RtoL.1.Op = Ack  (52.2)  'AND SPLIT'
 52.3  AND NOT PRINTED  (52.2-2 52.2-1 52.1-8 52.1-7 52.1-6 52.1-5 52.1-4
                                        52.1-3 52.1-2)  'SIMPLIFIED EFFECT'
 52.3-1  N" RtoL = RtoL:2  (52.3)  'AND SPLIT'
 52.3-2  (LState = SYNRECEIVED & LAckTest & LSeqTest
        => N" LOldUnack = LOldUnack+1 & N" LState = ESTABLISHED
        <> N" LOldUnack = LOldUnack & N" LState = LState)  (52.3)  'AND SPLIT'
 52.3-3  N" LtoR
 =  (     LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                          | LState = SYNRECEIVED & LSeqTest & ~LAckTest
        => LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
        <> (LState = SYNRECEIVED & ~LSeqTest
                => LtoR;.MakePkt(LSeqToSend, LSeqToRcv, Ack) <> LtoR))
                                                              (52.3)  'AND SPLIT'
.p *-2 B1!
  52.4.1  LState ~= SYNRECEIVED | ~LAckTest | ~LSeqTest  (9 2 36 38)  ASSUME
  52.4.2  AND NOT PRINTED  (52.3-3 52.4.1 52.3-1 52.3)  'SIMPLIFIED EFFECT'
  52.4.2-1  N" LOldUnack = LOldUnack  (52.4.2)  'AND SPLIT'
  52.4.2-2  N" LState = LState  (52.4.2)  'AND SPLIT'
  52.4.3  False  (52.1-1 52.4.2-1 52.1-8 52.4.2-2 52.1-9 52.1-10)  'SIMPLIFIED
                                                                    NEW CRITERION'
 52.4  AND NOT PRINTED  (52.4.3)  'Q.E.D.'
 52.4-1  LState = SYNRECEIVED  (52.4)  'AND SPLIT'
 52.4-2  LAckTest  (52.4)  'AND SPLIT'
 52.4-3  LSeqTest  (52.4)  'AND SPLIT'
 52.5  AND NOT PRINTED  (52.4-3 52.4-1 52.4-2 52.3-1 52.3)  'SIMPLIFIED EFFECT'
 52.5-1  N" LOldUnack = LOldUnack+1  (52.5)  'AND SPLIT'
 52.5-2  N" LState = ESTABLISHED  (52.5)  'AND SPLIT'
 52.5-3  N" LtoR = (LState = CLOSED | LState = LISTEN
        => LtoR;.MakePkt(RtoL.1.Ack, 0, Reset) <> LtoR)  (52.5)  'AND SPLIT'

.subst .1-9 *-2.1!
 52.6  ESTABLISHED = SYNSENT | ESTABLISHED = SYNRECEIVED  (52.1-9 52.5-2)
                                                          SUBSTITUTION(52.5-2L)
 52.6-1  False  (52.6)  SIMPLIFICATION
52  NOT PRINTED  'Q.E.D.'

THEOREM FROM LEVEL MECHANISM FOR: RCVSYN
```

```
READY FOR INPUT 53.1-8

.1 *-6 *-8!
 53.1-6  N" RState = RState  (53.1)  'AND SPLIT'
 53.1-7  N" LState = SYNSENT | N" LState = SYNRECEIVED  (53.1)  'AND SPLIT'
 53.1-8  N" LSeqToSend ~= N" LOldUnack+1  (53.1)  'AND SPLIT'

.p *-2 B1!
  53.2.1  RtoL = NIL | RtoL.1.Op ~= Syn  (18 1)  ASSUME
  53.2.2  AND NOT PRINTED  (53.2.1 53.1-6 53.1-5 53.1-4 53.1-3 53.1-2)
                                                    'SIMPLIFIED EFFECT'
  53.2.2-1  N" RtoL = RtoL  (53.2.2)  'AND SPLIT'
  53.2.2-2  N" LSeqToSend = LSeqToSend  (53.2.2)  'AND SPLIT'
  53.2.2-3  N" LOldUnack = LOldUnack  (53.2.2)  'AND SPLIT'
  53.2.2-4  N" LSeqToRcv = LSeqToRcv  (53.2.2)  'AND SPLIT'
  53.2.2-5  N" LState = LState  (53.2.2)  'AND SPLIT'
  53.2.2-6  N" LtoR = LtoR  (53.2.2)  'AND SPLIT'
  53.2.3  False  (53.1-1 53.2.2-3 53.2.2-2 53.2.2-5 53.1-7 53.1-8)  'SIMPLIFIED
                                                              NEW CRITERION'
 53.2  AND NOT PRINTED  (53.2.3)  'Q.E.D.'
 53.2-1  RtoL ~= NIL  (53.2)  'AND SPLIT'
 53.2-2  RtoL.1.Op = Syn  (53.2)  'AND SPLIT'
 53.3  AND NOT PRINTED  (53.2-2 53.2-1 53.1-6 53.1-5 53.1-4 53.1-3 53.1-2)
                                                    'SIMPLIFIED EFFECT'
 53.3-1  N" RtoL = RtoL:2  (53.3)  'AND SPLIT'
 53.3-2  N" LSeqToSend = (LState = LISTEN  =>  LSeqToSend+2 <> LSeqToSend)
                                                    (53.3)  'AND SPLIT'
 53.3-3  N" LOldUnack = (LState = LISTEN  =>  LSeqToSend+1 <> LOldUnack)
                                                    (53.3)  'AND SPLIT'
 53.3-4  N" LSeqToRcv = (LState = LISTEN | LState = SYNSENT
                        =>  RtoL.1.SEQ+1 <> LSeqToRcv)  (53.3)  'AND SPLIT'
 53.3-5  N" LState = (LState = LISTEN | LState = SYNSENT
                        =>  SYNRECEIVED <> LState)  (53.3)  'AND SPLIT'
 53.3-6  N" LtoR
 =  ( LState = SYNSENT  =>  LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)
      <> (LState = LISTEN => LtoR;.MakePkt(LSeqToSend+1, RtoL.1.SEQ+1, SynAck)
           <> (LState = CLOSED  =>  LtoR;.MakePkt(0, RtoL.1.SEQ+1, Reset)
                <> LtoR)))  (53.3)  'AND SPLIT'

.t LState~=LISTEN!
  53.4.1  LState = LISTEN  (9 2)  ASSUME
  53.4.2  AND NOT PRINTED  (53.4.1 53.3-1 53.3)  'SIMPLIFIED EFFECT'
  53.4.2-1  N" LSeqToSend = LSeqToSend+2  (53.4.2)  'AND SPLIT'
  53.4.2-2  N" LOldUnack = LSeqToSend+1  (53.4.2)  'AND SPLIT'
  53.4.2-3  N" LSeqToRcv = RtoL.1.SEQ+1  (53.4.2)  'AND SPLIT'
  53.4.2-4  N" LState = SYNRECEIVED  (53.4.2)  'AND SPLIT'
  53.4.2-5  N" LtoR
  =  (LState = SYNSENT  =>  LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)
     <> LtoR;.MakePkt(LSeqToSend+1, RtoL.1.SEQ+1, SynAck)) (53.4.2) 'AND SPLIT'

.subst .1-8 *-1.1,*-2.1!
  53.4.3  LSeqToSend+2 ~= LSeqToSend+1+1  (53.1-8 53.4.2-1 53.4.2-2)
                                        SUBSTITUTION(53.4.2-1L, 53.4.2-2L)
  53.4.3-1  False  (53.4.3)  SIMPLIFICATION
 53.4  LState ~= LISTEN  (53.4.3-1)  'Q.E.D.'
```

```
53.5  AND NOT PRINTED  (53.4 53.3-1 53.3)  'SIMPLIFIED EFFECT'
53.5-1  N" LSeqToSend = LSeqToSend  (53.5)  'AND SPLIT'
53.5-2  N" LOldUnack = LOldUnack  (53.5)  'AND SPLIT'
53.5-3  N" LSeqToRcv = (LState = SYNSENT  =>  RtoL.1.SEQ+1 <> LSeqToRcv)
                                                    (53.5)  'AND SPLIT'
53.5-4  N" LState = (LState = SYNSENT  =>  SYNRECEIVED <> LState)  (53.5)
                                                    'AND SPLIT'
53.5-5  N" LtoR
=  ( LState = SYNSENT  =>  LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)
     <> (LState = CLOSED  =>  LtoR;.MakePkt(0, RtoL.1.SEQ+1, Reset) <> LtoR))
                                                    (53.5)  'AND SPLIT'
53.6  LSeqToSend ~= LOldUnack+1  (53.1-7 53.5-2 53.5-1 53.1-8)  'SIMPLIFIED
                                                    NEW CRITERION'
.t LState~=SYNSENT!
 53.7.1  LState = SYNSENT  (9 2)  ASSUME
 53.7.2  AND NOT PRINTED  (53.7.1 53.5-2 53.5-1 53.5)  'SIMPLIFIED EFFECT'
 53.7.2-1  N" LSeqToRcv = RtoL.1.SEQ+1  (53.7.2)  'AND SPLIT'
 53.7.2-2  N" LState = SYNRECEIVED  (53.7.2)  'AND SPLIT'
 53.7.2-3  N" LtoR = LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)  (53.7.2)
                                                    'AND SPLIT'
.si .1-1!
 53.7.3  False  (53.6 53.7.1 53.1-1)  SIMPLIFICATION
53.7  LState ~= SYNSENT  (53.7.3)  'Q.E.D.'
53.8  AND NOT PRINTED  (53.7 53.5-2 53.5-1 53.5)  'SIMPLIFIED EFFECT'
53.8-1  N" LSeqToRcv = LSeqToRcv  (53.8)  'AND SPLIT'
53.8-2  N" LState = LState  (53.8)  'AND SPLIT'
53.8-3  N" LtoR = (LState = CLOSED  =>  LtoR;.MakePkt(0, RtoL.1.SEQ+1, Reset)
                                    <> LtoR)  (53.8)  'AND SPLIT'
.su .1-7 *-2.1!
53.9  LState = SYNSENT | LState = SYNRECEIVED  (53.1-7 53.8-2)
                                                    SUBSTITUTION(53.8-2L)
53.9-1  LState = SYNRECEIVED  (53.7 53.9)  SIMPLIFICATION

.si .1-1!
53.10  False  (53.6 53.9-1 53.7 53.1-1)  SIMPLIFICATION
53  NOT PRINTED  'Q.E.D.'

THEOREM FROM LEVEL MECHANISM FOR: LRCVSYNACK
READY FOR INPUT 54.1-11

.1 *-9 *-11!
54.1-9  N" LState = SYNSENT | N" LState = SYNRECEIVED  (54.1)  'AND SPLIT'
54.1-10  N" LSeqToSend ~= N" LOldUnack+1  (54.1)  'AND SPLIT'
54.1-11  LSeqToSend ~= N" LOldUnack+1  (54.1-8 54.1-10)  'NEW SUBSTITUTION'

.p *-2 B1!
 54.2.1  RtoL = NIL | RtoL.1.Op ~= SynAck  (18 1)  ASSUME
 54.2.2  AND NOT PRINTED  (54.2.1 54.1-8 54.1-7 54.1-6 54.1-5 54.1-4 54.1-2)
                                                    'SIMPLIFIED EFFECT'
 54.2.2-1  N" RtoL = RtoL  (54.2.2)  'AND SPLIT'
 54.2.2-2  N" LOldUnack = LOldUnack  (54.2.2)  'AND SPLIT'
 54.2.2-3  N" LState = LState  (54.2.2)  'AND SPLIT'
 54.2.2-4  N" LSeqToRcv = LSeqToRcv  (54.2.2)  'AND SPLIT'
 54.2.2-5  N" LtoR = LtoR  (54.2.2)  'AND SPLIT'
 54.2.3  False  (54.1-1 54.2.2-2 54.1-8 54.2.2-3 54.1-9 54.1-10)  'SIMPLIFIED
                                                    NEW CRITERION'
```

```
54.2  AND NOT PRINTED  (54.2.3)  'Q.E.D.'
54.2-1  RtoL ~= NIL  (54.2)  'AND SPLIT'
54.2-2  RtoL.1.Op = SynAck  (54.2)  'AND SPLIT'
54.3  AND NOT PRINTED  (54.2-2 54.2-1 54.1-8 54.1-7 54.1-6 54.1-5 54.1-4
                                                    54.1-2)  'SIMPLIFIED EFFECT'
54.3-1  N" RtoL = RtoL:2  (54.3)  'AND SPLIT'
54.3-2  (LState = SYNSENT & LAckTest =>  N" LOldUnack = LOldUnack+1
                        & N" LState = ESTABLISHED & N" LSeqToRcv = RtoL.1.SEQ+1
    <> N" LOldUnack = LOldUnack & N" LState = LState & N" LSeqToRcv=LSeqToRcv)
                                                              (54.3)  'AND SPLIT'
54.3-3  (LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                        | LState = SYNRECEIVED & LSeqTest & ~LAckTest
    =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
    <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
              | LState = SYNSENT & LAckTest
       =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack)
       <> N" LtoR = LtoR)  (54.3)  'AND SPLIT'

.p *-2 B1!
 54.4.1  LState ~= SYNSENT | ~LAckTest  (9 2 36)  ASSUME
 54.4.2  AND NOT PRINTED  (54.4.1 54.3-1 54.3)  'SIMPLIFIED EFFECT'
 54.4.2-1  N" LOldUnack = LOldUnack  (54.4.2)  'AND SPLIT'

 54.4.2-2  N" LState = LState  (54.4.2)  'AND SPLIT'
 54.4.2-3  N" LSeqToRcv = LSeqToRcv  (54.4.2)  'AND SPLIT'
 54.4.2-4  (LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                        | LState = SYNRECEIVED & LSeqTest & ~LAckTest
     =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
     <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
               =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack)
               <> N" LtoR = LtoR)  (54.4.2)  'AND SPLIT'
 54.4.2-5  (LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                        | LState = SYNRECEIVED & LSeqTest & ~LAckTest
     =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
     <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
               =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, LSeqToRcv, Ack)
               <> N" LtoR = LtoR)  (54.4.2-3 54.4.2-4)  'NEW SUBSTITUTION'
 54.4.3  False  (54.1-1 54.4.2-1 54.1-8 54.4.2-2 54.1-9 54.1-10)  'SIMPLIFIED
                                                              NEW CRITERION'
54.4  AND NOT PRINTED  (54.4.3)  'Q.E.D.'
54.4-1  LState = SYNSENT  (54.4)  'AND SPLIT'
54.4-2  LAckTest  (54.4)  'AND SPLIT'
54.5  AND NOT PRINTED  (54.4-2 54.4-1 54.3-1 54.3)  'SIMPLIFIED EFFECT'
54.5-1  N" LOldUnack = LOldUnack+1  (54.5)  'AND SPLIT'
54.5-2  N" LState = ESTABLISHED  (54.5)  'AND SPLIT'
54.5-3  N" LSeqToRcv = RtoL.1.SEQ+1  (54.5)  'AND SPLIT'
54.5-4  (LState = CLOSED | LState = LISTEN
       =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
       <> N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack))  (54.5)
                                                              'AND SPLIT'
.subst .1-9 *-2.1!
54.6  ESTABLISHED = SYNSENT | ESTABLISHED = SYNRECEIVED  (54.1-9 54.5-2)
                                                    SUBSTITUTION(54.5-2L)
54.6-1  False  (54.6)  SIMPLIFICATION
```

```
54  NOT PRINTED  'Q.E.D.'


Command log for proof of Lemma1
{ event labels in {} added manually }

LEAVE!
        {initial}
SUBST 48.1-11 48.1-1.L!
        {active open}
PROVE 49.1-2 B1!
SUBST 49.1-10 49.3-2.L 49.3-4.L!
        {passive open}
PROVE 50.1-2 B1!
SIMPLIFY 50.1-1!
SUBST 50.1-12 50.3.L!
        {reset}
PROVE 51.1-2 B1!
SIMPLIFY 51.1-1!
PROVE 51.3-2 B1!
SIMPLIFY 51.1-1!
SUBST 51.1-11 51.5.L!
        {ack}
PROVE 52.1-2 B1!
PROVE 52.3-2 B1!
SUBST 52.1-9 52.4.2-2.L!
        {syn}
PROVE 53.1-2 B1!
THEOREM  LState ~= LISTEN!
SUBST 53.1-8 53.4.2-1.L 53.4.2-2.L!
THEOREM  LState ~= SYNSENT!
SIMPLIFY 53.1-1!
SUBST 53.1-7 53.8-2.L!
SIMPLIFY 53.1-1!
        {synack}
PROVE 54.1-2 B1!
PROVE 54.3-2 B1!
SUBST 54.1-9 54.5-2.L!
```

II. AFFIRM PROOF TRANSCRIPT (EDITED) FOR SIMPLE LEMMA

```
62 U: profile autocases=Tell;
     AutoCases: Tell

64 U: profile AutoINvokeIH=Tell;
     AutoInvokeIH: Tell

65 U: profile autonext=Tell;
     AutoNext: Tell

78 U: theorem LSTS,
LState(t)=SYNSENT or LState(t)=SYNRECEIVED imp LSeqToSend(t)=LOldUnack(t)+1;

theorem LSTS,      (LState(t) = SYNSENT) or (LState(t) = SYNRECEIVED)
               imp LSeqToSend(t) = LOldUnack(t) + 1;

80 U: try LSTS;
LSTS is untried.

all t
  (     (LState(t) = SYNSENT) or (LState(t) = SYNRECEIVED)
    imp LSeqToSend(t) = LOldUnack(t) + 1)

81 U: employ Induction(t);
Case Init: Prop(Init) proven.
Case LActiveOpen: all u (IH(u) imp Prop(LActiveOpen(u))) remains to be shown.
Case LPassiveOpen: all u (IH(u) imp Prop(LPassiveOpen(u))) remains to be shown.
Case LRcvReset: all u (IH(u) imp Prop(LRcvReset(u))) remains to be shown.
Case LRcvAck: all u (IH(u) imp Prop(LRcvAck(u))) remains to be shown.
Case LRcvSyn: all u (IH(u) imp Prop(LRcvSyn(u))) remains to be shown.
Case LRcvSynAck: all u (IH(u) imp Prop(LRcvSynAck(u))) remains to be shown.
 (LActiveOpen:)
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

TRUE
(will go to the next proposition to prove)
Going to leaf LPassiveOpen:.
(will raise embedded if-expressions)

By the way, this command has generated the following children which already
existed, and which already have proof attempts.
Thus, you are not currently at a leaf.
           39  used in the proof of theorem LSTS
(will invoke induction hypothesis IH)

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvReset:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)
```

```
TRUE
(will go to the next proposition to prove)
Going to leaf LRcvAck:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvSyn:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

all u
  (          (LState(u) ~= SYNSENT) and (LState(u) ~= SYNRECEIVED)
         and RtoL(u) ~= NewQueueOfPackets
         and Control(Front(RtoL(u))) = Syn
         and LIorSS(u)
    imp (LState(u) = LISTEN) or (LSeqToSend(u) = LOldUnack(u) + 1))

82 U: invoke LIorSS;

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvSynAck:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

TRUE

LSTS proved.

No theorems are untried.
No theorems are tried.
No theorems are awaiting lemma proof.

83 U: print proof;
theorem LSTS.      (LState(t) = SYNSENT) or (LState(t) = SYNRECEIVED)
               imp LSeqToSend(t) = LOldUnack(t) + 1;

proof tree:
81:!  LSTS
          employ Induction(t)
        Init:
          immediate
81:     LActiveOpen:
          33  cases
81:       39  invoke IH | all  |
81:       (proven!)
81:     LPassiveOpen: {LSTS}
          34  cases
81:       39  invoke IH | all  |
81:       (proven!)
81:     LRcvReset: {LSTS}
          35  cases
```

```
81:         41  invoke IH | all  |
81:         (proven!)
81:      LRcvAck: {LSTS}
            36  cases
81:         43  invoke IH | all  |
81:         (proven!)
81:      LRcvSyn: {LSTS}
            37  cases
81:         45  invoke IH | all  |
82:         46  invoke LIorSS
82:         (proven!)
82:      LRcvSynAck: {LSTS}
            38  cases
82:         48  invoke IH | all  |
82:->       (proven!)
```

III. FDM ITP PROOF TRANSCRIPT (EDITED) FOR MAIN THREE-WAY HANDSHAKE PROPERTY

(See Appendix I for a list of commands.)

```
SDC'S INTERACTIVE THEOREM PROVER, RELEASE 11.35I
ENTER LIBRARY MODE

/* reading from a file of commands (lemmas) now -- input not shown */
 50.1  AND NOT PRINTED  (9 2 2 10 12)  ASSUME
 50.1-1  LState = SYNSENT | LState = SYNRECEIVED  (50.1)  'AND SPLIT'
 50.1-2  LSeqToSend ~= LOldUnack+1  (50.1)  'AND SPLIT'

50  LState = SYNSENT | LState = SYNRECEIVED -> LSeqToSend = LOldUnack+1 THEOREM

 51.1  SOME NOT PRINTED  (18 1 38 40 9 2 16 15)  ASSUME
 51.1-1  AND NOT PRINTED  (51.1)  'EXISTENTIAL INSTANTIATION'  (P')
 51.1-2  P' <<: Packet  (51.1-1)  'AND SPLIT'
 51.1-3  P' <: RtoL  (51.1-1)  'AND SPLIT'
 51.1-4  P'.Op = Ack  (51.1-1)  'AND SPLIT'
 51.1-5  LAckTest  (51.1-1)  'AND SPLIT'
 51.1-6  LSeqTest  (51.1-1)  'AND SPLIT'
 51.1-7  LState = SYNRECEIVED  (51.1-1)  'AND SPLIT'
 51.1-8  P'.Ack ~= RSeqToRcv | P'.SEQ ~= RSeqToSend  (51.1-1)  'AND SPLIT'

51  A" P:Packet(P <: RtoL & P.Op = Ack & LAckTest & LSeqTest
       & LState = SYNRECEIVED
    -> P.Ack = RSeqToRcv & P.SEQ = RSeqToSend) THEOREM

 52.1  SOME NOT PRINTED  (18 1 38 9 2 16 15)  ASSUME
 52.1-1  AND NOT PRINTED  (52.1)  'EXISTENTIAL INSTANTIATION'  (P'')
 52.1-2  P'' <<: Packet  (52.1-1)  'AND SPLIT'
 52.1-3  P'' <: RtoL  (52.1-1)  'AND SPLIT'
 52.1-4  P''.Op = SynAck  (52.1-1)  'AND SPLIT'
 52.1-5  LAckTest  (52.1-1)  'AND SPLIT'
 52.1-6  LState = SYNSENT  (52.1-1)  'AND SPLIT'
 52.1-7  P''.Ack ~= RSeqToRcv | P''.SEQ+1 ~= RSeqToSend  (52.1-1)  'AND SPLIT'

52  A" P:Packet(P <: RtoL & P.Op = SynAck & LAckTest & LState = SYNSENT
    -> P.Ack = RSeqToRcv & P.SEQ+1 = RSeqToSend)  THEOREM

 53.1  AND NOT PRINTED  (18)  ASSUME
 53.1-1  RtoL ~= NIL  (53.1)  'AND SPLIT'
 53.1-2  RtoL.1 ~<: RtoL  (53.1)  'AND SPLIT'

53  RtoL ~= NIL -> RtoL.1 <: RtoL  THEOREM

LEAVE LIBRARY MODE
/* back to manual entry of commands */

THEOREM FROM LEVEL MECHANISM FOR: INITIAL CONDITIONS
READY FOR INPUT 54.1-12
.1 *-1 *-12!
```

PRECEDING PAGE BLANK-NOT FILMED

```
 54.1-1  LState = CLOSED  (54.1)  'AND SPLIT'
 54.1-2  RState = CLOSED  (54.1)  'AND SPLIT'
 54.1-3  LSeqToSend = 0  (54.1)  'AND SPLIT'
 54.1-4  LSeqToRcv = 0  (54.1)  'AND SPLIT'
 54.1-5  LOldUnack = 0  (54.1)  'AND SPLIT'
 54.1-6  RSeqToSend = 0  (54.1)  'AND SPLIT'
 54.1-7  RSeqToRcv = 0  (54.1)  'AND SPLIT'
 54.1-8  ROldUnack = 0  (54.1)  'AND SPLIT'
 54.1-9  RtoL = NIL  (54.1)  'AND SPLIT'
 54.1-10  LtoR = NIL  (54.1)  'AND SPLIT'
 54.1-11  LState = ESTABLISHED  (54.1)  'AND SPLIT'
 54.1-12  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= LSeqToRcv  (54.1) 'AND SPLIT'

.subst *-11 *-1.1!
 54.2  CLOSED = ESTABLISHED  (54.1-11 54.1-1)  SUBSTITUTION(54.1-1L)
 54.2-1  False  (54.2)  SIMPLIFICATION

54  NOT PRINTED  'Q.E.D.'
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****

THEOREM FROM LEVEL MECHANISM FOR: LACTIVEOPEN
READY FOR INPUT 55.1-11

.1 *-1 *-11!
 55.1-1  LState = ESTABLISHED -> LSeqToSend = RSeqToRcv & RSeqToSend
                                        =LSeqToRcv  (55.1)  'AND SPLIT'
 55.1-2  (LState = CLOSED
     =>  N" LState = SYNSENT & N" LSeqToSend = LSeqToSend+2 & N" LtoR
                 = LtoR;.MakePkt(0, 0, Syn) & N" LOldUnack = LSeqToSend+1
     <> N" LState = LState & N" LSeqToSend = LSeqToSend & N" LtoR = LtoR
                       & N" LOldUnack = LOldUnack)  (55.1)  'AND SPLIT'
 55.1-3  N" RtoL = RtoL  (55.1)  'AND SPLIT'
 55.1-4  N" ROldUnack = ROldUnack  (55.1)  'AND SPLIT'
 55.1-5  N" RSeqToRcv = RSeqToRcv  (55.1)  'AND SPLIT'
 55.1-6  N" RSeqToSend = RSeqToSend  (55.1)  'AND SPLIT'
 55.1-7  N" RState = RState  (55.1)  'AND SPLIT'
 55.1-8  N" LSeqToRcv = LSeqToRcv  (55.1)  'AND SPLIT'
 55.1-9  N" LState = ESTABLISHED  (55.1)  'AND SPLIT'
 55.1-10  N" LSeqToSend ~= N" RSeqToRcv | N" RSeqToSend ~= N" LSeqToRcv  (55.1)
                                                          'AND SPLIT'
 55.1-11  N" LSeqToSend ~= RSeqToRcv | RSeqToSend ~= LSeqToRcv  (55.1-8 55.1-6
                                        55.1-5 55.1-10)  'NEW SUBSTITUTION'

.p *-2 B1!
  55.2.1  LState ~= CLOSED  (9 2)  ASSUME
  55.2.2  AND NOT PRINTED  (55.2.1 55.1-8 55.1-7 55.1-6 55.1-5 55.1-4 55.1-3
                                                  55.1-2)  'SIMPLIFIED EFFECT'
  55.2.2-1  N" LState = LState  (55.2.2)  'AND SPLIT'
  55.2.2-2  N" LSeqToSend = LSeqToSend  (55.2.2)  'AND SPLIT'
  55.2.2-3  N" LtoR = LtoR  (55.2.2)  'AND SPLIT'
  55.2.2-4  N" LOldUnack = LOldUnack  (55.2.2)  'AND SPLIT'
  55.2.3  False  (55.1-1 55.1-8 55.1-6 55.1-5 55.2.2-2 55.2.2-1 55.1-9 55.1-10)
                                                  'SIMPLIFIED NEW CRITERION'
 55.2  LState = CLOSED  (55.2.3)  'Q.E.D.'
```

```
 55.3  AND NOT PRINTED  (55.2 55.1-8 55.1-7 55.1-6 55.1-5 55.1-4 55.1-3 55.1-2)
                                                        'SIMPLIFIED EFFECT'
 55.3-1  N" LState = SYNSENT  (55.3)  'AND SPLIT'
 55.3-2  N" LSeqToSend = LSeqToSend+2  (55.3)  'AND SPLIT'
 55.3-3  N" LtoR = LtoR;.MakePkt(0, 0, Syn)  (55.3)  'AND SPLIT'
 55.3-4  N" LOldUnack = LSeqToSend+1  (55.3)  'AND SPLIT'

.subst .1-9 *-1.1!
 55.4  SYNSENT = ESTABLISHED  (55.1-9 55.3-1)  SUBSTITUTION(55.3-1L)
 55.4-1  False  (55.4)  SIMPLIFICATION

55  NOT PRINTED  'Q.E.D.'
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****

THEOREM FROM LEVEL MECHANISM FOR: LPASSIVEOPEN
READY FOR INPUT 56.1-14

.p *-2 B1!
  56.2.1  LState ~= CLOSED  (9 2)  ASSUME
  56.2.2  N" LState = LState  (56.2.1 56.1-11 56.1-10 56.1-9 56.1-8 56.1-7
                56.1-6 56.1-5 56.1-4 56.1-3 56.1-2)  'SIMPLIFIED EFFECT'
  56.2.3  LState = ESTABLISHED  (56.1-14 56.1-10 56.1-6 56.1-5 56.1-11 56.2.2
                                 56.1-12 56.1-13)  'SIMPLIFIED NEW CRITERION'

.si .1-1!
  56.2.4  False  (56.1-14 56.2.3 56.1-1)  SIMPLIFICATION
 56.2  LState = CLOSED  (56.2.4)  'Q.E.D.'
 56.3  N" LState = LISTEN  (56.2 56.1-11 56.1-10 56.1-9 56.1-8 56.1-7 56.1-6
                         56.1-5 56.1-4 56.1-3 56.1-2)  'SIMPLIFIED EFFECT'

.subst .1-12 *.1!
 56.4  LISTEN = ESTABLISHED  (56.1-12 56.3)  SUBSTITUTION(56.3L)
 56.4-1  False  (56.4)  SIMPLIFICATION

56  NOT PRINTED  'Q.E.D.'
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****

THEOREM FROM LEVEL MECHANISM FOR: LRCVRESET
READY FOR INPUT 57.1-13

.1 *-1 *-13!
 57.1-1  LState = ESTABLISHED -> LSeqToSend = RSeqToRcv & RSeqToSend
                                        = LSeqToRcv  (57.1)  'AND SPLIT'
 57.1-2  (RtoL ~= NIL & RtoL.1.Op = Reset
     =>   N" RtoL = RtoL:2
       &  (LState=SYNSENT & LAckTest | LState~=SYNSENT & LState~=LISTEN
                                                          & LSeqTest
                =>  N" LState = CLOSED
                <>  N" LState = LState)
    <> N" RtoL = RtoL & N" LState = LState)  (57.1)  'AND SPLIT'
 57.1-3  N" ROldUnack = ROldUnack  (57.1)  'AND SPLIT'
 57.1-4  N" RSeqToRcv = RSeqToRcv  (57.1)  'AND SPLIT'
 57.1-5  N" RSeqToSend = RSeqToSend  (57.1)  'AND SPLIT'
 57.1-6  N" RState = RState  (57.1)  'AND SPLIT'
 57.1-7  N" LtoR = LtoR  (57.1)  'AND SPLIT'
```

```
 57.1-8  N" LOldUnack = LOldUnack  (57.1)  'AND SPLIT'
 57.1-9  N" LSeqToRcv = LSeqToRcv  (57.1)  'AND SPLIT'
 57.1-10  N" LSeqToSend = LSeqToSend  (57.1)  'AND SPLIT'
 57.1-11  N" LState = ESTABLISHED  (57.1)  'AND SPLIT'
 57.1-12  N" LSeqToSend ~= N" RSeqToRcv | N" RSeqToSend ~= N" LSeqToRcv  (57.1)
                                                          'AND SPLIT'
 57.1-13  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= LSeqToRcv  (57.1-9 57.1-5
                              57.1-4 57.1-10 57.1-12)  'NEW SUBSTITUTION'

.p *-2 B1!
  57.2.1  RtoL = NIL | RtoL.1.Op ~= Reset  (18 1)  ASSUME
  57.2.2  AND NOT PRINTED  (57.2.1 57.1-10 57.1-9 57.1-8 57.1-7 57.1-6 57.1-5
                              57.1-4 57.1-3 57.1-2)  'SIMPLIFIED EFFECT'
  57.2.2-1  N" RtoL = RtoL  (57.2.2)  'AND SPLIT'
  57.2.2-2  N" LState = LState  (57.2.2)  'AND SPLIT'
  57.2.3  LState = ESTABLISHED  (57.1-13 57.1-9 57.1-5 57.1-4 57.1-10 57.2.2-2
                              57.1-11 57.1-12)  'SIMPLIFIED NEW CRITERION'

.si .1-1!
  57.2.4  False  (57.1-13 57.2.3 57.1-1)  SIMPLIFICATION
 57.2  AND NOT PRINTED  (57.2.4)  'Q.E.D.'
 57.2-1  RtoL ~= NIL  (57.2)  'AND SPLIT'
 57.2-2  RtoL.1.Op = Reset  (57.2)  'AND SPLIT'
 57.3  AND NOT PRINTED  (57.2-2 57.2-1 57.1-10 57.1-9 57.1-8 57.1-7 57.1-6
                        57.1-5 57.1-4 57.1-3 57.1-2)  'SIMPLIFIED EFFECT'
 57.3-1  N" RtoL = RtoL:2  (57.3)  'AND SPLIT'
 57.3-2  (LState = SYNSENT & LAckTest | LState ~= SYNSENT & LState ~= LISTEN
                                                          & LSeqTest
        =>  N" LState = CLOSED
        <> N" LState = LState)  (57.3)  'AND SPLIT'

.p *-2 B1!
  57.4.1  AND NOT PRINTED  (9 2 38 2 40)  ASSUME
  57.4.1-1  LState ~= SYNSENT | ~LAckTest  (57.4.1)  'AND SPLIT'
  57.4.1-2  LState = SYNSENT | LState = LISTEN | ~LSeqTest (57.4.1) 'AND SPLIT'
  57.4.2  N" LState = LState (57.4.1-2 57.4.1-1 57.3-1 57.3)'SIMPLIFIED EFFECT'
  57.4.3  LState = ESTABLISHED  (57.1-13 57.1-9 57.1-5 57.1-4 57.1-10 57.4.2
                              57.1-11 57.1-12)  'SIMPLIFIED NEW CRITERION'

.si .1-1!
  57.4.4  False  (57.1-13 57.4.3 57.1-1)  SIMPLIFICATION
 57.4  LState = SYNSENT & LAckTest | LState ~= SYNSENT & LState ~= LISTEN
                                          & LSeqTest  (57.4.4)  'Q.E.D.'
 57.5  N" LState = CLOSED  (57.4 57.3-1 57.3)  'SIMPLIFIED EFFECT'

.subst .1-11 *.1!
 57.6  CLOSED = ESTABLISHED  (57.1-11 57.5)  SUBSTITUTION(57.5L)
 57.6-1  False  (57.6)  SIMPLIFICATION

57  NOT PRINTED  'Q.E.D.'
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****

THEOREM FROM LEVEL MECHANISM FOR: LRCVACK
READY FOR INPUT 58.1-11   58.2
```

```
.l .1-1 *!
 58.1-1  LState = ESTABLISHED -> LSeqToSend = RSeqToRcv & RSeqToSend
                                        = LSeqToRcv  (58.1)  'AND SPLIT'
 58.1-2  (RtoL ~= NIL & RtoL.1.Op = Ack
     =>   N" RtoL = RtoL:2
        & (  LState = SYNRECEIVED & LAckTest & LSeqTest
                =>  N" LOldUnack = LOldUnack+1 & N" LState = ESTABLISHED
                <> N" LOldUnack = LOldUnack & N" LState = LState)
        & N" LtoR
          = ( LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                | LState = SYNRECEIVED & LSeqTest & ~LAckTest
                 =>  LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
                 <> (LState = SYNRECEIVED & ~LSeqTest
                        =>  LtoR;.MakePkt(LSeqToSend, LSeqToRcv, Ack) <> LtoR))
     <> N" RtoL = RtoL & N" LOldUnack = LOldUnack & N" LState = LState
                                      & N" LtoR = LtoR)  (58.1)  'AND SPLIT'
 58.1-3  N" ROldUnack = ROldUnack  (58.1)  'AND SPLIT'
 58.1-4  N" RSeqToRcv = RSeqToRcv  (58.1)  'AND SPLIT'
 58.1-5  N" RSeqToSend = RSeqToSend  (58.1)  'AND SPLIT'
 58.1-6  N" RState = RState  (58.1)  'AND SPLIT'
 58.1-7  N" LSeqToRcv = LSeqToRcv  (58.1)  'AND SPLIT'
 58.1-8  N" LSeqToSend = LSeqToSend  (58.1)  'AND SPLIT'
 58.1-9  N" LState = ESTABLISHED  (58.1)  'AND SPLIT'
 58.1-10  N" LSeqToSend ~= N" RSeqToRcv | N" RSeqToSend ~= N" LSeqToRcv  (58.1)
                                                          'AND SPLIT'
 58.1-11  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= LSeqToRcv  (58.1-7 58.1-5
                                58.1-4 58.1-8 58.1-10)  'NEW SUBSTITUTION'

 58.2  (RtoL ~= NIL & RtoL.1.Op = Ack
     =>   N" RtoL = RtoL:2
       & ( LState = SYNRECEIVED & LAckTest & LSeqTest
                =>  N" LOldUnack = LOldUnack+1
                <> N" LOldUnack = LOldUnack & N" LState = LState)
       & N" LtoR
         = ( LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                | LState = SYNRECEIVED & LSeqTest & ~LAckTest
                 =>  LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
                 <> (LState = SYNRECEIVED & ~LSeqTest
                        =>  LtoR;.MakePkt(LSeqToSend, LSeqToRcv, Ack) <> LtoR))
     <> N" RtoL = RtoL & N" LOldUnack = LOldUnack & N" LState = LState
                       & N" LtoR = LtoR)  (58.1-9 58.1-8 58.1-7 58.1-6 58.1-5
                                58.1-4 58.1-3 58.1-2)  'SIMPLIFIED EFFECT'

.p * B1!
  58.3.1  RtoL = NIL | RtoL.1.Op ~= Ack  (18 1)  ASSUME
  58.3.2  AND NOT PRINTED  (58.3.1 58.2)  'SIMPLIFIED EFFECT'
  58.3.2-1  N" RtoL = RtoL  (58.3.2)  'AND SPLIT'
  58.3.2-2  N" LOldUnack = LOldUnack  (58.3.2)  'AND SPLIT'
  58.3.2-3  N" LState = LState  (58.3.2)  'AND SPLIT'
  58.3.2-4  N" LtoR = LtoR  (58.3.2)  'AND SPLIT'
  58.3.3  LState = ESTABLISHED  (58.1-11 58.1-7 58.1-5 58.1-4 58.1-8 58.3.2-3
                                58.1-9 58.1-10)  'SIMPLIFIED NEW CRITERION'

.si .1-1!
  58.3.4  False  (58.1-11 58.3.3 58.1-1)  SIMPLIFICATION
```

```
58.3  AND NOT PRINTED  (58.3.4)  'Q.E.D.'
58.3-1  RtoL ~= NIL  (58.3)  'AND SPLIT'
58.3-2  RtoL.1.Op = Ack  (58.3)  'AND SPLIT'
58.4  AND NOT PRINTED  (58.3-2 58.3-1 58.2)  'SIMPLIFIED EFFECT'
58.4-1  N" RtoL = RtoL:2  (58.4)  'AND SPLIT'
58.4-2  (LState = SYNRECEIVED & LAckTest & LSeqTest
       => N" LOldUnack = LOldUnack+1
       <> N" LOldUnack = LOldUnack & N" LState = LState)  (58.4)  'AND SPLIT'
58.4-3  N" LtoR
=  ( LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                     | LState = SYNRECEIVED & LSeqTest & ~LAckTest
       => LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
       <> (LState = SYNRECEIVED & ~LSeqTest
              => LtoR;.MakePkt(LSeqToSend, LSeqToRcv, Ack) <> LtoR))
                                                    (58.4)  'AND SPLIT'

.p *-2 B1!
 58.5.1  LState ~= SYNRECEIVED | ~LAckTest | ~LSeqTest  (9 2 38 40)  ASSUME
 58.5.2  AND NOT PRINTED  (58.4-3 58.5.1 58.4-1 58.4)  'SIMPLIFIED EFFECT'
 58.5.2-1  N" LOldUnack = LOldUnack  (58.5.2)  'AND SPLIT'
 58.5.2-2  N" LState = LState  (58.5.2)  'AND SPLIT'
 58.5.3  LState = ESTABLISHED  (58.1-11 58.1-7 58.1-5 58.1-4 58.1-8 58.5.2-2
                               58.1-9 58.1-10)  'SIMPLIFIED NEW CRITERION'

.si .1-1!
 58.5.4  False  (58.1-11 58.5.3 58.1-1)  SIMPLIFICATION
58.5  AND NOT PRINTED  (58.5.4)  'Q.E.D.'
58.5-1  LState = SYNRECEIVED  (58.5)  'AND SPLIT'
58.5-2  LAckTest  (58.5)  'AND SPLIT'
58.5-3  LSeqTest  (58.5)  'AND SPLIT'
58.6  AND NOT PRINTED  (58.5-3 58.5-1 58.5-2 58.4-1 58.4)  'SIMPLIFIED EFFECT'
58.6-1  N" LOldUnack = LOldUnack+1  (58.6)  'AND SPLIT'
58.6-2  N" LtoR = (LState = CLOSED | LState = LISTEN
       => LtoR;.MakePkt(RtoL.1.Ack, 0, Reset) <> LtoR)  (58.6)  'AND SPLIT'

.si 53!
58.7  RtoL.1 <: RtoL  (58.3-1 53)  SIMPLIFICATION
58.7-1  RtoL.1 <<: STRUCTURE OF (INTEGER = SEQ, T" #0:INTEGER(#0 >= 0) = Ack,
                               (Syn, SynAck, Ack, Reset) = Op)  (58.7)  TYPE2
.i 51 RtoL.1!
58.8  RtoL.1<:RtoL & RtoL.1.Op=Ack & LAckTest & LSeqTest & LState=SYNRECEIVED
-> RtoL.1.Ack = RSeqToRcv & RtoL.1.SEQ = RSeqToSend  (51 18)  INSTANTIATION
58.8-1  AND NOT PRINTED (58.5-1 58.5-3 58.5-2 58.3-2 58.7 58.8)SIMPLIFICATION
58.8-2  RtoL.1.Ack = RSeqToRcv  (58.8-1)  'AND SPLIT'
58.8-3  RtoL.1.SEQ = RSeqToSend  (58.8-1)  'AND SPLIT'

.su .5-2 LAckTest.1!
58.9  RtoL.1.Ack = LOldUnack+1  (58.5-2 38)  SUBSTITUTION(38L)

.su .5-3 LSeqTest.1!
58.10  RtoL.1.SEQ = LSeqToRcv  (58.5-3 40)  SUBSTITUTION(40L)

.su * .8-3.1!
58.11  RSeqToSend = LSeqToRcv  (58.10 58.8-3)  SUBSTITUTION(58.8-3L)
```

```
.su .9 .8-2.1!
 58.12  RSeqToRcv = LOldUnack+1  (58.9 58.8-2)  SUBSTITUTION(58.8-2L)

.si .1-11!
 58.13  LSeqToSend ~= RSeqToRcv  (58.11 58.1-11)  SIMPLIFICATION

.si 50!
 58.14  LSeqToSend = LOldUnack+1  (58.5-1 50)  SIMPLIFICATION

.su * .12.r!
 58.15  LSeqToSend = RSeqToRcv  (58.14 58.12)  SUBSTITUTION(58.12R)
 58.15-1  False  (58.15 58.13)  CONTRADICTION

58  NOT PRINTED  'Q.E.D.'
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****

THEOREM FROM LEVEL MECHANISM FOR: RCVSYN
READY FOR INPUT 59.1-9

.1 .1-6 .1-8!
 59.1-6  N" RState = RState  (59.1)  'AND SPLIT'
 59.1-7  N" LState = ESTABLISHED  (59.1)  'AND SPLIT'
 59.1-8  N" LSeqToSend ~= N" RSeqToRcv | N" RSeqToSend ~= N" LSeqToRcv  (59.1)
                                                          'AND SPLIT'
.p *-2 B1!
 59.2.1  RtoL = NIL | RtoL.1.Op ~= Syn  (18 1)  ASSUME
  59.2.2  AND NOT PRINTED  (59.2.1 59.1-6 59.1-5 59.1-4 59.1-3 59.1-2)
                                                          'SIMPLIFIED EFFECT'
  59.2.2-1  N" RtoL = RtoL  (59.2.2)  'AND SPLIT'
  59.2.2-2  N" LSeqToSend = LSeqToSend  (59.2.2)  'AND SPLIT'
  59.2.2-3  N" LOldUnack = LOldUnack  (59.2.2)  'AND SPLIT'
  59.2.2-4  N" LSeqToRcv = RSeqToRcv  (59.2.2)  'AND SPLIT'
  59.2.2-5  N" LState = LState  (59.2.2)  'AND SPLIT'
  59.2.2-6  N" LtoR = LtoR  (59.2.2)  'AND SPLIT'
  59.2.3  False  (59.1-1 59.2.2-4 59.1-5 59.1-4 59.2.2-2 59.2.2-5 59.1-7
                                        59.1-8)  'SIMPLIFIED NEW CRITERION'
 59.2  AND NOT PRINTED  (59.2.3)  'Q.E.D.'
 59.2-1  RtoL ~= NIL  (59.2)  'AND SPLIT'
 59.2-2  RtoL.1.Op = Syn  (59.2)  'AND SPLIT'
 59.3  AND NOT PRINTED  (59.2-2 59.2-1 59.1-6 59.1-5 59.1-4 59.1-3 59.1-2)
                                                          'SIMPLIFIED EFFECT'
 59.3-1  N" RtoL = RtoL:2  (59.3)  'AND SPLIT'
 59.3-2  N" LSeqToSend = (LState = LISTEN  =>  LSeqToSend+2 <> LSeqToSend)
                                                          (59.3)  'AND SPLIT'
 59.3-3  N" LOldUnack = (LState = LISTEN  =>  LSeqToSend+1 <> LOldUnack)
                                                          (59.3)  'AND SPLIT'
 59.3-4  N" LSeqToRcv = (LState = LISTEN | LState = SYNSENT
              =>  RtoL.1.SEQ+1 <> LSeqToRcv)  (59.3)  'AND SPLIT'
 59.3-5  N" LState = (LState = LISTEN | LState = SYNSENT
              =>  SYNRECEIVED <> LState)  (59.3)  'AND SPLIT'
 59.3-6  N" LtoR
 =  (  LState = SYNSENT  =>  LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)
       <> (LState=LISTEN => LtoR;.MakePkt(LSeqToSend+1, RtoL.1.SEQ+1, SynAck)
            <> (LState = CLOSED  =>  LtoR;.MakePkt(0, RtoL.1.SEQ+1, Reset)
                                        <> LtoR)))  (59.3)  'AND SPLIT'
```

```
.t LState~=LISTEN!
  59.4.1  LState = LISTEN  (9 2)  ASSUME
  59.4.2  AND NOT PRINTED  (59.4.1 59.3-1 59.3)  'SIMPLIFIED EFFECT'
  59.4.2-1  N" LSeqToSend = LSeqToSend+2  (59.4.2)  'AND SPLIT'
  59.4.2-2  N" LOldUnack = LSeqToSend+1  (59.4.2)  'AND SPLIT'
  59.4.2-3  N" LSeqToRcv = RtoL.1.SEQ+1  (59.4.2)  'AND SPLIT'
  59.4.2-4  N" LState = SYNRECEIVED  (59.4.2)  'AND SPLIT'
  59.4.2-5  N" LtoR
  =  (LState = SYNSENT  =>  LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)
      <> LtoR;.MakePkt(LSeqToSend+1, RtoL.1.SEQ+1, SynAck)) (59.4.2)'AND SPLIT'

.su .1-7 *-4.1!
  59.4.3  SYNRECEIVED = ESTABLISHED  (59.1-7 59.4.2-4)  SUBSTITUTION(59.4.2-4L)
  59.4.3-1  False  (59.4.3)  SIMPLIFICATION
 59.4  LState ~= LISTEN  (59.4.3-1)  'Q.E.D.'
 59.5  AND NOT PRINTED  (59.4 59.3-1 59.3)  'SIMPLIFIED EFFECT'
 59.5-1  N" LSeqToSend = LSeqToSend  (59.5)  'AND SPLIT'
 59.5-2  N" LOldUnack = LOldUnack  (59.5)  'AND SPLIT'
 59.5-3  N" LSeqToRcv = (LState = SYNSENT  =>  RtoL.1.SEQ+1 <> LSeqToRcv)
                                                    (59.5)  'AND SPLIT'
 59.5-4  N" LState = (LState = SYNSENT  =>  SYNRECEIVED <> LState)  (59.5)
                                                          'AND SPLIT'
 59.5-5  N" LtoR
 =  (  LState = SYNSENT  =>  LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)
       <> (LState = CLOSED  =>  LtoR;.MakePkt(0, RtoL.1.SEQ+1, Reset) <> LtoR))
                                                    (59.5)  'AND SPLIT'
 59.6  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= N" LSeqToRcv  (59.1-7 59.1-5
                      59.1-4 59.5-1 59.1-8)  'SIMPLIFIED NEW CRITERION'

.t LState~=SYNSENT!
  59.7.1  LState = SYNSENT  (9 2)  ASSUME
  59.7.2  AND NOT PRINTED  (59.7.1 59.5-2 59.5-1 59.5)  'SIMPLIFIED EFFECT'
  59.7.2-1  N" LSeqToRcv = RtoL.1.SEQ+1  (59.7.2)  'AND SPLIT'
  59.7.2-2  N" LState = SYNRECEIVED  (59.7.2)  'AND SPLIT'
  59.7.2-3  N" LtoR = LtoR;.MakePkt(LSeqToSend, RtoL.1.SEQ+1, Ack)  (59.7.2)
                                                          'AND SPLIT'
.su .1-7 *-2.1!
  59.7.3  SYNRECEIVED = ESTABLISHED  (59.1-7 59.7.2-2)  SUBSTITUTION(59.7.2-2L)
  59.7.3-1  False  (59.7.3)  SIMPLIFICATION
 59.7  LState ~= SYNSENT  (59.7.3-1)  'Q.E.D.'
 59.8  AND NOT PRINTED  (59.7 59.5-2 59.5-1 59.5)  'SIMPLIFIED EFFECT'
 59.8-1  N" LSeqToRcv = LSeqToRcv  (59.8)  'AND SPLIT'
 59.8-2  N" LState = LState  (59.8)  'AND SPLIT'
 59.8-3  N" LtoR = (LState = CLOSED  =>  LtoR;.MakePkt(0, RtoL.1.SEQ+1, Reset)
                                      <> LtoR)  (59.8)  'AND SPLIT'
 59.9  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= LSeqToRcv  (59.8-1 59.6)
                                                 'SIMPLIFIED NEW CRITERION'
.si .1-7!
 59.10  LState = ESTABLISHED  (59.8-2 59.1-7)  SIMPLIFICATION

.si .1-1!
 59.11  False  (59.9 59.10 59.1-1)  SIMPLIFICATION

59  NOT PRINTED  'Q.E.D.'
```

```
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****

THEOREM FROM LEVEL MECHANISM FOR: LRCVSYNACK
READY FOR INPUT 60.1-11   60.2
.] .1-8 *
 60.1-8  N" LSeqToSend = LSeqToSend  (60.1)  'AND SPLIT'
 60.1-9  N" LState = ESTABLISHED  (60.1)  'AND SPLIT'
 60.1-10  N" LSeqToSend ~= N" RSeqToRcv | N" RSeqToSend ~= N" LSeqToRcv
                                                    (60.1)  'AND SPLIT'
 60.1-11  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= N" LSeqToRcv
                      (60.1-6 60.1-5 60.1-8 60.1-10)  'NEW SUBSTITUTION'

 60.2  (RtoL ~= NIL & RtoL.1.Op = SynAck
     => N" RtoL = RtoL:2
        &  ( LState = SYNSENT & LAckTest
                =>  N" LOldUnack = LOldUnack+1 & N" LSeqToRcv = RtoL.1.SEQ+1
                <>  N" LOldUnack = LOldUnack & N" LState = LState
                        & N" LSeqToRcv = LSeqToRcv)
        &  ( LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                        | LState = SYNRECEIVED & LSeqTest & ~LAckTest
              =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
              <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
                        | LState = SYNSENT & LAckTest
                    =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack)
                    <> N"LtoR = LtoR)
     <> N" RtoL = RtoL & N" LOldUnack = LOldUnack & N" LState = LState
                              & N" LSeqToRcv = LSeqToRcv & N" LtoR = LtoR)
        (60.1-9 60.1-8  60.1-7 60.1-6 60.1-5 60.1-4 60.1-2) 'SIMPLIFIED EFFECT'

.p * B1!
  60.3.1  RtoL = NIL | RtoL.1.Op ~= SynAck  (18 1)  ASSUME
  60.3.2  AND NOT PRINTED  (60.3.1 60.2)  'SIMPLIFIED EFFECT'
  60.3.2-1  N" RtoL = RtoL  (60.3.2)  'AND SPLIT'
  60.3.2-2  N" LOldUnack = LOldUnack  (60.3.2)  'AND SPLIT'
  60.3.2-3  N" LState = LState  (60.3.2)  'AND SPLIT'
  60.3.2-4  N" LSeqToRcv = LSeqToRcv  (60.3.2)  'AND SPLIT'
  60.3.2-5  N" LtoR = LtoR  (60.3.2)  'AND SPLIT'
  60.3.3  False  (60.1-1 60.3.2-4 60.1-6 60.1-5 60.1-8 60.3.2-3 60.1-9 60.1-10)
                                                    'SIMPLIFIED NEW CRITERION'
 60.3  AND NOT PRINTED  (60.3.3)  'Q.E.D.'
 60.3-1  RtoL ~= NIL  (60.3)  'AND SPLIT'
 60.3-2  RtoL.1.Op = SynAck  (60.3)  'AND SPLIT'
 60.4  AND NOT PRINTED  (60.3-2 60.3-1 60.2)  'SIMPLIFIED EFFECT'
 60.4-1  N" RtoL = RtoL:2  (60.4)  'AND SPLIT'
 60.4-2  (LState = SYNSENT & LAckTest
        =>  N" LOldUnack = LOldUnack+1 & N" LSeqToRcv = RtoL.1.SEQ+1
        <> N" LOldUnack = LOldUnack & N" LState = LState & N" LSeqToRcv
                                    = LSeqToRcv)  (60.4)  'AND SPLIT'
 60.4-3  (LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
                | LState = SYNRECEIVED & LSeqTest & ~LAckTest
     =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
     <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
                | LState = SYNSENT & LAckTest
         =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack)
         <>  N" LtoR = LtoR)  (60.4)  'AND SPLIT'
```

```
.p *-2 B1!
 60.5.1  LState ~= SYNSENT | ~LAckTest  (9 2 38)  ASSUME
 60.5.2  AND NOT PRINTED  (60.5.1 60.4-1 60.4)  'SIMPLIFIED EFFECT'
 60.5.2-1  N" LOldUnack = LOldUnack  (60.5.2)  'AND SPLIT'
 60.5.2-2  N" LState = LState  (60.5.2)  'AND SPLIT'
 60.5.2-3  N" LSeqToRcv = LSeqToRcv  (60.5.2)  'AND SPLIT'
 60.5.2-4  (LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
             | LState = SYNRECEIVED & LSeqTest & ~LAckTest
     =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
     <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
         =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack)
         <> N" LtoR = LtoR)  (60.5.2)  'AND SPLIT'
 60.5.2-5  (LState = CLOSED | LState = LISTEN | LState = SYNSENT & ~LAckTest
             | LState = SYNRECEIVED & LSeqTest & ~LAckTest
     =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
     <> (LState = SYNRECEIVED | LState = ESTABLISHED) & ~LSeqTest
         =>  N" LtoR = LtoR;.MakePkt(LSeqToSend, LSeqToRcv, Ack)
         <> N" LtoR = LtoR)  (60.5.2-3 60.5.2-4)  'NEW SUBSTITUTION'
 60.5.3  False  (60.1-1 60.5.2-3 60.1-6 60.1-5 60.1-8 60.5.2-2 60.1-9 60.1-10)
                                                  'SIMPLIFIED NEW CRITERION'
60.5  AND NOT PRINTED  (60.5.3)  'Q.E.D.'
60.5-1  LState = SYNSENT  (60.5)  'AND SPLIT'
60.5-2  LAckTest  (60.5)  'AND SPLIT'
60.6  AND NOT PRINTED  (60.5-2 60.5-1 60.4-1 60.4)  'SIMPLIFIED EFFECT'
60.6-1  N" LOldUnack = LOldUnack+1  (60.6)  'AND SPLIT'
60.6-2  N" LSeqToRcv = RtoL.1.SEQ+1  (60.6)  'AND SPLIT'
60.6-3  (LState = CLOSED | LState = LISTEN
       =>  N" LtoR = LtoR;.MakePkt(RtoL.1.Ack, 0, Reset)
       <> N" LtoR = LtoR;.MakePkt(LSeqToSend, N" LSeqToRcv, Ack))  (60.6)
                                                                 'AND SPLIT'
.su .1-11 *-2.1!
60.7  LSeqToSend ~= RSeqToRcv | RSeqToSend ~= RtoL.1.SEQ+1  (60.1-11 60.6-2)
                                                  SUBSTITUTION(60.6-2L)
.si 50!
60.8  LSeqToSend = LOldUnack+1  (60.5-1 50)  SIMPLIFICATION

.si 53!
60.9  RtoL.1 <: RtoL  (60.3-1 53)  SIMPLIFICATION
60.9-1  RtoL.1 <<: STRUCTURE OF (INTEGER = SEQ, T" #0:INTEGER(#0 >= 0) = Ack,
                            (Syn, SynAck, Ack, Reset) = Op)  (60.9)  TYPE2
.i 52 RtoL.1!
60.10  RtoL.1 <: RtoL & RtoL.1.Op = SynAck & LAckTest & LState = SYNSENT
  -> RtoL.1.Ack = RSeqToRcv & RtoL.1.SEQ+1 = RSeqToSend  (52 18) INSTANTIATION
60.10-1  AND NOT PRINTED  (60.5-1 60.5-2 60.3-2 60.9 60.10)  SIMPLIFICATION
60.10-2  RtoL.1.Ack = RSeqToRcv  (60.10-1)  'AND SPLIT'
60.10-3  RtoL.1.SEQ+1 = RSeqToSend  (60.10-1)  'AND SPLIT'

.su .5-2 LAckTest.1!
60.11  RtoL.1.Ack = LOldUnack+1  (60.5-2 38)  SUBSTITUTION(38L)

.su * .10-2.1!
60.12  RSeqToRcv = LOldUnack+1  (60.11 60.10-2)  SUBSTITUTION(60.10-2L)

.su * .8.r!
```

```
 60.13  RSeqToRcv = LSeqToSend  (60.12 60.8)  SUBSTITUTION(60.8R)

.c .7,.13,.10-3!
 60.14  False  (60.7 60.13 60.10-3)  CONTRADICTION

60  NOT PRINTED  'Q.E.D.'
***** DANGER!  LIBRARY THEOREMS NOT PROVED *****
```

IV. AFFIRM PROOF TRANSCRIPT (EDITED) FOR MAIN THREE-WAY HANDSHAKE PROPERTY

```
76 U: read threeway.lemmas;
(Reading AFFIRM commands from <INC-PROJECT>THREEWAY.LEMMAS.4)

theorem Sync1, all t(      LState(t) = ESTABLISHED
                       imp      LSeqToSend(t) = RSeqToRcv(t)
                            and LSeqToRcv(t) = RSeqToSend(t));

78 U: read threeway.lemmas.2;
(Reading AFFIRM commands from <INC-PROJECT>THREEWAY.LEMMAS.2)

theorem GoodAck1, all p(all t(          p in RtoL(t)
                                    and Control(p) = Ack
                                    and Ack(p) = LOldUnack(t) + 1
                                    and Seq(p) = LSeqToRcv(t)
                                    and LState(t) = SYNRECEIVED
                                imp      Seq(p) = RSeqToSend(t)
                                    and Ack(p) = RSeqToRcv(t)));

theorem GoodSynAck1, all p(all t(          p in RtoL(t)
                                       and Control(p) = SynAck
                                       and Ack(p) = LOldUnack(t) + 1
                                       and LState(t) = SYNSENT
                                   imp      Seq(p) + 1 = RSeqToSend(t)
                                       and Ack(p) = RSeqToRcv(t)));

79 U: try Sync1;
Sync1 is untried.

all t
  (     LState(t) = ESTABLISHED
    imp (LSeqToSend(t) = RSeqToRcv(t)) and (LSeqToRcv(t) = RSeqToSend(t)))

80 U: employ Induction(t);
Case Init: Prop(Init) proven.
Case LActiveOpen: all u (IH(u) imp Prop(LActiveOpen(u))) remains to be shown.
Case LPassiveOpen: all u (IH(u) imp Prop(LPassiveOpen(u))) remains to be shown.
Case LRcvReset: all u (IH(u) imp Prop(LRcvReset(u))) remains to be shown.
Case LRcvAck: all u (IH(u) imp Prop(LRcvAck(u))) remains to be shown.
Case LRcvSyn: all u (IH(u) imp Prop(LRcvSyn(u))) remains to be shown.
Case LRcvSynAck: all u (IH(u) imp Prop(LRcvSynAck(u))) remains to be shown.
 (LActiveOpen:)
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

TRUE
(will go to the next proposition to prove)
Going to leaf LPassiveOpen:.
(will raise embedded if-expressions)

By the way, this command has generated the following children which already
```

PRECEDING PAGE BLANK-NOT FILMED

```
existed, and which already have proof attempts.
Thus, you are not currently at a leaf.
            43  used in the proof of theorem Sync1
(will invoke induction hypothesis IH)

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvReset:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvAck:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

all u
  (          (LState(u) ~= ESTABLISHED) and (RtoL(u) ~= NewQueueOfPackets)
         and Control(Front(RtoL(u))) = Ack
         and LState(u) = SYNRECEIVED
         and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
         and Seq(Front(RtoL(u))) = LSeqToRcv(u)
    imp (LSeqToSend(u) = RSeqToRcv(u)) and (LSeqToRcv(u) = RSeqToSend(u)))

81 U: apply GoodAck1;

some p, t
  (          p in RtoL(t) and (Control(p) = Ack)
         and Ack(p) = LOldUnack(t) + 1
         and Seq(p) = LSeqToRcv(t)
         and LState(t) = SYNRECEIVED
    imp (Seq(p) = RSeqToSend(t)) and (Ack(p) = RSeqToRcv(t)))

82 U: put p=Front(RtoL(u)) and t=u;

all u
  (if Front(RtoL(u)) in RtoL(u)
      then           Control(Front(RtoL(u))) = Ack
                 and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
                 and Seq(Front(RtoL(u))) = LSeqToRcv(u)
                 and LState(u) = SYNRECEIVED
                 and Seq(Front(RtoL(u))) = RSeqToSend(u)
                 and Ack(Front(RtoL(u))) = RSeqToRcv(u)
           imp     LState(u) = ESTABLISHED
                 or RtoL(u) = NewQueueOfPackets
                 or      LSeqToSend(u) = RSeqToRcv(u)
                     and LSeqToRcv(u) = RSeqToSend(u)
      else           LState(u) ~= ESTABLISHED
                 and RtoL(u) ~= NewQueueOfPackets
                 and Control(Front(RtoL(u))) = Ack
                 and LState(u) = SYNRECEIVED
                 and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
                 and Seq(Front(RtoL(u))) = LSeqToRcv(u)
           imp      LSeqToSend(u) = RSeqToRcv(u)
                 and LSeqToRcv(u) = RSeqToSend(u))
```

```
88 U: apply Q2, ~Empty(q) imp Front(q) in q;

some q ((q = NewQueueOfPackets) or Front(q) in q)
<<collecting lists..., 33 pages left>>

89 U: put q=RtoL(u);
<<collecting lists..., 33 pages left>>

all u
  (          RtoL(u) ~= NewQueueOfPackets
         and Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = Ack
         and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
         and Seq(Front(RtoL(u))) = LSeqToRcv(u)
         and LState(u) = SYNRECEIVED
         and Seq(Front(RtoL(u))) = RSeqToSend(u)
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
    imp     LState(u) = ESTABLISHED
         or      LSeqToSend(u) = RSeqToRcv(u)
             and LSeqToRcv(u) = RSeqToSend(u))

90 U: replace;

all u
  (          RtoL(u) ~= NewQueueOfPackets
         and Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = Ack
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
         and Seq(Front(RtoL(u))) = RSeqToSend(u)
         and LState(u) = SYNRECEIVED
         and LSeqToRcv(u) = RSeqToSend(u)
         and LOldUnack(u) + 1 = RSeqToRcv(u)
    imp LSeqToSend(u) = RSeqToRcv(u))

91 U: apply LSTS;

some t
  (     (LState(t) = SYNSENT) or (LState(t) = SYNRECEIVED)
    imp LSeqToSend(t) = LOldUnack(t) + 1)

92 U: put t=u;

all u
  (if LState(u) = SYNSENT
      then          LSeqToSend(u) = LOldUnack(u) + 1
                and RtoL(u) ~= NewQueueOfPackets
                and Front(RtoL(u)) in RtoL(u)
                and Control(Front(RtoL(u))) = Ack
                and Ack(Front(RtoL(u))) = RSeqToRcv(u)
                and Seq(Front(RtoL(u))) = RSeqToSend(u)
                and LState(u) = SYNRECEIVED
                and LSeqToRcv(u) = RSeqToSend(u)
                and LOldUnack(u) + 1 = RSeqToRcv(u)
```

```
            imp LSeqToSend(u) = RSeqToRcv(u)
      else           LState(u) = SYNRECEIVED
                 and LSeqToSend(u) = LOldUnack(u) + 1
                 and RtoL(u) ~= NewQueueOfPackets
                 and Front(RtoL(u)) in RtoL(u)
                 and Control(Front(RtoL(u))) = Ack
                 and Ack(Front(RtoL(u))) = RSeqToRcv(u)
                 and Seq(Front(RtoL(u))) = RSeqToSend(u)
                 and LSeqToRcv(u) = RSeqToSend(u)
                 and LOldUnack(u) + 1 = RSeqToRcv(u)
            imp LSeqToSend(u) = RSeqToRcv(u))

93 U: split;
 (first:)
all u
  (          LState(u) = SYNSENT
         and LSeqToSend(u) = LOldUnack(u) + 1
         and RtoL(u) ~= NewQueueOfPackets
         and Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = Ack
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
         and Seq(Front(RtoL(u))) = RSeqToSend(u)
         and LState(u) = SYNRECEIVED
         and LSeqToRcv(u) = RSeqToSend(u)
         and LOldUnack(u) + 1 = RSeqToRcv(u)
    imp LSeqToSend(u) = RSeqToRcv(u))

94 U: replace LState(u);

TRUE
(will go to the next proposition to prove)
Going to leaf second:.

all u
  (          LState(u) ~= SYNSENT
         and LState(u) = SYNRECEIVED
         and LSeqToSend(u) = LOldUnack(u) + 1
         and RtoL(u) ~= NewQueueOfPackets
         and Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = Ack
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
         and Seq(Front(RtoL(u))) = RSeqToSend(u)
         and LSeqToRcv(u) = RSeqToSend(u)
         and LOldUnack(u) + 1 = RSeqToRcv(u)
    imp LSeqToSend(u) = RSeqToRcv(u))

95 U: replace LOldUnack(u)+1;

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvSyn:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

all u
```

```
  (          (LState(u) = ESTABLISHED) and (LSeqToSend(u) = RSeqToRcv(u))
         and LSeqToRcv(u) = RSeqToSend(u)
         and RtoL(u) ~= NewQueueOfPackets
         and Control(Front(RtoL(u))) = Syn
         and ~LIorSS(u)
         and LState(u) = LISTEN
    imp LSeqToSend(u) + 2 = RSeqToRcv(u))

96 U: invoke LIorSS;

TRUE
(will go to the next proposition to prove)
Going to leaf LRcvSynAck:.
(will raise embedded if-expressions)
(will invoke induction hypothesis IH)

all u
  (if LState(u) = ESTABLISHED
      then           (LSeqToSend(u) = RSeqToRcv(u)) and   LSeqToRcv(u)
                                                        = RSeqToSend(u)
                 and RtoL(u) ~= NewQueueOfPackets
                 and Control(Front(RtoL(u))) = SynAck
                 and LState(u) = SYNSENT
                 and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
           imp Seq(Front(RtoL(u))) + 1 = RSeqToSend(u)
      else           RtoL(u) ~= NewQueueOfPackets
                 and Control(Front(RtoL(u))) = SynAck
                 and LState(u) = SYNSENT
                 and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
           imp       LSeqToSend(u) = RSeqToRcv(u)
                 and Seq(Front(RtoL(u))) + 1 = RSeqToSend(u))

104 U: apply Q2;

some q ((q = NewQueueOfPackets) or Front(q) in q)

105 U: put q=RtoL(u);

all u
  (          RtoL(u) ~= NewQueueOfPackets
         and Front(RtoL(u)) in RtoL(u)
    imp if LState(u) = ESTABLISHED
           then           LSeqToSend(u) = RSeqToRcv(u)
                      and LSeqToRcv(u) = RSeqToSend(u)
                      and Control(Front(RtoL(u))) = SynAck
                      and LState(u) = SYNSENT
                      and   Ack(Front(RtoL(u)))
                          = LOldUnack(u) + 1
                 imp Seq(Front(RtoL(u))) + 1 = RSeqToSend(u)
           else           Control(Front(RtoL(u))) = SynAck
                      and LState(u) = SYNSENT
                      and   Ack(Front(RtoL(u)))
                          = LOldUnack(u) + 1
                 imp      LSeqToSend(u) = RSeqToRcv(u)
                      and   Seq(Front(RtoL(u))) + 1
                          = RSeqToSend(u))
```

```
106 U: apply GoodSynAck1;

some p, t
  (          p in RtoL(t) and (Control(p) = SynAck)
         and Ack(p) = LOldUnack(t) + 1
         and LState(t) = SYNSENT
    imp (Seq(p) + 1 = RSeqToSend(t)) and (Ack(p) = RSeqToRcv(t)))

107 U: put t=u and p=Front(RtoL(u));

all u
  (          Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = SynAck
         and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
         and LState(u) = SYNSENT
         and Seq(Front(RtoL(u))) + 1 = RSeqToSend(u)
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
    imp     (RtoL(u) = NewQueueOfPackets) or (LState(u) = ESTABLISHED)
         or LSeqToSend(u) = RSeqToRcv(u))

108 U: replace LState(u);

all u
  (          Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = SynAck
         and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
         and LState(u) = SYNSENT
         and Seq(Front(RtoL(u))) + 1 = RSeqToSend(u)
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
    imp (RtoL(u) = NewQueueOfPackets) or (LSeqToSend(u) = RSeqToRcv(u)))

109 U: apply LSTS;

some t
  (     (LState(t) = SYNSENT) or (LState(t) = SYNRECEIVED)
    imp LSeqToSend(t) = LOldUnack(t) + 1)

110 U: put t=u;

all u
  (          LState(u) = SYNSENT
         and LSeqToSend(u) = LOldUnack(u) + 1
         and Front(RtoL(u)) in RtoL(u)
         and Control(Front(RtoL(u))) = SynAck
         and Ack(Front(RtoL(u))) = LOldUnack(u) + 1
         and Seq(Front(RtoL(u))) + 1 = RSeqToSend(u)
         and Ack(Front(RtoL(u))) = RSeqToRcv(u)
    imp (RtoL(u) = NewQueueOfPackets) or (LSeqToSend(u) = RSeqToRcv(u)))

111 U: replace;
Sync1 is awaiting the proof of lemmas GoodAck1, Q2, and GoodSynAck1.

TRUE
```

```
118 U: print proof;
theorem Sync1, all t(     LState(t) = ESTABLISHED
                      imp      LSeqToSend(t) = RSeqToRcv(t)
                           and LSeqToRcv(t) = RSeqToSend(t));
Sync1 uses GoodAck1?, Q2?, GoodSynAck1?, and LSTS!.

proof tree:
80:|  Sync1
         employ Induction(t)
->     Init:
         immediate
80:    LActiveOpen:
         37  cases
80:      43  invoke IH | all  |
80:      (proven!)
80:    LPassiveOpen: {Sync1}
         38  cases
80:      43  invoke IH | all  |
80:      (proven!)
80:    LRcvReset: {Sync1}
         39  cases
80:      45  invoke IH | all  |
80:      (proven!)
80:    LRcvAck: {Sync1}
         40  cases
80:      47  invoke IH | all  |
81:      48  apply GoodAck1
82:      49  put (p = Front(RtoL(u))) and (t=u)
88:      50  apply Q2
89:      52  put q = RtoL(u)
90:      53  replace
91:      54  apply LSTS
92:      55  put t=u
93:      56  split
94:       first:
             57  replace LState(u)
->           (proven!)
95:       second:
             58  replace LOldUnack(u) + 1
->           (proven!)
95:    LRcvSyn: {Sync1}
         41  cases
95:      59  invoke IH | all  |
96:      60  invoke LIorSS
96:      (proven!)
96:    LRcvSynAck: {Sync1}
         42  cases
96:      62  invoke IH | all  |
104:     63  apply Q2
105:     66  put q = RtoL(u)
106:     67  apply GoodSynAck1
107:     68  put (t=u) and (p = Front(RtoL(u)))
108:     69  replace LState(u)
109:     70  apply LSTS
```

```
110:        71  put t=u
117:        72  replace
->          (proven!)

119 U: print status;
The untried theorems are GoodAck1, GoodSynAck1, and Q2.
No theorems are tried.
No theorems are assumed.
The awaiting lemma proof theorem is Sync1.
The proved theorem is LSTS.
```

V. GYPSY SPECIFICATION OF THE ALTERNATING BIT PROTOCOL

```
scope AB_Protocol =
begin

 const timeout : integer = pending; { but > 0 }
 type bit = (zero, one);
 const initialbit : bit = pending; { either will do }
 type info = pending; { probably sequence of character }
 type msgpacket = record (packseqno : bit; packmsg : info);
 type ackpacket = bit;
 type infobuff = buffer of info;
 type msgpacketbuff = buffer of msgpacket;
 type ackpacketbuff = buffer of ackpacket;
 type clockbuff = buffer of integer;

 procedure Protocol (var info_to_send : infobuff<input>;
                     var info_rcvd : infobuff<output>) =

  begin
   entry timeout > 0;
   block lag(outto(info_rcvd,myid),infrom(info_to_send,myid));
   exit false;  { never stops }

   var msgs_from_sender, msgs_to_rcvr : msgpacketbuff;
   var acks_from_rcvr, acks_to_sender : ackpacketbuff;
   var clock_in, clock_out : clockbuff;

     cobegin
     Msg_Medium (msgs_from_sender, msgs_to_rcvr);
     Ack_Medium (acks_from_rcvr, acks_to_sender);
     Timer (clock_in, clock_out);
     Sender (info_to_send, msgs_from_sender, acks_to_sender,
             clock_in, clock_out);
     Receiver (info_rcvd, msgs_to_rcvr, acks_from_rcvr);
     end
    end { procedure Protocol };

   procedure Timer (var clock_in :  clockbuff<input>;
                    var clock_out : clockbuff<output>) =

   begin
    exit false; { never terminates }
    pending;
   end; { Timer }

   procedure Sender(var info_to_send : infobuff<input>;
                    var msgs_from_sender : msgpacketbuff<output>;
                    var acks_to_sender : ackpacketbuff<input>;
                    var clock_in : clockbuff<output>;
                    var clock_out : clockbuff<input> ) =

   begin
     block lag(firstmsgs(outto(msgs_from_sender,myid)),
                         infrom(info_to_send,myid))  and
```

```
        lag(firstacks(infrom(acks_to_sender,myid)),
                        firstseqnos(outto(msgs_from_sender,myid))) and
        size(infrom(info_to_send,myid)) -
           size(firstacks(infrom(acks_to_sender,myid))) le 1  and
        repeatedmsgs(outto(msgs_from_sender,myid)) ;
    exit false; { never terminates }

    var expect : bit := initialbit;
    var msgx : msgpacket;
    var ack : ackpacket;
    var clock_seqno : integer := pending;
    var clock_return : integer;
      loop
      assert infrom(info_to_send,myid) =
                firstmsgs(outto(msgs_from_sender,myid)) and
             firstseqnos(outto(msgs_from_sender,myid)) =
                firstacks(infrom(acks_to_sender,myid)) and
             size(infrom(info_to_send,myid)) =
                size(firstacks(infrom(acks_to_sender,myid))) and
             expect = expecting(outto(msgs_from_sender,myid)) ;
      receive msgx.packmsg from info_to_send;
      msgx.packseqno := expect;
      send msgx to msgs_from_sender;
      clock_seqno := clock_seqno + 1;
      send clock_seqno to clock_in;
      loop
        await
          on receive ack from acks_to_sender then
            if ack = expect then
              expect := flip(expect);
              leave
            end;
          on receive clock_return from clock_out then
            if clock_return = clock_seqno then
              send msgx to msgs_from_sender;
              send clock_seqno to clock_in;
            end
        end;  { await }
      end;  { await loop }
    end  { main loop }
    end; { Sender }

  procedure Receiver(var info_rcvd : infobuff<output>;
                     var msgs_to_rcvr : msgpacketbuff<input>;
                     var acks_from_rcvr : ackpacketbuff<output> ) =
  begin
    block lag(outto(info_rcvd,myid),firstmsgs(infrom(msgs_to_rcvr,myid))) and
          lag(outto(acks_from_rcvr,myid),allseqnos(infrom(msgs_to_rcvr,myid)))
          and size(outto(info_rcvd,myid)) -
                size(firstacks(outto(acks_from_rcvr,myid))) in [0..1] ;
    exit false; { never terminates }

    var msgx : msgpacket;
    var expect : bit := initialbit;
    loop
```

```
      assert outto(info_rcvd,myid) = firstmsgs(infrom(msgs_to_rcvr,myid)) and
             outto(acks_from_rcvr,myid) = allseqnos(infrom(msgs_to_rcvr,myid))
             and size(outto(info_rcvd,myid)) =
                   size(firstacks(outto(acks_from_rcvr,myid))) and
             expect = expecting(infrom(msgs_to_rcvr,myid));
      receive msgx from msgs_to_rcvr;
      if expect = msgx.packseqno then
        send msgx.packmsg to info_rcvd;
        expect := flip(expect)
      end;
      send msgx.packseqno to acks_from_rcvr;
    end;
  end;  { Receiver }

  procedure Msg_Medium (var pkts_in : msgpacketbuff<input>;
                        var pkts_out : msgpacketbuff<output>) =
   begin
    block outto(pkts_out,myid) sub infrom(pkts_in,myid);
    exit false;  { never stops }
    pending;
   end;

  procedure Ack_Medium (var pkts_in : ackpacketbuff<input>;
                        var pkts_out : ackpacketbuff<output>) =
   begin
    block outto(pkts_out,myid) sub infrom(pkts_in,myid);
    exit false;  { never stops }
    pending;
   end;

  function flip(b : bit) : bit =
   begin
    exit result = if b = zero then one else zero fi;
    result := if b = zero then one else zero fi
   end;

 {***** specification functions *****}
  type msgpackseq = sequence of msgpacket;
  type ackpackseq = sequence of ackpacket;
  type infoseq = sequence of info;
  type anything = pending;
  type anyseq = sequence of anything;

  function lag(s1,s2 : anyseq) : boolean =
   begin
     exit (assume result = [ s1 = s2  or  s1 = nonlast(s2) ]);
   end;

  function allseqnos(mpseq : msgpackseq) : ackpackseq =
   begin
     exit (assume result =
         if mpseq = null(msgpackseq)
           then null(ackpackseq)
           else allseqnos(nonlast(mpseq)) <: last(mpseq).packseqno
         fi );
   end;
```

```
function firstmsgs(mpseq : msgpackseq) : infoseq =
 begin
   exit (assume result =
      if mpseq = null(msgpackseq)
        then null(infoseq)
        else if last(mpseq).packseqno = expecting(nonlast(mpseq))
               then firstmsgs(nonlast(mpseq)) <: last(mpseq).packmsg
               else firstmsgs(nonlast(mpseq))
             fi
      fi );
 end;

function firstseqnos(mpseq : msgpackseq) : ackpackseq =
 begin
   exit (assume result =
      if mpseq = null(msgpackseq)
        then null(ackpackseq)
        else if last(mpseq).packseqno = expecting(nonlast(mpseq))
               then firstseqnos(nonlast(mpseq)) <: last(mpseq).packseqno
               else firstseqnos(nonlast(mpseq))
             fi
      fi );
 end;

function firstacks(apseq : ackpackseq) : ackpackseq =
 begin
  exit (assume result =
      if apseq = null(ackpackseq)
        then null(ackpackseq)
        else if last(apseq) = seqexpecting(nonlast(apseq))
               then firstacks(nonlast(apseq)) <: last(apseq)
               else firstacks(nonlast(apseq))
             fi
      fi );
 end;

function repeatedmsgs(mpseq : msgpackseq) : boolean =
 begin
   exit (assume result =
       if mpseq = null(msgpackseq)  or  nonlast(mpseq) = null(msgpackseq)
         then true
         else (last(mpseq).packseqno ne expecting(mpseq))
            -> last(mpseq) = last(nonlast(mpseq))
       fi );
 end;

function expecting(mpseq : msgpackseq) : bit =
 begin
   exit (assume result = seqexpecting(allseqnos(mpseq)));
 end;

function seqexpecting(apseq : ackpackseq) : bit =
 begin
```

```
    exit (assume result = if apseq = null(ackpackseq)
                          then initialbit
                          else flip(last(apseq))
                         fi );
  end;

end { scope AB_Protocol };


scope lemmas =
begin
 name msgpackseq, ackpackseq, lag, firstacks, firstseqnos, firstmsgs,
        allseqnos, repeatedmsgs  from AB_Protocol;

 lemma mainlemma (mps1, mps2 : msgpackseq) =
    (mps1 sub mps2  and  lag(firstseqnos(mps1),firstseqnos(mps2))  and
       repeatedmsgs(mps2))  ->  lag(firstmsgs(mps1),firstmsgs(mps2));

{ properties about the functions which remove duplicate packets }
 lemma first1 (mps1, mps2 : msgpackseq) =
   mps1 sub mps2  ->  firstseqnos(mps1) sub firstseqnos(mps2);
 lemma first2 (aps1, aps2 : ackpackseq) =
   aps1 sub aps2  ->  firstacks(aps1) sub firstacks(aps2);
 lemma first3 (mps1, mps2 : msgpackseq) =
   mps1 sub mps2  ->  firstmsgs(mps1) sub firstmsgs(mps2);
 lemma first4 (mps : msgpackseq) = firstacks(allseqnos(mps))=firstseqnos(mps);

{ properties about the lag function }
 type anything = pending;
 type anyseq = sequence of anything;
 type anybuff = buffer of anything;

 lemma lag1 (s1, s2 : anyseq) =  lag(s1,s2)  ->  s1 sub s2;
 lemma lag2 (a, b, c : anyseq) =
   (a sub b  and  b sub c  and  lag(a,c))  ->  lag(b,c);
 lemma lag3 (a,b,c,d : anyseq) =
   (lag(a,b)  and  lag(b,c)  and  lag(c,d)  and  size(d) - size(a) le 1)
      ->  lag(a,d);

{ properties about sequences }
 lemma seq1(s : anyseq; b : anybuff) =  allto(b) sub s  ->  allfrom(b) sub s ;
 lemma seq2(s : anyseq; b : anybuff) =  s sub allfrom(b)  ->  s sub allto(b);
 lemma seq3(s1, s2 : anyseq) =  s1 sub s2  ->  size(s1) le size(s2);
 lemma seq4(s1, s2, s3 : anyseq) =  (s1 sub s2  and  s2 sub s3) -> s1 sub s3;
 lemma seq5 ( a,b,c : anyseq ) =  a  c  sub  b  c   iff   a sub b;
 lemma seqcases (s : anyseq) =
   s = null(anyseq)  or  some s1 : anyseq, some x : anything, s = s1 <: x;
 lemma nullsize (s : anyseq) =  s = null(anyseq)  iff  size(s) = 0;
 lemma sizelemma (s : anyseq) =  size(s) ge 0;

{ a property about integers }
 lemma squeeze (a,b,c,n : integer) =
    (a le b  and  b le c  and  c - a le n) ->  (c - b le n);

{ Gypsy doesn't reason well about enumerated types }
```

```
   name bit from AB_Protocol;
   lemma bitlemma (b : bit) =  b = zero  or  b = one;

end; { scope lemmas }
```

VI. GYPSY SPECIFICATION OF THE THREE-WAY HANDSHAKE

```
scope threeway =
begin
 type packetop = (reset, syn, ack, synack);
 type command = (activeopen, passiveopen);
 type nodestate = (closed, listening, synsent, synrcvd, estab);
 type packet = record (
                op : packetop;
                seqno : integer;
                ackno : integer;
                Vislast : boolean );  { tells the Medium to exit }
 type packetseq = sequence of packet;
 type packetbuff = buffer of packet;
 const dontcare : integer = pending;
 const Linitseqno : integer = pending;
 const Rinitseqno : integer = pending;

procedure Protocol(Lcmd, Rcmd : command;
                   var VLseqno, VRseqno, VLackno, VRackno : integer) =
 begin
  exit VLseqno = VRackno  and  VRseqno = VLackno;
  var Lout, Rin, Rout, Lin : packetbuff;
  cobegin
    Medium(Lout,Rin);
    Medium(Rout,Lin);
    Node(Lcmd,Lout,Lin,Linitseqno,VLseqno,VLackno);
    Node(Rcmd,Rout,Rin,Rinitseqno,VRseqno,VRackno);
  end
 end; { Protocol }

procedure Node(const cmd : command;
               var outbuff : packetbuff<output>;
               var inbuff : packetbuff<input>;
               const initseqno : integer;
               var Vseqno : integer;
               var Vackno : integer    ) =
 begin
  exit
    estab = mystateof(nonlast(outto(outbuff,myid)),infrom(inbuff,myid))  &
    Vseqno = seqnotosendof(nonlast(outto(outbuff,myid)),infrom(inbuff,myid)) &
    Vackno = seqnotorcvof(nonlast(outto(outbuff,myid)),infrom(inbuff,myid)) &
    seqnotosendprop(nonlast(outto(outbuff,myid)),infrom(inbuff,myid)) &
    seqnotorcvprop(nonlast(outto(outbuff,myid)),infrom(inbuff,myid));
  var mystate : nodestate := closed;
  var seqnotosend : integer := dontcare;
  var oldestunack : integer := dontcare;
  var seqnotorcv : integer := dontcare;
  var pin : packet;

  docmd(outbuff,cmd,mystate,initseqno,seqnotosend,oldestunack);
  loop
```

```
    assert
     properstate(mystate,seqnotosend,oldestunack,seqnotorcv,
                 outto(outbuff,myid),infrom(inbuff,myid));
    if mystate = estab then
      leave
    end;
    receive pin from inbuff;
    case mystate
     is closed: doclosed(outbuff,pin,mystate);
     is listening: dolistening(outbuff,pin,mystate,initseqno,seqnotosend,
                               oldestunack,seqnotorcv);
     is synsent:
       dosynsent(outbuff,pin,mystate,seqnotosend,oldestunack,seqnotorcv);
     is synrcvd:
       dosynrcvd(outbuff,pin,mystate,seqnotosend,oldestunack,seqnotorcv);
     is estab: ; { can't happen with this setup }
    end; { case mystate }
   end; { loop }
   Vseqno := seqnotosend;
   Vackno := seqnotorcv;
   send packlast(ack,seqnotosend,seqnotorcv) to outbuff;
  end; { Node }

procedure docmd(var outbuff : packetbuff<output>; cmd : command;
                var mystate : nodestate; initseqno : integer;
                var seqnotosend : integer;
                var oldestunack : integer ) =
  begin
   entry mystate = closed;
   exit if cmd = activeopen
          then outto(outbuff,myid) = [seq:pack(syn,initseqno,dontcare)]  and
               oldestunack = initseqno  and  seqnotosend = initseqno+1  and
               mystate = synsent
          else outto(outbuff,myid) = null(packetseq) and mystate = listening
               oldestunack = oldestunack'  and  seqnotosend = seqnotosend'
        fi;
   case cmd
    is activeopen:
      send pack(syn,initseqno,dontcare) to outbuff;
      oldestunack := initseqno;
      seqnotosend := initseqno + 1;
      mystate := synsent;
    is passiveopen: mystate := listening;
   end; { case cmd }
  end; { docmd }

procedure doclosed(var outbuff : packetbuff<output>; pin : packet;
                   mystate : nodestate ) =
  begin
   entry mystate = closed;
   exit if pin.op = syn
          then outto(outbuff,myid) = [seq:pack(reset,dontcare,pin.seqno+1)]
          else if pin.op = reset
                then outto(outbuff,myid) = null(packetseq)
                else outto(outbuff,myid) = [seq:pack(reset,pin.ackno,dontcare)]
               fi
```

```
            fi
       fi;
  case pin.op
    is reset: ; { do nothing }
    is syn: send pack(reset,dontcare,pin.seqno+1) to outbuff;
    is ack: send pack(reset,pin.ackno,dontcare) to outbuff;
    is synack: send pack(reset,pin.ackno,dontcare) to outbuff;
   end; { case pin.op }
 end; { doclosed }

procedure dolistening(var outbuff : packetbuff<output>; pin : packet;
                      var mystate : nodestate; const initseqno : integer;
                      var seqnotosend : integer; var oldestunack : integer;
                      var seqnotorcv : integer) =
 begin
  entry mystate = listening;
  exit if pin.op = syn
         then outto(outbuff,myid) = [seq:pack(synack,initseqno,pin.seqno+1)] &
              seqnotorcv = pin.seqno+1  and  oldestunack = initseqno  and
              seqnotosend = initseqno+1  and  mystate = synrcvd
         else mystate = listening  and  seqnotosend = seqnotosend'  and
              oldestunack = oldestunack'  and  seqnotorcv = seqnotorcv'  and
              if pin.op = reset
               then outto(outbuff,myid) = null(packetseq)
               else outto(outbuff,myid) = [seq:pack(reset,pin.ackno,dontcare)]
              fi
       fi;
  case pin.op
   is reset: ; { do nothing }
   is syn:
     seqnotorcv := pin.seqno + 1;
     send pack(synack,initseqno,seqnotorcv) to outbuff;
     oldestunack := initseqno;
     seqnotosend := initseqno + 1;
     mystate := synrcvd;
   is ack: send pack(reset,pin.ackno,dontcare) to outbuff;
   is synack: send pack(reset,pin.ackno,dontcare) to outbuff;
  end; { case pin.op }
 end; { dolistening }

procedure dosynsent(var outbuff : packetbuff<output>; pin : packet;
                    var mystate : nodestate;
                    seqnotosend : integer; var oldestunack : integer;
                    var seqnotorcv : integer) =
 begin
  entry mystate = synsent;
  exit if (pin.op = ack or pin.op = synack)  and  pin.ackno = oldestunack' + 1
         then oldestunack = oldestunack' + 1
         else oldestunack = oldestunack'
       fi  and
       if pin.op = syn or (pin.op = synack  and  pin.ackno = oldestunack' + 1)
         then seqnotorcv = pin.seqno + 1  and
              outto(outbuff,myid) = [seq:pack(ack,seqnotosend,pin.seqno+1)]
         else seqnotorcv = seqnotorcv'  and
              if pin.op ne reset  and  pin.ackno ne oldestunack' + 1
```

```
                then outto(outbuff,myid) = [seq:pack(reset,pin.ackno,dontcare)]
                else outto(outbuff,myid) = null(packetseq)
               fi
        fi and
        if pin.op = syn
          then mystate = synrcvd
          else if pin.ackno = oldestunack' + 1
                 then if pin.op = reset
                        then mystate = closed
                        else if pin.op = synack
                               then mystate = estab
                               else mystate = synsent
                             fi
                      fi
                 else mystate = synsent
               fi
        fi;
    case pin.op
     is reset:
       if pin.ackno = oldestunack + 1 then
         mystate := closed
       end;
     is syn:
       seqnotorcv := pin.seqno + 1;
       send pack(ack,seqnotosend,seqnotorcv) to outbuff;
       mystate := synrcvd
     is ack:
       if pin.ackno = oldestunack + 1 then
         oldestunack := oldestunack + 1
        else
         send pack(reset,pin.ackno,dontcare) to outbuff
       end;
     is synack:
       if pin.ackno = oldestunack + 1 then
         seqnotorcv := pin.seqno + 1;
         send pack(ack,seqnotosend,seqnotorcv) to outbuff;
         oldestunack := oldestunack + 1;
         mystate := estab;
        else
         send pack(reset,pin.ackno,dontcare) to outbuff;
       end;
    end; { case pin.op }
   end; { dosynsent }

procedure dosynrcvd(var outbuff : packetbuff<output>; pin : packet;
                    var mystate : nodestate;
                    seqnotosend : integer; var oldestunack : integer;
                    seqnotorcv : integer) =
   begin
    entry mystate = synrcvd;
    exit if (pin.op = ack or pin.op = synack)  and  pin.seqno = seqnotorcv
              and pin.ackno ne oldestunack'+1
           then outto(outbuff,myid) = [seq:pack(reset,pin.ackno,dontcare)]
           else if if pin.seqno = seqnotorcv
                     then pin.op = synack  and  pin.ackno = oldestunack' + 1
```

```
                    else pin.op ne reset
                  fi
                 then outto(outbuff,myid)=[seq:pack(ack,seqnotosend,seqnotorcv)]
                 else outto(outbuff,myid)=null(packetseq)
                fi
        fi  and
        if (pin.op = ack or pin.op = synack)  and  pin.seqno = seqnotorcv
              and pin.ackno = oldestunack' + 1
          then oldestunack = oldestunack' + 1
          else oldestunack = oldestunack'
        fi  and
        if pin.seqno = seqnotorcv
          then if pin.op = ack  and  pin.ackno = oldestunack' + 1
                 then mystate = estab
                 else if pin.op = reset
                        then mystate = closed
                        else mystate = synrcvd
                      fi
               fi
          else mystate = synrcvd
        fi;
  case pin.op
   is reset:
     if pin.seqno = seqnotorcv then
       mystate := closed
     end;
   is syn:
     if pin.seqno ne seqnotorcv then
       send pack(ack,seqnotosend,seqnotorcv) to outbuff
     end;
   is ack:
     if pin.seqno = seqnotorcv then
       if pin.ackno = oldestunack + 1 then
         oldestunack := oldestunack + 1;
         mystate := estab;
        else
         send pack(reset,pin.ackno,dontcare) to outbuff
       end;
      else
       send pack(ack,seqnotosend,seqnotorcv) to outbuff
     end;
   is synack:
     if pin.seqno = seqnotorcv then
       if pin.ackno = oldestunack + 1 then
         oldestunack := oldestunack + 1;
         send pack(ack,seqnotosend,seqnotorcv) to outbuff;
        else
         send pack(reset,pin.ackno,dontcare) to outbuff
       end;
      else
       send pack(a   seqnotosend,seqnotorcv) to outbuff
     end;
  end; { case pin.op }
 end; { dosynrcvd }
```

```
procedure Medium(var packets_in : packetbuff<input>;
                 var packets_out : packetbuff<output> ) =
 begin
  exit
    all p : packet, p in outto(packets_out,myid)
        ->  p in nonlast(infrom(packets_in,myid));
  pending;
 end; { Medium }

function pack(pop : packetop; pseqno : integer; packno : integer) : packet =
 begin
  exit (assume  result.op = pop  and  result.seqno = pseqno  and
          result.ackno = packno  and  not result.Vislast);
  result.op := pop;
  result.seqno := pseqno;
  result.ackno := packno;
  result.Vislast := false;
 end;

function packlast(pop: packetop; pseqno: integer; packno: integer): packet =
 begin
  exit (assume  result.op = pop  and  result.seqno = pseqno  and
           result.ackno = packno  and  result.Vislast);
  result.op := pop;
  result.seqno := pseqno;
  result.ackno := packno;
  result.Vislast := true;
 end;

{******** specification functions ********}
function properstate(mystate : nodestate; seqnotosend : integer;
                     oldestunack : integer; seqnotorcv : integer;
                     oseq : packetseq; iseq : packetseq) : boolean =
 begin
  exit ( assume result = [
             mystate = mystateof(oseq,iseq)  and
             seqnotosend = seqnotosendof(oseq,iseq)  and
             oldestunack = oldestunackof(oseq,iseq)  and
             seqnotorcv = seqnotorcvof(oseq,iseq)  and
             seqnotosendprop(oseq,iseq)  and
             seqnotorcvprop(oseq,iseq) ] );
 end;

function seqnotosendprop(oseq, iseq : packetseq) : boolean =
 begin
  exit ( assume result = [
         (mystateof(oseq,iseq) = synsent  or
          mystateof(oseq,iseq) = synrcvd  or
          mystateof(oseq,iseq) = estab )
         -> ((all p : packet, p in oseq  and  (p.op = syn  or  p.op = synack)
              -> seqnotosendof(oseq,iseq) = p.seqno + 1)) ] );
 end;

function seqnoto rcvprop(oseq,iseq : packetseq) : boolean =
 begin
```

```
  exit ( assume result = [
    (mystateof(oseq,iseq) = estab  or  mystateof(oseq,iseq) = synrcvd)
    -> some p : packet, (p.op = syn  or  p.op = synack)  and  p in iseq
         and seqnotorcvof(oseq,iseq) = p.seqno + 1 ] );
 end;

function mystateof(oseq, iseq : packetseq) : nodestate =
 begin
  pending; {restates state transitions--see page 24}
 end;
function seqnotosendof(oseq, iseq : packetseq) : integer =
 begin
  pending; {see page 24}
 end;
function oldestunackof(oseq, iseq : packetseq) : integer =
 begin
  pending; {see page 24}
 end;
function seqnotorcvof(oseq, iseq : packetseq) : integer =
 begin
  pending; {see page 24}
 end;

{******** lemmas ********}
lemma mainlemma(oseq1, iseq1, oseq2, iseq2 : packetseq) =
   [ estab = mystateof(oseq1,iseq1)  and
     estab = mystateof(oseq2,iseq2)  and
     seqnotosendprop(oseq1,iseq1)  and
     seqnotorcvprop(oseq2,iseq2)  and
     [ all p : packet,  p in iseq2  ->  p in oseq1 ] ]
   ->  seqnotosendof(oseq1,iseq1) = seqnotorcvof(oseq2,iseq2);

{ The following lemma is used to reduce the VC for the
  procedure Protocol to two instances of mainlemma.  It
  is formulated so that it can easily be used in the proof,
  and says that if everything sent to buffer b1 was received
  from buffer b2 (other than the last one received), then
  everything received from b1 was sent to b2 (other than the
  last one sent). }
lemma fromto(b1, b2 : packetbuff; p : packet) =
   [ p in allto(b1) -> p in nonlast(allfrom(b2)) ]
   -> [ p in allfrom(b1) -> p in nonlast(allto(b2)) ];

{ The following basic facts about sequences and enumerated
  types are neither built into nor provable in Gypsy. }
lemma seqprop(s : packetseq) =
   s = null(packetseq)  or  some s1: packetseq, some p: packet, s = s1 <: p;
lemma packetop_cases(x : packetop) =
   x = reset  or  x = syn  or  x = ack  or  x = synack;

end; { scope threeway }
```

VII. PROOF OF MAINLEMMA IN GYPSY

Notes: User commands appear after the prompt "->" and are put in upper case here. (Note that other arrows in this transcript represent implication, not prompts.) This proof uses no lemmas, although the properties seqnotosendprop and seqnotorcvprop act as lemmas in that they are strong enough to prove the theorem but must themselves be proved true. Since the proof steps consist of substitutions and simplifications with no case splitting or other branching of the proof tree, the final status line "(. E . E . ... QED .)" represents the whole proof tree. Normally, the current path from the root of the proof tree is only a small part of the whole tree. The QED command at the end of the proof tells the prover itself to try to prove the current theorem. The prover was actually able to do this, but only because we had manually brought to a point where a couple of substitutions would finish the proof.

```
1 Vsys -> PROVE
       Unit or vc names -> MAINLEMMA
  Entering Prover with lemma MAINLEMMA
all OSEQ1, ISEQ1, OSEQ2, ISEQ2 : PACKETSEQ,
        (all P#1 : PACKET, P#1 in ISEQ2 -> P#1 in OSEQ1)
      & MYSTATEOF (OSEQ1, ISEQ1) = ESTAB & MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
      & SEQNOTORCVPROP (OSEQ2, ISEQ2) & SEQNOTOSENDPROP (OSEQ1, ISEQ1)
   -> SEQNOTOSENDOF (OSEQ1, ISEQ1) = SEQNOTORCVOF (OSEQ2, ISEQ2)
  H1: MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
  H2: MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
  H3: SEQNOTORCVPROP (OSEQ2, ISEQ2)
  H4: SEQNOTOSENDPROP (OSEQ1, ISEQ1)
  H5: P#12$ in ISEQ2 -> P#12$ in OSEQ1
->
  C1: SEQNOTOSENDOF (OSEQ1, ISEQ1) = SEQNOTORCVOF (OSEQ2, ISEQ2)
Backup point
(.)

Prvr -> EXPAND
       Unit name -> SEQNOTORCVPROP
  Backup point
(. E .)

Prvr -> EXPAND
       Unit name -> SEQNOTOSENDPROP
  Backup point
(. E . E .)

Prvr -> THEOREM
  H1: MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
  H2: MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
  H3:       MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
          or MYSTATEOF (OSEQ2, ISEQ2) = SYNRCVD
       ->   SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1 & P#13 in ISEQ2
          & (P#13.OP = SYN or P#13.OP = SYNACK)
```

PRECEDING PAGE BLANK-NOT FILMED

```
  H4:       MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
          or MYSTATEOF (OSEQ1, ISEQ1) = SYNRCVD
          or MYSTATEOF (OSEQ1, ISEQ1) = SYNSENT
       -> (   P#15$ in OSEQ1 & (P#15$.OP = SYN or P#15$.OP = SYNACK)
           -> SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#15$.SEQNO + 1)
  H5: P#12$ in ISEQ2 -> P#12$ in OSEQ1
->
  C1: SEQNOTOSENDOF (OSEQ1, ISEQ1) = SEQNOTORCVOF (OSEQ2, ISEQ2)

Prvr -> INTERACTIVE SIMPLIFY HYPOTHESIS
What hypotheses (by label) would you like to simplify?
        hypothesis labels -> 3 4
  What hypotheses would you like to assume?
        hypothesis labels -> 1 2
Backup point
(. E . E . SIMP .)

Prvr -> THEOREM
  H1: P#12$ in ISEQ2 -> P#12$ in OSEQ1
  H2: SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1
  H3:    P#15$ in OSEQ1 & (P#15$.OP = SYN or P#15$.OP = SYNACK)
      -> SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#15$.SEQNO + 1
  H4: P#13 in ISEQ2
  H5: P#13.OP = SYN or P#13.OP = SYNACK
  H6: MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
  H7: MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
->
  C1: SEQNOTOSENDOF (OSEQ1, ISEQ1) = SEQNOTORCVOF (OSEQ2, ISEQ2)

Prvr -> PUT
    For what? **P#12$;
Put what? *P#13;
  For what? *P#15$;
Put what? *P#13;
  For what? *$DONE
  Typelist equalities
    SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1
Backup point
(. E . E . SIMP . PUT .)

Prvr -> THEOREM
  H1: MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
  H2: MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
  H3: P#13.OP = SYN or P#13.OP = SYNACK
  H4: P#13 in ISEQ2
  H5:    P#13 in OSEQ1 & (P#13.OP = SYN or P#13.OP = SYNACK)
      -> SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
  H6: SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1
  H7: P#13 in ISEQ2 -> P#13 in OSEQ1
->
  C1: SEQNOTOSENDOF (OSEQ1, ISEQ1) = SEQNOTORCVOF (OSEQ2, ISEQ2)

Prvr -> FORWARDCHAIN H7
  Forward chaining gives
    P#13 in OSEQ1
```

```
Backup point
(. E . E . SIMP . PUT . FC .)

Prvr -> HYPOTHESES
  H1: P#13 in OSEQ1
  H2: MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
  H3: MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
  H4: P#13.OP = SYN or P#13.OP = SYNACK
  H5: P#13 in ISEQ2
  H6:    P#13 in OSEQ1 & (P#13.OP = SYN or P#13.OP = SYNACK)
      -> SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
  H7: SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1
  H8: P#13 in ISEQ2 -> P#13 in OSEQ1

Prvr -> FORWARDCHAIN H6
  Forward chaining gives
    SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
Typelist equalities
    SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
Backup point
(. E . E . SIMP . PUT . FC . FC .)

Prvr -> HYPOTHESES
  H1: SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
  H2: P#13 in OSEQ1
  H3: MYSTATEOF (OSEQ2, ISEQ2) = ESTAB
  H4: MYSTATEOF (OSEQ1, ISEQ1) = ESTAB
  H5: P#13.OP = SYN or P#13.OP = SYNACK
  H6: P#13 in ISEQ2
  H7:    P#13 in OSEQ1 & (P#13.OP = SYN or P#13.OP = SYNACK)
      -> SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
  H8: SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1
  H9: P#13 in ISEQ2 -> P#13 in OSEQ1

Prvr -> RETAIN
         hypothesis labels -> 1 8
  Typelist equalities
      SEQNOTORCVOF (OSEQ2, ISEQ2) = P#13.SEQNO + 1
  & SEQNOTOSENDOF (OSEQ1, ISEQ1) = P#13.SEQNO + 1
Backup point
(. E . E . SIMP . PUT . FC . FC . D .)

Prvr -> QED
  (. E . E . SIMP . PUT . FC . FC . D . QED)
::Equality proved TRUE
QED
MAINLEMMA proved in theorem prover.
```

VIII. PROOF OF SIMPLE ALTERNATING BIT PROTOCOL IN CSD

Notes: Each item (CSDa, CSDb) represents the currently active CSDs in each processor. The processor with the different CSD from the last item has been active and has completed a CSD. When a CSD has two (or more) exits (e.g., SB and RB), then a "disjunction" branch is created for each. When either processor could finish a CSD next, then an "interleaving" branch is created. The first number in the left column (before the slash) is an increasing branch point ID, while the second number refers to the earlier branch point for which this is a later alternative.

The proof tree at the end summarizes the symbolic execution paths discovered by listing just the CSD pairs sequentially on a line, with branch points matched vertically under one another. Numbered lines are continuations of the top lines.

```
_assume (TIMEOUT>MAXDELAY MAXDELAY ge MINDELAY MINDELAY>0 MAXLOSS=1)
_prove SPEC
          (SA RA) (SB RA)
     2-way disjunction branch            SenderPC114=Bt
                                         UNDEFINED113~=TRUE and SenderPC114=Bf
 d1/1     Start disjunction branch #1    SenderPC114=Bt
          (SBt RA) (SC RA) (SC RAt) (SC RB)
     2-way disjunction branch            ReceiverPC128=SenderPC114
                                         UNDEFINED127~=TRUE and
                                           ReceiverPC128=Bf
 d2/2     Start disjunction branch #1    ReceiverPC128=SenderPC114
          (SC RBt) (SC RA) (SCt RA) (SA RA)
     Goal reached at time                SB112+RB126
 d3/2     Start disjunction branch #2    UNDEFINED127~=TRUE and
                                           ReceiverPC128=Bf
          (SC RBf) (SC RA) (SB RA)
     2-way disjunction branch            SenderPC142=SenderPC114
                                         UNDEFINED141~=TRUE and
                                           SenderPC142=ReceiverPC128
 d4/4     Start disjunction branch #1    SenderPC142=SenderPC114
          (SBt RA) (SC RA) (SC RB) (SC RBt) (SC RA) (SCt RA) (SA RA)
     Goal reached at time                SB140+TIMEOUT+RB151+SB112
 d5/4     Start disjunction branch #2    UNDEFINED141~=TRUE and
                                           SenderPC142=ReceiverPC128
          (SBf RA) (SC RA) (SB RA) (SBt RA) (SC RA) (SC RB) (SC RBt) (SC RA)
          (SCt RA) (SA RA)
     Goal reached at time                SB165+SB112+RB176+2*TIMEOUT+SB140
 d6/1     Start disjunction branch #2    UNDEFINED113~=TRUE and SenderPC114=Bf
          (SBf RA) (SC RA) (SB RA) (SBt RA) (SC RA) (SC RAt) (SC RB)
     2-way disjunction branch            ReceiverPC206=SenderPC192
                                         UNDEFINED205~=TRUE and
                                           ReceiverPC206=SenderPC114
 d7/7     Start disjunction branch #1    ReceiverPC206=SenderPC192
```

PRECEDING PAGE BLANK-NOT FILMED

```
          (SC RBt) (SC RA) (SCt RA) (SA RA)
     Goal reached at time                SB190+TIMEOUT+RB204+SB112
 d8/7     Start disjunction branch #2    UNDEFINED205~=TRUE and
                                           ReceiverPC206=SenderPC114
          (SC RBf) (SC RA) (SB RA)
     2-way disjunction branch            SenderPC220=SenderPC192
                                         UNDEFINED219~=TRUE and
                                           SenderPC220=ReceiverPC206
 d9/9     Start disjunction branch #1    SenderPC220=SenderPC192
          (SBt RA) (SC RA) (SC RB) (SC RBt) (SC RA) (SCt RA) (SA RA)
     Goal reached at time                SB218+SB190+RB229+2*TIMEOUT+SB112
 d10/9    Start disjunction branch #2    UNDEFINED219~=TRUE and
                                           SenderPC220=ReceiverPC206
          (SBf RA) (SC RA) (SB RA) (SBt RA) (SC RA) (SC RB) (SC RBt) (SC RA)
          (SCt RA) (SA RA)
     Goal reached at time                SB243+SB112+3*TIMEOUT+RB254+SB190+
                                           SB218
TRUE

_tree
 (SA RA) (SB RA) d1 (SBt RA) (SC RA) (SC RAt) (SC RB) d2 (SC RBt) (SC RA) 1
                                                      d3 (SC RBf) (SC RA) 2
                 d6 (SBf RA) (SC RA) (SB RA) (SBt RA) (SC RA) (SC RAt) 3
1    (SCt RA) (SA RA)
2    (SB RA) d4 (SBt RA) (SC RA) (SC RB) (SC RBt) (SC RA) (SCt RA) (SA RA)
             d5 (SBf RA) (SC RA) (SB RA) (SBt RA) (SC RA) (SC RB) (SC RBt) 4
3    (SC RB) d7 (SC RBt) (SC RA) (SCt RA) (SA RA)
             d8 (SC RBf) (SC RA) (SB RA) d9 (SBt RA) (SC RA) (SC RB) 5
                                          d10 (SBf RA) (SC RA) (SB RA) 6
4    (SC RA) (SCt RA) (SA RA)
5    (SC RBt) (SC RA) (SCt RA) (SA RA)
6    (SBt RA) (SC RA) (SC RB) (SC RBt) (SC RA) (SCt RA) (SA RA)
```

IX. CSDS FOR THE THREE-WAY HANDSHAKE

```
sw
[WAIT pre:       (.SIn.type=Empty .SSendFlag~=TRUE
      exp:       .SIn.type~=Empty
      procs:     (S)
      wait:      (SIn.type)

smed
[WAIT pre:       (.SSendFlag=TRUE)
      exp:       .TIn.type=Empty
      sig:       (TIn.type TIn.seq TIn.ack SSendFlag)
      procs:     (S)
      thenpost:  (#TIn.type=.SBuf.type #TIn.seq=.SBuf.seq #TIn.ack=.SBuf.ack
                   #SSendFlag=FALSE)
      wait:      (TIn.type SBuf.type SBuf.seq SBuf.ack)]

sa
[PSD  pre:       (.SState=Closed .SIn.type=ActiveOpen)
      mod:       (SIn.type SState SSeqToSend SBuf.seq SBuf.ack SBuf.type
                          SSendFlag SOldUnack)
      procs:     (S)
      post:      (#SIn.type=Empty #SState=SynSent #SSeqToSend=SMaxval+1
                   #SBuf.seq=SMaxval SBuf.ack=0 #SBuf.type=Syn
                   (SSendFlag=True #SoldUnack=SMaxval)
      time:      (Range 0)]

sb
[PSD  pre:       (.SState~=Closed .SIn.type=ActiveOpen)
      mod:       (SIn.type)
      procs:     (S)
      post:      (#SIn.type=Empty)
      time:      (RANGE 0)]

sc
[PSD  pre:       (.SState=Closed .SIn.type=PassiveOpen)
      mod:       (SIn.type SState)
      procs:     (S)
      post:      (#SIn.type=Empty #SState=Listen)
      time:      (RANGE 0)]

sd
[PSD  pre:       (.SState~=Closed .SIn.type=PassiveOpen)
      mod:       (SIn.type)
      procs:     (S)
      post:      (#SIn.type=Empty)
      time:      (Range 0)]

se
[PSD  pre:       (.SState=Listen or .SState=Closed .SIn.type=Rst)
      mod:       (SIn.type)
      procs:     (S)
      post:      (#SIn.type=Empty)
      time:      (RANGE 0)]
```

```
sf
[PSD  pre:      (.SState=SynSent .SIn.type=Rst)
      read:     (SIn.ack SoldUnack SState)
      mod:      (SIn.type SState)
      procs:    (S)
      post:     (#SIn.type=Empty
                  (if .SIn.ack=.SOldUnack+1 then #SState=Closed
                    else #SState=.SState))
      time:     (RANGE 0)]

sg
[PSD  pre:      (.SState=SynReceived or .SState=Established .SIn.type=Rst)
      read:     (SIn.seq SSeqToReceive SState)
      mod:      (SIn.type SState)
      procs:    (S)
      post:     (#SIn.type=Empty
                  (if .SIn.seq=.SSeqToReceive then #SState=Closed
                    else #SState=.SState))
      time:     (RANGE 0)]

sh
[PSD  pre:      (.SState=Closed or .SState=Listen .SIn.type=Ack)
      read:     (SIn.ack)
      mod:      (SIn.type SSendFlag SBuf.seq SBuf.ack SBuf.type)
      procs:    (S)
      post:     (#SIn.type=Empty SSendFlag=TRUE SSBuf.seq=.SIn.ack
                  #SBuf.ack=0 #SBuf.type=Rst)
      time:     (RANGE 0)]

si
[PSD  pre:    (.SState=SynSent .SIn.type=Ack)
      read:     (SIn.ack SOldUnack SSendFlag SBuf.seq SBuf.ack
                  SBuf.type)
      mod:      (SIn.type SSendFlag SBuf.seq SBuf.ack SBuf.type)
      procs:    (S)
      post:     (SIn.type=Empty
                (if ~(.SIn.ack=.SOLDUnack+1)
                  then
                    #SSendFlag=TRUE and #SBuf.seq=.SIn.ack and
                      #SBuf.ack=0 and #SBuf.type=Rst
                  else
                    #SSendFlag=.SSendFlag and #SBuf.seq=.SBuf.seq
                       and #SBuf.ack=.SBuf.ack and #SBuf.type=.SBuf.type))
     time:      (RANGE 0)]

sj
[PSD pre:       (.SState=SynReceived .SIn.type=Ack)
     read:      (SIn.ack SOldUnack SIn.seq SSeqToReceive SSendFlag
                  SBuf.type SBuf.seq SBuf.ack SState SSeqToSend)
     mod:       (SIn.type SOldUnack SState SSendFlag SBuf.seq SBuf.ack
                       SBuf.type
     procs:     (S)
     post:      (#SIn.type=Empty
                  (if .SIn.ack=.SOldUnack+1 and .SIn.seq=.SSeqToReceive
```

```
                    then
                      #SOldUnack=.SOldUnack+1 and #SState=Established
                        and #SSendFlag=.SSendFlag and #SBuf.type=.SBuf.type
                        and #SBuf.seq=.SBuf.seq and #SBuf.ack=.SBuf.ack
                    else
                      #SOldUnack=.SOldUnack and #SState=.SState and
                      #SSendFlag=TRUE and
                      (if ~(.SIn.seq=.SSeqToReceive)
                        then
                          #SBuf.seq=.SSeqToSend and #SBuf.ack=.SSeqToReceive
                            and #SBuf.type=Ack
                        else
                          #SBuf.seq=.SIn.ack and #SBuf.ack=0 and
                            #SBuf.type=Rst)))
      time:     (RANGE 0)]

sk
[PSD pre:       (.SState=Established .SIn.type=Ack)
      read:     (SIn.seq SSeqToReceive SSeqToSend SBuf.seq SBuf.ack
                   SBuf.type SSendFlag)
      mod:      (SIn.type SBuf.seq SBuf.ack SBuf.type SSendFlag)
      procs:    (S)
      post:     (#SIn.type=Empty
                  (if ~(.SIn.seq=.SSeqToReceive
                    then
                      #SBuf.seq=.SSeqToSend and #SBuf.ack=.SSeqToReceive
                        and #SBuf.type=Ack and #SSendFlag=TRUE
                    else
                      #SBuf.seq=.SBuf.seq and #SBuf.ack=.SBuf.ack
                        and #SBuf.type=.SBuf.type and #SSendFlag=.SSendFlag))
      time:     (RANGE 0)]

sl
[PSD pre:       (.SState=Closed .SIn.type=Syn)
      read:     (SIn.seq)
      mod:      (SIn.type SSendFlag SBuf.seq SBuf.ack SBuf.type)
      procs:    (S)
      post:     (#SIn.type=Empty #SSendFlag=TRUE #SBuf.seq=0
                  #SBuf.ack=.SIn.seq+1 #SBuf.type=Rst)
      time:     (RANGE 0)]

sm
[PSD pre:       (.SState=Listen .SIn.type=Syn)
      read:     (SIn.seq)
      mod:      (SIn.type SSendFlag SState SSeqToSendToReceive SBuf.seq
                           SBuf.ack SBuf.type SOldUnack)

      procs:    (S)
      post:     (#SIn.type=Empty #SSendFlag=TRUE #SState=SynReceived
                  #SSeqToSend=SMaxval+1 #SSeqToReceive=.Sin.seq+1
                  #SBuf.seq=SMaxval #SBuf.ack=#SSeqToReceive #S
                  #SBuf.type=SynAck #SOldUnack=SMaxval)
      time:     (RANGE 0)]

sn
```

```
[PSD pre:      (.SState=SynSent .SIn.type=Syn)
     read:     (SIn.seq SSeqToSend)
     mod:      (SIn.type SState SSeqToReceive SSendFlag SBuf.seq
                        SBuf.ack SBuf.type)
     procs:    (S)
     post:     (#SIn.type=Empty #SState=SynReceived #
                #SSeqToReceive=.SIn.seq+1 #SSendFlag=TRUE
                #SBuf.seq=.SSeqToSend #SBuf.ack=#SSeqToReceive
                #SBuf.type=Ack)
     time:     (RANGE 0)]

so
[PSD pre:      (.SState=SynReceived or.SState=Established .SIn.type=Syn)
     read:     (SSeqToSend SSeqToReceive)
     mod:      (SIn.type SSendFlag SBuf.seq SBuf.ack SBuf.type)
     procs:    (S)
     post:     (#SIn.type=Empty #SSendFlag=TRUE #SBuf.seq=.SSeqToSend
                 #SBuf.ack=.SSeqToReceive #SBuf.type=Ack)
     time:     RANGE 0)]

sp
[PSD pre:      (.SState=Closed or .SState=Listen .SIn.type=SynAck)
     read:     (SIn.ack)
     mod:      (SIn.type SSendFlag SBuf.seq SBuf.ack SBuf.type)
     procs:    (S)
     post:     (#SIn.type=Empty #SSendFlag=TRUE #SBuf.seq=.SIn.ack
                      #SBuf.ack=0 #SBuf.type=Rst)
     time:     (RANGE 0)]

sq
[PSD pre:      (.SState=SynSent .SIn.type=SynAck)
     read:     (SIn.ack SOldUnack SSeqToSend SIn.seq SState SSeqToReceive)
     mod:      (SIn.type SSendFlag SState SSeqToReceive SBuf.seq
                       SBuf.ack SBuf.ty   SOldUnack)
     procs:    (S)
     post:     (#SIn.type=Empty #SSendFlag=TRUE
                 (if .SIn.ack=.SOldUnack+1
                   then
                     #SState=Established and #SSeqToReceive=.SIn.seq+1
                       and SSBuf.seq=.SSeqToSend and #SSbuf.ack=.SIn.
                       seq+1 and #SBuf.type=Ack and #SOldUnack=.SoldUnack+1
                   else
                     #SState=.SState and #SSeqToReceive=.SSeqToReceive and
                       #SBuf.seq=.SIn.ack and #SBuf.ack=0 and
                       #SBuf.type=Rst and #SOldUnack=.SOldUnack))
     time:    (RANGE 0)]

sr
[PSD pre:      (.SState=SynReceived or .SState=Established .SIn.type=SynAck)
     read:     (SSeqToSend SSeqToReceive)
     mod:      (SIn.type SSendFlag SBuf.seq SBuf.ack SBuf.type)
     procs:    (S)
     post:     (#SIn.type=Empty #SSendFlag=TRUE #SBuf.seq=.SSeqToSend
                 #SBuf.ack=.SSeqToReceive #SBuf.type=Ack)
     time:     (RANGE 0)]
```

X. PROOF OF FIRSTHALF OF A SIMPLE THREE-WAY HANDSHAKE IN CSD

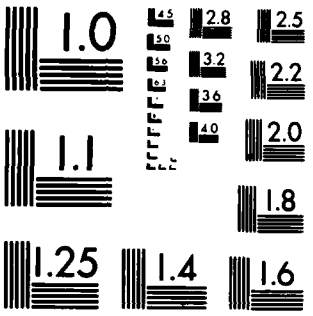
Notes: See Appendix VIII for explanations. The trace shows which CSD has just completed in each state pair (after the ;) in addition to the information described in Appendix VIII. This proof covers only the case in which both nodes are performing an Active Open. (The Active/Passive case is much simpler.)

```
29_*prove FirstHalf
          (ta sa)
     2-way interleave
i1/1      Assume first to finish is    (T)
          (tmed sa ; ta) (tmed smed ; sa)
     2-way interleave
i2/2      Assume first to finish is    (T)
          (tw =SynSent smed ; tmed) (tw =SynSent sn ; smed)
     2-way interleave
i3/3      Assume first to finish is    (T)
          (tn sn ; tw)
     2-way interleave
i4/4      Assume first to finish is    (T)
          (tmed sn ; tn) (tmed smed ; sn)
     Goal reached at time               sa114+sn115
i5/4      Assume first to finish is    (S)
          (tn smed ; sn) (tmed smed ; tn)
     Goal reached at time               sa114+tn116
i6/3      Assume first to finish is    (S)
          (tw =SynSent smed ; sn) (tn smed ; tw) (tmed smed ; tn)
     Goal reached at time               sn115+sa114+tn117
i7/2      Assume first to finish is    (S)
          (tmed sw =SynSent ; smed) (tn sw=SynSent ; tmed)
     2-way interleave
i8/8      Assume first to finish is     (T)
          (tmed sw =SynSent ; tn) (tmed sn ; sw) (tmed smed ; sn)
     Goal reached at time              tn118+sa114+sn119
i9/8      Assume first to finish is     (S)
          (tn sn ; sw)
     2-way interleave
i10/10    Assume first to finish is     (T)
          (tmed sn ; tn) (tmed smed ; sn)
     Goal reached at time               sa114+sn120
i11/10    Assume first to finish is     (S)
          (tn smed ; sn) (tmed smed ; tn)
     Goal reached at time               sa114+tn118
i12/1     Assume first to finish is     (S)
          (ta smed ; sa) (tmed smed ; ta)
     2-way interleave
i13/13    Assume first to finish is     (T)
          (tw =SynSent smed ; tmed) (tw =SynSent sn ; smed)
     2-way interleave
i14/14    Assume first to finish is     (T)
```

AD-A126 271 AUTOMATED PROTOCOL VERIFICATION(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST C A SUNSHINE ET AL. OCT 82 ISI/RR-83-110 2/2

UNCLASSIFIED MDA903-81-C-0335 F/G 9/2 NL

END
DATE FILMED
4-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
         (tn sn ; tw)
    2-way interleave
i15/15   Assume first to finish is    (T)
         (tmed sn ; tn) (tmed smed ; sn)
    Goal reached at time              ta113+sn121
i16/15   Assume first to finish is    (S)
         (tn smed ; sn) (tmed smed ; tn)
    Goal reached at time              ta113+tn122
i17/14   Assume first to finish is    (S)
         (tw =SynSent smed ; sn) (tn smed ; tw) (tmed smed ; tn)
    Goal reached at time              sn121+ta113+tn123
i18/13   Assume first to finish is    (S)
         (tmed sw =SynSent ; smed) (tn sw =SynSent ; tmed)
    2-way interleave
i19/19   Assume first to finish is    (T)
         (tmed sw =SynSent ; tn) (tmed sn ; sw) (tmed smed ; sn)
    Goal reached at time              tn124+ta113+sn125
i20/19   Assume first to finish is    (S)
         (tn sn ; sw)
    2-way interleave
i21/21   Assume first to finish is    (T)
         (tmed sn ; tn) (tmed smed ; sn)
    Goal reached at time              ta113+sn126
i22/21   Assume first to finish is    (S)
         (tn smed ; sn) (tmed smed ; tn)
    Goal reached at time              ta113+tn124
455886 conses
346.754 seconds
106.979 seconds, garbage collection time
TRUE

30_*tree
 (ta sa) i1 (tmed sa ; ta) (tmed smed ; sa) i2 (tw =SynSent smed ; tmed) 1
                                            i7 (tmed sw =SynSent ; smed) 2
         i12 (ta smed ; sa) tmed smed ; ta) i13 (tw =SynSent smed ; tmed) 3
                                            i18 (tmed sw =SynSent ; smed) 4

1  (tw=SynSent sn ;smed) i3 (tn sn ; tw)  i4 (tmed sn ; tn) (tmed smed ; sn)
                                          i5 (tn smed ; sn) (tmed smed ; tn)
                         i6 (tw=SynSent smed ;sn) (tn smed ;tw) (tmed smed ;tn)
2  (tn sw=SynSent ;tmed) i8 (tmed sw=SynSent ;tn) (tmed sn ;sw) (tmed smed ;sn)
                         i9 (tn sn ; sw)  i10 (tmed sn ; tn) (tmed smed ; sn)
                                          i11 (tn smed ; sn) (tmed smed ; tn)
3  (tw=SynSent sn ;smed) i14 (tn sn ; tw)  i15 (tmed sn ; tn) (tmed smed ; sn)
                                           i16 (tn smed ; sn) (tmed smed ; tn)
                         i17 (tw=SynSent smed ;sn) (tn smed ;tw) (tmed smed ; tn)
4  (tn sw=SynSent ;tmed) i19 (tmed sw=SynSent ;tn) (tmed sn ;sw) (tmed smed ;sn)
                         i20 (tn sn ; sw) i21 (tmed sn ; tn) (tmed smed ; sn)
                                          i22 (tn smed ; sn) (tmed smed ; tn)
NIL
```

XI. CSDS FOR A THREE-WAY HANDSHAKE USING QUEUES

Notes: Predicates, defined first, are substituted wherever their name appears in the CSDs, with actual arguments replacing the formal ones (&1, &2, ...).

```
pred SSend
 (if .SLost ge SMaxloss then #SLost=0 and (pred SEnqueue &1 &2 &3)
  else #SLost=0 and (pred SEnqueue &1 &2 &3)
       or #SLost=.SLost+1 and #TIn.type=.TIn.type and #TIn.seq=.TIn.seq
          and #TIn.ack=.TIn.ack))
pred SEnqueue
 #TIn.type=(ENQUEUE .TIn.type &1) and #TIn.seq=(ENQUEUE .TIn.seq &2)
       and #TIn.ack=(ENQUEUE .TIn.ack &3))
pred SDequeue #SIn.type=(DEQUEUE .SIn.type) and #SIn.seq=(DEQUEUE .SIn.seq)
       and #SIn.ack=(DEQUEUE .SIn.ack))
pred SIn (if (EMPTYQUEUE .SIn.type) or .STimeoutFlag=TRUE
          then FALSE else (FRONTQUEUE .SIn.type)=&1))
pred SSetPending #SPending.type=&1 and #SPending.seq=&2 and #SPending.ack=&3)
pred SClearPending #SPending.type=Empty)
pred SNothingSent #SLost=.SLost and #TIn.type=.TIn.type
       and #TIn.seq=.TIn.seq and #TIn.ack=.TIn.ack)


sw
[WAIT pre:      ((EMPTYQUEUE .SIn.type) .STimeoutFlag=FALSE)
      exp:      ~(EMPTYQUEUE .SIn.type)
      sig:      (STimeoutFlag)
      procs:    (S)
      thenpost: (#STimeoutFlag=FALSE)
      elsepost: (#STimeoutFlag=TRUE)
      wait:     (SIn.type)
      time:     STimeout]

sz
[PSD  pre:      (.STimeoutFlag=TRUE)
      read:     (SPending.type SLost TIn.type TIn.seq TIn.ack
                             SPending.seq SPending.ack)
      mod:      (STimeoutFlag SLost TIn.type TIn.seq TIn.ack)
      procs:    (S)
      post:     (#STimeoutFlag=FALSE
                 (if .SPending.type=Empty then (pred SNothingSent)
                   else
                     (pred SSend .SPending.type .SPending.seq
                       .SPending.ack)))
      time:     (RANGE SMin SMax)]

sa
[PSD  pre:      (.SState=Closed (pred SIn ActiveOpen))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack SState SSeqToSend SLost
                          TIn.type TIn.seq TIn.ack SPending.type
                          SPending.seq SPending.ack SOldUnack)
```

```
      procs:    (S)
      post:     ((pred SDequeue) #SState=SynSent #SSeqToSend=SMaxval+1
                  (pred SSend Syn SMaxval 0)
                  (pred SSetPending Syn SMaxval 0) #SOldUnack=SMaxval)
      time:     (RANGE SMin SMax)]

sb
[PSD  pre:      (.SState~=Closed (pred SIn ActiveOpen))
      read:     (SIn.type SIn.seq SIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack)
      procs:    (S)
      post:     ((pred SDequeue))
      time:     (RANGE SMin SMax)]

sc
[PSD  pre:      (.SState=Closed (pred SIn PassiveOpen))
      read:     (SIn.type SIn.seq SIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack SState)
      procs:    (S)
      post:     ((pred SDequeue) #SState=Listen)
      time:     (RANGE SMin SMax)]

sd
[PSD  pre:      (.SState~=Closed (pred SIn PassiveOpen))
      read:     (SIn.type SIn.seq SIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack)
      procs:    (S)
      post:     ((pred SDequeue))
      time:     (RANGE SMin SMax)]

se
[PSD  pre:      (.SState=Listen or .SState=Closed (pred SIn Rst))
      read:     (SIn.type SIn.seq SIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack)
      procs:    (S)
      post:     ((pred SDequeue))
      time:     (RANGE SMin SMax)]

sf
[PSD  pre:      (.SState=SynSent (pred SIn Rst))
      read:     (SIn.type SIn.seq SIn.ack SOldUnack SState)
      mod:      (SIn.type SIn.seq SIn.ack SState SPending.type)
      procs:    (S)
      post:     ((pred SDequeue)
                  (if (FRONTQUEUE .SIn.ack)=.SOldUnack+1
                    then #SState=Closed and (pred SClearPending)
                    else #SState=.SState))
      time:     (RANGE SMin SMax)]

sg
[PSD  pre:      (.SState=SynReceived or .SState=Established
                  (pred SIn Rst))
      read:     (SIn.type SIn.seq SIn.ack SSeqToReceive SState)
      mod:      (SIn.type SIn.seq SIn.ack SState SPending.type)
      procs:    (S)
```

```
        post:     ((pred SDequeue)
                    (if (FRONTQUEUE .SIn.seq)=.SSeqToReceive
                      then #SState=Closed and (pred SClearPending)
                      else #SState=.SState))
        time:     (RANGE SMin SMax)]

sh
[PSD    pre:      (.SState=Closed or .SState=Listen (pred SIn Ack))
        read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                            TIn.ack)
        mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                            TIn.ack)
        procs:    (S)
        post:     ((pred SDequeue)
                    (pred SSend Rst (FRONTQUEUE .SIn.ack) 0))
        time:     (RANGE SMin SMax)]

si
[PSD    pre:      (.SState=SynSent (pred SIn Ack))
        read:     (SIn.type SIn.seq SIn.ack SOldUnack SLost TIn.type
                            TIn.seq TIn.ack)
        mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                            TIn.ack)
        procs:    (S)
        post:     ((pred SDequeue)
                    (if ~((FRONTQUEUE .SIn.ack)=.SOldUnack+1)
                      then (pred SSend Rst (FRONTQUEUE .SIn.ack) 0)
                      else (pred SNothingSent)))
        time:     (RANGE SMin SMax)]

sj
[PSD    pre:      (.SState=SynReceived (pred SIn Ack))
        read:     (SIn.type SIn.seq SIn.ack SOldUnack SSeqToReceive
                            TIn.type TIn.seq TIn.ack SState SLost
                            SSeqToSend)
        mod:      (SIn.type SIn.seq SIn.ack SOldUnack SState TIn.type
                            TIn.seq TIn.ack SPending.type SLost)
        procs:    (S)
        post:     ((pred SDequeue)
                    (if
                      (FRONTQUEUE .SIn.ack)=.SOldUnack+1 and
                        (FRONTQUEUE .SIn.seq)=.SSeqToReceive
                      then
                        #SOldUnack=.SOldUnack+1 and #SState=Established
                          and (pred SNothingSent) and
                          (pred SClearPending)
                      else
                        #SOldUnack=.SOldUnack and #SState=.SState and
                          (if ~((FRONTQUEUE .SIn.seq)=.SSeqToReceive)
                            then
                               and
                                (pred SSend Ack .SSeqToSend
                                  .SSeqToReceive)
                            else (pred SSend Rst (FRONTQUEUE .SIn.ack) 0)
                            )))
```

```
      time:     (RANGE SMin SMax)]

sk
[PSD  pre:      (.SState=Established (pred SIn Ack))
      read:     (SIn.type SIn.seq SIn.ack SSeqToReceive SLost TIn.type
                          TIn.seq TIn.ack SSeqToSend)
      mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      procs:    (S)
      post:     ((pred SDequeue)
                  (if ~((FRONTQUEUE .SIn.seq)=.SSeqToReceive)
                    then (pred SSend Ack .SSeqToSend .SSeqToReceive)
                    else (pred SNothingSent)))
      time:     (RANGE SMin SMax)]

sl
[PSD  pre:      (.SState=Closed (pred SIn Syn))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      procs:    (S)
      post:     ((pred SDequeue)
                  (pred SSend Rst 0 (FRONTQUEUE .SIn.seq+1)))
      time:     (RANGE SMin SMax)]

sm
[PSD  pre:      (.SState=Listen (pred SIn Syn))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack SState SSeqToSend
                          SSeqToReceive SLost TIn.type TIn.seq TIn.ack
                          SOldUnack SPending.type SPending.seq
                          SPending.ack)
      procs:    (S)
      post:     ((pred SDequeue) #SState=SynReceived
                  #SSeqToSend=SMaxval+1
                  #SSeqToReceive=(FRONTQUEUE .SIn.seq)+1
                  (pred SSend SynAck SMaxval #SSeqToReceive)
                  #SOldUnack=SMaxval
                  (pred SSetPending SynAck SMaxval #SSeqToReceive))
      time:     (RANGE SMin SMax)]

sn
[PSD  pre:      (.SState=SynSent (pred SIn Syn))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack SSeqToSend)
      mod:      (SIn.type SIn.seq SIn.ack SState SSeqToReceive SLost
                          TIn.type TIn.seq TIn.ack)
      procs:    (S)
      post:     ((pred SDequeue) #SState=SynReceived
                  #SSeqToReceive=(FRONTQUEUE .SIn.seq)+1
                  (pred SSend Ack .SSeqToSend #SSeqToReceive))
      time:     (RANGE SMin SMax)]
```

```
so
[PSD  pre:      (.SState=SynReceived or .SState=Established
                  (pred SIn Syn))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack SSeqToSend SSeqToReceive)
      mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      procs:    (S)
      post:     ((pred SDequeue)
                  (pred SSend Ack .SSeqToSend .SSeqToReceive))
      time:     (RANGE SMin SMax)]

sp
[PSD  pre:      (.SState=Closed or .SState=Listen (pred SIn SynAck))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      procs:    (S)
      post:     ((pred SDequeue)
                  (pred SSend Rst (FRONTQUEUE .SIn.ack) 0))
      time:     (RANGE SMin SMax)]

sq
[PSD  pre:      (.SState=SynSent (pred SIn SynAck))
      read:     (SIn.type SIn.seq SIn.ack SOldUnack SLost TIn.type
                          TIn.seq TIn.ack SSeqToSend SState
                          SSeqToReceive)
      mod:      (SIn.type SIn.seq SIn.ack SState SSeqToReceive SLost
                          TIn.type TIn.seq TIn.ack SPending.type
                          SOldUnack)
      procs:    (S)
      post:     ((pred SDequeue)
                  (if (FRONTQUEUE .SIn.ack)=.SOldUnack+1
                    then
                      #SState=Established and
                        #SSeqToReceive=(FRONTQUEUE .SIn.seq)+1 and
                        (pred SSend Ack .SSeqToSend #SSeqToReceive) and
                        (pred SClearPending) and
                        #SOldUnack=.SOldUnack+1
                    else
                      #SState=.SState and #SSeqToReceive=.SSeqToReceive
                        and (pred SSend Rst (FRONTQUEUE .SIn.ack) 0)
                        and #SOldUnack=.SOldUnack))
      time:     (RANGE SMin SMax)]

sr
[PSD  pre:      (.SState=SynReceived or .SState=Established
                  (pred SIn SynAck))
      read:     (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack SSeqToSend SSeqToReceive)
      mod:      (SIn.type SIn.seq SIn.ack SLost TIn.type TIn.seq
                          TIn.ack)
      procs:    (S)
      post:     ((pred SDequeue)
```

```
                (pred SSend Ack .SSeqToSend .SSeqToReceive))
        time:   (RANGE SMin SMax)]
```

REFERENCES

[BoMo 79] Boyer, R., and J. Moore, *A Theorem Prover for Recursive Functions: A User's Manual*, SRI International, Technical Report CSL-91, June 1979.

[Crai 81] Craigen, D., *Formal Verification of Programs: Report # 5*, I. P. Sharp, Ltd., Ottawa, Canada, Technical Report TR-81-5605-5, June 1981.

[Croc 77] Crocker, S., *State Deltas: A Formalism for Representing Segments of Computation*, Ph.D. thesis, University of California, Los Angeles, 1977.

[Diaz 82] Diaz, M., "Modeling and analysis of communication and cooperation protocols using Petri net based models," in *Proceedings of the Second International Workshop on Protocol Specification, Testing, and Verification*, Idyllwild, California, North-Holland, May 1982, pp. 465-510. Revised version to appear in *Computer Networks*, 1982.

[Divi 81] DiVito, B., *A Mechanical Verification of the Alternating Bit Protocol*, University of Texas at Austin, Technical Report ICSCA-CMP-21, June 1981.

[Gerh 80] Gerhart, S., et al., "An overview of Affirm: A specification and verification system," in *Information Processing 80: Proceedings of the IFIP Congress*, Melbourne, Australia, October 1980, pp. 343-347.

[Gogu 81] Goguen, J., and R. Burstall, *An Ordinary Design*, SRI International, Technical Report, 1981.

[Good 78] Good, D., et al., *Report on the Language Gypsy*, University of Texas at Austin, Technical Report ISCMA-CMP-10, September 1978.

[GoDi 81] Good, D., and B. DiVito, Using the Gypsy methodology, University of Texas at Austin, October 1981. Draft report.

[Haje 78] Hajek, J., "Automatically verified data transfer protocols," in *Proceedings of the Fourth International Computer Communication Conference*, Kyoto, September 1978, pp. 749-756.

[HDM 79] Levitt, K., B. Silverberg, and L. Robinson, *The HDM Handbook*, SRI International, Technical Report, June 1979. (Three volumes.)

[Loca 80] Locasso, R., et al., *The Ina Jo Specification Language Reference Manual*, System Development Corporation, Technical Manual TM-(L)-6021/001/00, June 1980.

[Over 81] Overman, W., *Verification of Concurrent Systems: Function and Timing*, Ph.D. thesis, University of California, Los Angeles, 1981.

[OvCr 82] Overman, W., and S. Crocker, "Verification of concurrent systems: Function and timing," in *Proceedings of the Second International Workshop on Protocol Specification, Testing, and Verification*, Idyllwild, California, North-Holland, May 1982, pp. 401-409.

[QuSi 81] Queille, J., and J. Sifakis, *Specification and Verification of Concurrent Systems in CESAR: An Example*, Laboratoire D'Informatique et de Mathematique Appliquees de Grenoble, France, RR 254, June 1981.

[Post 81] Postel, J., ed., "Transmission Control Protocol," USC/Information Sciences Institute RFC 793, September 1981.

[RaEs 80] Razouk, R., and G. Estrin, "Modeling and verification of communication protocols in SARA: The X.21 interface," *IEEE Transactions on Computers* C-29 (12), December 1980, 1038-1051.

[Schw 81] Schwabe, D., "Formal specification and verification of a connection-establishment protocol," in *Proceedings of the Seventh Data Communications Symposium*, Mexico City, October 1981, pp. 11-26. Also USC/Information Sciences Institute RR-81-91, April 1981.

[ScMe 81] Schwartz, R. L., and P. M. Melliar-Smith, "Temporal logic specifications of distributed systems," in *Proceedings of the Second International Conference on Distributed Computing Systems*, Paris, April 1981, pp. 446-454.

[Suns 75] Sunshine, C., *Interprocess Communication Protocols for Computer Networks*, Ph.D. thesis, Stanford University, 1975.

[Suns 81a] Sunshine, C., et al., "Specification and verification of communication protocols in Affirm using state transition models," IEEE Transactions on Software Engineering SE-8 (9), September 1982, 460-489. Also USC/Information Sciences Institute RR-81-88, March 1981.

[Suns 81b] Sunshine, C., "The Restaurant Example Revisited," USC/Information Sciences Institute Affirm Memo 52, September 1981.

[Suns 81c] Sunshine, C., *Formal Modeling of Communication Protocols*, USC/Information Sciences Institute RR-81-89, March 1981.

[Suns 82a] Sunshine, C., "Experience with four automated verification systems," in *Proceedings of the Second International Workshop on Protocol Specification, Testing, and Verification*, Idyllwild, California, North-Holland, May 1982, pp. 373-379.

[Suns 82b] Sunshine, C., "Protocol Specification and Verification Work at USC/ISI: Summary Report," DARPA Internet Experiment Note # 211, August 1982.

[SVG 79] Stanford Verification Group, *Stanford Pascal Verifier User Manual*, Stanford University, Technical Report STAN-CS-79-731, March 1979.

[West 82] West, C. H., "Applications and limitations of automated protocol validation," in *Proceedings of the Second International Workshop on Protocol Specification, Testing, and Verification*, Idyllwild, California, North-Holland, May 1982, pp. 361-371.

[Yemi 82] Yemini, Y., and J. Kurose, "Towards the unification of the functional and performance analysis of protocols, or Is the Alternating Bit protocol really correct?" in *Proceedings of the Second International Workshop on Protocol Specification, Testing, and Verification*, Idyllwild, California, North-Holland, May 1982, pp. 189-196. Revised version to appear in *Computer Networks*, 1983.