TRANSCRIPTS FOR THE PROOF OF THE ALTERNATING BIT PROTOCOL

Benedetto L. DiVito

Technical Report #22@June 1981

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

The sole purpose of this report is to bring together the many pages of verifier transcripts that were generated in our proof of the Alternating Bit Protocol. A summary of this work is contained in a companion report [DiVito 81]. Refer to that report for a description of the problem and a basic outline of the approach. The present report is not self-contained; it is primarily a raw collection of transcripts and is not intended to contain explanatory material.

As mentioned in the summary report, the proof was a fully mechanical verification effort. In addition, there were two separate verification systems used in this work: the Gypsy system and the Affirm system. We present complete transcripts showing the work done on both systems. It is divided into the following four parts:

1. Text of the Gypsy model of the protocol and the intermediate specifications. Includes the supporting lemmas used in the Gypsy part of the proof. Followed by the output of the verification condition (VC) generation.
2. Prover transcripts for the proofs of the VCs. These were done with the Gypsy prover.
3. Prover transcripts for the proofs of those supporting lemmas which were proved with the Gypsy prover.
4. Transcripts from the Affirm system. Includes a listing of the type specifications expressed in Affirm. Followed by sessions with the Affirm prover in which the remaining lemmas were proved.

Documentation on the use of both verification systems can be found in the various manuals cited in the references.

References

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

[Good 78] D. I. Good, et al. "Report on the Language Gypsy Version 2.0," ICSCA-CMP-10, University of Texas at Austin, September 1978.

[CMP 80] Certifiable Minicomputer Project. "Gypsy 2.0 Verification Environment 6.0 Users Manual," ICSCA, University of Texas at Austin, February 1980.

[Thompson 81] D. H. Thompson, et al. "The Affirm Reference

Library," USC-ISI, 1981. (Five volume set.)
[PHOTO: Recording initiated Tue 5-May-81 9:05AM]
LINK FROM CMP.DIVITO, TTY 20
TOPS-20 Command processor 4(560)
[PHOTO: Logging disabled Tue 5-May-81 9:05AM]
PHOTO: Logging enabled Tue 5-May-81 9:13AM]
44444 The following is a transcript of the modeling and verification of
4 the Alternating Bit Protocol in Gypsy. We begin by displaying the
4 Gypsy text and the generation of verification conditions. This is
4 followed by the various proof transcripts. Most of the proofs are
4 carried out with the Gypsy prover. However, some are performed
4 with the help of the Affirm system. These are documented in a
4 separate set of transcripts.
44444@ty abp.gyp,abp.lem
Abp.Gyp.2
scope alt_bit_protocol =
begin
type bit = (zero, one);
type message = sequence of character;
type packet = record (mssg : message;
seqno : bit);
type pkt_buf = buffer of packet;
type clk_buf = buffer of boolean;
type msg_buf = buffer of message;
type msg_seq = sequence of message;
type pkt_seq = sequence of packet;
type bit_seq = sequence of bit;
procedure ab_protocol (var source : msg_buf<input>;
var sink : msg_buf<output>) =
begin
block msg_lag (outto (sink, myid), infrom (source, myid), 1);
exit false;
var pkt_send, pkt_rcv, ack_send, ack_rcv : pkt_buf;
var clock_in, clock_out : clk_buf;
cobegin
sender (source, pkt_send, ack_send, clock_in, clock_out);
medium (pkt_send, pkt_rcv);
medium (ack_rcv, ack_send);
receiver (sink, pkt_rcv, ack_rcv);
timer (clock_in, clock_out)
end
end;

```
procedure sender (var source : msg_buf<input>;
var pkt_send : pkt_buf<output>;
var ack_send : pkt_buf<input>;
var start : clk_buf<output>;
var tick : clk_buf<input>) =
begin
block proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1);
exit false;
var pack, ack : packet;
var b : boolean;
var next : bit := one;
loop
assert proper_transmission (infrom (source, myid),
outto (pkt_send, myid),
infrom (ack_send, myid), 0)
& next = next_seqnum (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid));
receive pack.mssg from source;
pack.seqno := next;
send pack to pkt_send;
send true to start;
loop
assert proper_transmission (infrom (source, myid),
outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid)
= unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid))
& pack.seqno = next;
await
on receive ack from ack_send
then if ack.seqno = next
then leave
end
on receive b from tick
then send pack to pkt_send;
send true to start
```

```
end
end;
next := comp (next)
end
end;
procedure medium (var pkt_in : pkt_buf <input>;
var pkt_out : pkt_buf <output>) =
begin
block outto (pkt_out, myid) sub infrom (pkt_in, myid);
exit false;
pending
end;
procedure receiver (var sink : msg_buf<output>;
var pkt_rcv : pkt_buf<input>;
var ack_rcv : pkt_buf<output>) =
begin
block proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 1);
exit false;
var pack : packet;
var exp : bit := one;
loop
assert proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& exp = next_seqnum (infrom (pkt_rcv, myid))
& exp = next_seqnum (outto (ack_rcv, myid));
receive pack from pkt_rcv;
if exp = pack.seqno
then send pack.mssg to sink;
exp := comp (exp)
end;
send pack to ack_rcv
end
end;
procedure timer (var clock_in : clk_buf<input>;
var clock_out : clk_buf<output>) =
begin
block true;
exit false;
pending
end;
function comp (b : bit) : bit =
```

```
begin
exit result = if b = zero then one else zero fi;
result := if b = zero then one else zero fi;
end;
name proper_transmission, proper_reception, next_seqnum, msg_lag,
seqnums, last_bit from alt_bit_specs;
name nchanges, unique_msg, repeats, initial_subseq from alt_bit_specs;
end;
scope alt_bit_specs =
begin
name bit, message, packet, pkt_buf, msg_buf, clk_buf, pkt_seq,
msg_seq, bit_seq, comp from alt_bit_protocol;
function msg_lag (s, t : msg_seq; n : integer) : boolean =
begin
exit (assume result
iff initial_subseq (s, t)
& size (t) - size (s) in [0..n]);
end;
function initial_subseq (u, v : msg_seq) : boolean = pending;
{begin
exit (assume result iff some s : msg_seq, u  s = v);
end;}
function proper_transmission (source : msg_seq;
pkt_send, ack_send : pkt_seq; n : integer)
: boolean =
begin
exit (assume result
iff msg_lag (unique_msg (pkt_send), source, n)
& size (unique_msg (pkt_send))
- size (unique_msg (ack_send))
in [0..n]
& size (source) - size (unique_msg (ack_send))
in [0..n] & repeats (pkt_send));
end;
function proper_reception (sink : msg_seq;
pkt_rcv, ack_rcv : pkt_seq; n : integer) : boolean =
begin
exit (assume result
iff msg_lag (sink, unique_msg (pkt_rcv), n)
& ack_rcv sub pkt_rcv
& size (sink) - size (unique_msg (ack_rcv))
in [0..n]);
```

```
end;
function next_seqnum (ps : pkt_seq) : bit =
begin
exit (assume result = comp (last_bit (seqnums (ps))));
end;
function last_bit (bs : bit_seq) : bit =
begin
exit (assume result = if bs = null (bit_seq) then zero else last (bs) fi);
end;
function seqnums (ps : pkt_seq) : bit_seq =
begin
exit (assume result
= if ps = null (pkt_seq)
then null (bit_seq)
else seqnums (nonlast (ps)) <: last (ps).seqno
fi);
end;
function nchanges (bs : bit_seq) : integer =
begin
exit (assume result
= if bs = null (bit_seq)
then 0
else if last (bs) = last_bit (nonlast (bs))
then nchanges (nonlast (bs))
else nchanges (nonlast (bs)) + 1
fi
fi);
end;
function unique_msg (ps : pkt_seq) : msg_seq =
begin
exit (assume result
= if ps = null (pkt_seq)
then null (msg_seq)
else if last (ps).seqno
= next_seqnum (nonlast (ps))
then unique_msg (nonlast (ps)) <: last (ps).mssg
else unique_msg (nonlast (ps))
fi
fi);
end;
function repeats (ps : pkt_seq) : boolean =
begin
```

```
exit (assume result
iff if ps = null (pkt_seq) or nonlast (ps) = null (pkt_seq)
then true
else repeats (nonlast (ps))
& [ last (ps).seqno
ne next_seqnum (nonlast (ps))
-> last (ps) = last (nonlast (ps)) ]
fi);
end;
end;
Abp.Lem.5
scope lemmas =
begin
lemma prop_trans_1 (u: msg_seq; x, y: pkt_seq; p:packet; m: message) =
p.seqno = next_seqnum (x) & p.mssg = m
& proper_transmission (u, x, y, 0)
-> u <: m = unique_msg (x <: p);
lemma prop_trans_2 (u: msg_seq; x, y: pkt_seq; p:packet; m: message) =
p.seqno = next_seqnum (x) & p.mssg = m
& proper_transmission (u, x, y, 0)
-> proper_transmission (u <: m, x <: p, y, 1);
lemma prop_trans_3 (u: msg_seq; x, y: pkt_seq; p: packet) =
proper_transmission (u, x, y, 1)
& x ne null (pkt_seq) & p = last(x)
-> proper_transmission (u, x <: p, y, 1);
lemma prop_trans_4 (u: msg_seq; x, y: pkt_seq; p:packet) =
p.seqno ne next_seqnum (y)
& proper_transmission (u, x, y, 1)
-> proper_transmission (u, x, y <: p, 1);
lemma prop_trans_5 (u: msg_seq; x, y: pkt_seq; m: message) =
proper_transmission (u, x, y, 0)
-> proper_transmission (u <: m, x, y, 1);
lemma prop_trans_6 (u: msg_seq; x, y: pkt_seq) =
proper_transmission (u, x, y, 0)
-> proper_transmission (u, x, y, 1);
lemma prop_trans_7 (u: msg_seq; x, y: pkt_seq; p: packet) =
unique_msg (x) = u
& size (unique_msg (x)) = size (unique_msg (y)) + 1
& p.seqno = next_seqnum (y)
& proper_transmission (u, x, y, 1)
-> proper_transmission (u, x, y <: p, 0);
lemma prop_rec_1 (u: msg_seq; x, y: pkt_seq; p: packet; m: message) =
```

```
p.seqno = next_seqnum (x) & p.mssg = m
& proper_reception (u, x, y, 0)
-> proper_reception (u <: m, x <: p, y, 1);
lemma prop_rec_2 (u: msg_seq; x, y: pkt_seq; p: packet) =
proper_reception (u, x, y, 0)
-> proper_reception (u, x <: p, y, 1);
lemma prop_rec_3 (u: msg_seq; x, y: pkt_seq) =
proper_reception (u, x, y, 0) -> proper_reception (u, x, y, 1);
lemma prop_rec_4 (u: msg_seq; x, y: pkt_seq; p: packet; m: message) =
p.seqno = next_seqnum (x) & p.seqno = next_seqnum (y)
& proper_reception (u, x, y, 0) & p.mssg = m
-> proper_reception (u <: m, x <: p, y <: p, 0);
lemma prop_rec_5 (u: msg_seq; x, y: pkt_seq; p: packet) =
p.seqno ne next_seqnum (x) & p.seqno ne next_seqnum (y)
& proper_reception (u, x, y, 0)
-> proper_reception (u, x <: p, y <: p, 0);
lemma abp_1 (u, v: msg_seq; w, x, y, z: pkt_seq) =
proper_transmission (u, w, z, 1)
& proper_reception (v, x, y, 1)
& x sub w & z sub y
-> msg_lag (v, u, 1);
lemma main_lemma (s: bit_seq; x, y: pkt_seq) =
s sub seqnums (x) & x sub y & repeats (y)
& nchanges (seqnums (y)) - nchanges (s) in [0..1]
-> msg_lag (unique_msg (x), unique_msg (y), 1);
lemma interpolate (s: bit_seq; x, y: pkt_seq) =
s sub seqnums (x) & x sub y & repeats (y)
& nchanges (seqnums (y)) - nchanges (s) in [0..1]
-> nchanges (seqnums (y)) - nchanges (seqnums (x)) in [0..1] ;
lemma next_comp (x: pkt_seq; p: packet) =
next_seqnum (x <: p) = comp (p.seqno);
lemma ne_next (x: pkt_seq; p: packet) =
p.seqno ne next_seqnum (x)
-> next_seqnum (x <: p) = next_seqnum (x);
lemma last_next (x: pkt_seq; p: packet) =
x ne null (pkt_seq) & p = last (x)
-> p.seqno ne next_seqnum (x);
lemma last_unique (x: pkt_seq; p: packet) =
x ne null (pkt_seq) & p = last(x)
-> unique_msg (x <: p) = unique_msg (x);
lemma last_repeats (x: pkt_seq; p: packet) =
x ne null (pkt_seq) & p = last (x) & repeats (x)
```

```
-> repeats (x <: p);
lemma eq_iss (u, v: msg_seq) =
u = v -> initial_subseq (u, v);
lemma eq_iss_app (u, v: msg_seq; m: message) =
u = v -> initial_subseq (u <: m, v <: m);
lemma iss_app (u, v: msg_seq; m: message) =
initial_subseq (u, v) -> initial_subseq (u, v <: m);
lemma iss_trans (u, v, w: msg_seq) =
initial_subseq (u, v) & initial_subseq (v, w)
-> initial_subseq (u, w);
lemma msg_lag_eq (u, v: msg_seq) =
msg_lag (u, v, 0) iff u = v;
lemma comp_ne (b1, b2: bit) = b1 ne b2 iff comp (b1) = b2;
lemma bit_cases (b: bit) = b = zero or b = one;
lemma hist_sub (x, y: pkt_buf) =
allfrom (x) sub allfrom (y) -> allfrom (x) sub allto (y);
lemma app_pkt_nonnull (x: pkt_seq; p: packet) =
[(x <: p) = null (pkt_seq)] = false;
lemma app_msg_nonnull (u: msg_seq; m: message) =
[(u <: m) = null (msg_seq)] = false;
lemma app_bit_nonnull (s: bit_seq; b: bit) =
[(s <: b) = null (bit_seq)] = false;
lemma sub_app (x, y: pkt_seq; p: packet) =
(assume x sub y -> x <: p sub y <: p);
lemma size_null (u: msg_seq) =
(assume size (u) = 0 iff u = null (msg_seq));
lemma sub_seqnum (x, y: pkt_seq) =
(assume x sub y -> seqnums (x) sub seqnums (y));
lemma sub_nchanges (s, t: bit_seq) =
(assume s sub t -> nchanges (s) le nchanges (t));
lemma nchanges_unique (x: pkt_seq) =
(assume size (unique_msg (x)) = nchanges (seqnums (x)) );
lemma sub_to_lag (x, y: pkt_seq) =
(assume x sub y & repeats (y)
& nchanges (seqnums (y)) - nchanges (seqnums (x)) in [0..1]
-> initial_subseq (unique_msg (x), unique_msg (y)) );
name proper_transmission, proper_reception, next_seqnum, last_bit, seqnums,
nchanges, unique_msg, repeats, initial_subseq, msg_lag
from alt_bit_specs;
name bit, message, packet, pkt_seq, msg_seq, bit_seq, comp, pkt_buf
from alt_bit_protocol;
end;
```

44444@vsysxxx
[Continuing]
translate abp.gyp
No syntax errors detected
No semantic errors detected
Exec-> translate abp.lem
No syntax errors detected
No semantic errors detected
Exec-> show status all
The current design and verification status is:
SCOPE ALT_BIT_PROTOCOL
Waiting for VC generation: AB_PROTOCOL, COMP, RECEIVER, SENDER
Waiting for pending body to be filled in: MEDIUM, TIMER
Types, constants: BIT, BIT_SEQ, CLK_BUF, MESSAGE, MSG_BUF, MSG_SEQ, PACKET,
PKT_BUF, PKT_SEQ
SCOPE ALT_BIT_SPECS
Waiting for pending body to be filled in: INITIAL_SUBSEQ
For specifications only: LAST_BIT, MSG_LAG, NCHANGES, NEXT_SEQNUM,
PROPER_RECEPTION, PROPER_TRANSMISSION, REPEATS, SEQNUMS, UNIQUE_MSG
SCOPE LEMMAS
Waiting for VC generation: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4,
PROP_REC_5, PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4,
PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7, SUB_APP, SIZE_NULL,
SUB_SEQNUM, SUB_TO_LAG, SUB_NCHANGES
Exec-> set scope alt_bit_protocol
Exec-> vcs sender
Generating VCs for procedure sender
Found 1st path
Found 2nd path
Found 3rd path
Found 4th path
Found 5th path
Note: loop has no exit paths
Found 6th path

Beginning new path...
Assume (unit entry specification)
true

initializing local variables
entering loop...
Evaluating next_seqnum (infrom (ack_send, myid))
Continuing in path...
Evaluating next_seqnum (outto (pkt_send, myid))
Continuing in path...
Evaluating proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Continuing in path...
Assert proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
& next = next_seqnum (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid))
- -
Must verify Assert condition
Verification condition sender#4
1: next_seqnum (null (#seqtype#)) = one
2: proper_transmission (null (#seqtype#), null (#seqtype#), null (#seqtype#),
0)
- -
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
& next = next_seqnum (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid))
Receive pack.mssg from source
- -
Must verify (Receive blocked) condition
Verification condition sender#5
H1: empty (source#1)
H2: next_seqnum (infrom (ack_send, myid)) = next
H3: next_seqnum (outto (pkt_send, myid)) = next
H4: infrom (source, myid) = infrom (source#1, myid)
H5: outto (source, myid) = outto (source#1, myid)
H6: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
-->

C1: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
- -
Continuing in path...
binding due to receive..
pack := pack with (.mssg:=pack#1.mssg)
pack := pack with (.seqno:=next)
Send pack to pkt_send
- -
Must verify (Send blocked) condition
Verification condition sender#6
H1: infrom (source, myid) [seq: pack#1.mssg] = infrom (source#2, myid)
H2: next_seqnum (infrom (ack_send, myid)) = next
H3: next_seqnum (outto (pkt_send, myid)) = next
H4: infrom (pkt_send, myid) = infrom (pkt_send#1, myid)
H5: outto (pkt_send, myid) = outto (pkt_send#1, myid)
H6: outto (source, myid) = outto (source#2, myid)
H7: full (pkt_send#1)
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
-->
C1: proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid), infrom (ack_send, myid), 1)
- -
Continuing in path...
Send true to start
- -
Must verify (Send blocked) condition
Verification condition sender#7
H1: infrom (source, myid) [seq: pack#1.mssg] = infrom (source#2, myid)
H2: outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)]
= outto (pkt_send#2, myid)
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: next_seqnum (outto (pkt_send, myid)) = next
H5: infrom (pkt_send, myid) = infrom (pkt_send#2, myid)
H6: infrom (start, myid) = infrom (start#1, myid)
H7: outto (source, myid) = outto (source#2, myid)
H8: outto (start, myid) = outto (start#1, myid)
H9: full (start#1)
H10: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)

-->
C1: proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)],
infrom (ack_send, myid), 1)
- -
Continuing in path...
entering loop...
Evaluating next_seqnum (infrom (ack_send, myid))
Continuing in path...
Evaluating unique_msg (infrom (ack_send, myid))
Continuing in path...
Evaluating unique_msg (outto (pkt_send, myid))
Continuing in path...
Evaluating unique_msg (outto (pkt_send, myid))
Continuing in path...
Evaluating proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Continuing in path...
Assert proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
- -
Must verify Assert condition
Verification condition sender#13
H1: infrom (source, myid) [seq: pack#1.mssg] = infrom (source#2, myid)
H2: outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)]
= outto (pkt_send#2, myid)
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: next_seqnum (outto (pkt_send, myid)) = next
H5: infrom (pkt_send, myid) = infrom (pkt_send#2, myid)
H6: outto (source, myid) = outto (source#2, myid)
H7: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
-->

C1: infrom (source, myid) [seq: pack#1.mssg]
= unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)])
C2: size (unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)]))
= size (unique_msg (infrom (ack_send, myid))) + 1
C3: proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)],
infrom (ack_send, myid), 1)
- -
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
Assume (Await blocked)
empty (tick) & empty (ack_send)
Blockage Assertion is
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))

= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
Receive ack from ack_send
binding due to receive..
ack := ack#1
Assume (If test failed)
not ack.seqno = next
entering next iteration of loop...
Evaluating next_seqnum (infrom (ack_send, myid))
Continuing in path...
Evaluating unique_msg (infrom (ack_send, myid))
Continuing in path...
Evaluating unique_msg (outto (pkt_send, myid))
Continuing in path...
Evaluating unique_msg (outto (pkt_send, myid))
Continuing in path...
Evaluating proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Continuing in path...
Assert proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
- -
Must verify Assert condition
Verification condition sender#20
H1: infrom (ack_send, myid) [seq: ack#1] = infrom (ack_send#3, myid)
H2: pack.seqno = next
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: outto (ack_send, myid) = outto (ack_send#3, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)

H9: ack#1.seqno ne next
H10: null (pkt_seq) ne outto (pkt_send, myid)
-->
C1: next_seqnum (infrom (ack_send, myid) [seq: ack#1]) = next
C2: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid) [seq: ack#1])) + 1
C3: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 1)
- -
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
Receive b from tick
binding due to receive..
b := b#1
Send pack to pkt_send
Continuing in path...
Send true to start
- -
Must verify (Send blocked) condition
Verification condition sender#22
H1: outto (pkt_send, myid) [seq: pack] = outto (pkt_send#4, myid)
H2: pack.seqno = next
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: infrom (pkt_send, myid) = infrom (pkt_send#4, myid)
H6: infrom (start, myid) = infrom (start#3, myid)
H7: outto (start, myid) = outto (start#3, myid)
H8: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H9: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1

H10: full (start#3)
H11: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H12: null (pkt_seq) ne outto (pkt_send, myid)
-->
C1: proper_transmission (infrom (source, myid),
outto (pkt_send, myid) [seq: pack],
infrom (ack_send, myid), 1)
- -
Continuing in path...
entering next iteration of loop...
Evaluating next_seqnum (infrom (ack_send, myid))
Continuing in path...
Evaluating unique_msg (infrom (ack_send, myid))
Continuing in path...
Evaluating unique_msg (outto (pkt_send, myid))
Continuing in path...
Evaluating unique_msg (outto (pkt_send, myid))
Continuing in path...
Evaluating proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Continuing in path...
Assert proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
- -
Must verify Assert condition
Verification condition sender#28
H1: outto (pkt_send, myid) [seq: pack] = outto (pkt_send#4, myid)
H2: pack.seqno = next
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: infrom (pkt_send, myid) = infrom (pkt_send#4, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),

infrom (ack_send, myid), 1)
H9: null (pkt_seq) ne outto (pkt_send, myid)
-->
C1: unique_msg (outto (pkt_send, myid) [seq: pack])
= infrom (source, myid)
C2: size (unique_msg (outto (pkt_send, myid) [seq: pack]))
= size (unique_msg (infrom (ack_send, myid))) + 1
C3: proper_transmission (infrom (source, myid),
outto (pkt_send, myid) [seq: pack],
infrom (ack_send, myid), 1)
- -
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& infrom (source, myid) = unique_msg (outto (pkt_send, myid))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& outto (pkt_send, myid) ne null (pkt_seq)
& pack = last (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid)) & pack.seqno = next
Receive ack from ack_send
binding due to receive..
ack := ack#1
Assume (If test succeeded)
ack.seqno = next
Leaving loop...
Evaluating comp (next)
Continuing in path...
next := comp (next)
entering next iteration of loop...
Evaluating next_seqnum (infrom (ack_send, myid))
Continuing in path...
Evaluating next_seqnum (outto (pkt_send, myid))
Continuing in path...
Evaluating proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Continuing in path...

Assert proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
& next = next_seqnum (outto (pkt_send, myid))
& next = next_seqnum (infrom (ack_send, myid))
- -
Must verify Assert condition
Verification condition sender#33
H1: infrom (ack_send, myid) [seq: ack#1] = infrom (ack_send#3, myid)
H2: ack#1.seqno = next
H3: pack.seqno = next
H4: next_seqnum (infrom (ack_send, myid)) = next
H5: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H6: outto (ack_send, myid) = outto (ack_send#3, myid)
H7: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H8: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H9: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H10: null (pkt_seq) ne outto (pkt_send, myid)
-->
C1: comp (next)
= next_seqnum (infrom (ack_send, myid) [seq: ack#1])
C2: comp (next) = next_seqnum (outto (pkt_send, myid))
C3: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 0)
- -
End of path

Exec-> vcs receiver
Generating VCs for procedure receiver
Found 1st path
Note: loop has no exit paths
Found 2nd path
Found 3rd path

Beginning new path...
Assume (unit entry specification)
true
initializing local variables
entering loop...
Evaluating next_seqnum (outto (ack_rcv, myid))
Continuing in path...

Evaluating next_seqnum (infrom (pkt_rcv, myid))
Continuing in path...
Evaluating proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Continuing in path...
Assert proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& exp = next_seqnum (infrom (pkt_rcv, myid))
& exp = next_seqnum (outto (ack_rcv, myid))
- -
Must verify Assert condition
Verification condition receiver#4
1: next_seqnum (null (#seqtype#)) = one
2: proper_reception (null (#seqtype#), null (#seqtype#), null (#seqtype#),
0)
- -
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& exp = next_seqnum (infrom (pkt_rcv, myid))
& exp = next_seqnum (outto (ack_rcv, myid))
Receive pack from pkt_rcv
- -
Must verify (Receive blocked) condition
Verification condition receiver#5
H1: empty (pkt_rcv#1)
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: infrom (pkt_rcv, myid) = infrom (pkt_rcv#1, myid)
H5: outto (pkt_rcv, myid) = outto (pkt_rcv#1, myid)
H6: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
-->
C1: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 1)
- -
Continuing in path...

binding due to receive..
pack := pack#1
Assume (If test succeeded)
exp = pack.seqno
Send pack.mssg to sink
- -
Must verify (Send blocked) condition
Verification condition receiver#6
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: pack#1.seqno = exp
H3: next_seqnum (infrom (pkt_rcv, myid)) = exp
H4: next_seqnum (outto (ack_rcv, myid)) = exp
H5: infrom (sink, myid) = infrom (sink#1, myid)
H6: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H7: outto (sink, myid) = outto (sink#1, myid)
H8: full (sink#1)
H9: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
-->
C1: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid), 1)
- -
Continuing in path...
Evaluating comp (exp)
Continuing in path...
exp := comp (exp)
Send pack to ack_rcv
- -
Must verify (Send blocked) condition
Verification condition receiver#8
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: outto (sink, myid) [seq: pack#1.mssg] = outto (sink#2, myid)
H3: pack#1.seqno = exp
H4: next_seqnum (infrom (pkt_rcv, myid)) = exp
H5: next_seqnum (outto (ack_rcv, myid)) = exp
H6: infrom (ack_rcv, myid) = infrom (ack_rcv#1, myid)
H7: infrom (sink, myid) = infrom (sink#2, myid)
H8: outto (ack_rcv, myid) = outto (ack_rcv#1, myid)
H9: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H10: full (ack_rcv#1)
H11: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),

outto (ack_rcv, myid), 0)
-->
C1: proper_reception (outto (sink, myid) [seq: pack#1.mssg],
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid), 1)
- -
Continuing in path...
entering next iteration of loop...
Evaluating next_seqnum (outto (ack_rcv, myid))
Continuing in path...
Evaluating next_seqnum (infrom (pkt_rcv, myid))
Continuing in path...
Evaluating proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Continuing in path...
Assert proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& exp = next_seqnum (infrom (pkt_rcv, myid))
& exp = next_seqnum (outto (ack_rcv, myid))
- -
Must verify Assert condition
Verification condition receiver#12
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: outto (ack_rcv, myid) [seq: pack#1] = outto (ack_rcv#2, myid)
H3: outto (sink, myid) [seq: pack#1.mssg] = outto (sink#2, myid)
H4: pack#1.seqno = exp
H5: next_seqnum (infrom (pkt_rcv, myid)) = exp
H6: next_seqnum (outto (ack_rcv, myid)) = exp
H7: infrom (ack_rcv, myid) = infrom (ack_rcv#2, myid)
H8: infrom (sink, myid) = infrom (sink#2, myid)
H9: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H10: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
-->
C1: comp (exp)
= next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1])
C2: comp (exp) = next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
C3: proper_reception (outto (sink, myid) [seq: pack#1.mssg],
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
- -
End of path

Beginning new path...
continuing in loop ...
Assume (from last assertion)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& exp = next_seqnum (infrom (pkt_rcv, myid))
& exp = next_seqnum (outto (ack_rcv, myid))
Receive pack from pkt_rcv
- -
Must verify (Receive blocked) condition
Verification condition receiver#13
H1: empty (pkt_rcv#1)
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: infrom (pkt_rcv, myid) = infrom (pkt_rcv#1, myid)
H5: outto (pkt_rcv, myid) = outto (pkt_rcv#1, myid)
H6: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
-->
C1: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 1)
- -
Continuing in path...
binding due to receive..
pack := pack#1
Assume (If test failed)
not exp = pack.seqno
Send pack to ack_rcv
- -
Must verify (Send blocked) condition
Verification condition receiver#14
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: infrom (ack_rcv, myid) = infrom (ack_rcv#3, myid)
H5: outto (ack_rcv, myid) = outto (ack_rcv#3, myid)
H6: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H7: full (ack_rcv#3)
H8: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)

H9: pack#1.seqno ne exp
-->
C1: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid), 1)
- -
Continuing in path...
entering next iteration of loop...
Evaluating next_seqnum (outto (ack_rcv, myid))
Continuing in path...
Evaluating next_seqnum (infrom (pkt_rcv, myid))
Continuing in path...
Evaluating proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Continuing in path...
Assert proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& exp = next_seqnum (infrom (pkt_rcv, myid))
& exp = next_seqnum (outto (ack_rcv, myid))
- -
Must verify Assert condition
Verification condition receiver#18
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: outto (ack_rcv, myid) [seq: pack#1] = outto (ack_rcv#4, myid)
H3: next_seqnum (infrom (pkt_rcv, myid)) = exp
H4: next_seqnum (outto (ack_rcv, myid)) = exp
H5: infrom (ack_rcv, myid) = infrom (ack_rcv#4, myid)
H6: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H7: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
H8: pack#1.seqno ne exp
-->
C1: next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1]) = exp
C2: next_seqnum (outto (ack_rcv, myid) [seq: pack#1]) = exp
C3: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
- -
End of path

Exec-> vcs ab_protocol
Generating VCs for procedure ab_protocol

Found 1st path
Found 2nd path

Beginning new path...
Assume (unit entry specification)
true
initializing local variables
entering cobegin...
timer (clock_in, clock_out)
receiver (sink, pkt_rcv, ack_rcv)
medium (ack_rcv, ack_send)
medium (pkt_send, pkt_rcv)
sender (source, pkt_send, ack_send, clock_in, clock_out)
Continuing in path...
assume cobegin blocks.
Assume (blockage specification for subprocess sender#1)
if isblocked (sender#1)
then proper_transmission (infrom (source#1, sender#1),
outto (pkt_send#1, sender#1),
infrom (ack_send#1, sender#1), 1)
else false
fi
Assume (blockage specification for subprocess medium#1)
if isblocked (medium#1)
then outto (pkt_rcv#1, medium#1) sub infrom (pkt_send#1, medium#1)
else false
fi
Assume (blockage specification for subprocess medium#2)
if isblocked (medium#2)
then outto (ack_send#1, medium#2) sub infrom (ack_rcv#1, medium#2)
else false
fi
Assume (blockage specification for subprocess receiver#1)
if isblocked (receiver#1)
then proper_reception (outto (sink#1, receiver#1),
infrom (pkt_rcv#1, receiver#1),
outto (ack_rcv#1, receiver#1), 1)
else false
fi
Assume (blockage specification for subprocess timer#1)
if isblocked (timer#1) then true else false fi
Assume (blockage specification for unit ab_protocol)

isblocked (sender#1) or isblocked (medium#1) or isblocked (medium#2)
or isblocked (receiver#1) or isblocked (timer#1)
Blockage Assertion is
msg_lag (outto (sink, myid), infrom (source, myid), 1)
- -
Must verify (process blockage) condition
Verification condition ab_protocol#2
H1: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H2: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)
H3: isblocked (medium#1)
H4: isblocked (medium#2)
H5: isblocked (receiver#1)
H6: isblocked (sender#1)
H7: isblocked (timer#1)
H8: allto (ack_send#1) sub allfrom (ack_rcv#1)
H9: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
-->
C1: msg_lag (outto (sink#1, receiver#1), infrom (source#1, sender#1), 1)
- -
End of path

Beginning new path...
Assume (unit entry specification)
true
initializing local variables
entering cobegin...
timer (clock_in, clock_out)
receiver (sink, pkt_rcv, ack_rcv)
medium (ack_rcv, ack_send)
medium (pkt_send, pkt_rcv)
sender (source, pkt_send, ack_send, clock_in, clock_out)
Continuing in path...
Assume Procedure Exit Specifications
false & false & false & false & false
assume all procedure activations terminate normally.
Leaving Unit ab_protocol
Assert false
End of path

Exec-> vcs comp
Generating VCs for function comp
Found 1st path

Beginning new path...
Assume (unit entry specification)
true
result := if b = zero then one else zero fi
Leaving Unit comp
Assert result = if b = zero then one else zero fi
End of path

Exec-> show status all
The current design and verification status is:
SCOPE ALT_BIT_PROTOCOL
Waiting for pending body to be filled in: MEDIUM, TIMER
Proved: COMP
Types, constants: BIT, BIT_SEQ, CLK_BUF, MESSAGE, MSG_BUF, MSG_SEQ, PACKET,
PKT_BUF, PKT_SEQ
Check VCs (not fully checked): AB_PROTOCOL, RECEIVER, SENDER
Check VCs (path pending): AB_PROTOCOL, RECEIVER, SENDER
ab_protocol
Waiting to be proved: AB_PROTOCOL#2
Proved in VC generator: AB_PROTOCOL#1, AB_PROTOCOL#3, AB_PROTOCOL#4
receiver
Waiting to be proved: RECEIVER#4, RECEIVER#5, RECEIVER#6, RECEIVER#8,
RECEIVER#12, RECEIVER#13, RECEIVER#14, RECEIVER#18
Proved in VC generator: RECEIVER#1, RECEIVER#2, RECEIVER#3, RECEIVER#7,
RECEIVER#9, RECEIVER#10, RECEIVER#11, RECEIVER#15, RECEIVER#16,
RECEIVER#17
sender
Waiting to be proved: SENDER#4, SENDER#5, SENDER#6, SENDER#7, SENDER#13,
SENDER#20, SENDER#22, SENDER#28, SENDER#33
Proved in VC generator: SENDER#1, SENDER#2, SENDER#3, SENDER#8,
SENDER#9, SENDER#10, SENDER#11, SENDER#12, SENDER#14, SENDER#15,
SENDER#16, SENDER#17, SENDER#18, SENDER#19, SENDER#21, SENDER#23,
SENDER#24, SENDER#25, SENDER#26, SENDER#27, SENDER#29, SENDER#30,
SENDER#31, SENDER#32
SCOPE ALT_BIT_SPECS
Waiting for pending body to be filled in: INITIAL_SUBSEQ
For specifications only: LAST_BIT, MSG_LAG, NCHANGES, NEXT_SEQNUM,
PROPER_RECEPTION, PROPER_TRANSMISSION, REPEATS, SEQNUMS, UNIQUE_MSG

SCOPE LEMMAS
Waiting for VC generation: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4,
PROP_REC_5, PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4,
PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7, SUB_APP, SIZE_NULL,
SUB_SEQNUM, SUB_TO_LAG, SUB_NCHANGES
Exec-> save abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
[PHOTO: Logging disabled Tue 5-May-81 9:41AM]
[PHOTO: Recording terminated Tue 5-May-81 9:43AM]
[PHOTO: Recording initiated Thu 7-May-81 7:52AM]
LINK FROM CMP.DIVITO, TTY 20
TOPS-20 Command processor 4(560)
[PHOTO: Logging disabled Thu 7-May-81 7:52AM]
PHOTO: Logging enabled Thu 7-May-81 7:55AM]
% We are now ready to do the proofs. We will first knock off the VCs
% and then take on the various lemmas. These will go in roughly a
% top down order. The lemmas which get proved in the Affirm system
% will simply be assumed here.
show status all
The current design and verification status is:
SCOPE ALT_BIT_PROTOCOL
Waiting for pending body to be filled in: MEDIUM, TIMER
Proved: COMP
Types, constants: BIT, BIT_SEQ, CLK_BUF, MESSAGE, MSG_BUF, MSG_SEQ, PACKET,
PKT_BUF, PKT_SEQ
Check VCs (not fully checked): AB_PROTOCOL, RECEIVER, SENDER
Check VCs (path pending): AB_PROTOCOL, RECEIVER, SENDER
ab_protocol
Waiting to be proved: AB_PROTOCOL#2
Proved in VC generator: AB_PROTOCOL#1, AB_PROTOCOL#3, AB_PROTOCOL#4
receiver
Waiting to be proved: RECEIVER#4, RECEIVER#5, RECEIVER#6, RECEIVER#8,
RECEIVER#12, RECEIVER#13, RECEIVER#14, RECEIVER#18
Proved in VC generator: RECEIVER#1, RECEIVER#2, RECEIVER#3, RECEIVER#7,

RECEIVER#9, RECEIVER#10, RECEIVER#11, RECEIVER#15, RECEIVER#16,
RECEIVER#17
sender
Waiting to be proved: SENDER#4, SENDER#5, SENDER#6, SENDER#7, SENDER#13,
SENDER#20, SENDER#22, SENDER#28, SENDER#33
Proved in VC generator: SENDER#1, SENDER#2, SENDER#3, SENDER#8,
SENDER#9, SENDER#10, SENDER#11, SENDER#12, SENDER#14, SENDER#15,
SENDER#16, SENDER#17, SENDER#18, SENDER#19, SENDER#21, SENDER#23,
SENDER#24, SENDER#25, SENDER#26, SENDER#27, SENDER#29, SENDER#30,
SENDER#31, SENDER#32
SCOPE ALT_BIT_SPECS
Waiting for pending body to be filled in: INITIAL_SUBSEQ
For specifications only: LAST_BIT, MSG_LAG, NCHANGES, NEXT_SEQNUM,
PROPER_RECEPTION, PROPER_TRANSMISSION, REPEATS, SEQNUMS, UNIQUE_MSG
SCOPE LEMMAS
Waiting to be proved: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4,
PROP_REC_5, PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4,
PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7, SUB_APP, SIZE_NULL,
SUB_SEQNUM, SUB_TO_LAG, SUB_NCHANGES
Exec-> prove sender#4
Entering Prover with verification condition sender#4
C1: next_seqnum (null (#seqtype#)) = one
C2: proper_transmission (null (#seqtype#), null (#seqtype#),
null (#seqtype#), 0)
Backup point
(.)
Prvr-> $Proceeding
Backup point
(. 1 .)
Prvr-> expand next_seqnum
Backup point
(. 1 . E .)
Prvr-> theorem
C1: comp (last_bit (seqnums (null (#seqtype#)))) = one
Prvr-> expand last_bit
Backup point
(. 1 . E . E .)
Prvr-> expand seqnums

Backup point
(. 1 . E . E . E .)
Prvr-> theorem
C1: comp (if null (bit_seq) = null (bit_seq)
then zero
else null (bit_seq)[size (null (bit_seq))]
fi)
= one
Prvr-> simplify theorem
Prvr-> expand comp
Backup point
(. 1 . E . E . E . E .)
Prvr-> $Proceeding
(. 1 .)
next_seqnum (null (#seqtype#)) = one
Proved
Prvr-> $Proceeding
Backup point
(. 2 .)
Prvr-> theorem
C1: proper_transmission (null (#seqtype#), null (#seqtype#),
null (#seqtype#), 0)
Prvr-> expand proper_transmission
Backup point
(. 2 . E .)
Prvr->theorem
C1: 0 = size (unique_msg (null (#seqtype#)))
C2: msg_lag (unique_msg (null (#seqtype#)), null (#seqtype#), 0)
C3: repeats (null (#seqtype#))
Prvr-> expand unique_msg
Backup point
(. 2 . E . E .)
Prvr-> expand repeats
Backup point
(. 2 . E . E . E .)
Prvr-> simplify theorem
Prvr->theorem
C1: msg_lag (null (msg_seq), null (#seqtype#), 0)
Prvr-> expand msg_lag
Backup point
(. 2 . E . E . E . E .)
Prvr-> theorem

C1: initial_subseq (null (msg_seq), null (#seqtype#))
Prvr-> use eq_iss
**** EQ_ISS is not known in scope ALT_BIT_PROTOCOL
Illegal argument to USE
Prvr-> use eq_iss::lemmas
Backup point
(. 2 . E . E . E . E . U .)
Prvr-> qed
(. 2 . E . E . E . E . U . QED)
(. 2 . E . E . E . E . U . QED BC)
(. 2 . E . E . E . E . U . QED BC)
null (msg_seq) = null (#seqtype#)
Proved
QED
Prvr-> $Proceeding
(. 2 .)
proper_transmission (null (#seqtype#), null (#seqtype#), null (#seqtype#),
0)
Proved
Prvr-> $Proceeding
sender#4
proved in theorem prover.
Exec-> prove sender#5
Entering Prover with verification condition sender#5
H1: empty (source#1)
H2: next_seqnum (infrom (ack_send, myid)) = next
H3: next_seqnum (outto (pkt_send, myid)) = next
H4: infrom (source, myid) = infrom (source#1, myid)
H5: outto (source, myid) = outto (source#1, myid)
H6: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Backup point
(.)
Prvr-> retain h6
Backup point
(. D .)
Prvr-> use prop_trans_6::lemmas
Backup point
(. D . U .)

```
Prvr-> qed
(. D . U . QED)
::.(. D . U . QED BC)
(. D . U . QED BC)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Proved
QED
Prvr-> $Proceeding
sender#5
proved in theorem prover.
Exec-> prove sender#6
Entering Prover with verification condition sender#6
H1: infrom (source, myid)  [seq: pack#1.mssg] = infrom (source#2, myid)
H2: next_seqnum (infrom (ack_send, myid)) = next
H3: next_seqnum (outto (pkt_send, myid)) = next
H4: infrom (pkt_send, myid) = infrom (pkt_send#1, myid)
H5: outto (pkt_send, myid) = outto (pkt_send#1, myid)
H6: outto (source, myid) = outto (source#2, myid)
H7: full (pkt_send#1)
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: proper_transmission (infrom (source, myid)  [seq: pack#1.mssg],
outto (pkt_send, myid), infrom (ack_send, myid), 1)
Backup point
(.)
Prvr-> retain h8
Backup point
(. D .)
Prvr-> use prop_trans_5::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
::.(. D . U . QED BC)
(. D . U . QED BC)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Proved
QED
Prvr-> $Proceeding
```

sender#6
proved in theorem prover.
Exec-> prove sender#7
Entering Prover with verification condition sender#7
H1: infrom (source, myid) [seq: pack#1.mssg] = infrom (source#2, myid)
H2: outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)]
= outto (pkt_send#2, myid)
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: next_seqnum (outto (pkt_send, myid)) = next
H5: infrom (pkt_send, myid) = infrom (pkt_send#2, myid)
H6: infrom (start, myid) = infrom (start#1, myid)
H7: outto (source, myid) = outto (source#2, myid)
H8: outto (start, myid) = outto (start#1, myid)
H9: full (start#1)
H10: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)],
infrom (ack_send, myid), 1)
Backup point
(.)
Prvr-> retain h4,h10
Backup point
(. D .)
Prvr-> use prop_trans_2::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
:::.(. D . U . QED BC)
(. D . U . QED BC 1)
(. D . U . QED BC 1)
pack with (.mssg:=pack#1.mssg; .seqno:=next).mssg = pack#1.mssg
Proved
(. D . U . QED BC 2)
(. D . U . QED BC 2)
pack with (.mssg:=pack#1.mssg; .seqno:=next).seqno
= next_seqnum (outto (pkt_send, myid))

Proved
(. D . U . QED BC 3)
(. D . U . QED BC 3)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Proved
(. D . U . QED BC)
pack with (.mssg:=pack#1.mssg; .seqno:=next).mssg = pack#1.mssg
& pack with (.mssg:=pack#1.mssg; .seqno:=next).seqno
= next_seqnum (outto (pkt_send, myid))
& proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Proved
QED
Prvr-> $Proceeding
sender#7
proved in theorem prover.
Exec-> save
Enter file name-> abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec-> show status sender
Check VCs (not fully checked): SENDER
Check VCs (path pending): SENDER
sender
Waiting to be proved: SENDER#13, SENDER#20, SENDER#22, SENDER#28,
SENDER#33
Proved in VC generator: SENDER#1, SENDER#2, SENDER#3, SENDER#8,
SENDER#9, SENDER#10, SENDER#11, SENDER#12, SENDER#14, SENDER#15,
SENDER#16, SENDER#17, SENDER#18, SENDER#19, SENDER#21, SENDER#23,
SENDER#24, SENDER#25, SENDER#26, SENDER#27, SENDER#29, SENDER#30,
SENDER#31, SENDER#32
Proved in theorem prover: SENDER#4, SENDER#5, SENDER#6, SENDER#7
Exec-> prove sender#13
Entering Prover with verification condition sender#13
H1: infrom (source, myid) [seq: pack#1.mssg] = infrom (source#2, myid)
H2: outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)]
= outto (pkt_send#2, myid)

H3: next_seqnum (infrom (ack_send, myid)) = next
H4: next_seqnum (outto (pkt_send, myid)) = next
H5: infrom (pkt_send, myid) = infrom (pkt_send#2, myid)
H6: outto (source, myid) = outto (source#2, myid)
H7: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: infrom (source, myid) [seq: pack#1.mssg]
= unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)])
C2: size (unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)]))
= size (unique_msg (infrom (ack_send, myid))) + 1
C3: proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)],
infrom (ack_send, myid), 1)
Backup point
(.)
Prvr-> retain h4,h7
Backup point
(. D .)
Prvr-> $Proceeding
Backup point
(. D . 1 .)
Prvr-> theorem
H1: next_seqnum (outto (pkt_send, myid)) = next
H2: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: infrom (source, myid) [seq: pack#1.mssg]
= unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)])
Prvr-> use prop_trans_1::lemmas
Backup point
(. D . 1 . U .)
Prvr-> qed
(. D . 1 . U . QED)
:::Conclusion simplified to:
infrom (source, myid) [seq: pack#1.mssg]

= unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)]).(. D . 1 .
U . QED BC)
(. D . 1 . U . QED BC 1)
(. D . 1 . U . QED BC 1)
pack with (.mssg:=pack#1.mssg; .seqno:=next).mssg = pack#1.mssg
Proved
(. D . 1 . U . QED BC 2)
(. D . 1 . U . QED BC 2)
pack with (.mssg:=pack#1.mssg; .seqno:=next).seqno
= next_seqnum (outto (pkt_send, myid))
Proved
(. D . 1 . U . QED BC 3)
::(. D . 1 . U . QED BC 3)
proper_transmission (infrom (source, myid), outto (pkt_send, myid), y#8$, 0)
Proved
(. D . 1 . U . QED BC)
pack with (.mssg:=pack#1.mssg; .seqno:=next).mssg = pack#1.mssg
& pack with (.mssg:=pack#1.mssg; .seqno:=next).seqno
= next_seqnum (outto (pkt_send, myid))
& proper_transmission (infrom (source, myid), outto (pkt_send, myid), y#8$,
0)
Proved
QED
Prvr-> $Proceeding
(. D . 1 .)
infrom (source, myid) [seq: pack#1.mssg]
= unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)])
Proved
Prvr-> $Proceeding
Backup point
(. D . 2 .)
Prvr-> theorem
H1: next_seqnum (outto (pkt_send, myid)) = next
H2: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: size (unique_msg (outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)]))
= size (unique_msg (infrom (ack_send, myid))) + 1

```
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? show-choices
1: unique_msg ( outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)])
2: unique_msg (infrom (ack_send, myid))
Which do you want to expand? 1
Backup point
(. D . 2 . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: next_seqnum (outto (pkt_send, myid)) = next
H2: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1:  size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid)))
Prvr-> expand proper_transmission
Backup point
(. D . 2 . E . E .)
Prvr-> simplify hypothesis
Prvr-> theorem
H1: next_seqnum (outto (pkt_send, myid)) = next
H2:  size (unique_msg (infrom (ack_send, myid)))
= size (unique_msg (outto (pkt_send, myid)))
H3: size (unique_msg (infrom (ack_send, myid))) = size (infrom (source,
myid))
H4: msg_lag (unique_msg (outto (pkt_send, myid)), infrom (source, myid), 0)
H5: repeats (outto (pkt_send, myid))
->
C1:  size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid)))
Prvr-> equality substitute h2
H2 can be solved for:
T1: size (unique_msg (infrom (ack_send, myid)))
T2: size (unique_msg (outto (pkt_send, myid)))
Which term (by label) do you want to substitute for?
t1
size (unique_msg (infrom (ack_send, myid))) :=
size (unique_msg (outto (pkt_send, myid)))
OK??
y
```

Typelist equalities
size (infrom (source, myid)) = size (unique_msg (outto (pkt_send, myid)))
Backup point
(. D . 2 . E . E . =S .)
Prvr-> $Proceeding
(. D . 2 .)
size (unique_msg (outto (pkt_send, myid)
[seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)]))
= size (unique_msg (infrom (ack_send, myid))) + 1
Proved
Prvr-> $Proceeding
Backup point
(. D . 3 .)
Prvr-> theorem
H1: next_seqnum (outto (pkt_send, myid)) = next
H2: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
->
C1: proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid)
[seq: pack with (.mssg:=pack#1.mssg;
.seqno:=next)],
infrom (ack_send, myid), 1)
Prvr-> use prop_trans_2::lemmas
Backup point
(. D . 3 . U .)
Prvr-> qed
(. D . 3 . U . QED)
:::.(. D . 3 . U . QED BC)
(. D . 3 . U . QED BC 1)
(. D . 3 . U . QED BC 1)
pack with (.mssg:=pack#1.mssg; .seqno:=next).mssg = pack#1.mssg
Proved
(. D . 3 . U . QED BC 2)
(. D . 3 . U . QED BC 2)
pack with (.mssg:=pack#1.mssg; .seqno:=next).seqno
= next_seqnum (outto (pkt_send, myid))
Proved
(. D . 3 . U . QED BC 3)
(. D . 3 . U . QED BC 3)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)

Proved
(. D . 3 . U . QED BC)
pack with (.mssg:=pack#1.mssg; .seqno:=next).mssg = pack#1.mssg
& pack with (.mssg:=pack#1.mssg; .seqno:=next).seqno
= next_seqnum (outto (pkt_send, myid))
& proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 0)
Proved
QED
Prvr-> $Proceeding
(. D . 3 .)
proper_transmission (infrom (source, myid) [seq: pack#1.mssg],
outto (pkt_send, myid)
 [seq: pack with (.mssg:=pack#1.mssg; .seqno:=next)],
infrom (ack_send, myid), 1)
Proved
Prvr-> $Proceeding
sender#13
proved in theorem prover.
Exec-> prove sender#20
Entering Prover with verification condition sender#20
H1: infrom (ack_send, myid) [seq: ack#1] = infrom (ack_send#3, myid)
H2: pack.seqno = next
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: outto (ack_send, myid) = outto (ack_send#3, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H9: ack#1.seqno ne next
H10: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: next_seqnum (infrom (ack_send, myid) [seq: ack#1]) = next
C2: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid) [seq: ack#1])) + 1
C3: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 1)
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))

Backup point
(.)
Prvr-> retain h3,h7,h8,h9
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(. D .)
Prvr-> $Proceeding
Backup point
(. D . 1 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: next_seqnum (infrom (ack_send, myid)) = next
H3: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H4: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H5: ack#1.seqno ne next
->
C1: next_seqnum (infrom (ack_send, myid) [seq: ack#1]) = next
Prvr-> retain h2,h5
Backup point
(. D . 1 . D .)
Prvr-> use ne_next::lemmas
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
Backup point
(. D . 1 . D . U .)
Prvr-> hypothesis
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H3: p#8$.seqno ne next_seqnum (x#12$)
-> next_seqnum (x#12$ [seq: p#8$]) = next_seqnum (x#12$)
H4: next_seqnum (infrom (ack_send, myid)) = next
H5: ack#1.seqno ne next

```
Prvr-> reorder h4,h3,h5
Backup point
(. D . 1 . D . U . REORDER .)
Prvr-> qed
(. D . 1 . D . U . REORDER . QED)
:::::.
next := next_seqnum (infrom (ack_send, myid))
(. D . 1 . D . U . REORDER . QED =S)
::::.
size (unique_msg (infrom (ack_send, myid))) :=
size (unique_msg (outto (pkt_send, myid))) - 1
(. D . 1 . D . U . REORDER . QED =S =S)
::.(. D . 1 . D . U . REORDER . QED =S =S BC)
(. D . 1 . D . U . REORDER . QED =S =S BC)
ack#1.seqno ne next_seqnum (infrom (ack_send, myid))
Proved
QED
Prvr-> $Proceeding
(. D . 1 .)
next_seqnum (infrom (ack_send, myid)  [seq: ack#1]) = next
Proved
Prvr-> $Proceeding
Backup point
(. D . 2 .)
Prvr-> theorem
H1:  size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: next_seqnum (infrom (ack_send, myid)) = next
H3:  size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H4: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H5: ack#1.seqno ne next
->
C1:  size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid)  [seq: ack#1])) + 1
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? show-choices
1: unique_msg (infrom (ack_send, myid))
2: unique_msg (outto (pkt_send, myid))
3: unique_msg (infrom (ack_send, myid)  [seq: ack#1])
```

Which do you want to expand? 3
Backup point
(. D . 2 . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: next_seqnum (infrom (ack_send, myid)) = next
H2: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H3: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H4: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H5: ack#1.seqno ne next
->
C1: size (unique_msg (outto (pkt_send, myid)))
= size (if ack#1.seqno = next_seqnum (infrom (ack_send, myid))
then unique_msg (infrom (ack_send, myid))
 [seq: ack#1.mssg]
else unique_msg (infrom (ack_send, myid))
fi) + 1
Prvr-> equality substitute h1
H1 yields
next_seqnum (infrom (ack_send, myid)) := next
Backup point
(. D . 2 . E . =S .)
Prvr-> simplify theorem
Prvr->theorem
C1: true
Prvr-> $Proceeding
(. D . 2 .)
size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid) [seq: ack#1])) + 1
Proved
Prvr-> $Proceeding
Backup point
(. D . 3 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: next_seqnum (infrom (ack_send, myid)) = next
H3: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1

H4: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H5: ack#1.seqno ne next
->
C1: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 1)
Prvr-> use prop_trans_4::lemmas
Backup point
(. D . 3 . U .)
Prvr-> qed
(. D . 3 . U . QED)
:::::::(. D . 3 . U . QED BC)
(. D . 3 . U . QED BC 1)
(. D . 3 . U . QED BC 1)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Proved
(. D . 3 . U . QED BC 2)
::::::.
size (unique_msg (infrom (ack_send, myid))) :=
size (unique_msg (outto (pkt_send, myid))) - 1
(. D . 3 . U . QED BC 2 =S)
:::.
next_seqnum (infrom (ack_send, myid)) := next
(. D . 3 . U . QED BC 2 =S =S)
(. D . 3 . U . QED BC 2)
ack#1.seqno ne next_seqnum (infrom (ack_send, myid))
Proved
(. D . 3 . U . QED BC)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& ack#1.seqno ne next_seqnum (infrom (ack_send, myid))
Proved
QED
Prvr-> $Proceeding
(. D . 3 .)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 1)
Proved
Prvr-> $Proceeding
sender#20
proved in theorem prover.

Exec->
save
Enter file name-> abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec-> prove sender#22
Entering Prover with verification condition sender#22
H1: outto (pkt_send, myid) [seq: pack] = outto (pkt_send#4, myid)
H2: pack.seqno = next
H3: next_seqnum (infrom (ack_send, myid)) = next
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: infrom (pkt_send, myid) = infrom (pkt_send#4, myid)
H6: infrom (start, myid) = infrom (start#3, myid)
H7: outto (start, myid) = outto (start#3, myid)
H8: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H9: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H10: full (start#3)
H11: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H12: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: proper_transmission (infrom (source, myid),
outto (pkt_send, myid) [seq: pack],
infrom (ack_send, myid), 1)
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(.)
Prvr-> retain h8,h11,h12
Backup point
(. D .)
Prvr-> use prop_trans_3::lemmas
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1

Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
:::::::.
size (unique_msg (infrom (ack_send, myid))) :=
size (unique_msg (outto (pkt_send, myid))) - 1
(. D . U . QED =S)
::::.
pack := outto (pkt_send, myid)[size (outto (pkt_send, myid))]
(. D . U . QED =S =S)
:::.....::...(. D . U . QED =S =S BC)
(. D . U . QED =S =S BC 1)
(. D . U . QED =S =S BC 1)
outto (pkt_send, myid)[size (outto (pkt_send, myid))]
= outto (pkt_send, myid)[size (outto (pkt_send, myid))]
Proved
(. D . U . QED =S =S BC 2)
(. D . U . QED =S =S BC 2)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Proved
(. D . U . QED =S =S BC 3)
(. D . U . QED =S =S BC 3)
null (pkt_seq) ne outto (pkt_send, myid)
Proved
(. D . U . QED =S =S BC)
outto (pkt_send, myid)[size (outto (pkt_send, myid))]
= outto (pkt_send, myid)[size (outto (pkt_send, myid))]
& proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& null (pkt_seq) ne outto (pkt_send, myid)
Proved
QED
Prvr-> $Proceeding
sender#22
proved in theorem prover.
Exec-> prove sender#28
Entering Prover with verification condition sender#28
H1: outto (pkt_send, myid) [seq: pack] = outto (pkt_send#4, myid)
H2: pack.seqno = next
H3: next_seqnum (infrom (ack_send, myid)) = next

H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: infrom (pkt_send, myid) = infrom (pkt_send#4, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H9: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: unique_msg (outto (pkt_send, myid) [seq: pack])
= infrom (source, myid)
C2: size (unique_msg (outto (pkt_send, myid) [seq: pack]))
= size (unique_msg (infrom (ack_send, myid))) + 1
C3: proper_transmission (infrom (source, myid),
outto (pkt_send, myid) [seq: pack],
infrom (ack_send, myid), 1)
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(.)
Prvr-> retain h4,h6,h7,h8,h9
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(. D .)
Prvr-> $Proceeding
Backup point
(. D . 1 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H3: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H4: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H5: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H6: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: unique_msg (outto (pkt_send, myid) [seq: pack])

= infrom (source, myid)
Prvr-> retain h2,h3,h6
Backup point
(. D . 1 . D .)
Prvr-> use last_unique::lemmas
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
Backup point
(. D . 1 . D . U .)
Prvr-> hypothesis
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H3: x#18$[size (x#18$)] = p#14$ & null (pkt_seq) ne x#18$
-> unique_msg (x#18$ [seq: p#14$]) = unique_msg (x#18$)
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H6: null (pkt_seq) ne outto (pkt_send, myid)
Prvr-> delete h1,h2
Backup point
(. D . 1 . D . U . D .)
Prvr-> qed
(. D . 1 . D . U . D . QED)
:::::..
infrom (source, myid) := unique_msg (outto (pkt_send, myid))
(. D . 1 . D . U . D . QED =S)
:::.
pack := outto (pkt_send, myid)[size (outto (pkt_send, myid))]
(. D . 1 . D . U . D . QED =S =S)
::.(. D . 1 . D . U . D . QED =S =S BC)
(. D . 1 . D . U . D . QED =S =S BC 1)
(. D . 1 . D . U . D . QED =S =S BC 1)
outto (pkt_send, myid)[size (outto (pkt_send, myid))]
= outto (pkt_send, myid)[size (outto (pkt_send, myid))]
Proved
(. D . 1 . D . U . D . QED =S =S BC 2)
(. D . 1 . D . U . D . QED =S =S BC 2)
null (pkt_seq) ne outto (pkt_send, myid)

Proved
(. D . 1 . D . U . D . QED =S =S BC)
outto (pkt_send, myid)[size (outto (pkt_send, myid))]
= outto (pkt_send, myid)[size (outto (pkt_send, myid))]
& null (pkt_seq) ne outto (pkt_send, myid)
Proved
QED
Prvr-> $Proceeding
(. D . 1 .)
unique_msg (outto (pkt_send, myid) [seq: pack]) = infrom (source, myid)
Proved
Prvr-> $Proceeding
Backup point
(. D . 2 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H3: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H4: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H5: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H6: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: size (unique_msg (outto (pkt_send, myid) [seq: pack]))
= size (unique_msg (infrom (ack_send, myid))) + 1
Prvr-> delete h4,h5
Typelist equalities
size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
Backup point
(. D . 2 . D .)
Prvr-> use last_unique::lemmas
Backup point
(. D . 2 . D . U .)
Prvr-> hypothesis
H1: x#20$[size (x#20$)] = p#16$ & null (pkt_seq) ne x#20$
-> unique_msg (x#20$ [seq: p#16$]) = unique_msg (x#20$)
H2: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H3: size (unique_msg (infrom (ack_send, myid))) + 1

= size (unique_msg (outto (pkt_send, myid)))
H4: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H5: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H6: null (pkt_seq) ne outto (pkt_send, myid)
Prvr-> forwardchain h1
:::::Forward chaining gives
unique_msg (outto (pkt_send, myid) [seq: pack])
= unique_msg (outto (pkt_send, myid))
Backup point
(. D . 2 . D . U . FC .)
Prvr-> theorem
H1: unique_msg (outto (pkt_send, myid) [seq: pack])
= unique_msg (outto (pkt_send, myid))
H2: x#20$[size (x#20$)] = p#16$ & null (pkt_seq) ne x#20$
-> unique_msg (x#20$ [seq: p#16$]) = unique_msg (x#20$)
H3: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H4: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H5: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: size (unique_msg (outto (pkt_send, myid) [seq: pack]))
= size (unique_msg (infrom (ack_send, myid))) + 1
Prvr-> equality substitute h1
H1 yields
unique_msg (outto (pkt_send, myid) [seq: pack]) :=
unique_msg (outto (pkt_send, myid))
Backup point
(. D . 2 . D . U . FC . =S .)
Prvr-> $Proceeding
(. D . 2 .)
size (unique_msg (outto (pkt_send, myid) [seq: pack]))
= size (unique_msg (infrom (ack_send, myid))) + 1
Proved
Prvr-> $Proceeding
Backup point
(. D . 3 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))

H2: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H3: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H4: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H5: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H6: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: proper_transmission (infrom (source, myid),
outto (pkt_send, myid) [seq: pack],
infrom (ack_send, myid), 1)
Prvr-> use prop_trans_3::lemmas
Backup point
(. D . 3 . U .)
Prvr-> qed
(. D . 3 . U . QED)
::::::::.(. D . 3 . U . QED BC)
(. D . 3 . U . QED BC 1)
(. D . 3 . U . QED BC 1)
outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
Proved
(. D . 3 . U . QED BC 2)
(. D . 3 . U . QED BC 2)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Proved
(. D . 3 . U . QED BC 3)
(. D . 3 . U . QED BC 3)
null (pkt_seq) ne outto (pkt_send, myid)
Proved
(. D . 3 . U . QED BC)
outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
& proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
& null (pkt_seq) ne outto (pkt_send, myid)
Proved
QED
Prvr-> $Proceeding
(. D . 3 .)
proper_transmission (infrom (source, myid),
outto (pkt_send, myid) [seq: pack],
infrom (ack_send, myid), 1)

Proved
Prvr-> $Proceeding
sender#28
proved in theorem prover.
[PHOTO: Recording initiated Sat 9-May-81 9:55AM]
LINK FROM CMP.DIVITO, TTY 20
TOPS-20 Command processor 4(560)
[PHOTO: Logging disabled Sat 9-May-81 9:55AM]
PHOTO: Logging enabled Sat 9-May-81 10:04AM]
prove sender#33
Entering Prover with verification condition sender#33
H1: infrom (ack_send, myid) [seq: ack#1] = infrom (ack_send#3, myid)
H2: ack#1.seqno = next
H3: pack.seqno = next
H4: next_seqnum (infrom (ack_send, myid)) = next
H5: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H6: outto (ack_send, myid) = outto (ack_send#3, myid)
H7: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H8: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H9: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H10: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: comp (next)
= next_seqnum (infrom (ack_send, myid) [seq: ack#1])
C2: comp (next) = next_seqnum (outto (pkt_send, myid))
C3: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 0)
Typelist equalities
size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(.)
Prvr-> delete h1,h6
Backup point
(. D .)
Prvr-> $Proceeding
Backup point
(. D . 1 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1

= size (unique_msg (outto (pkt_send, myid)))
H2: ack#1.seqno = next
H3: pack.seqno = next
H4: next_seqnum (infrom (ack_send, myid)) = next
H5: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H9: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: comp (next)
= next_seqnum (infrom (ack_send, myid) [seq: ack#1])
Prvr-> retain h2
Backup point
(. D . 1 . D .)
Prvr-> use next_comp::lemmas
Typelist equalities
size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(. D . 1 . D . U .)
Prvr-> theorem
H1: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H2: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H3: comp (p#20$.seqno) = next_seqnum (x#24$ [seq: p#20$])
H4: ack#1.seqno = next
->
C1: comp (next)
= next_seqnum (infrom (ack_send, myid) [seq: ack#1])
Prvr-> delete h1,h2
Backup point
(. D . 1 . D . U . D .)
Prvr-> qed
(. D . 1 . D . U . D . QED)
::..
next := ack#1.seqno

(. D . 1 . D . U . D . QED =S)
QED
Prvr-> $Proceeding
(. D . 1 .)
comp (next) = next_seqnum (infrom (ack_send, myid) [seq: ack#1])
Proved
Prvr-> $Proceeding
Backup point
(. D . 2 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: ack#1.seqno = next
H3: pack.seqno = next
H4: next_seqnum (infrom (ack_send, myid)) = next
H5: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H9: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: comp (next) = next_seqnum (outto (pkt_send, myid))
Prvr-> retain h3,h6,h9
Backup point
(. D . 2 . D .)
Prvr-> use last_next::lemmas
Typelist equalities
size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
Backup point
(. D . 2 . D . U .)
Prvr-> theorem
H1: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H2: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H3: x#26$[size (x#26$)] = p#22$ & null (pkt_seq) ne x#26$
-> p#22$.seqno ne next_seqnum (x#26$)

H4: pack.seqno = next
H5: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H6: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: comp (next) = next_seqnum (outto (pkt_send, myid))
Prvr-> forwardchain h3
:::::Forward chaining gives
pack.seqno ne next_seqnum (outto (pkt_send, myid))
Backup point
(. D . 2 . D . U . FC .)
Prvr-> hypothesis
H1: pack.seqno ne next_seqnum (outto (pkt_send, myid))
H2: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H3: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H4: x#26$[size (x#26$)] = p#22$ & null (pkt_seq) ne x#26$
-> p#22$.seqno ne next_seqnum (x#26$)
H5: pack.seqno = next
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: null (pkt_seq) ne outto (pkt_send, myid)
Prvr-> retain h1,h5
Backup point
(. D . 2 . D . U . FC . D .)
Prvr-> hypothesis
H1: pack.seqno ne next_seqnum (outto (pkt_send, myid))
H2: pack.seqno = next
Prvr-> equality substitute h2
H2 yields
next := pack.seqno
Backup point
(. D . 2 . D . U . FC . D . =S .)
Prvr-> theorem
H1: pack.seqno ne next_seqnum (outto (pkt_send, myid))
->
C1: comp (pack.seqno) = next_seqnum (outto (pkt_send, myid))
Prvr-> use comp_ne::lemmas
Typelist equalities
size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))

Backup point
(. D . 2 . D . U . FC . D . =S . U .)
Prvr-> theorem
H1: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H2: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H3: comp (b1#2$) = b2#2$ iff b1#2$ ne b2#2$
H4: pack.seqno ne next_seqnum (outto (pkt_send, myid))
->
C1: comp (pack.seqno) = next_seqnum (outto (pkt_send, myid))
Prvr-> forwardchain h3
Which way? (left or right)
left
::::Forward chaining gives
comp (pack.seqno) = next_seqnum (outto (pkt_send, myid))
Backup point
(. D . 2 . D . U . FC . D . =S . U . FC .)
Prvr-> $Proceeding
(. D . 2 .)
comp (next) = next_seqnum (outto (pkt_send, myid))
Proved
Prvr-> $Proceeding
Backup point
(. D . 3 .)
Prvr-> theorem
H1: size (unique_msg (infrom (ack_send, myid))) + 1
= size (unique_msg (outto (pkt_send, myid)))
H2: ack#1.seqno = next
H3: pack.seqno = next
H4: next_seqnum (infrom (ack_send, myid)) = next
H5: unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
H6: outto (pkt_send, myid)[size (outto (pkt_send, myid))] = pack
H7: size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
H8: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
H9: null (pkt_seq) ne outto (pkt_send, myid)
->
C1: proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid) [seq: ack#1], 0)
Prvr-> use prop_trans_7::lemmas

Backup point
(. D . 3 . U .)
Prvr-> qed
(. D . 3 . U . QED)
::::::::::.(. D . 3 . U . QED BC)
(. D . 3 . U . QED BC 1)
::::::::::.
size (unique_msg (infrom (ack_send, myid))) :=
size (unique_msg (outto (pkt_send, myid))) - 1
(. D . 3 . U . QED BC 1 =S)
::::::::.
ack#1.seqno := next
(. D . 3 . U . QED BC 1 =S =S)
(. D . 3 . U . QED BC 1)
ack#1.seqno = next_seqnum (infrom (ack_send, myid))
Proved
(. D . 3 . U . QED BC 2)
(. D . 3 . U . QED BC 2)
unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
Proved
(. D . 3 . U . QED BC 3)
(. D . 3 . U . QED BC 3)
size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
Proved
(. D . 3 . U . QED BC 4)
(. D . 3 . U . QED BC 4)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Proved
(. D . 3 . U . QED BC)
ack#1.seqno = next_seqnum (infrom (ack_send, myid))
& unique_msg (outto (pkt_send, myid)) = infrom (source, myid)
& size (unique_msg (outto (pkt_send, myid)))
= size (unique_msg (infrom (ack_send, myid))) + 1
& proper_transmission (infrom (source, myid), outto (pkt_send, myid),
infrom (ack_send, myid), 1)
Proved
QED
Prvr-> $Proceeding
(. D . 3 .)
proper_transmission (infrom (source, myid), outto (pkt_send, myid),

infrom (ack_send, myid) [seq: ack#1], 0)
Proved
Prvr-> $Proceeding
sender#33
proved in theorem prover.
Exec-> show status sender
Proved: SENDER
sender#1 . . . sender#33 proved.
Exec-> show status receiver
receiver
Waiting to be proved: RECEIVER#4, RECEIVER#5, RECEIVER#6, RECEIVER#8,
RECEIVER#12, RECEIVER#13, RECEIVER#14, RECEIVER#18
Proved in VC generator: RECEIVER#1, RECEIVER#2, RECEIVER#3, RECEIVER#7,
RECEIVER#9, RECEIVER#10, RECEIVER#11, RECEIVER#15, RECEIVER#16,
RECEIVER#17
Exec-> prove receiver#4
Entering Prover with verification condition receiver#4
C1: next_seqnum (null (#seqtype#)) = one
C2: proper_reception (null (#seqtype#), null (#seqtype#), null (#seqtype#),
0)
Backup point
(.)
Prvr-> $Proceeding
Backup point
(. 1 .)
Prvr-> expand next_seqnum
Backup point
(. 1 . E .)
Prvr-> theorem
C1: comp (last_bit (seqnums (null (#seqtype#)))) = one
Prvr-> expand seqnums
Backup point
(. 1 . E . E .)
Prvr->expand last_bit
Backup point
(. 1 . E . E . E .)
Prvr-> expand comp
Backup point
(. 1 . E . E . E . E .)
Prvr-> simplify theorem
Prvr-> $Proceeding
(. 1 .)

next_seqnum (null (#seqtype#)) = one
Proved
Prvr-> $Proceeding
Backup point
(. 2 .)
Prvr-> theorem
C1: proper_reception (null (#seqtype#), null (#seqtype#), null (#seqtype#),
0)
Prvr-> expand proper_reception
Backup point
(. 2 . E .)
Prvr-> theorem
C1: 0 = size (unique_msg (null (#seqtype#)))
C2: msg_lag (null (#seqtype#), unique_msg (null (#seqtype#)), 0)
Prvr-> expand unique_msg
Backup point
(. 2 . E . E .)
Prvr->expand msg_lag
Backup point
(. 2 . E . E . E .)
Prvr-> theorem
C1: 0 = size (null (msg_seq))
C2: initial_subseq (null (#seqtype#), null (msg_seq))
Prvr-> simplify theorem
Prvr-> use eq_iss::lemmas
Backup point
(. 2 . E . E . E . U .)
Prvr-> qed
(. 2 . E . E . E . U . QED)
(. 2 . E . E . E . U . QED BC)
(. 2 . E . E . E . U . QED BC)
null (#seqtype#) = null (msg_seq)
Proved
QED
Prvr-> $Proceeding
(. 2 .)
proper_reception (null (#seqtype#), null (#seqtype#), null (#seqtype#), 0)
Proved
Prvr-> $Proceeding
receiver#4
proved in theorem prover.
Exec-> prove receiver#5

Entering Prover with verification condition receiver#5
H1: empty (pkt_rcv#1)
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: infrom (pkt_rcv, myid) = infrom (pkt_rcv#1, myid)
H5: outto (pkt_rcv, myid) = outto (pkt_rcv#1, myid)
H6: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 1)
Backup point
(.)
Prvr-> retain h6
Backup point
(. D .)
Prvr-> use prop_rec_3::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
::.(. D . U . QED BC)
(. D . U . QED BC)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
QED
Prvr-> $Proceeding
receiver#5
proved in theorem prover.
Exec-> prove receiver#6
Entering Prover with verification condition receiver#6
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: pack#1.seqno = exp
H3: next_seqnum (infrom (pkt_rcv, myid)) = exp
H4: next_seqnum (outto (ack_rcv, myid)) = exp
H5: infrom (sink, myid) = infrom (sink#1, myid)
H6: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H7: outto (sink, myid) = outto (sink#1, myid)
H8: full (sink#1)
H9: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)

->
C1: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid), 1)
Backup point
(.)
Prvr-> retain h9
Backup point
(. D .)
Prvr-> use prop_rec_2::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
::.(. D . U . QED BC)
(. D . U . QED BC)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
QED
Prvr-> $Proceeding
receiver#6
proved in theorem prover.
Exec-> prove receiver#8
Entering Prover with verification condition receiver#8
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: outto (sink, myid) [seq: pack#1.mssg] = outto (sink#2, myid)
H3: pack#1.seqno = exp
H4: next_seqnum (infrom (pkt_rcv, myid)) = exp
H5: next_seqnum (outto (ack_rcv, myid)) = exp
H6: infrom (ack_rcv, myid) = infrom (ack_rcv#1, myid)
H7: infrom (sink, myid) = infrom (sink#2, myid)
H8: outto (ack_rcv, myid) = outto (ack_rcv#1, myid)
H9: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H10: full (ack_rcv#1)
H11: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: proper_reception (outto (sink, myid) [seq: pack#1.mssg],
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid), 1)
Backup point

(.)
Prvr-> retain h3,h4,h11
Backup point
(. D .)
Prvr-> use prop_rec_1::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
::::.(. D . U . QED BC)
(. D . U . QED BC 1)
(. D . U . QED BC 1)
pack#1.mssg = pack#1.mssg
Proved
(. D . U . QED BC 2)
:::.
pack#1.seqno := exp
(. D . U . QED BC 2 =S)
(. D . U . QED BC 2)
pack#1.seqno = next_seqnum (infrom (pkt_rcv, myid))
Proved
(. D . U . QED BC 3)
(. D . U . QED BC 3)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
(. D . U . QED BC)
pack#1.mssg = pack#1.mssg
& pack#1.seqno = next_seqnum (infrom (pkt_rcv, myid))
& proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
QED
Prvr-> $Proceeding
receiver#8
proved in theorem prover.
Exec-> prove receiver#13
Entering Prover with verification condition receiver#13
H1: empty (pkt_rcv#1)
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: infrom (pkt_rcv, myid) = infrom (pkt_rcv#1, myid)

H5: outto (pkt_rcv, myid) = outto (pkt_rcv#1, myid)
H6: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 1)
Backup point
(.)
Prvr-> retain h6
Backup point
(. D .)
Prvr-> use prop_rec_3::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
::.(. D . U . QED BC)
(. D . U . QED BC)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
QED
Prvr-> $Proceeding
receiver#13
proved in theorem prover.
Exec-> prove receiver#14
Entering Prover with verification condition receiver#14
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: infrom (ack_rcv, myid) = infrom (ack_rcv#3, myid)
H5: outto (ack_rcv, myid) = outto (ack_rcv#3, myid)
H6: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H7: full (ack_rcv#3)
H8: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
H9: pack#1.seqno ne exp
->
C1: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid), 1)
Backup point

(.)
Prvr-> retain h8
Backup point
(. D .)
Prvr-> use prop_rec_2::lemmas
Backup point
(. D . U .)
Prvr-> qed
(. D . U . QED)
::.(. D . U . QED BC)
(. D . U . QED BC)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
QED
Prvr-> $Proceeding
receiver#14
proved in theorem prover.
Exec-> save abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec-> show status receiver
receiver
Waiting to be proved: RECEIVER#12, RECEIVER#18
Proved in VC generator: RECEIVER#1, RECEIVER#2, RECEIVER#3, RECEIVER#7,
RECEIVER#9, RECEIVER#10, RECEIVER#11, RECEIVER#15, RECEIVER#16,
RECEIVER#17
Proved in theorem prover: RECEIVER#4, RECEIVER#5, RECEIVER#6,
RECEIVER#8, RECEIVER#13, RECEIVER#14
Exec-> prove receiver#12
Entering Prover with verification condition receiver#12
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: outto (ack_rcv, myid) [seq: pack#1] = outto (ack_rcv#2, myid)
H3: outto (sink, myid) [seq: pack#1.mssg] = outto (sink#2, myid)
H4: pack#1.seqno = exp
H5: next_seqnum (infrom (pkt_rcv, myid)) = exp
H6: next_seqnum (outto (ack_rcv, myid)) = exp
H7: infrom (ack_rcv, myid) = infrom (ack_rcv#2, myid)
H8: infrom (sink, myid) = infrom (sink#2, myid)

H9: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H10: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: comp (exp)
= next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1])
C2: comp (exp) = next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
C3: proper_reception (outto (sink, myid) [seq: pack#1.mssg],
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
Backup point
(.)
Prvr-> retain h4,h5,h6,h10
Backup point
(. D .)
Prvr-> $Proceeding
Backup point
(. D . 1 .)
Prvr-> use next_comp::lemmas
Backup point
(. D . 1 . U .)
Prvr-> theorem
H1: comp (p#8$.seqno) = next_seqnum (x#12$ [seq: p#8$])
H2: pack#1.seqno = exp
H3: next_seqnum (infrom (pkt_rcv, myid)) = exp
H4: next_seqnum (outto (ack_rcv, myid)) = exp
H5: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: comp (exp)
= next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1])
Prvr-> put
For what?* p#8$;
Put what?* pack#1;
For what?* $done
Backup point
(. D . 1 . U . PUT .)
Prvr-> equality substitute h3
H3 yields
exp := next_seqnum (infrom (pkt_rcv, myid))
Backup point
(. D . 1 . U . PUT . =S .)

Prvr-> theorem
H1: pack#1.seqno = next_seqnum (infrom (pkt_rcv, myid))
H2: comp (pack#1.seqno) = next_seqnum (x#12$ [seq: pack#1])
H3: next_seqnum (infrom (pkt_rcv, myid))
= next_seqnum (outto (ack_rcv, myid))
H4: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: comp (next_seqnum (infrom (pkt_rcv, myid)))
= next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1])
Prvr-> equality substitute h1
H1 yields
next_seqnum (infrom (pkt_rcv, myid)) := pack#1.seqno
Backup point
(. D . 1 . U . PUT . =S . =S .)
Prvr-> $Proceeding
::(. D . 1 .)
comp (exp) = next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1])
Proved
Prvr-> $Proceeding
Backup point
(. D . 2 .)
Prvr-> $Proceeding
::::.....Ran out of tricks
Prvr-> theorem
H1: pack#1.seqno = exp
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: comp (exp) = next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
Prvr-> equality substitute h1
H1 yields
exp := pack#1.seqno
Backup point
(. D . 2 . =S .)
Prvr-> theorem
H1: pack#1.seqno = next_seqnum (infrom (pkt_rcv, myid))
H2: pack#1.seqno = next_seqnum (outto (ack_rcv, myid))
H3: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)

->
C1: comp (pack#1.seqno)
= next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
Prvr-> retain h2
Backup point
(. D . 2 . =S . D .)
Prvr-> use next_comp::lemmas
Backup point
(. D . 2 . =S . D . U .)
Prvr-> theorem
H1: comp (p#10$.seqno) = next_seqnum (x#14$ [seq: p#10$])
H2: pack#1.seqno = next_seqnum (outto (ack_rcv, myid))
->
C1: comp (pack#1.seqno)
= next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
Prvr-> $Proceeding
:(. D . 2 .)
comp (exp) = next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
Proved
Prvr-> $Proceeding
Backup point
(. D . 3 .)
Prvr-> theorem
H1: pack#1.seqno = exp
H2: next_seqnum (infrom (pkt_rcv, myid)) = exp
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
->
C1: proper_reception (outto (sink, myid) [seq: pack#1.mssg],
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
Prvr-> use prop_rec_4::lemmas
Backup point
(. D . 3 . U .)
Prvr-> qed
(. D . 3 . U . QED)
::::::(. D . 3 . U . QED BC)
(. D . 3 . U . QED BC 1)
(. D . 3 . U . QED BC 1)
pack#1.mssg = pack#1.mssg
Proved

(. D . 3 . U . QED BC 2)
:::::.
pack#1.seqno := exp
(. D . 3 . U . QED BC 2 =S)
(. D . 3 . U . QED BC 2)
pack#1.seqno = next_seqnum (infrom (pkt_rcv, myid))
Proved
(. D . 3 . U . QED BC 3)
:::::.
pack#1.seqno := exp
(. D . 3 . U . QED BC 3 =S)
(. D . 3 . U . QED BC 3)
pack#1.seqno = next_seqnum (outto (ack_rcv, myid))
Proved
(. D . 3 . U . QED BC 4)
(. D . 3 . U . QED BC 4)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
(. D . 3 . U . QED BC)
pack#1.mssg = pack#1.mssg
& pack#1.seqno = next_seqnum (infrom (pkt_rcv, myid))
& pack#1.seqno = next_seqnum (outto (ack_rcv, myid))
& proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
QED
Prvr-> $Proceeding
(. D . 3 .)
proper_reception (outto (sink, myid) [seq: pack#1.mssg],
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
Proved
Prvr-> $Proceeding
receiver#12
proved in theorem prover.
Exec-> prove receiver#18
Entering Prover with verification condition receiver#18
H1: infrom (pkt_rcv, myid) [seq: pack#1] = infrom (pkt_rcv#2, myid)
H2: outto (ack_rcv, myid) [seq: pack#1] = outto (ack_rcv#4, myid)
H3: next_seqnum (infrom (pkt_rcv, myid)) = exp
H4: next_seqnum (outto (ack_rcv, myid)) = exp

H5: infrom (ack_rcv, myid) = infrom (ack_rcv#4, myid)
H6: outto (pkt_rcv, myid) = outto (pkt_rcv#2, myid)
H7: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
H8: pack#1.seqno ne exp
->
C1: next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1]) = exp
C2: next_seqnum (outto (ack_rcv, myid) [seq: pack#1]) = exp
C3: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
Backup point
(.)
Prvr-> retain h3,h4,h7,h8
Backup point
(. D .)
Prvr-> $Proceeding
Backup point
(. D . 1 .)
Prvr-> theorem
H1: next_seqnum (infrom (pkt_rcv, myid)) = exp
H2: next_seqnum (outto (ack_rcv, myid)) = exp
H3: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
H4: pack#1.seqno ne exp
->
C1: next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1]) = exp
Prvr-> retain h1,h4
Backup point
(. D . 1 . D .)
Prvr-> use next_comp::lemmas
Backup point
(. D . 1 . D . U .)
Prvr-> use comp_ne::lemmas
Backup point
(. D . 1 . D . U . U .)
Prvr-> theorem
H1: comp (b1#2$) = b2#2$ iff b1#2$ ne b2#2$
H2: comp (p#14$.seqno) = next_seqnum (x#18$ [seq: p#14$])
H3: next_seqnum (infrom (pkt_rcv, myid)) = exp
H4: pack#1.seqno ne exp
->

C1: next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1]) = exp
Prvr-> forwardchain h1
Which way? (left or right)
left
::::Forward chaining gives
comp (pack#1.seqno) = exp
Backup point
(. D . 1 . D . U . U . FC .)
Prvr-> theorem
H1: comp (pack#1.seqno) = exp
H2: comp (b1#2$) = b2#2$ iff b1#2$ ne b2#2$
H3: comp (p#14$.seqno) = next_seqnum (x#18$ [seq: p#14$])
H4: next_seqnum (infrom (pkt_rcv, myid)) = exp
H5: pack#1.seqno ne exp
->
C1: next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1]) = exp
Prvr-> equality substitute h1
H1 yields
exp := comp (pack#1.seqno)
Backup point
(. D . 1 . D . U . U . FC . =S .)
Prvr-> theorem
H1: comp (p#14$.seqno) = next_seqnum (x#18$ [seq: p#14$])
H2: comp (pack#1.seqno) = next_seqnum (infrom (pkt_rcv, myid))
H3: comp (b1#2$) = b2#2$ iff b1#2$ ne b2#2$
H4: pack#1.seqno ne comp (pack#1.seqno)
->
C1: comp (pack#1.seqno)
= next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1])
Prvr-> retain h1,h2
Backup point
(. D . 1 . D . U . U . FC . =S . D .)
Prvr-> $Proceeding
:(. D . 1 .)
next_seqnum (infrom (pkt_rcv, myid) [seq: pack#1]) = exp
Proved
Prvr-> $Proceeding
Backup point
(. D . 2 .)
Prvr-> theorem
H1: next_seqnum (infrom (pkt_rcv, myid)) = exp
H2: next_seqnum (outto (ack_rcv, myid)) = exp

H3: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
H4: pack#1.seqno ne exp
->
C1: next_seqnum (outto (ack_rcv, myid) [seq: pack#1]) = exp
Prvr-> retain h2,h4
Backup point
(. D . 2 . D .)
Prvr-> use next_comp::lemmas
Backup point
(. D . 2 . D . U .)
Prvr-> use comp_ne::lemmas
Backup point
(. D . 2 . D . U . U .)
Prvr-> theorem
H1: comp (b1#4$) = b2#4$ iff b1#4$ ne b2#4$
H2: comp (p#16$.seqno) = next_seqnum (x#20$ [seq: p#16$])
H3: next_seqnum (outto (ack_rcv, myid)) = exp
H4: pack#1.seqno ne exp
->
C1: next_seqnum (outto (ack_rcv, myid) [seq: pack#1]) = exp
Prvr-> forwardchain h1
Which way? (left or right)
left
::::Forward chaining gives
comp (pack#1.seqno) = exp
Backup point
(. D . 2 . D . U . U . FC .)
Prvr-> hypothesis
H1: comp (pack#1.seqno) = exp
H2: comp (b1#4$) = b2#4$ iff b1#4$ ne b2#4$
H3: comp (p#16$.seqno) = next_seqnum (x#20$ [seq: p#16$])
H4: next_seqnum (outto (ack_rcv, myid)) = exp
H5: pack#1.seqno ne exp
Prvr-> equality substitute h1
H1 yields
exp := comp (pack#1.seqno)
Backup point
(. D . 2 . D . U . U . FC . =S .)
Prvr-> theorem
H1: comp (p#16$.seqno) = next_seqnum (x#20$ [seq: p#16$])
H2: comp (pack#1.seqno) = next_seqnum (outto (ack_rcv, myid))

H3: comp (b1#4$) = b2#4$ iff b1#4$ ne b2#4$
H4: pack#1.seqno ne comp (pack#1.seqno)
->
C1: comp (pack#1.seqno)
= next_seqnum (outto (ack_rcv, myid) [seq: pack#1])
Prvr-> retain h1,h2
Backup point
(. D . 2 . D . U . U . FC . =S . D .)
Prvr-> $Proceeding
:(. D . 2 .)
next_seqnum (outto (ack_rcv, myid) [seq: pack#1]) = exp
Proved
Prvr-> $Proceeding
Backup point
(. D . 3 .)
Prvr-> theorem
H1: next_seqnum (infrom (pkt_rcv, myid)) = exp
H2: next_seqnum (outto (ack_rcv, myid)) = exp
H3: proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
H4: pack#1.seqno ne exp
->
C1: proper_reception (outto (sink, myid),
infrom (pkt_rcv, myid) [seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
Prvr-> use prop_rec_5::lemmas
Backup point
(. D . 3 . U .)
Prvr-> qed
(. D . 3 . U . QED)
:::::.(. D . 3 . U . QED BC)
(. D . 3 . U . QED BC 1)
(. D . 3 . U . QED BC 1)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
Proved
(. D . 3 . U . QED BC 2)
::::.
next_seqnum (infrom (pkt_rcv, myid)) := exp
(. D . 3 . U . QED BC 2 =S)
(. D . 3 . U . QED BC 2)
pack#1.seqno ne next_seqnum (infrom (pkt_rcv, myid))

Proved
(. D . 3 . U . QED BC 3)
:::::..
next_seqnum (outto (ack_rcv, myid)) := exp
(. D . 3 . U . QED BC 3 =S)
(. D . 3 . U . QED BC 3)
pack#1.seqno ne next_seqnum (outto (ack_rcv, myid))
Proved
(. D . 3 . U . QED BC)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid),
outto (ack_rcv, myid), 0)
& pack#1.seqno ne next_seqnum (infrom (pkt_rcv, myid))
& pack#1.seqno ne next_seqnum (outto (ack_rcv, myid))
Proved
QED
Prvr-> $Proceeding
(. D . 3 .)
proper_reception (outto (sink, myid), infrom (pkt_rcv, myid)
[seq: pack#1],
outto (ack_rcv, myid) [seq: pack#1], 0)
Proved
Prvr-> $Proceeding
receiver#18
proved in theorem prover.
Exec-> show status receiver
Proved: RECEIVER
receiver#1 . . . receiver#18 proved.
Exec-> prove ab_protocol
ab_protocol#1
proved in VC generator.
Used in proof: TIMER, RECEIVER, MEDIUM, SENDER
Entering Prover with verification condition ab_protocol#2
H1: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H2: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)
H3: isblocked (medium#1)
H4: isblocked (medium#2)
H5: isblocked (receiver#1)
H6: isblocked (sender#1)
H7: isblocked (timer#1)
H8: allto (ack_send#1) sub allfrom (ack_rcv#1)

H9: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: msg_lag (outto (sink#1, receiver#1), infrom (source#1, sender#1), 1)
Backup point
(.)
Prvr-> retain h1,h2,h8,h9
Backup point
(. D .)
Prvr-> use abp_1::lemmas
Backup point
(. D . U .)
Prvr-> theorem
H1: proper_reception (v#2$, x#2$, y#2$, 1)
& proper_transmission (u#2$, w#2$, z#2$, 1) & x#2$ sub w#2$
& z#2$ sub y#2$
-> msg_lag (v#2$, u#2$, 1)
H2: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H3: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)
H4: allto (ack_send#1) sub allfrom (ack_rcv#1)
H5: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: msg_lag (outto (sink#1, receiver#1), infrom (source#1, sender#1), 1)
Prvr-> qed
(. D . U . QED)
:::::.(. D . U . QED BC)
(. D . U . QED BC 1)
:(. D . U . QED BC 1)
proper_reception (outto (sink#1, receiver#1), x#2$, y#2$, 1)
Proved
(. D . U . QED BC 2)
::(. D . U . QED BC 2)
proper_transmission (infrom (source#1, sender#1), w#2$, z#2$, 1)
Proved
(. D . U . QED BC 3)
::::.....::....Ran out of tricks
Prvr-> theorem
H1: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H2: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)

H3: allto (ack_send#1) sub allfrom (ack_rcv#1)
H4: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: allfrom (pkt_rcv#1) sub allto (pkt_send#1)
Prvr-> label
(. D . U . QED BC 3)
Prvr-> back up 3
Restored
Backup point
(. D . U . QED .)
Prvr-> back up2
**** Illegal command - UP2
Back UP or establish a back up POINT to return to? up
How many levels would you like to back up? 2
Restored
Backup point
(. D . U .)
Prvr-> theorem
H1: proper_reception (v#2$, x#2$, y#2$, 1)
& proper_transmission (u#2$, w#2$, z#2$, 1) & x#2$ sub w#2$
& z#2$ sub y#2$
-> msg_lag (v#2$, u#2$, 1)
H2: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H3: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)
H4: allto (ack_send#1) sub allfrom (ack_rcv#1)
H5: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: msg_lag (outto (sink#1, receiver#1), infrom (source#1, sender#1), 1)
Prvr-> claim
Newgoal:
* allfrom(ack_send#1) sub allto(ack_rcv#1)
* & allfrom(pkt_rcv#1) sub allto(pkt_send#1);
Defer proof of claim? y
Backup point
(. D . U . .)
Prvr-> theorem
H1: allfrom (ack_send#1) sub allto (ack_rcv#1)
H2: allfrom (pkt_rcv#1) sub allto (pkt_send#1)
H3: proper_reception (v#2$, x#2$, y#2$, 1)
& proper_transmission (u#2$, w#2$, z#2$, 1) & x#2$ sub w#2$

& z#2$ sub y#2$
-> msg_lag (v#2$, u#2$, 1)
H4: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H5: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)
H6: allto (ack_send#1) sub allfrom (ack_rcv#1)
H7: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: msg_lag (outto (sink#1, receiver#1), infrom (source#1, sender#1), 1)
Prvr-> qed
(. D . U . . QED)
::::::::...(. D . U . . QED BC)
(. D . U . . QED BC 1)
:::(. D . U . . QED BC 1)
proper_reception (outto (sink#1, receiver#1), x#2$, y#2$, 1)
Proved
(. D . U . . QED BC 2)
::::(. D . U . . QED BC 2)
proper_transmission (infrom (source#1, sender#1), w#2$, z#2$, 1)
Proved
(. D . U . . QED BC 3)
(. D . U . . QED BC 3)
allfrom (pkt_rcv#1) sub allto (pkt_send#1)
Proved
(. D . U . . QED BC 4)
(. D . U . . QED BC 4)
allfrom (ack_send#1) sub allto (ack_rcv#1)
Proved
(. D . U . . QED BC)
proper_reception (outto (sink#1, receiver#1), x#2$, y#2$, 1)
& proper_transmission (infrom (source#1, sender#1), w#2$, z#2$, 1)
& x#2$ sub w#2$ & z#2$ sub y#2$
Proved
QED
Prvr-> $Proceeding
Proved using CLAIM. Now prove the claim.
Backup point
(CLAIM .)
Prvr-> theorem
H1: proper_reception (v#2$, x#2$, y#2$, 1)
& proper_transmission (u#2$, w#2$, z#2$, 1) & x#2$ sub w#2$

& z#2$ sub y#2$
-> msg_lag (v#2$, u#2$, 1)
H2: proper_reception (outto (sink#1, receiver#1), allfrom (pkt_rcv#1),
allto (ack_rcv#1), 1)
H3: proper_transmission (infrom (source#1, sender#1), allto (pkt_send#1),
allfrom (ack_send#1), 1)
H4: allto (ack_send#1) sub allfrom (ack_rcv#1)
H5: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: allfrom (ack_send#1) sub allto (ack_rcv#1)
C2: allfrom (pkt_rcv#1) sub allto (pkt_send#1)
Prvr-> retain h4,h5
Backup point
(CLAIM . D .)
Prvr-> $Proceeding
Backup point
(CLAIM . D . 1 .)
Prvr-> theorem
H1: allto (ack_send#1) sub allfrom (ack_rcv#1)
H2: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: allfrom (ack_send#1) sub allto (ack_rcv#1)
Prvr-> delete h2
Backup point
(CLAIM . D . 1 . D .)
Prvr-> $Proceeding
Ran out of tricks
Prvr-> simplify theorem
Prvr->theorem
H1: allto (ack_send#1) sub allfrom (ack_rcv#1)
->
C1: allfrom (ack_send#1) sub allto (ack_rcv#1)
Prvr-> use hist_sub::lemmas
Backup point
(CLAIM . D . 1 . D . U .)
Prvr-> qed
(CLAIM . D . 1 . D . U . QED)
::.(CLAIM . D . 1 . D . U . QED BC)
(CLAIM . D . 1 . D . U . QED BC)
allfrom (ack_send#1) sub allfrom (ack_rcv#1)
Proved
QED

Prvr-> $Proceeding
(CLAIM . D . 1 .)
allfrom (ack_send#1) sub allto (ack_rcv#1)
Proved
Prvr-> $Proceeding
Backup point
(CLAIM . D . 2 .)
Prvr-> theorem
H1: allto (ack_send#1) sub allfrom (ack_rcv#1)
H2: allto (pkt_rcv#1) sub allfrom (pkt_send#1)
->
C1: allfrom (pkt_rcv#1) sub allto (pkt_send#1)
Prvr-> delete h1
Backup point
(CLAIM . D . 2 . D .)
Prvr-> use hist_sub::lemmas
Backup point
(CLAIM . D . 2 . D . U .)
Prvr-> qed
(CLAIM . D . 2 . D . U . QED)
::.(CLAIM . D . 2 . D . U . QED BC)
(CLAIM . D . 2 . D . U . QED BC)
allfrom (pkt_rcv#1) sub allfrom (pkt_send#1)
Proved
QED
Prvr-> $Proceeding
(CLAIM . D . 2 .)
allfrom (pkt_rcv#1) sub allto (pkt_send#1)
Proved
Prvr-> $Proceeding
(CLAIM .)
allfrom (ack_send#1) sub allto (ack_rcv#1)
& allfrom (pkt_rcv#1) sub allto (pkt_send#1)
Established claim
ab_protocol#2
proved in theorem prover.
Used in proof: TIMER, RECEIVER, MEDIUM, SENDER
ab_protocol#3
proved in VC generator.
Used in proof: TIMER, RECEIVER, MEDIUM, SENDER
ab_protocol#4
proved in VC generator.

Used in proof: TIMER, RECEIVER, MEDIUM, SENDER
Exec-> show status scope $
SCOPE ALT_BIT_PROTOCOL
Waiting for pending body to be filled in: MEDIUM, TIMER
Proved: AB_PROTOCOL, COMP, RECEIVER, SENDER
Types, constants: BIT, BIT_SEQ, CLK_BUF, MESSAGE, MSG_BUF, MSG_SEQ, PACKET,
PKT_BUF, PKT_SEQ
Exec-> show status scope lemmas
SCOPE LEMMAS
Waiting to be proved: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4,
PROP_REC_5, PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4,
PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7, SUB_APP, SIZE_NULL,
SUB_SEQNUM, SUB_TO_LAG, SUB_NCHANGES
Exec-> save abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec-> exit no
4@pop
@pop
[PHOTO: Recording terminated Sat 9-May-81 10:44AM]
[PHOTO: Recording initiated Sat 9-May-81 10:54AM]
LINK FROM CMP.DIVITO, TTY 20
TOPS-20 Command processor 4(560)
[PHOTO: Logging disabled Sat 9-May-81 10:54AM]
PHOTO: Logging enabled Sat 9-May-81 10:57AM]
% It is now time to prove the lemmas. We will start with the
% higher level ones and move down to the lower level ones.
show status all
The current design and verification status is:
SCOPE ALT_BIT_PROTOCOL
Waiting for pending body to be filled in: MEDIUM, TIMER
Proved: AB_PROTOCOL, COMP, RECEIVER, SENDER
Types, constants: BIT, BIT_SEQ, CLK_BUF, MESSAGE, MSG_BUF, MSG_SEQ, PACKET,
PKT_BUF, PKT_SEQ
SCOPE ALT_BIT_SPECS

Waiting for pending body to be filled in: INITIAL_SUBSEQ
For specifications only: LAST_BIT, MSG_LAG, NCHANGES, NEXT_SEQNUM,
PROPER_RECEPTION, PROPER_TRANSMISSION, REPEATS, SEQNUMS, UNIQUE_MSG
SCOPE LEMMAS
Waiting to be proved: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4,
PROP_REC_5, PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4,
PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7, SUB_APP, SIZE_NULL,
SUB_SEQNUM, SUB_TO_LAG, SUB_NCHANGES
Exec-> set scope lemmas
Exec-> prove prop_trans_1
Entering Prover with lemma prop_trans_1
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
all m : message,
p.mssg = m & p.seqno = next_seqnum (x)
& proper_transmission (u, x, y, 0)
-> u [seq: m] = unique_msg (x [seq: p])
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: proper_transmission (u, x, y, 0)
->
C1: u [seq: m] = unique_msg (x [seq: p])
Backup point
(.)
Prvr-> expand proper_transmission
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: size (unique_msg (y)) = size (unique_msg (x))
H4: size (unique_msg (y)) = size (u)
H5: msg_lag (unique_msg (x), u, 0)
H6: repeats (x)
->
C1: u [seq: m] = unique_msg (x [seq: p])

```
Prvr-> retain h1,h2,h5
Backup point
(. E . D .)
Prvr-> use msg_lag_eq
Backup point
(. E . D . U .)
Prvr-> hypothesis
H1: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H2: p.mssg = m
H3: p.seqno = next_seqnum (x)
H4: msg_lag (unique_msg (x), u, 0)
Prvr-> forwardchain h1
Which way? (left or right)
left
::::Forward chaining gives
unique_msg (x) = u
Backup point
(. E . D . U . FC .)
Prvr-> theorem
H1: unique_msg (x) = u
H2: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H3: p.mssg = m
H4: p.seqno = next_seqnum (x)
H5: msg_lag (unique_msg (x), u, 0)
->
C1: u  [seq: m] = unique_msg (x  [seq: p])
Prvr-> equality substitute htheorem
HTHEOREM not in last output
**** Command Aborted
Prvr-> equality substitute h1
H1 yields
u := unique_msg (x)
Backup point
(. E . D . U . FC . =S .)
Prvr-> theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H4: msg_lag (unique_msg (x), unique_msg (x), 0)
->
C1: unique_msg (x)  [seq: m] = unique_msg (x  [seq: p])
Prvr-> delete h3,h4
```

Backup point
(. E . D . U . FC . =S . D .)
Prvr-> equality substitute h1
H1 yields
m := p.mssg
Backup point
(. E . D . U . FC . =S . D . =S .)
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? show-choices
1: unique_msg (x)
2: unique_msg (x [seq: p])
Which do you want to expand? 2
Backup point
(. E . D . U . FC . =S . D . =S . E .)
Prvr-> conclusion
C1: unique_msg (x) [seq: p.mssg]
= if p.seqno = next_seqnum (x)
then unique_msg (x) [seq: p.mssg]
else unique_msg (x)
fi
Prvr-> simplify theorem
Prvr->theorem
C1: true
Prvr-> $Proceeding
prop_trans_1 proved in theorem prover.
Exec-> prove prop_trans_2
Entering Prover with lemma prop_trans_2
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
all m : message,
p.mssg = m & p.seqno = next_seqnum (x)
& proper_transmission (u, x, y, 0)
-> proper_transmission (u [seq: m], x [seq: p], y, 1)
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: proper_transmission (u, x, y, 0)
->
C1: proper_transmission (u [seq: m], x [seq: p], y, 1)
Backup point
(.)

Prvr-> expand proper_transmission
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: size (unique_msg (y)) = size (unique_msg (x))
H4: size (unique_msg (y)) = size (u)
H5: msg_lag (unique_msg (x), u, 0)
H6: repeats (x)
->
C1: msg_lag (unique_msg (x [seq: p]), u [seq: m], 1)
C2: repeats (x [seq: p])
C3: size (unique_msg (x [seq: p])) - size (unique_msg (y))
in [0..1]
C4: (size (u) - size (unique_msg (y))) + 1 in [0..1]
Prvr-> equality substitute h4
H4 can be solved for:
T1: size (unique_msg (y))
T2: size (u)
Which term (by label) do you want to substitute for?
t1
size (unique_msg (y)) := size (u)
OK??
y
Backup point
(. E . =S .)
Prvr-> use msg_lag_eq
Backup point
(. E . =S . U .)
Prvr-> hypothesis
H1: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H2: p.mssg = m
H3: p.seqno = next_seqnum (x)
H4: size (u) = size (unique_msg (x))
H5: msg_lag (unique_msg (x), u, 0)
H6: repeats (x)
Prvr-> forwardchain h1
Which way? (left or right)

left
:::::Forward chaining gives
unique_msg (x) = u
Backup point
(. E . =S . U . FC .)
Prvr-> theorem
H1: unique_msg (x) = u
H2: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H3: p.mssg = m
H4: p.seqno = next_seqnum (x)
H5: size (u) = size (unique_msg (x))
H6: msg_lag (unique_msg (x), u, 0)
H7: repeats (x)
->
C1: msg_lag (unique_msg (x [seq: p]), u [seq: m], 1)
C2: repeats (x [seq: p])
C3: size (unique_msg (x [seq: p])) - size (u) in [0..1]
Prvr-> equality substitute h1,h3
H1 yields
u := unique_msg (x)
H3 yields
m := p.mssg
Backup point
(. E . =S . U . FC . =S .)
Prvr-> theorem
H1: p.seqno = next_seqnum (x)
H2: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H3: msg_lag (unique_msg (x), unique_msg (x), 0)
H4: repeats (x)
->
C1: msg_lag (unique_msg (x [seq: p]), unique_msg (x) [seq: p.mssg],
1)
C2: repeats (x [seq: p])
C3: size (unique_msg (x [seq: p])) - size (unique_msg (x))
in [0..1]
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? show-choices
1: unique_msg (x)
2: unique_msg (x [seq: p])
Which do you want to expand? 2
Backup point

(. E . =S . U . FC . =S . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: p.seqno = next_seqnum (x)
H2: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H3: msg_lag (unique_msg (x), unique_msg (x), 0)
H4: repeats (x)
->
C1: msg_lag (unique_msg (x) [seq: p.mssg],
unique_msg (x) [seq: p.mssg], 1)
C2: repeats (x [seq: p])
Prvr-> retain h1,h4
Backup point
(. E . =S . U . FC . =S . E . D .)
Prvr-> $Proceeding
Backup point
(. E . =S . U . FC . =S . E . D . 1 .)
Prvr-> theorem
H1: p.seqno = next_seqnum (x)
H2: repeats (x)
->
C1: msg_lag (unique_msg (x) [seq: p.mssg],
unique_msg (x) [seq: p.mssg], 1)
Prvr-> expand msg_lag
Backup point
(. E . =S . U . FC . =S . E . D . 1 . E .)
Prvr-> use eq_iss
Typelist equalities
size (u) = size (unique_msg (x))
Backup point
(. E . =S . U . FC . =S . E . D . 1 . E . U .)
Prvr-> theorem
H1: size (u) = size (unique_msg (x))
H2: u#4$ = v#4$ -> initial_subseq (u#4$, v#4$)
H3: p.seqno = next_seqnum (x)
H4: repeats (x)
->
C1: initial_subseq (unique_msg (x) [seq: p.mssg],
unique_msg (x) [seq: p.mssg])
Prvr-> retain h2
Backup point
(. E . =S . U . FC . =S . E . D . 1 . E . U . D .)

```
Prvr-> qed
(. E . =S . U . FC . =S . E . D . 1 . E . U . D . QED)
(. E . =S . U . FC . =S . E . D . 1 . E . U . D . QED BC)
(. E . =S . U . FC . =S . E . D . 1 . E . U . D . QED BC)
unique_msg (x)  [seq: p.mssg] = unique_msg (x)  [seq: p.mssg]
Proved
QED
Prvr-> $Proceeding
(. E . =S . U . FC . =S . E . D . 1 .)
msg_lag (unique_msg (x)  [seq: p.mssg],
unique_msg (x)  [seq: p.mssg], 1)
Proved
Prvr-> $Proceeding
Backup point
(. E . =S . U . FC . =S . E . D . 2 .)
Prvr-> theorem
H1: p.seqno = next_seqnum (x)
H2: repeats (x)
->
C1: repeats (x  [seq: p])
Prvr-> expand repeats
More than one to expand.
Which do you want to expand? show-choices
1: repeats (x)
2: repeats (x  [seq: p])
Which do you want to expand? 2
Typelist equalities
size (u) = size (unique_msg (x))
Backup point
(. E . =S . U . FC . =S . E . D . 2 . E .)
Prvr-> theorem
H1: size (u) = size (unique_msg (x))
H2: null (pkt_seq) ne x
H3: p.seqno = next_seqnum (x)
H4: repeats (x)
->
C1: repeats (x)
C2: p.seqno ne next_seqnum (x) -> x[size (x)] = p
Prvr-> back up 2
Restored
Backup point
(. E . =S . U . FC . =S . E . D . 2 .)
```

```
Prvr-> theorem
H1: p.seqno = next_seqnum (x)
H2: repeats (x)
->
C1: repeats (x  [seq: p])
Prvr-> use last_repeats
Typelist equalities
size (u) = size (unique_msg (x))
Backup point
(. E . =S . U . FC . =S . E . D . 2 . U .)
Prvr-> qed
(. E . =S . U . FC . =S . E . D . 2 . U . QED)
:::::..(. E . =S . U . FC . =S . E . D . 2 . U . QED BC)
(. E . =S . U . FC . =S . E . D . 2 . U . QED BC 1)
:::...::Ran out of tricks
Prvr-> back up 4
Restored
Backup point
(. E . =S . U . FC . =S . E . D . 2 . U .)
Prvr-> theorem
H1: size (u) = size (unique_msg (x))
H2:  x#2$[size (x#2$)] = p#2$ & repeats (x#2$) & null (pkt_seq)
ne x#2$
-> repeats (x#2$  [seq: p#2$])
H3: p.seqno = next_seqnum (x)
H4: repeats (x)
->
C1: repeats (x  [seq: p])
Prvr-> back up 2
Restored
Backup point
(. E . =S . U . FC . =S . E . D . 2 .)
Prvr-> theorem
H1: p.seqno = next_seqnum (x)
H2: repeats (x)
->
C1: repeats (x  [seq: p])
Prvr-> qed
(. E . =S . U . FC . =S . E . D . 2 . QED)
::..::(. E . =S . U . FC . =S . E . D . 2 . QED E)
Typelist equalities
size (u) = size (unique_msg (x))
```

(. E . =S . U . FC . =S . E . D . 2 . QED E P->)
(. E . =S . U . FC . =S . E . D . 2 . QED E P-> 1)
(. E . =S . U . FC . =S . E . D . 2 . QED E P-> 1)
repeats (x)
Proved
(. E . =S . U . FC . =S . E . D . 2 . QED E P-> 2)
(. E . =S . U . FC . =S . E . D . 2 . QED E P-> 2 P->)
(. E . =S . U . FC . =S . E . D . 2 . QED E P-> 2)
p.seqno ne next_seqnum (x) -> x[size (x)] = p
Proved
QED
Prvr-> $Proceeding
(. E . =S . U . FC . =S . E . D . 2 .)
repeats (x [seq: p])
Proved
Prvr-> $Proceeding
prop_trans_2 proved in theorem prover.
Exec-> prove prop_trans_3
Entering Prover with lemma prop_trans_3
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
x[size (x)] = p & proper_transmission (u, x, y, 1)
& null (pkt_seq) ne x
-> proper_transmission (u, x [seq: p], y, 1)
H1: x[size (x)] = p
H2: proper_transmission (u, x, y, 1)
H3: null (pkt_seq) ne x
->
C1: proper_transmission (u, x [seq: p], y, 1)
Backup point
(.)
Prvr-> expand proper_transmission
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
H1: x[size (x)] = p
H2: msg_lag (unique_msg (x), u, 1)
H3: repeats (x)

H4: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H5: size (u) - size (unique_msg (y)) in [0..1]
H6: null (pkt_seq) ne x
->
C1: msg_lag (unique_msg (x [seq: p]), u, 1)
C2: repeats (x [seq: p])
C3: size (unique_msg (x [seq: p])) - size (unique_msg (y))
in [0..1]
Prvr-> use last_unique
Backup point
(. E . U .)
Prvr-> hypothesis
H1: x#2$[size (x#2$)] = p#2$ & null (pkt_seq) ne x#2$
-> unique_msg (x#2$ [seq: p#2$]) = unique_msg (x#2$)
H2: x[size (x)] = p
H3: msg_lag (unique_msg (x), u, 1)
H4: repeats (x)
H5: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H6: size (u) - size (unique_msg (y)) in [0..1]
H7: null (pkt_seq) ne x
Prvr-> forwardchain h1
::Forward chaining gives
unique_msg (x [seq: p]) = unique_msg (x)
Backup point
(. E . U . FC .)
Prvr-> theorem
H1: unique_msg (x [seq: p]) = unique_msg (x)
H2: x#2$[size (x#2$)] = p#2$ & null (pkt_seq) ne x#2$
-> unique_msg (x#2$ [seq: p#2$]) = unique_msg (x#2$)
H3: x[size (x)] = p
H4: msg_lag (unique_msg (x), u, 1)
H5: repeats (x)
H6: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H7: size (u) - size (unique_msg (y)) in [0..1]
H8: null (pkt_seq) ne x
->
C1: msg_lag (unique_msg (x [seq: p]), u, 1)
C2: repeats (x [seq: p])
C3: size (unique_msg (x [seq: p])) - size (unique_msg (y))
in [0..1]
Prvr-> equality substitute h1
H1 yields

unique_msg (x [seq: p]) := unique_msg (x)
Backup point
(. E . U . FC . =S .)
Prvr-> simplify theorem
Prvr->theorem
H1: x[size (x)] = p
H2: msg_lag (unique_msg (x), u, 1)
H3: repeats (x)
H4: x#2$[size (x#2$)] = p#2$ & null (pkt_seq) ne x#2$
-> unique_msg (x#2$ [seq: p#2$]) = unique_msg (x#2$)
H5: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H6: size (u) - size (unique_msg (y)) in [0..1]
H7: null (pkt_seq) ne x
->
C1: repeats (x [seq: p])
Prvr-> retain h1,h3,h7
Backup point
(. E . U . FC . =S . D .)
Prvr-> use last_repeats
Backup point
(. E . U . FC . =S . D . U .)
Prvr-> qed
(. E . U . FC . =S . D . U . QED)
:::::(. E . U . FC . =S . D . U . QED BC)
(. E . U . FC . =S . D . U . QED BC 1)
(. E . U . FC . =S . D . U . QED BC 1)
x[size (x)] = p
Proved
(. E . U . FC . =S . D . U . QED BC 2)
(. E . U . FC . =S . D . U . QED BC 2)
repeats (x)
Proved
(. E . U . FC . =S . D . U . QED BC 3)
(. E . U . FC . =S . D . U . QED BC 3)
null (pkt_seq) ne x
Proved
(. E . U . FC . =S . D . U . QED BC)
x[size (x)] = p & repeats (x) & null (pkt_seq) ne x
Proved
QED
Prvr-> $Proceeding
prop_trans_3 proved in theorem prover.

```
Exec-> prove prop_trans_4
Entering Prover with lemma prop_trans_4
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
proper_transmission (u, x, y, 1) & p.seqno
ne next_seqnum (y)
-> proper_transmission (u, x, y  [seq: p], 1)
H1: proper_transmission (u, x, y, 1)
H2: p.seqno ne next_seqnum (y)
->
C1: proper_transmission (u, x, y  [seq: p], 1)
Backup point
(.)
Prvr-> expand proper_transmission
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
H1: msg_lag (unique_msg (x), u, 1)
H2: repeats (x)
H3: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H4: size (u) - size (unique_msg (y)) in [0..1]
H5: p.seqno ne next_seqnum (y)
->
C1:  size (unique_msg (x)) - size (unique_msg (y  [seq: p]))
in [0..1]
C2: size (u) - size (unique_msg (y  [seq: p])) in [0..1]
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? show-choices
1: unique_msg (x)
2: unique_msg (y)
3: unique_msg (y  [seq: p])
Which do you want to expand? 3
Backup point
(. E . E .)
Prvr-> simplify theorem
Prvr->theorem
C1: true
```

```
Prvr-> $Proceeding
prop_trans_4 proved in theorem prover.
[Your LOG file is job11.LOG;T;P777700]
[Recording initiated at Sat 9-May-81 12:17:42]
prove prop_trans_5
Entering Prover with lemma prop_trans_5
all u : msg_seq,
all x, y : pkt_seq,
all m : message,
proper_transmission (u, x, y, 0)
-> proper_transmission (u  [seq: m], x, y, 1)
H1: proper_transmission (u, x, y, 0)
->
C1: proper_transmission (u  [seq: m], x, y, 1)
Backup point
(.)
Prvr-> expand proper_transmission
More than one to expand.
Which do you want to expand?all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) = size (unique_msg (x))
H2: size (unique_msg (y)) = size (u)
H3: msg_lag (unique_msg (x), u, 0)
H4: repeats (x)
->
C1: msg_lag (unique_msg (x), u  [seq: m], 1)
C2: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
C3: (size (u) - size (unique_msg (y))) + 1 in [0..1]
Prvr-> equality substitute h1,h2
H1 can be solved for:
T1: size (unique_msg (y))
T2: size (unique_msg (x))
Which term (by label) do you want to substitute for?
t2
size (unique_msg (x)) := size (unique_msg (y))
OK??
y
H2 can be solved for:
T1: size (unique_msg (y))
```

T2: size (u)
Which term (by label) do you want to substitute for?
t2
size (u) := size (unique_msg (y))
OK??
y
Backup point
(. E . =S .)
Prvr-> theorem
H1: msg_lag (unique_msg (x), u, 0)
H2: repeats (x)
->
C1: msg_lag (unique_msg (x), u [seq: m], 1)
Prvr-> retain h1
Backup point
(. E . =S . D .)
Prvr-> use eq_iss
Backup point
(. E . =S . D . U .)
Prvr-> hypothesis
H1: u#8$ = v#8$ -> initial_subseq (u#8$, v#8$)
H2: msg_lag (unique_msg (x), u, 0)
Prvr-> forwardchain h1
::Forward chaining gives
initial_subseq (v#8$, v#8$)
Backup point
(. E . =S . D . U . FC .)
Prvr-> back up 3
Restored
Backup point
(. E . =S . D .)
Prvr-> use msg_lag_eq
Backup point
(. E . =S . D . U .)
Prvr-> theorem
H1: u#10$ = v#10$ iff msg_lag (u#10$, v#10$, 0)
H2: msg_lag (unique_msg (x), u, 0)
->
C1: msg_lag (unique_msg (x), u [seq: m], 1)
Prvr-> forwardchain h1
Which way? (left or right)
left

::Forward chaining gives
unique_msg (x) = u
Backup point
(. E . =S . D . U . FC .)
Prvr-> equality substitute h1
H1 can be solved for:
T1: u#10$ = v#10$
T2: msg_lag (u#10$, v#10$, 0)
Which term (by label) do you want to substitute for?
t1
u#10$ = v#10$:= msg_lag (u#10$, v#10$, 0)
OK??
y
Backup point
(. E . =S . D . U . FC . =S .)
Prvr-> theorem
H1: unique_msg (x) = u
H2: msg_lag (unique_msg (x), u, 0)
->
C1: msg_lag (unique_msg (x), u [seq: m], 1)
Prvr-> equality substitute h1
H1 can be solved for:
T1: unique_msg (x)
T2: u
Which term (by label) do you want to substitute for?
t1
unique_msg (x) := u
OK??
y
Backup point
(. E . =S . D . U . FC . =S . =S .)
Prvr-> expand msg_lag
More than one to expand.
Which do you want to expand? 2
Backup point
(. E . =S . D . U . FC . =S . =S . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: msg_lag (u, u, 0)
->
C1: initial_subseq (u, u [seq: m])
Prvr-> use eq_iss

Backup point
(. E . =S . D . U . FC . =S . =S . E . U .)
Prvr-> use iss_app
Backup point
(. E . =S . D . U . FC . =S . =S . E . U . U .)
Prvr->qed
(. E . =S . D . U . FC . =S . =S . E . U . U . QED)
:::.(. E . =S . D . U . FC . =S . =S . E . U . U . QED BC)
::.(. E . =S . D . U . FC . =S . =S . E . U . U . QED BC BC)
(. E . =S . D . U . FC . =S . =S . E . U . U . QED BC BC)
u = u
Proved
(. E . =S . D . U . FC . =S . =S . E . U . U . QED BC)
initial_subseq (u, u)
Proved
QED
Prvr-> $Proceeding
prop_trans_5 proved in theorem prover.
Exec-> prove prop_trans_6
Entering Prover with lemma prop_trans_6
all u : msg_seq,
all x, y : pkt_seq,
proper_transmission (u, x, y, 0) -> proper_transmission (u, x, y, 1)
H1: proper_transmission (u, x, y, 0)
->
C1: proper_transmission (u, x, y, 1)
Backup point
(.)
Prvr-> expand proper_transmission
More than one to expand.
Which do you want to expand?all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) = size (unique_msg (x))
H2: size (unique_msg (y)) = size (u)
H3: msg_lag (unique_msg (x), u, 0)
H4: repeats (x)
->
C1: msg_lag (unique_msg (x), u, 1)
C2: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]

C3: size (u) - size (unique_msg (y)) in [0..1]
Prvr-> equality substitute h1,h2
H1 can be solved for:
T1: size (unique_msg (y))
T2: size (unique_msg (x))
Which term (by label) do you want to substitute for?
t2
size (unique_msg (x)) := size (unique_msg (y))
OK??
y
H2 can be solved for:
T1: size (unique_msg (y))
T2: size (u)
Which term (by label) do you want to substitute for?
t2
size (u) := size (unique_msg (y))
OK??
y
Backup point
(. E . =S .)
Prvr-> theorem
H1: msg_lag (unique_msg (x), u, 0)
H2: repeats (x)
->
C1: msg_lag (unique_msg (x), u, 1)
Prvr-> retain h1
Backup point
(. E . =S . D .)
Prvr-> expand msg_lag
More than one to expand.
Which do you want to expand? all
Backup point
(. E . =S . D . E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (x)) = size (u)
H2: initial_subseq (unique_msg (x), u)
->
C1: size (u) - size (unique_msg (x)) in [0..1]
Prvr-> equality substitute h1
H1 can be solved for:
T1: size (unique_msg (x))

T2: size (u)
Which term (by label) do you want to substitute for?
t1
size (unique_msg (x)) := size (u)
OK??
y
Backup point
(. E . =S . D . E . =S .)
Prvr-> $Proceeding
prop_trans_6 proved in theorem prover.
Exec-> prove prop_trans_7
Entering Prover with lemma prop_trans_7
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
p.seqno = next_seqnum (y) & unique_msg (x) = u
& size (unique_msg (x)) = size (unique_msg (y)) + 1
& proper_transmission (u, x, y, 1)
-> proper_transmission (u, x, y [seq: p], 0)
H1: p.seqno = next_seqnum (y)
H2: unique_msg (x) = u
H3: size (unique_msg (x)) = size (unique_msg (y)) + 1
H4: proper_transmission (u, x, y, 1)
->
C1: proper_transmission (u, x, y [seq: p], 0)
Typelist equalities
size (unique_msg (y)) + 1 = size (unique_msg (x))
Backup point
(.)
Prvr-> expand proper_transmission
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: p.seqno = next_seqnum (y)
H2: unique_msg (x) = u
H3: size (unique_msg (x)) = size (unique_msg (y)) + 1
H4: size (unique_msg (y)) + 1 = size (unique_msg (x))
H5: msg_lag (unique_msg (x), u, 1)
H6: repeats (x)

H7: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H8: size (u) - size (unique_msg (y)) in [0..1]
->
C1: size (unique_msg (y [seq: p])) = size (unique_msg (x))
C2: size (unique_msg (y [seq: p])) = size (u)
C3: msg_lag (unique_msg (x), u, 0)
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand?show-choices
1: unique_msg (x)
2: unique_msg (y)
3: unique_msg (y [seq: p])
Which do you want to expand? 3
Backup point
(. E . E .)
Prvr->simplify theorem
Prvr->theorem
H1: p.seqno = next_seqnum (y)
H2: unique_msg (x) = u
H3: size (unique_msg (x)) = size (unique_msg (y)) + 1
H4: size (unique_msg (y)) + 1 = size (unique_msg (x))
H5: msg_lag (unique_msg (x), u, 1)
H6: repeats (x)
H7: size (unique_msg (x)) - size (unique_msg (y)) in [0..1]
H8: size (u) - size (unique_msg (y)) in [0..1]
->
C1: size (unique_msg (y)) + 1 = size (u)
C2: msg_lag (unique_msg (x), u, 0)
Prvr-> equality substitute h2
H2 can be solved for:
T1: unique_msg (x)
T2: u
Which term (by label) do you want to substitute for?
t1
unique_msg (x) := u
OK??
y
Backup point
(. E . E . =S .)
Prvr-> theorem
H1: p.seqno = next_seqnum (y)
H2: size (unique_msg (y)) + 1 = size (u)

H3: size (u) = size (unique_msg (y)) + 1
H4: msg_lag (u, u, 1)
H5: repeats (x)
H6: size (u) - size (unique_msg (y)) in [0..1]
->
C1: size (unique_msg (y)) + 1 = size (u)
C2: msg_lag (u, u, 0)
Prvr-> retain
What hypotheses would you like to retain? h4
Backup point
(. E . E . =S . D .)
Prvr-> simplify theorem
Prvr->theorem
H1: msg_lag (u, u, 1)
->
C1: size (unique_msg (y)) + 1 = size (u)
C2: msg_lag (u, u, 0)
Prvr-> back up2
**** Illegal command - UP2
Back UP or establish a back up POINT to return to? up
How many levels would you like to back up? 2
Restored
Backup point
(. E . E . =S .)
Prvr-> theorem
H1: p.seqno = next_seqnum (y)
H2: size (unique_msg (y)) + 1 = size (u)
H3: size (u) = size (unique_msg (y)) + 1
H4: msg_lag (u, u, 1)
H5: repeats (x)
H6: size (u) - size (unique_msg (y)) in [0..1]
->
C1: size (unique_msg (y)) + 1 = size (u)
C2: msg_lag (u, u, 0)
Prvr-> retain h2
Typelist equalities
size (u) = size (unique_msg (y)) + 1
Backup point
(. E . E . =S . D .)
Prvr-> simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) + 1 = size (u)

H2: size (u) = size (unique_msg (y)) + 1
->
C1: msg_lag (u, u, 0)
Prvr-> use msg_lag_eq
Backup point
(. E . E . =S . D . U .)
Prvr-> qed
(. E . E . =S . D . U . QED)
:::..
size (unique_msg (y)) := size (u) - 1
(. E . E . =S . D . U . QED =S)
(. E . E . =S . D . U . QED =S E)
Ran out of tricks
Prvr-> theorem
H1: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
->
C1: initial_subseq (u, u)
Prvr-> back up 4
Restored
Backup point
(. E . E . =S . D . U .)
Prvr-> theorem
H1: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H2: size (unique_msg (y)) + 1 = size (u)
H3: size (u) = size (unique_msg (y)) + 1
->
C1: msg_lag (u, u, 0)
Prvr-> use eq_iss
Backup point
(. E . E . =S . D . U . U .)
Prvr-> hypothesis
H1: u#6$ = v#6$ -> initial_subseq (u#6$, v#6$)
H2: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H3: size (unique_msg (y)) + 1 = size (u)
H4: size (u) = size (unique_msg (y)) + 1
Prvr-> retain h1
Backup point
(. E . E . =S . D . U . U . D .)
Prvr-> expand msg_lag
Backup point
(. E . E . =S . D . U . U . D . E .)
Prvr-> qed

(. E . E . =S . D . U . U . D . E . QED)
(. E . E . =S . D . U . U . D . E . QED BC)
(. E . E . =S . D . U . U . D . E . QED BC)
u = u
Proved
QED
Prvr-> $Proceeding
prop_trans_7 proved in theorem prover.
Exec-> show status scope $
SCOPE LEMMAS
Waiting to be proved: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4,
PROP_REC_5, SUB_APP, SIZE_NULL, SUB_SEQNUM, SUB_TO_LAG, SUB_NCHANGES
Proved: PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4,
PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7
Exec-> prove prop_rec_1
Entering Prover with lemma prop_rec_1
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
all m : message,
p.mssg = m & p.seqno = next_seqnum (x)
& proper_reception (u, x, y, 0)
-> proper_reception (u [seq: m], x [seq: p], y, 1)
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: proper_reception (u, x, y, 0)
->
C1: proper_reception (u [seq: m], x [seq: p], y, 1)
Backup point
(.)
Prvr-> expand proper_reception
More than one to expand.
Which do you want to expand?all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: p.mssg = m

H2: p.seqno = next_seqnum (x)
H3: size (unique_msg (y)) = size (u)
H4: msg_lag (u, unique_msg (x), 0)
H5: y sub x
->
C1: msg_lag (u [seq: m], unique_msg (x [seq: p]), 1)
C2: (size (u) - size (unique_msg (y))) + 1 in [0..1]
Prvr-> equality substitute h3
H3 can be solved for:
T1: size (unique_msg (y))
T2: size (u)
Which term (by label) do you want to substitute for?
t1
size (unique_msg (y)) := size (u)
OK??
y
Backup point
(. E . =S .)
Prvr-> theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: msg_lag (u, unique_msg (x), 0)
H4: y sub x
->
C1: msg_lag (u [seq: m], unique_msg (x [seq: p]), 1)
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? 2
Backup point
(. E . =S . E .)
Prvr-> use msg_lag_eq
Backup point
(. E . =S . E . U .)
Prvr-> theorem
H1: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H2: p.mssg = m
H3: p.seqno = next_seqnum (x)
H4: msg_lag (u, unique_msg (x), 0)
H5: y sub x
->
C1: msg_lag (u [seq: m],
if p.seqno = next_seqnum (x)

then unique_msg (x) [seq: p.mssg]
else unique_msg (x)
fi, 1)
Prvr-> forwardchain h1
Which way? (left or right)
left
::::Forward chaining gives
u = unique_msg (x)
Backup point
(. E . =S . E . U . FC .)
Prvr-> simplify theorem
Prvr->theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: unique_msg (x) = u
H4: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H5: msg_lag (u, unique_msg (x), 0)
H6: y sub x
->
C1: msg_lag (u [seq: m], unique_msg (x) [seq: p.mssg], 1)
Prvr-> retain h1,h3
Backup point
(. E . =S . E . U . FC . D .)
Prvr-> hypothesis
H1: p.mssg = m
H2: unique_msg (x) = u
Prvr-> equality substitute h1,h2
H1 can be solved for:
T1: p.mssg
T2: m
Which term (by label) do you want to substitute for?
t1
p.mssg := m
OK??
y
H2 can be solved for:
T1: unique_msg (x)
T2: u
Which term (by label) do you want to substitute for?
t1
unique_msg (x) := u
OK??

y
Backup point
(. E . =S . E . U . FC . D . =S .)
Prvr-> theorem
H1: true
->
C1: msg_lag (u [seq: m], u [seq: m], 1)
Prvr-> expand msg_lag
Backup point
(. E . =S . E . U . FC . D . =S . E .)
Prvr->use eq_iss
Backup point
(. E . =S . E . U . FC . D . =S . E . U .)
Prvr-> qed
(. E . =S . E . U . FC . D . =S . E . U . QED)
(. E . =S . E . U . FC . D . =S . E . U . QED BC)
(. E . =S . E . U . FC . D . =S . E . U . QED BC)
u [seq: m] = u [seq: m]
Proved
QED
Prvr-> $Proceeding
prop_rec_1 proved in theorem prover.
Exec-> prove prop_rec_2
Entering Prover with lemma prop_rec_2
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
proper_reception (u, x, y, 0)
-> proper_reception (u, x [seq: p], y, 1)
H1: proper_reception (u, x, y, 0)
->
C1: proper_reception (u, x [seq: p], y, 1)
Backup point
(.)
Prvr-> expand proper_reception
More than one to expand.
Which do you want to expand?all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) = size (u)

H2: msg_lag (u, unique_msg (x), 0)
H3: y sub x
->
C1: msg_lag (u, unique_msg (x [seq: p]), 1)
C2: size (u) - size (unique_msg (y)) in [0..1]
Prvr-> equality substitute h1
H1 can be solved for:
T1: size (unique_msg (y))
T2: size (u)
Which term (by label) do you want to substitute for?
t1
size (unique_msg (y)) := size (u)
OK??
y
Backup point
(. E . =S .)
Prvr-> theorem
H1: msg_lag (u, unique_msg (x), 0)
H2: y sub x
->
C1: msg_lag (u, unique_msg (x [seq: p]), 1)
Prvr-> retain h1
Backup point
(. E . =S . D .)
Prvr-> use msg_lag_eq
Backup point
(. E . =S . D . U .)
Prvr-> hypothesis
H1: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H2: msg_lag (u, unique_msg (x), 0)
Prvr-> forwardchain h1
Which way? (left or right)
left
::Forward chaining gives
u = unique_msg (x)
Backup point
(. E . =S . D . U . FC .)
Prvr-> hypothesis
H1: u = unique_msg (x)
H2: u#2$ = v#2$ iff msg_lag (u#2$, v#2$, 0)
H3: msg_lag (u, unique_msg (x), 0)
Prvr-> retain h1

Backup point
(. E . =S . D . U . FC . D .)
Prvr-> equality substitute h1
H1 yields
u := unique_msg (x)
Backup point
(. E . =S . D . U . FC . D . =S .)
Prvr-> theorem
H1: true
->
C1: msg_lag (unique_msg (x), unique_msg (x [seq: p]), 1)
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand?show-choices
1: unique_msg (x)
2: unique_msg (x [seq: p])
Which do you want to expand? 2
Backup point
(. E . =S . D . U . FC . D . =S . E .)
Prvr-> theorem
H1: true
->
C1: msg_lag (unique_msg (x),
if p.seqno = next_seqnum (x)
then unique_msg (x) [seq: p.mssg]
else unique_msg (x)
fi, 1)
Prvr-> cases
Case:
* p.seqno=next_seqnum(x);
Case:
* $done
Attempting CASE1
p.seqno = next_seqnum (x)
Backup point
(. E . =S . D . U . FC . D . =S . E . CASE1 .)
Prvr-> theorem
H1: p.seqno = next_seqnum (x)
->
C1: msg_lag (unique_msg (x),
if p.seqno = next_seqnum (x)
then unique_msg (x) [seq: p.mssg]

```
else unique_msg (x)
fi, 1)
Prvr-> simplify theorem
Prvr->theorem
H1: p.seqno = next_seqnum (x)
->
C1: msg_lag (unique_msg (x), unique_msg (x)  [seq: p.mssg], 1)
Prvr-> use eq_iss
Backup point
(. E . =S . D . U . FC . D . =S . E . CASE1 . U .)
Prvr->use iss_app
Backup point
(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U .)
Prvr-> qed
(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U . QED)
:::...(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U . QED E)
:::.(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U . QED E BC)
::.(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U . QED E BC BC)
(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U . QED E BC BC)
unique_msg (x) = unique_msg (x)
Proved
(. E . =S . D . U . FC . D . =S . E . CASE1 . U . U . QED E BC)
initial_subseq (unique_msg (x), unique_msg (x))
Proved
QED
Prvr-> $Proceeding
(. E . =S . D . U . FC . D . =S . E . CASE1 .)
p.seqno = next_seqnum (x)
-> msg_lag (unique_msg (x),
if p.seqno = next_seqnum (x)
then unique_msg (x)  [seq: p.mssg]
else unique_msg (x)
fi, 1)
Proved
Attempting CASE2
p.seqno ne next_seqnum (x)
Backup point
(. E . =S . D . U . FC . D . =S . E . CASE2 .)
Prvr-> theorem
H1: p.seqno ne next_seqnum (x)
->
C1: msg_lag (unique_msg (x),
```

```
if p.seqno = next_seqnum (x)
then unique_msg (x)  [seq: p.mssg]
else unique_msg (x)
fi, 1)
Prvr-> simplify theorem
Prvr->theorem
H1: p.seqno ne next_seqnum (x)
->
C1: msg_lag (unique_msg (x), unique_msg (x), 1)
Prvr-> use eq_iss
Backup point
(. E . =S . D . U . FC . D . =S . E . CASE2 . U .)
Prvr-> qed
(. E . =S . D . U . FC . D . =S . E . CASE2 . U . QED)
::..(. E . =S . D . U . FC . D . =S . E . CASE2 . U . QED E)
::.(. E . =S . D . U . FC . D . =S . E . CASE2 . U . QED E BC)
(. E . =S . D . U . FC . D . =S . E . CASE2 . U . QED E BC)
unique_msg (x) = unique_msg (x)
Proved
QED
Prvr-> $Proceeding
(. E . =S . D . U . FC . D . =S . E . CASE2 .)
p.seqno ne next_seqnum (x)
-> msg_lag (unique_msg (x),
if p.seqno = next_seqnum (x)
then unique_msg (x)  [seq: p.mssg]
else unique_msg (x)
fi, 1)
Proved
Done with cases
prop_rec_2 proved in theorem prover.
Exec->$
**** Illegal command - <ESCAPE>
Exec-> prove prop_rec_3
Entering Prover with lemma prop_rec_3
all u : msg_seq,
all x, y : pkt_seq,
proper_reception (u, x, y, 0) -> proper_reception (u, x, y, 1)
H1: proper_reception (u, x, y, 0)
->
C1: proper_reception (u, x, y, 1)
Backup point
```

(.)
Prvr-> expand proper_reception
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) = size (u)
H2: msg_lag (u, unique_msg (x), 0)
H3: y sub x
->
C1: msg_lag (u, unique_msg (x), 1)
C2: size (u) - size (unique_msg (y)) in [0..1]
Prvr-> equality substitute h1
H1 can be solved for:
T1: size (unique_msg (y))
T2: size (u)
Which term (by label) do you want to substitute for?
t1
size (unique_msg (y)) := size (u)
OK??
y
Backup point
(. E . =S .)
Prvr-> theorem
H1: msg_lag (u, unique_msg (x), 0)
H2: y sub x
->
C1: msg_lag (u, unique_msg (x), 1)
Prvr-> expand msg_lag
More than one to expand.
Which do you want to expand? all
Backup point
(. E . =S . E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (u) = size (unique_msg (x))
H2: initial_subseq (u, unique_msg (x))
H3: y sub x
->
C1: size (unique_msg (x)) - size (u) in [0..1]

```
Prvr-> equality substitute h1
H1 can be solved for:
T1: size (u)
T2: size (unique_msg (x))
Which term (by label) do you want to substitute for?
t1
size (u) := size (unique_msg (x))
OK??
y
Backup point
(. E . =S . E . =S .)
Prvr-> $Proceeding
prop_rec_3 proved in theorem prover.
Exec->$
**** Illegal command - <ESCAPE>
Exec-> prove prop_rec_4
Entering Prover with lemma prop_rec_4
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
all m : message,
p.mssg = m & p.seqno = next_seqnum (x)
& p.seqno = next_seqnum (y) & proper_reception (u, x, y, 0)
-> proper_reception (u  [seq: m], x  [seq: p], y  [seq: p], 0)
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: p.seqno = next_seqnum (y)
H4: proper_reception (u, x, y, 0)
->
C1: proper_reception (u  [seq: m], x  [seq: p], y  [seq: p], 0)
Backup point
(.)
Prvr-> expand proper_reception
More than one to expand.
Which do you want to expand?all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: p.seqno = next_seqnum (y)
```

H4: size (unique_msg (y)) = size (u)
H5: msg_lag (u, unique_msg (x), 0)
H6: y sub x
->
C1: size (unique_msg (y [seq: p])) = size (u) + 1
C2: msg_lag (u [seq: m], unique_msg (x [seq: p]), 0)
C3: y [seq: p] sub x [seq: p]
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand?show-choices
1: unique_msg (y)
2: unique_msg (x)
3: unique_msg (y [seq: p])
4: unique_msg (x [seq: p])
Which do you want to expand? 3,4
Backup point
(. E . E .)
Prvr->simplify theorem
Prvr->theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: p.seqno = next_seqnum (y)
H4: size (unique_msg (y)) = size (u)
H5: msg_lag (u, unique_msg (x), 0)
H6: y sub x
->
C1: msg_lag (u [seq: m], unique_msg (x) [seq: p.mssg], 0)
C2: y [seq: p] sub x [seq: p]
Prvr-> use eq_iss
Backup point
(. E . E . U .)
Prvr-> hypothesis
H1: u#2$ = v#2$ -> initial_subseq (u#2$, v#2$)
H2: p.mssg = m
H3: p.seqno = next_seqnum (x)
H4: p.seqno = next_seqnum (y)
H5: size (unique_msg (y)) = size (u)
H6: msg_lag (u, unique_msg (x), 0)
H7: y sub x
Prvr-> forwardchain h1
:::Forward chaining gives
initial_subseq (p.seqno, next_seqnum (x))

Backup point
(. E . E . U . FC .)
Prvr-> back up 2
Restored
Backup point
(. E . E . U .)
Prvr-> back up 2
Restored
Backup point
(. E . E .)
Prvr-> use msg_lag_eq
Backup point
(. E . E . U .)
Prvr-> hypothesis
H1: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H2: p.mssg = m
H3: p.seqno = next_seqnum (x)
H4: p.seqno = next_seqnum (y)
H5: size (unique_msg (y)) = size (u)
H6: msg_lag (u, unique_msg (x), 0)
H7: y sub x
Prvr->forwardchain h1
Which way? (left or right)
left
::::::Forward chaining gives
u = unique_msg (x)
Backup point
(. E . E . U . FC .)
Prvr-> hypothesis
H1: u = unique_msg (x)
H2: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H3: p.mssg = m
H4: p.seqno = next_seqnum (x)
H5: p.seqno = next_seqnum (y)
H6: size (unique_msg (y)) = size (u)
H7: msg_lag (u, unique_msg (x), 0)
H8: y sub x
Prvr-> equality substitute h1
H1 can be solved for:
T1: u
T2: unique_msg (x)
Which term (by label) do you want to substitute for?

t2
unique_msg (x) := u
OK??
y
Typelist equalities
size (u) = size (unique_msg (y))
Backup point
(. E . E . U . FC . =S .)
Prvr->theorem
H1: size (u) = size (unique_msg (y))
H2: p.mssg = m
H3: p.seqno = next_seqnum (x)
H4: p.seqno = next_seqnum (y)
H5: size (unique_msg (y)) = size (u)
H6: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H7: msg_lag (u, u, 0)
H8: y sub x
->
C1: size (if p.seqno = next_seqnum (y)
then unique_msg (y) [seq: p.mssg]
else unique_msg (y)
fi)
= size (u) + 1
C2: msg_lag (u [seq: m],
if p.seqno = next_seqnum (x)
then u [seq: p.mssg]
else u
fi, 0)
C3: y [seq: p] sub x [seq: p]
Prvr-> simplify theorem
Prvr->theorem
H1: p.mssg = m
H2: p.seqno = next_seqnum (x)
H3: p.seqno = next_seqnum (y)
H4: size (unique_msg (y)) = size (u)
H5: size (u) = size (unique_msg (y))
H6: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H7: msg_lag (u, u, 0)
H8: y sub x
->
C1: msg_lag (u [seq: m], u [seq: p.mssg], 0)
C2: y [seq: p] sub x [seq: p]

Prvr-> equality substitute h1
H1 can be solved for:
T1: p.mssg
T2: m
Which term (by label) do you want to substitute for?
t1
p.mssg := m
OK??
y
Backup point
(. E . E . U . FC . =S . =S .)
Prvr->hypothesis
H1: p.seqno = next_seqnum (x)
H2: p.seqno = next_seqnum (y)
H3: size (unique_msg (y)) = size (u)
H4: size (u) = size (unique_msg (y))
H5: u#4$ = v#4$ iff msg_lag (u#4$, v#4$, 0)
H6: msg_lag (u, u, 0)
H7: y sub x
Prvr-> retain h6,h7
Backup point
(. E . E . U . FC . =S . =S . D .)
Prvr-> $Proceeding
Backup point
(. E . E . U . FC . =S . =S . D . 1 .)
Prvr-> theorem
H1: msg_lag (u, u, 0)
H2: y sub x
->
C1: msg_lag (u [seq: m], u [seq: m], 0)
Prvr-> use eq_iss_app
Typelist equalities
size (u) = size (unique_msg (y))
Backup point
(. E . E . U . FC . =S . =S . D . 1 . U .)
Prvr->expand msg_lag
More than one to expand.
Which do you want to expand? all
Backup point
(. E . E . U . FC . =S . =S . D . 1 . U . E .)
Prvr->qed
(. E . E . U . FC . =S . =S . D . 1 . U . E . QED)

:::::..(. E . E . U . FC . =S . =S . D . 1 . U . E . QED BC)
(. E . E . U . FC . =S . =S . D . 1 . U . E . QED BC)
u = u
Proved
QED
Prvr-> $Proceeding
(. E . E . U . FC . =S . =S . D . 1 .)
msg_lag (u [seq: m], u [seq: m], 0)
Proved
Prvr-> $Proceeding
Backup point
(. E . E . U . FC . =S . =S . D . 2 .)
Prvr-> theorem
H1: msg_lag (u, u, 0)
H2: y sub x
->
C1: y [seq: p] sub x [seq: p]
Prvr-> retain h2
Backup point
(. E . E . U . FC . =S . =S . D . 2 . D .)
Prvr-> simplify theorem
Prvr->theorem
H1: y sub x
->
C1: y [seq: p] sub x [seq: p]
Prvr-> use sub_app
Typelist equalities
size (u) = size (unique_msg (y))
Backup point
(. E . E . U . FC . =S . =S . D . 2 . D . U .)
Prvr->qed
(. E . E . U . FC . =S . =S . D . 2 . D . U . QED)
:::..(. E . E . U . FC . =S . =S . D . 2 . D . U . QED BC)
(. E . E . U . FC . =S . =S . D . 2 . D . U . QED BC)
y sub x
Proved
QED
Prvr-> $Proceeding
(. E . E . U . FC . =S . =S . D . 2 .)
y [seq: p] sub x [seq: p]
Proved
Prvr-> $Proceeding

prop_rec_4 proved in theorem prover.
Exec-> prove prop_rec_5
Entering Prover with lemma prop_rec_5
all u : msg_seq,
all x, y : pkt_seq,
all p : packet,
proper_reception (u, x, y, 0) & p.seqno ne next_seqnum (x)
& p.seqno ne next_seqnum (y)
-> proper_reception (u, x [seq: p], y [seq: p], 0)
H1: proper_reception (u, x, y, 0)
H2: p.seqno ne next_seqnum (x)
H3: p.seqno ne next_seqnum (y)
->
C1: proper_reception (u, x [seq: p], y [seq: p], 0)
Backup point
(.)
Prvr-> expand proper_reception
More than one to expand.
Which do you want to expand?all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) = size (u)
H2: msg_lag (u, unique_msg (x), 0)
H3: p.seqno ne next_seqnum (x)
H4: p.seqno ne next_seqnum (y)
H5: y sub x
->
C1: size (unique_msg (y [seq: p])) = size (u)
C2: msg_lag (u, unique_msg (x [seq: p]), 0)
C3: y [seq: p] sub x [seq: p]
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand?show-choices
1: unique_msg (y)
2: unique_msg (x)
3: unique_msg (y [seq: p])
4: unique_msg (x [seq: p])
Which do you want to expand? 3,4
Backup point
(. E . E .)

```
Prvr->simplify theorem
Prvr->theorem
H1: size (unique_msg (y)) = size (u)
H2: msg_lag (u, unique_msg (x), 0)
H3: p.seqno ne next_seqnum (x)
H4: p.seqno ne next_seqnum (y)
H5: y sub x
->
C1: y  [seq: p] sub x  [seq: p]
Prvr-> retain h5
Backup point
(. E . E . D .)
Prvr-> use sub_app
Backup point
(. E . E . D . U .)
Prvr->qed
(. E . E . D . U . QED)
::.(. E . E . D . U . QED BC)
(. E . E . D . U . QED BC)
y sub x
Proved
QED
Prvr-> $Proceeding
prop_rec_5 proved in theorem prover.
Exec-> show status scope $
SCOPE LEMMAS
Waiting to be proved: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL,
APP_PKT_NONNULL, BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB,
ISS_APP, ISS_TRANS, INTERPOLATE, LAST_NEXT, LAST_UNIQUE,
LAST_REPEATS, MAIN_LEMMA, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, SUB_APP, SIZE_NULL, SUB_SEQNUM, SUB_TO_LAG,
SUB_NCHANGES
Proved: PROP_REC_1, PROP_REC_2, PROP_REC_3, PROP_REC_4, PROP_REC_5,
PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4, PROP_TRANS_5,
PROP_TRANS_6, PROP_TRANS_7
Exec-> save abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec->
```

```
prove abp_1
Entering Prover with lemma abp_1
all u, v : msg_seq,
all w, x, y, z : pkt_seq,
proper_reception (v, x, y, 1) & proper_transmission (u, w, z, 1)
& x sub w & z sub y
-> msg_lag (v, u, 1)
H1: proper_reception (v, x, y, 1)
H2: proper_transmission (u, w, z, 1)
H3: x sub w
H4: z sub y
->
C1: msg_lag (v, u, 1)
Backup point
(.)
Prvr-> expand proper_reception
Backup point
(. E .)
Prvr-> expand proper_transmission
Backup point
(. E . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: msg_lag (unique_msg (w), u, 1)
H2: msg_lag (v, unique_msg (x), 1)
H3: repeats (w)
H4: size (v) - size (unique_msg (y)) in [0..1]
H5: size (unique_msg (w)) - size (unique_msg (z)) in [0..1]
H6: size (u) - size (unique_msg (z)) in [0..1]
H7: x sub w
H8: y sub x
H9: z sub y
->
C1: msg_lag (v, u, 1)
Prvr-> use nchanges_unique
Backup point
(. E . E . U .)
Prvr->use nchanges_unique
Backup point
(. E . E . U . U .)
Prvr-> hypothesis
H1: size (unique_msg (x#4$)) = nchanges (seqnums (x#4$))
```

H2: size (unique_msg (x#2$)) = nchanges (seqnums (x#2$))
H3: msg_lag (unique_msg (w), u, 1)
H4: msg_lag (v, unique_msg (x), 1)
H5: repeats (w)
H6: size (v) - size (unique_msg (y)) in [0..1]
H7: size (unique_msg (w)) - size (unique_msg (z)) in [0..1]
H8: size (u) - size (unique_msg (z)) in [0..1]
H9: x sub w
H10: y sub x
H11: z sub y
Prvr-> put
For what?* x#4$;
Put what?* w;
For what?* x#2$;
Put what?* z;
For what?* $done
Backup point
(. E . E . U . U . PUT .)
Prvr-> hypothesis
H1: size (unique_msg (z)) = nchanges (seqnums (z))
H2: size (unique_msg (w)) = nchanges (seqnums (w))
H3: z sub y
H4: y sub x
H5: x sub w
H6: size (u) - size (unique_msg (z)) in [0..1]
H7: size (unique_msg (w)) - size (unique_msg (z)) in [0..1]
H8: size (v) - size (unique_msg (y)) in [0..1]
H9: repeats (w)
H10: msg_lag (v, unique_msg (x), 1)
H11: msg_lag (unique_msg (w), u, 1)
Prvr-> equality substitute h1,h2
H1 yields
size (unique_msg (z)) := nchanges (seqnums (z))
H2 yields
size (unique_msg (w)) := nchanges (seqnums (w))
Backup point
(. E . E . U . U . PUT . =S .)
Prvr-> theorem
H1: msg_lag (unique_msg (w), u, 1)
H2: msg_lag (v, unique_msg (x), 1)
H3: repeats (w)
H4: nchanges (seqnums (w)) - nchanges (seqnums (z))

in [0..1]
H5: size (u) - nchanges (seqnums (z)) in [0..1]
H6: size (v) - size (unique_msg (y)) in [0..1]
H7: x sub w
H8: y sub x
H9: z sub y
->
C1: msg_lag (v, u, 1)
Prvr-> claim
Newgoal:
* z sub x;
Defer proof of claim? y
Backup point
(. E . E . U . U . PUT . =S . .)
Prvr-> use sub_seqnum
Backup point
(. E . E . U . U . PUT . =S . . U .)
Prvr-> hypothesis
H1: x#6$ sub y#2$ -> seqnums (x#6$) sub seqnums (y#2$)
H2: z sub x
H3: msg_lag (unique_msg (w), u, 1)
H4: msg_lag (v, unique_msg (x), 1)
H5: repeats (w)
H6: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H7: size (u) - nchanges (seqnums (z)) in [0..1]
H8: size (v) - size (unique_msg (y)) in [0..1]
H9: x sub w
H10: y sub x
H11: z sub y
Prvr-> forwardchain h1
::Forward chaining gives
seqnums (z) sub seqnums (x)
Backup point
(. E . E . U . U . PUT . =S . . U . FC .)
Prvr-> use main_lemma
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U .)
Prvr-> theorem
H1: repeats (y#4$)
& nchanges (seqnums (y#4$)) - nchanges (s#2$)
in [0..1] & s#2$ sub seqnums (x#8$) & x#8$ sub y#4$

-> msg_lag (unique_msg (x#8$), unique_msg (y#4$), 1)
H2: seqnums (z) sub seqnums (x)
H3: x#6$ sub y#2$ -> seqnums (x#6$) sub seqnums (y#2$)
H4: z sub x
H5: msg_lag (unique_msg (w), u, 1)
H6: msg_lag (v, unique_msg (x), 1)
H7: repeats (w)
H8: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H9: size (u) - nchanges (seqnums (z)) in [0..1]
H10: size (v) - size (unique_msg (y)) in [0..1]
H11: x sub w
H12: y sub x
H13: z sub y
->
C1: msg_lag (v, u, 1)
Prvr-> forwardchain h1
::::::::::::::::::Forward chaining gives
msg_lag (unique_msg (x), unique_msg (w), 1)
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC .)
Prvr-> expand msg_lag
More than one to expand.
Which do you want to expand? all
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E .)
Prvr->theorem
H1: initial_subseq (unique_msg (x), unique_msg (w))
& size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H2: repeats (y#4$)
& nchanges (seqnums (y#4$)) - nchanges (s#2$)
in [0..1] & s#2$ sub seqnums (x#8$) & x#8$ sub y#4$
-> initial_subseq (unique_msg (x#8$), unique_msg (y#4$))
& size (unique_msg (y#4$)) - size (unique_msg (x#8$))
in [0..1]
H3: seqnums (z) sub seqnums (x)
H4: x#6$ sub y#2$ -> seqnums (x#6$) sub seqnums (y#2$)
H5: z sub x
H6: initial_subseq (unique_msg (w), u)
& size (u) - size (unique_msg (w)) in [0..1]
H7: initial_subseq (v, unique_msg (x))
& size (unique_msg (x)) - size (v) in [0..1]

H8: repeats (w)
H9: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H10: size (u) - nchanges (seqnums (z)) in [0..1]
H11: size (v) - size (unique_msg (y)) in [0..1]
H12: x sub w
H13: y sub x
H14: z sub y
->
C1: initial_subseq (v, u)
C2: size (u) - size (v) in [0..1]
Prvr-> delete h2,h8
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D .)
Prvr-> simplify theorem
Prvr->theorem
H1: initial_subseq (unique_msg (w), u)
H2: initial_subseq (unique_msg (x), unique_msg (w))
H3: initial_subseq (v, unique_msg (x))
H4: x#6$ sub y#2$ -> seqnums (x#6$) sub seqnums (y#2$)
H5: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H6: size (u) - nchanges (seqnums (z)) in [0..1]
H7: size (u) - size (unique_msg (w)) in [0..1]
H8: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H9: size (v) - size (unique_msg (y)) in [0..1]
H10: size (unique_msg (x)) - size (v) in [0..1]
H11: seqnums (z) sub seqnums (x)
H12: x sub w
H13: y sub x
H14: z sub x
H15: z sub y
->
C1: initial_subseq (v, u)
C2: size (u) - size (v) in [0..1]
Prvr-> $Proceeding
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 .)
Prvr-> retain h1,h2,h3
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D .)
Prvr-> use iss_trans

Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U .)
Prvr-> use iss_trans
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U .)
Prvr->theorem
H1: initial_subseq (u#4$, v#4$) & initial_subseq (v#4$, w#4$)
-> initial_subseq (u#4$, w#4$)
H2: initial_subseq (u#2$, v#2$) & initial_subseq (v#2$, w#2$)
-> initial_subseq (u#2$, w#2$)
H3: initial_subseq (unique_msg (w), u)
H4: initial_subseq (unique_msg (x), unique_msg (w))
H5: initial_subseq (v, unique_msg (x))
->
C1: initial_subseq (v, u)
Prvr-> qed
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U .
QED)
::::::(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U .
QED BC)
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 1)
::::(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . Q
ED BC 1)
initial_subseq (v, v#4$)
Proved
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 2)
:::::(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U .
QED BC 2 BC)
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 2 BC 1)
::(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED
BC 2 BC 1)
initial_subseq (unique_msg (x), v#2$)
Proved
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 2 BC 2)
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 2 BC 2)
initial_subseq (unique_msg (w), u)
Proved

(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 2 BC)
initial_subseq (unique_msg (x), v#2$) & initial_subseq (v#2$, u)
Proved
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C 2)
initial_subseq (unique_msg (x), u)
Proved
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 . D . U . U . QED B
C)
initial_subseq (v, v#4$) & initial_subseq (v#4$, u)
Proved
QED
Prvr-> $Proceeding
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 1 .)
initial_subseq (v, u)
Proved
Prvr-> $Proceeding
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 .)
Prvr-> theorem
H1: initial_subseq (unique_msg (w), u)
H2: initial_subseq (unique_msg (x), unique_msg (w))
H3: initial_subseq (v, unique_msg (x))
H4: x#6$ sub y#2$ -> seqnums (x#6$) sub seqnums (y#2$)
H5: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H6: size (u) - nchanges (seqnums (z)) in [0..1]
H7: size (u) - size (unique_msg (w)) in [0..1]
H8: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H9: size (v) - size (unique_msg (y)) in [0..1]
H10: size (unique_msg (x)) - size (v) in [0..1]
H11: seqnums (z) sub seqnums (x)
H12: x sub w
H13: y sub x
H14: z sub x
H15: z sub y
->
C1: size (u) - size (v) in [0..1]
Prvr-> delete h1,h2,h3,h4,h11,h12,h13,h14
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D .)

Prvr-> theorem
H1: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H2: size (u) - nchanges (seqnums (z)) in [0..1]
H3: size (u) - size (unique_msg (w)) in [0..1]
H4: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H5: size (v) - size (unique_msg (y)) in [0..1]
H6: size (unique_msg (x)) - size (v) in [0..1]
H7: z sub y
->
C1: size (u) - size (v) in [0..1]
Prvr-> use sub_seqnum
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U .)
Prvr-> use sub_nchanges
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U .)
Prvr-> hypothesis
H1: s#4$ sub t#2$ -> nchanges (s#4$) le nchanges (t#2$)
H2: x#10$ sub y#6$ -> seqnums (x#10$) sub seqnums (y#6$)
H3: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H4: size (u) - nchanges (seqnums (z)) in [0..1]
H5: size (u) - size (unique_msg (w)) in [0..1]
H6: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H7: size (v) - size (unique_msg (y)) in [0..1]
H8: size (unique_msg (x)) - size (v) in [0..1]
H9: z sub y
Prvr-> forwardchain h2
:::::::::Forward chaining gives
seqnums (z) sub seqnums (y)
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U . FC .)
Prvr->hypothesis
H1: seqnums (z) sub seqnums (y)
H2: s#4$ sub t#2$ -> nchanges (s#4$) le nchanges (t#2$)
H3: x#10$ sub y#6$ -> seqnums (x#10$) sub seqnums (y#6$)
H4: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H5: size (u) - nchanges (seqnums (z)) in [0..1]
H6: size (u) - size (unique_msg (w)) in [0..1]
H7: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]

H8: size (v) - size (unique_msg (y)) in [0..1]
H9: size (unique_msg (x)) - size (v) in [0..1]
H10: z sub y
Prvr->hypothesis
H1: seqnums (z) sub seqnums (y)
H2: s#4$ sub t#2$ -> nchanges (s#4$) le nchanges (t#2$)
H3: x#10$ sub y#6$ -> seqnums (x#10$) sub seqnums (y#6$)
H4: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H5: size (u) - nchanges (seqnums (z)) in [0..1]
H6: size (u) - size (unique_msg (w)) in [0..1]
H7: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H8: size (v) - size (unique_msg (y)) in [0..1]
H9: size (unique_msg (x)) - size (v) in [0..1]
H10: z sub y
Prvr-> forwardchain h2
:Forward chaining gives
nchanges (seqnums (z)) le nchanges (seqnums (y))
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U . FC .
FC .)
Prvr-> hypothesis
H1: nchanges (seqnums (z)) le nchanges (seqnums (y))
H2: seqnums (z) sub seqnums (y)
H3: s#4$ sub t#2$ -> nchanges (s#4$) le nchanges (t#2$)
H4: x#10$ sub y#6$ -> seqnums (x#10$) sub seqnums (y#6$)
H5: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H6: size (u) - nchanges (seqnums (z)) in [0..1]
H7: size (u) - size (unique_msg (w)) in [0..1]
H8: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H9: size (v) - size (unique_msg (y)) in [0..1]
H10: size (unique_msg (x)) - size (v) in [0..1]
H11: z sub y
Prvr-> delete h2,h3,h4,h11
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U . FC .
FC . D .)
Prvr->simplify theorem
Prvr->theorem
H1: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]

H2: size (u) - nchanges (seqnums (z)) in [0..1]
H3: size (u) - size (unique_msg (w)) in [0..1]
H4: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H5: size (v) - size (unique_msg (y)) in [0..1]
H6: size (unique_msg (x)) - size (v) in [0..1]
H7: nchanges (seqnums (z)) le nchanges (seqnums (y))
->
C1: size (u) - size (v) in [0..1]
Prvr-> use nchanges_unique
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U . FC .
FC . D . U .)
Prvr->put
For what?* $done
Prvr-> hypothesis
H1: size (unique_msg (x#12$)) = nchanges (seqnums (x#12$))
H2: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H3: size (u) - nchanges (seqnums (z)) in [0..1]
H4: size (u) - size (unique_msg (w)) in [0..1]
H5: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H6: size (v) - size (unique_msg (y)) in [0..1]
H7: size (unique_msg (x)) - size (v) in [0..1]
H8: nchanges (seqnums (z)) le nchanges (seqnums (y))
Prvr-> put
For what?* x#12$;
Put what?* y;
For what?* $done
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U . FC .
FC . D . U . PUT .)
Prvr->equality substitute h1
H1 no longer part of current HYPOTHESES.
**** Command Aborted
Prvr-> hypothesis
H1: size (unique_msg (y)) = nchanges (seqnums (y))
H2: nchanges (seqnums (z)) le nchanges (seqnums (y))
H3: size (unique_msg (x)) - size (v) in [0..1]
H4: size (v) - size (unique_msg (y)) in [0..1]
H5: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H6: size (u) - size (unique_msg (w)) in [0..1]
H7: size (u) - nchanges (seqnums (z)) in [0..1]

H8: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
Prvr-> equality substitute h1
H1 can be solved for:
T1: nchanges (seqnums (y))
T2: size (unique_msg (y))
Which term (by label) do you want to substitute for?
t1
nchanges (seqnums (y)) := size (unique_msg (y))
OK??
y
Backup point
(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 . D . U . U . FC .
FC . D . U . PUT . =S .)
Prvr->theorem
H1: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H2: size (u) - nchanges (seqnums (z)) in [0..1]
H3: size (u) - size (unique_msg (w)) in [0..1]
H4: size (unique_msg (w)) - size (unique_msg (x)) in [0..1]
H5: size (v) - size (unique_msg (y)) in [0..1]
H6: size (unique_msg (x)) - size (v) in [0..1]
H7: nchanges (seqnums (z)) le size (unique_msg (y))
->
C1: size (u) - size (v) in [0..1]
Prvr-> $Proceeding
:::::::(. E . E . U . U . PUT . =S . . U . FC . U . FC . E . D . 2 .)
size (u) - size (v) in [0..1]
Proved
Prvr-> $Proceeding
Proved using CLAIM. Now prove the claim.
Backup point
(CLAIM .)
Prvr-> theorem
H1: msg_lag (unique_msg (w), u, 1)
H2: msg_lag (v, unique_msg (x), 1)
H3: repeats (w)
H4: nchanges (seqnums (w)) - nchanges (seqnums (z))
in [0..1]
H5: size (u) - nchanges (seqnums (z)) in [0..1]
H6: size (v) - size (unique_msg (y)) in [0..1]
H7: x sub w

H8: y sub x
H9: z sub y
->
C1: z sub x
Prvr-> retain h8,h9
Backup point
(CLAIM . D .)
Prvr-> $Proceeding
(CLAIM .)
z sub x
Established claim
abp_1 proved in theorem prover.
Exec-> prove main_lemma
Entering Prover with lemma main_lemma
all s : bit_seq,
all x, y : pkt_seq,
repeats (y)
& nchanges (seqnums (y)) - nchanges (s) in [0..1]
& s sub seqnums (x) & x sub y
-> msg_lag (unique_msg (x), unique_msg (y), 1)
H1: repeats (y)
H2: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H3: s sub seqnums (x)
H4: x sub y
->
C1: msg_lag (unique_msg (x), unique_msg (y), 1)
Backup point
(.)
Prvr-> use interpolate
Backup point
(. U .)
Prvr-> hypothesis
H1: repeats (y#2$)
& nchanges (seqnums (y#2$)) - nchanges (s#2$)
in [0..1] & s#2$ sub seqnums (x#2$) & x#2$ sub y#2$
-> nchanges (seqnums (y#2$))
- nchanges (seqnums (x#2$))
in [0..1]
H2: repeats (y)
H3: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H4: s sub seqnums (x)
H5: x sub y

```
Prvr-> forwardchain h1
:::::::::Forward chaining gives
nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
Backup point
(. U . FC .)
Prvr-> expand msg_lag
Backup point
(. U . FC . E .)
Prvr-> theorem
H1:  nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
H2:  repeats (y#2$)
& nchanges (seqnums (y#2$)) - nchanges (s#2$)
in [0..1] & s#2$ sub seqnums (x#2$) & x#2$ sub y#2$
-> nchanges (seqnums (y#2$))
- nchanges (seqnums (x#2$))
in [0..1]
H3: repeats (y)
H4: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H5: s sub seqnums (x)
H6: x sub y
->
C1: initial_subseq (unique_msg (x), unique_msg (y))
C2: size (unique_msg (y)) - size (unique_msg (x)) in [0..1]
Prvr-> $Proceeding
Backup point
(. U . FC . E . 1 .)
Prvr-> use sub_to_lag
Backup point
(. U . FC . E . 1 . U .)
Prvr-> theorem
H1:  repeats (y#4$)
& nchanges (seqnums (y#4$))
- nchanges (seqnums (x#4$))
in [0..1] & x#4$ sub y#4$
-> initial_subseq (unique_msg (x#4$), unique_msg (y#4$))
H2:  nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
H3:  repeats (y#2$)
& nchanges (seqnums (y#2$)) - nchanges (s#2$)
in [0..1] & s#2$ sub seqnums (x#2$) & x#2$ sub y#2$
```

-> nchanges (seqnums (y#2$))
- nchanges (seqnums (x#2$))
in [0..1]
H4: repeats (y)
H5: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H6: s sub seqnums (x)
H7: x sub y
->
C1: initial_subseq (unique_msg (x), unique_msg (y))
Prvr-> qed
(. U . FC . E . 1 . U . QED)
::::::::(. U . FC . E . 1 . U . QED BC)
(. U . FC . E . 1 . U . QED BC 1)
(. U . FC . E . 1 . U . QED BC 1)
repeats (y)
Proved
(. U . FC . E . 1 . U . QED BC 2)
(. U . FC . E . 1 . U . QED BC 2)
nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
Proved
(. U . FC . E . 1 . U . QED BC 3)
(. U . FC . E . 1 . U . QED BC 3)
x sub y
Proved
(. U . FC . E . 1 . U . QED BC)
repeats (y)
& nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1] & x sub y
Proved
QED
Prvr-> $Proceeding
(. U . FC . E . 1 .)
initial_subseq (unique_msg (x), unique_msg (y))
Proved
Prvr-> $Proceeding
Backup point
(. U . FC . E . 2 .)
Prvr-> theorem
H1: nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
H2: repeats (y#2$)

& nchanges (seqnums (y#2$)) - nchanges (s#2$)
in [0..1] & s#2$ sub seqnums (x#2$) & x#2$ sub y#2$
-> nchanges (seqnums (y#2$))
- nchanges (seqnums (x#2$))
in [0..1]
H3: repeats (y)
H4: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H5: s sub seqnums (x)
H6: x sub y
->
C1: size (unique_msg (y)) - size (unique_msg (x)) in [0..1]
Prvr-> retain h1
Backup point
(. U . FC . E . 2 . D .)
Prvr-> use nchanges_unique
Backup point
(. U . FC . E . 2 . D . U .)
Prvr->use nchanges_unique
Backup point
(. U . FC . E . 2 . D . U . U .)
Prvr->hypothesis
H1: size (unique_msg (x#8$)) = nchanges (seqnums (x#8$))
H2: size (unique_msg (x#6$)) = nchanges (seqnums (x#6$))
H3: nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
Prvr-> put
For what?* x#8$;
Put what?* y;
For what?* x#6$;
Put what?* x;
For what?* $done
Backup point
(. U . FC . E . 2 . D . U . U . PUT .)
Prvr->hypothesis
H1: nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
H2: size (unique_msg (x)) = nchanges (seqnums (x))
H3: size (unique_msg (y)) = nchanges (seqnums (y))
Prvr-> equality substitute h2,h3
H2 yields
size (unique_msg (x)) := nchanges (seqnums (x))
H3 yields

size (unique_msg (y)) := nchanges (seqnums (y))
Backup point
(. U . FC . E . 2 . D . U . U . PUT . =S .)
Prvr-> $Proceeding
(. U . FC . E . 2 .)
size (unique_msg (y)) - size (unique_msg (x)) in [0..1]
Proved
Prvr-> $Proceeding
main_lemma proved in theorem prover.
Exec-> prove interpolate
Entering Prover with lemma interpolate
all s : bit_seq,
all x, y : pkt_seq,
repeats (y)
& nchanges (seqnums (y)) - nchanges (s) in [0..1]
& s sub seqnums (x) & x sub y
-> nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
H1: repeats (y)
H2: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H3: s sub seqnums (x)
H4: x sub y
->
C1: nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
Backup point
(.)
Prvr-> use sub_seqnum
Backup point
(. U .)
Prvr->use sub_nchanges
Backup point
(. U . U .)
Prvr-> hypothesis
H1: s#2$ sub t#2$ -> nchanges (s#2$) le nchanges (t#2$)
H2: x#2$ sub y#2$ -> seqnums (x#2$) sub seqnums (y#2$)
H3: repeats (y)
H4: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H5: s sub seqnums (x)
H6: x sub y
Prvr-> forwardchain h2
::::::Forward chaining gives

seqnums (x) sub seqnums (y)
Backup point
(. U . U . FC .)
Prvr-> hypothesis
H1: seqnums (x) sub seqnums (y)
H2: s#2$ sub t#2$ -> nchanges (s#2$) le nchanges (t#2$)
H3: x#2$ sub y#2$ -> seqnums (x#2$) sub seqnums (y#2$)
H4: repeats (y)
H5: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H6: s sub seqnums (x)
H7: x sub y
Prvr-> forwardchain h2
:Forward chaining gives
nchanges (seqnums (x)) le nchanges (seqnums (y))
Backup point
(. U . U . FC . FC .)
Prvr-> reorder h6
Backup point
(. U . U . FC . FC . REORDER .)
Prvr-> use sub_nchanges
Backup point
(. U . U . FC . FC . REORDER . U .)
Prvr->hypothesis
H1: s#4$ sub t#4$ -> nchanges (s#4$) le nchanges (t#4$)
H2: s sub seqnums (x)
H3: nchanges (seqnums (x)) le nchanges (seqnums (y))
H4: seqnums (x) sub seqnums (y)
H5: s#2$ sub t#2$ -> nchanges (s#2$) le nchanges (t#2$)
H6: x#2$ sub y#2$ -> seqnums (x#2$) sub seqnums (y#2$)
H7: repeats (y)
H8: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H9: x sub y
Prvr-> forwardchain h1
::Forward chaining gives
nchanges (s) le nchanges (seqnums (x))
Backup point
(. U . U . FC . FC . REORDER . U . FC .)
Prvr->theorem
H1: nchanges (s) le nchanges (seqnums (x))
H2: s#4$ sub t#4$ -> nchanges (s#4$) le nchanges (t#4$)
H3: s sub seqnums (x)
H4: nchanges (seqnums (x)) le nchanges (seqnums (y))

H5: seqnums (x) sub seqnums (y)
H6: s#2$ sub t#2$ -> nchanges (s#2$) le nchanges (t#2$)
H7: x#2$ sub y#2$ -> seqnums (x#2$) sub seqnums (y#2$)
H8: repeats (y)
H9: nchanges (seqnums (y)) - nchanges (s) in [0..1]
H10: x sub y
->
C1: nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
Prvr-> retain h1,h4,h9
Backup point
(. U . U . FC . FC . REORDER . U . FC . D .)
Prvr->theorem
H1: nchanges (s) le nchanges (seqnums (x))
H2: nchanges (seqnums (x)) le nchanges (seqnums (y))
H3: nchanges (seqnums (y)) - nchanges (s) in [0..1]
->
C1: nchanges (seqnums (y)) - nchanges (seqnums (x))
in [0..1]
Prvr-> $Proceeding
:::interpolate proved in theorem prover.
Exec-> show status scope $
SCOPE LEMMAS
Waiting to be proved: APP_BIT_NONNULL, APP_MSG_NONNULL, APP_PKT_NONNULL,
BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB, ISS_APP, ISS_TRANS,
LAST_NEXT, LAST_UNIQUE, LAST_REPEATS, MSG_LAG_EQ, NE_NEXT, NEXT_COMP,
NCHANGES_UNIQUE, SUB_APP, SIZE_NULL, SUB_SEQNUM, SUB_TO_LAG,
SUB_NCHANGES
Proved: ABP_1, INTERPOLATE, MAIN_LEMMA, PROP_REC_1, PROP_REC_2, PROP_REC_3,
PROP_REC_4, PROP_REC_5, PROP_TRANS_1, PROP_TRANS_2, PROP_TRANS_3,
PROP_TRANS_4, PROP_TRANS_5, PROP_TRANS_6, PROP_TRANS_7
Exec-> save abp.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec-> prove next_comp
Entering Prover with lemma next_comp
all x : pkt_seq,
all p : packet, comp (p.seqno) = next_seqnum (x [seq: p])
C1: comp (p.seqno) = next_seqnum (x [seq: p])

Backup point
(.)
Prvr-> qed
(. QED)
Ran out of tricks
Prvr-> expand next_seqnum
Backup point
(. QED E .)
Prvr->theorem
C1: comp (p.seqno)
= comp (last_bit (seqnums (x [seq: p])))
Prvr-> qed
(. QED E . QED)
Ran out of tricks
Prvr-> expand seqnums
Backup point
(. QED E . QED E .)
Prvr-> expand last_bit
Backup point
(. QED E . QED E . E .)
Prvr-> simplify theorem
Prvr->theorem
C1: true
Prvr-> $Proceeding
QED
Prvr-> $Proceeding
QED
Prvr-> $Proceeding
next_comp proved in theorem prover.
Exec-> prove ne_next
Entering Prover with lemma ne_next
all x : pkt_seq,
all p : packet,
p.seqno ne next_seqnum (x)
-> next_seqnum (x [seq: p]) = next_seqnum (x)
H1: p.seqno ne next_seqnum (x)
->
C1: next_seqnum (x [seq: p]) = next_seqnum (x)
Backup point
(.)
Prvr-> expand next_seqnum
More than one to expand.

Which do you want to expand?show-choices
1: next_seqnum (x)
2: next_seqnum (x [seq: p])
Which do you want to expand? 2
Backup point
(. E .)
Prvr-> theorem
H1: p.seqno ne next_seqnum (x)
->
C1: comp (last_bit (seqnums (x [seq: p])))
= next_seqnum (x)
Prvr-> expand seqnums
Backup point
(. E . E .)
Prvr->expand last_bit
Backup point
(. E . E . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: p.seqno ne next_seqnum (x)
->
C1: comp (p.seqno) = next_seqnum (x)
Prvr-> use comp_ne
Backup point
(. E . E . E . U .)
Prvr-> hypothesis
H1: comp (b1#2$) = b2#2$ iff b1#2$ ne b2#2$
H2: p.seqno ne next_seqnum (x)
Prvr-> forwardchain h1
Which way? (left or right)
left
::Forward chaining gives
comp (p.seqno) = next_seqnum (x)
Backup point
(. E . E . E . U . FC .)
Prvr-> $Proceeding
ne_next proved in theorem prover.
Exec-> prove last_next
Entering Prover with lemma last_next
all x : pkt_seq,
all p : packet,
x[size (x)] = p & null (pkt_seq) ne x

-> p.seqno ne next_seqnum (x)
H1: x[size (x)] = p
H2: null (pkt_seq) ne x
->
C1: p.seqno ne next_seqnum (x)
Backup point
(.)
Prvr-> expand next_seqnum
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
H1: x[size (x)] = p
H2: null (pkt_seq) ne x
->
C1: p.seqno ne comp (last_bit (seqnums (x)))
Prvr-> expand seqnums
Backup point
(. E . E .)
Prvr->expand last_bit
Backup point
(. E . E . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: x[size (x)] = p
H2: null (pkt_seq) ne x
->
C1: p.seqno ne comp (x[size (x)].seqno)
Prvr-> equality substitute h1
H1 can be solved for:
T1: x[size (x)]
T2: p
Which term (by label) do you want to substitute for?
t1
x[size (x)] := p
OK??
y
Backup point
(. E . E . E . =S .)
Prvr-> theorem
H1: null (pkt_seq) ne x
->

C1: p.seqno ne comp (p.seqno)
Prvr-> use comp_ne
Backup point
(. E . E . E . =S . U .)
Prvr-> theorem
H1: comp (b1#2$) = b2#2$ iff b1#2$ ne b2#2$
H2: null (pkt_seq) ne x
->
C1: p.seqno ne comp (p.seqno)
Prvr-> put
For what?* b1#2$;
Put what?* p.seqno;
For what?* b2#2$;
Put what?* p.seqno;
For what?* $done
Backup point
(. E . E . E . =S . U . PUT .)
Prvr-> theorem
H1: comp (p.seqno) = p.seqno iff p.seqno ne p.seqno
H2: null (pkt_seq) ne x
->
C1: p.seqno ne comp (p.seqno)
Prvr-> $Proceeding
last_next proved in theorem prover.
Exec-> prove last_unique
Entering Prover with lemma last_unique
all x : pkt_seq,
all p : packet,
x[size (x)] = p & null (pkt_seq) ne x
-> unique_msg (x [seq: p]) = unique_msg (x)
H1: x[size (x)] = p
H2: null (pkt_seq) ne x
->
C1: unique_msg (x [seq: p]) = unique_msg (x)
Backup point
(.)
Prvr-> expand unique_msg
More than one to expand.
Which do you want to expand? show-choices
1: unique_msg (x [seq: p])
2: unique_msg (x)
Which do you want to expand? 1

Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
H1: x[size (x)] = p
H2: null (pkt_seq) ne x
->
C1: unique_msg (x)
= if p.seqno = next_seqnum (x)
then unique_msg (x) [seq: p.mssg]
else unique_msg (x)
fi
Prvr-> use last_next
Backup point
(. E . U .)
Prvr-> hypothesis
H1: x#2$[size (x#2$)] = p#2$ & null (pkt_seq) ne x#2$
-> p#2$.seqno ne next_seqnum (x#2$)
H2: x[size (x)] = p
H3: null (pkt_seq) ne x
Prvr-> forwardchain h1
::Forward chaining gives
p.seqno ne next_seqnum (x)
Backup point
(. E . U . FC .)
Prvr-> simplify theorem
Prvr->theorem
C1: true
Prvr-> $Proceeding
last_unique proved in theorem prover.
Exec-> prove last_repeats
Entering Prover with lemma last_repeats
all x : pkt_seq,
all p : packet,
x[size (x)] = p & repeats (x) & null (pkt_seq) ne x
-> repeats (x [seq: p])
H1: x[size (x)] = p
H2: repeats (x)
H3: null (pkt_seq) ne x
->
C1: repeats (x [seq: p])
Backup point

(.)
Prvr-> expand repeats
More than one to expand.
Which do you want to expand? show-choices
1: repeats (x)
2: repeats (x [seq: p])
Which do you want to expand? 2
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
C1: true
Prvr-> $Proceeding
last_repeats proved in theorem prover.
Exec->
% Now we need to stop and fill in one last specification function.
% The definition of "initial_subseq" was deliberately left pending
% to avoid certain unstable prover behavior when is was expanded.
% We now give it a definition in order to prove the lemmas which
% use it.
translate tty:
Gypsy Text: (terminate with ^Z)
$extending scope alt_bit_specs =
begin
function initial_subseq (u, v: msg_seq) : boolean =
begin
exit (assume result iff some s: msg_seq, u s = v);
end;
end;
^Z
No syntax errors detected
No semantic errors detected
Exec-> set scope lemmas
Exec-> prove eq_iss
Entering Prover with lemma eq_iss
all u, v : msg_seq, u = v -> initial_subseq (u, v)
H1: u = v
->
C1: initial_subseq (u, v)
Backup point
(.)
Prvr-> expand initial_subseq

```
Backup point
(. E .)
Prvr-> theorem
H1: u = v
->
C1: u  s#4$ = v
Prvr-> $Proceeding
Ran out of tricks
Prvr-> put
For what?* s#4$;
Put what?* null(msg_seq);
For what?* $done
Backup point
(. E . PUT .)
Prvr-> $Proceeding
eq_iss proved in theorem prover.
Exec-> prove eq_iss_app
Entering Prover with lemma eq_iss_app
all u, v : msg_seq,
all m : message,
u = v -> initial_subseq (u  [seq: m], v  [seq: m])
H1: u = v
->
C1: initial_subseq (u  [seq: m], v  [seq: m])
Backup point
(.)
Prvr-> expand initial_subseq
Backup point
(. E .)
Prvr->theorem
H1: u = v
->
C1: u  [seq: m]  s#4$ = v  [seq: m]
Prvr-> put
For what?* s#4$;
Put what?* null(msg_seq);
For what?* $done
Backup point
(. E . PUT .)
Prvr-> $Proceeding
Conclusion simplified to:
u  [seq: m] = v  [seq: m]Ran out of tricks
```

```
Prvr-> theorem
H1: u = v
->
C1: u  [seq: m]  null (msg_seq) = v  [seq: m]
Prvr-> equality substitute h1
H1 can be solved for:
T1: u
T2: v
Which term (by label) do you want to substitute for?
t1
u := v
OK??
y
Backup point
(. E . PUT . =S .)
Prvr-> $Proceeding
eq_iss_app proved in theorem prover.
Exec-> prove iss_app
Entering Prover with lemma iss_app
all u, v : msg_seq,
all m : message,
initial_subseq (u, v) -> initial_subseq (u, v  [seq: m])
H1: initial_subseq (u, v)
->
C1: initial_subseq (u, v  [seq: m])
Backup point
(.)
Prvr-> expand initial_subseq
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr->theorem
H1: u  s#2 = v
->
C1: u  s#7$ = v  [seq: m]
Prvr-> simplify theorem
Prvr->theorem
H1: u  s#2 = v
->
C1: u  s#7$ = v  [seq: m]
Prvr-> equality substitute h1
```

H1 yields
v := u s#2
Backup point
(. E . =S .)
Prvr->theorem
H1: true
->
C1: u s#2 [seq: m] = u s#7$
Prvr-> $Proceeding
iss_app proved in theorem prover.
Exec-> prove iss_trans
Entering Prover with lemma iss_trans
all u, v, w : msg_seq,
initial_subseq (u, v) & initial_subseq (v, w)
-> initial_subseq (u, w)
H1: initial_subseq (u, v)
H2: initial_subseq (v, w)
->
C1: initial_subseq (u, w)
Backup point
(.)
Prvr-> expand initial_subseq
More than one to expand.
Which do you want to expand? all
Backup point
(. E .)
Prvr->simplify theorem
Prvr->theorem
H1: u s#2 = v
H2: v s#3 = w
->
C1: u s#10$ = w
Prvr-> $Proceeding
::..Ran out of tricks
Prvr-> equality substitute h1
H1 yields
v := u s#2
Backup point
(. E . =S .)
Prvr-> theorem
H1: u s#2 s#3 = w
->

C1: u s#10$ = w
Prvr-> equality substitute h1
H1 yields
w := u s#2 s#3
Backup point
(. E . =S . =S .)
Prvr-> theorem
H1: true
->
C1: u s#2 s#3 = u s#10$
Prvr-> $Proceeding
iss_trans proved in theorem prover.
Exec-> prove msg_lag_eq
Entering Prover with lemma msg_lag_eq
all u, v : msg_seq, u = v iff msg_lag (u, v, 0)
C1: u = v iff msg_lag (u, v, 0)
Backup point
(.)
Prvr-> expand msg_lag
Backup point
(. E .)
Prvr-> simplify theorem
Prvr->theorem
C1: size (u) = size (v) & initial_subseq (u, v) iff u = v
Prvr-> $Proceeding
Backup point
(. E . IFFC .)
Prvr-> theorem
C1: size (u) = size (v) & initial_subseq (u, v) -> u = v
C2: u = v -> size (u) = size (v) & initial_subseq (u, v)
Prvr-> $Proceeding
Backup point
(. E . IFFC . 1 .)
Prvr-> theorem
C1: size (u) = size (v) & initial_subseq (u, v) -> u = v
Prvr-> $Proceeding
Typelist equalities
size (v) = size (u)
Backup point
(. E . IFFC . 1 . P-> .)
Prvr-> theorem
H1: size (v) = size (u)

H2: size (u) = size (v)
H3: initial_subseq (u, v)
->
C1: u = v
Prvr-> expand initial_subseq
Backup point
(. E . IFFC . 1 . P-> . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: u s#2 = v
H2: size (u) = size (v)
H3: size (v) = size (u)
->
C1: u = v
Prvr-> use size_null
Backup point
(. E . IFFC . 1 . P-> . E . U .)
Prvr-> theorem
H1: null (msg_seq) = u#2$ iff 0 = size (u#2$)
H2: u s#2 = v
H3: size (u) = size (v)
H4: size (v) = size (u)
->
C1: u = v
Prvr-> $Proceeding
::::....Ran out of tricks
Prvr-> put
For what?* u#2$;
Put what?* s#2;
For what?* $done
Backup point
(. E . IFFC . 1 . P-> . E . U . PUT .)
Prvr-> theorem
H1: null (msg_seq) = s#2 iff 0 = size (s#2)
H2: size (v) = size (u)
H3: size (u) = size (v)
H4: u s#2 = v
->
C1: u = v
Prvr-> simplify theorem
Prvr->theorem
H1: u s#2 = v

H2: size (u) = size (v)
H3: size (v) = size (u)
H4: null (msg_seq) = s#2 iff 0 = size (s#2)
->
C1: u = v
Prvr-> equality substitute h1
H1 yields
v := u s#2
Backup point
(. E . IFFC . 1 . P-> . E . U . PUT . =S .)
Prvr-> theorem
H1: null (msg_seq) = s#2
H2: 0 = size (s#2)
->
C1: u s#2 = u
Prvr-> simplify theorem
Prvr->theorem
C1: true
Prvr-> $Proceeding
(. E . IFFC . 1 .)
size (u) = size (v) & initial_subseq (u, v) -> u = v
Proved
Prvr-> $Proceeding
Backup point
(. E . IFFC . 2 .)
Prvr-> theorem
C1: u = v -> size (u) = size (v) & initial_subseq (u, v)
Prvr-> $Proceeding
Backup point
(. E . IFFC . 2 . P-> .)
Prvr-> theorem
H1: u = v
->
C1: size (u) = size (v)
C2: initial_subseq (u, v)
Prvr-> equality substitute h1
H1 can be solved for:
T1: u
T2: v
Which term (by label) do you want to substitute for?
t1
u := v

OK??
y
Backup point
(. E . IFFC . 2 . P-> . =S .)
Prvr-> $Proceeding
Ran out of tricks
Prvr-> theorem
H1: true
->
C1: initial_subseq (v, v)
Prvr-> use eq_iss
Backup point
(. E . IFFC . 2 . P-> . =S . U .)
Prvr-> qed
(. E . IFFC . 2 . P-> . =S . U . QED)
(. E . IFFC . 2 . P-> . =S . U . QED BC)
(. E . IFFC . 2 . P-> . =S . U . QED BC)
v = v
Proved
QED
Prvr-> $Proceeding
(. E . IFFC . 2 .)
u = v -> size (u) = size (v) & initial_subseq (u, v)
Proved
Prvr-> $Proceeding
msg_lag_eq proved in theorem prover.
Exec-> prove comp_ne
Entering Prover with lemma comp_ne
all b1, b2 : bit, comp (b1) = b2 iff b1 ne b2
C1: comp (b1) = b2 iff b1 ne b2
Backup point
(.)
Prvr-> $Proceeding
Backup point
(. IFFC .)
Prvr-> $Proceeding
Backup point
(. IFFC . 1 .)
Prvr-> theorem
C1: comp (b1) = b2 -> b1 ne b2
Prvr-> $Proceeding
Backup point

```
(. IFFC . 1 . P-> .)
Prvr-> theorem
H1: comp (b1) = b2
->
C1: b1 ne b2
Prvr-> use bit_cases
Backup point
(. IFFC . 1 . P-> . U .)
Prvr-> theorem
H1: b#2$ = one or b#2$ = zero
H2: comp (b1) = b2
->
C1: b1 ne b2
Prvr-> expand comp
Backup point
(. IFFC . 1 . P-> . U . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: if b1 = zero then one else zero fi = b2
H2: b#2$ = one or b#2$ = zero
->
C1: b1 ne b2
Prvr-> put
For what?* b#2$;
Put what?* b1
* ;
For what?* $done
Backup point
(. IFFC . 1 . P-> . U . E . PUT .)
Prvr-> simplify theorem
Prvr->theorem
H1: if b1 = zero then one else zero fi = b2
H2: b1 = one or b1 = zero
->
C1: b1 ne b2
Prvr-> qed
(. IFFC . 1 . P-> . U . E . PUT . QED)
::.
b2 := if b1 = zero then one else zero fi
(. IFFC . 1 . P-> . U . E . PUT . QED =S)
Attempting CASE1
b1 = one
```

(. IFFC . 1 . P-> . U . E . PUT . QED =S CASE1)
Ran out of tricks
Prvr-> theorem
H1: b1 = one
->
C1: if b1 = zero then one else zero fi ne b1
Prvr-> simplify theorem
Prvr->theorem
H1: b1 = one
->
C1: if b1 = zero then one else zero fi ne b1
Prvr-> equality substitute h1
H1 can be solved for:
T1: b1
T2: one
Which term (by label) do you want to substitute for?
t1
b1 := one
OK??
y
Backup point
(. IFFC . 1 . P-> . U . E . PUT . QED =S CASE1 =S .)
Prvr-> $Proceeding
(. IFFC . 1 . P-> . U . E . PUT . QED =S CASE1 .)
b1 = one -> if b1 = zero then one else zero fi ne b1
Proved
Attempting CASE2
b1 = zero
Backup point
(. IFFC . 1 . P-> . U . E . PUT . QED =S CASE2 .)
Prvr-> theorem
H1: b1 = zero
->
C1: if b1 = zero then one else zero fi ne b1
Prvr-> equality substitute h1
H1 can be solved for:
T1: b1
T2: zero
Which term (by label) do you want to substitute for?
t1
b1 := zero
OK??

y
Backup point
(. IFFC . 1 . P-> . U . E . PUT . QED =S CASE2 . =S .)
Prvr-> $Proceeding
(. IFFC . 1 . P-> . U . E . PUT . QED =S CASE2 .)
b1 = zero -> if b1 = zero then one else zero fi ne b1
Proved
Done with cases
QED
Prvr-> $Proceeding
(. IFFC . 1 .)
comp (b1) = b2 -> b1 ne b2
Proved
Prvr-> $Proceeding
Backup point
(. IFFC . 2 .)
Prvr-> theorem
C1: b1 ne b2 -> comp (b1) = b2
Prvr-> $Proceeding
Backup point
(. IFFC . 2 . P-> .)
Prvr-> theorem
H1: b1 ne b2
->
C1: comp (b1) = b2
Prvr-> expand comp
Backup point
(. IFFC . 2 . P-> . E .)
Prvr-> simplify theorem
Prvr->theorem
H1: b1 ne b2
->
C1: if b1 = zero then one else zero fi = b2
Prvr-> use bit_cases
Backup point
(. IFFC . 2 . P-> . E . U .)
Prvr-> hypothesis
H1: b#4$ = one or b#4$ = zero
H2: b1 ne b2
Prvr-> put
For what?* b#4$;
Put what?* b1;

For what?* $done
Backup point
(. IFFC . 2 . P-> . E . U . PUT .)
Prvr-> theorem
H1: b1 = one or b1 = zero
H2: b1 ne b2
->
C1: if b1 = zero then one else zero fi = b2
Prvr-> qed
(. IFFC . 2 . P-> . E . U . PUT . QED)
::.Attempting CASE1
b1 = one
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1)
::..Ran out of tricks
Prvr-> theorem
H1: b1 = one
H2: b1 ne b2
->
C1: if b1 = zero then one else zero fi = b2
Prvr-> equality substitute h1
H1 yields
b1 := one
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S .)
Prvr-> $Proceeding
Ran out of tricks
Prvr-> theorem
H1: b2 ne one
->
C1: b2 = zero
Prvr-> use bit_cases
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S . U .)
Prvr-> theorem
H1: b#6$ = one or b#6$ = zero
H2: b2 ne one
->
C1: b2 = zero
Prvr-> put
For what?* b#6$;
Put what?* b2;
For what?* $done

Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S . U . PUT .)
Prvr-> $Proceeding
::.Attempting CASE1
b2 = one
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S . U . PUT . CASE1 .)
Prvr-> $Proceeding
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S . U . PUT . CASE1 .)
b2 = one -> b2 = zero
Proved
Attempting CASE2
b2 = zero
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S . U . PUT . CASE2 .)
Prvr-> $Proceeding
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 =S . U . PUT . CASE2 .)
b2 = zero -> b2 = zero
Proved
Done with cases
(. IFFC . 2 . P-> . E . U . PUT . QED CASE1 .)
b1 = one -> if b1 = zero then one else zero fi = b2
Proved
Attempting CASE2
b1 = zero
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 .)
Prvr-> $Proceeding
::..Ran out of tricks
Prvr-> theorem
H1: b1 = zero
H2: b1 ne b2
->
C1: if b1 = zero then one else zero fi = b2
Prvr-> equality substitute h1
H1 yields
b1 := zero
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S .)
Prvr-> $Proceeding
Ran out of tricks
Prvr-> theorem

H1: b2 ne zero
->
C1: b2 = one
Prvr-> use bit_cases
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S . U .)
Prvr-> hypothesis
H1: b#8$ = one or b#8$ = zero
H2: b2 ne zero
Prvr-> put
For what?* b#8$;
Put what?* b2;
For what?* $done
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S . U . PUT .)
Prvr-> $Proceeding
::.Attempting CASE1
b2 = one
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S . U . PUT . CASE1 .)
Prvr-> $Proceeding
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S . U . PUT . CASE1 .)
b2 = one -> b2 = one
Proved
Attempting CASE2
b2 = zero
Backup point
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S . U . PUT . CASE2 .)
Prvr-> $Proceeding
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 . =S . U . PUT . CASE2 .)
b2 = zero -> b2 = one
Proved
Done with cases
(. IFFC . 2 . P-> . E . U . PUT . QED CASE2 .)
b1 = zero -> if b1 = zero then one else zero fi = b2
Proved
Done with cases
QED
Prvr-> $Proceeding
(. IFFC . 2 .)
b1 ne b2 -> comp (b1) = b2
Proved

Prvr-> $Proceeding
comp_ne proved in theorem prover.
Exec-> prove hist_sub
Entering Prover with lemma hist_sub
all x, y : pkt_buf,
allfrom (x) sub allfrom (y) -> allfrom (x) sub allto (y)
C1: true
Backup point
(.)
Prvr-> $Proceeding
hist_sub proved in theorem prover.
Exec->
% The following lemma constitutes the entire basis for the proof.
prove bit_cases
Entering Prover with lemma bit_cases
all b : bit, b = one or b = zero
C1: b = one or b = zero
Backup point
(.)
Prvr-> simplify theorem
Prvr->theorem
C1: b = one or b = zero
Prvr->
% This lemma must be assumed since the prover does not have the
% information needed about the enumeration type "bit".
assume
Select assumed subgoal name bit_cases
Enter justification (terminate with ESCape)
see above.
$bit_cases proved in theorem prover.
Exec->
% The following lemmas were once needed but no longer are
% due to improvements in the simplifier.
prove app_pkt_nonnull
Entering Prover with lemma app_pkt_nonnull
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
app_pkt_nonnull proved in theorem prover.
Exec-> prove app_msg_nonnull

Entering Prover with lemma app_msg_nonnull
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
app_msg_nonnull proved in theorem prover.
Exec-> prove app_bit_nonnull
Entering Prover with lemma app_bit_nonnull
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
app_bit_nonnull proved in theorem prover.
Exec->
% Finally, we come to the remaining set of lemmas which are those
% proved under the Affirm system. Their definitions have noted
% that they are to be assumed so they will just fall through the
% prover here.
prove sub_app
Entering Prover with lemma sub_app
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
sub_app proved in theorem prover.
Exec-> prove size_null
Entering Prover with lemma size_null
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
size_null proved in theorem prover.
Exec-> prove sub_seqnum
Entering Prover with lemma sub_seqnum
true
C1: true
Backup point
(.)

Prvr->$Proceeding
sub_seqnum proved in theorem prover.
Exec-> prove sub_seqnum
Entering Prover with lemma sub_seqnum
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
sub_seqnum proved in theorem prover.
Exec-> prove sub_nchanges
Entering Prover with lemma sub_nchanges
true
C1: true
Backup point
(.)
Prvr->$Proceeding
sub_nchanges proved in theorem prover.
Exec-> prove nchanges_unique
Entering Prover with lemma nchanges_unique
true
C1: true
Backup point
(.)
Prvr->$Proceeding
nchanges_unique proved in theorem prover.
Exec-> prove sub_to_lag
Entering Prover with lemma sub_to_lag
true
C1: true
Backup point
(.)
Prvr-> $Proceeding
sub_to_lag proved in theorem prover.
Exec-> show status all
The current design and verification status is:
SCOPE ALT_BIT_PROTOCOL
Waiting for pending body to be filled in: MEDIUM, TIMER
Proved: AB_PROTOCOL, COMP, RECEIVER, SENDER
Types, constants: BIT, BIT_SEQ, CLK_BUF, MESSAGE, MSG_BUF, MSG_SEQ, PACKET,
PKT_BUF, PKT_SEQ
SCOPE LEMMAS

Proved: ABP_1, APP_BIT_NONNULL, APP_MSG_NONNULL, APP_PKT_NONNULL,
BIT_CASES, COMP_NE, EQ_ISS, EQ_ISS_APP, HIST_SUB, ISS_APP, ISS_TRANS,
INTERPOLATE, LAST_NEXT, LAST_UNIQUE, LAST_REPEATS, MAIN_LEMMA,
MSG_LAG_EQ, NE_NEXT, NEXT_COMP, NCHANGES_UNIQUE, PROP_REC_1,
PROP_REC_2, PROP_REC_3, PROP_REC_4, PROP_REC_5, PROP_TRANS_1,
PROP_TRANS_2, PROP_TRANS_3, PROP_TRANS_4, PROP_TRANS_5, PROP_TRANS_6,
PROP_TRANS_7, SUB_APP, SIZE_NULL, SUB_SEQNUM, SUB_TO_LAG,
SUB_NCHANGES

SCOPE ALT_BIT_SPECS
For specifications only: INITIAL_SUBSEQ, LAST_BIT, MSG_LAG, NCHANGES,
NEXT_SEQNUM, PROPER_RECEPTION, PROPER_TRANSMISSION, REPEATS, SEQNUMS,
UNIQUE_MSG
Exec-> save abv.dmp
File ABP.DMP already exists. Rewrite it? -> y
Saving.....
.....
.....
.....
Exec->
% This ends the Gypsy part of our proof of the Alternating Bit Protocol.
% Be sure to tune in next time when we try to prove bigger and better
% protocols.
[HDCOPY terminated at Sat 9-May-81 14:38:34]
Transcript file <DIVITO>AFFIRMTRANSCRIPT.8-FEB-81.1
is open in the AFFIRM system <AFFIRM>AFFIRM.EXE.120
1 U: load abpcontext;
file created for AFFIRM on 31-Jan-81 15:27:15
ABPCONTEXTCOMS
file created for AFFIRM on 4-Feb-81 21:48:05
BITCOMS
<DIVITO>BIT..2
file created for AFFIRM on 8-Jan-81 22:00:35
PACKETCOMS
compiled for AFFIRM on 20-Dec-80 15:42:13
file created for AFFIRM on 20-Dec-80 15:42:10
ELEMTYPECOMS
(File created under Affirm 121)
<PVLIBRARY>ELEMTYPE.COM.4<DIVITO>PACKET..2
file created for AFFIRM on 31-Jan-81 15:26:55
SEQUENCEOFELEMTYPECOMS
<DIVITO>SEQUENCEOFELEMTYPE..5

file created for AFFIRM on 31-Jan-81 15:25:36
SEQUENCEOFBITCOMS
<DIVITO>SEQUENCEOFBIT..5
file created for AFFIRM on 7-Feb-81 06:44:08
SEQUENCEOFPACKETCOMS
<DIVITO>SEQUENCEOFPACKET..7<DIVITO>ABPCONTEXT..3
2 U: print type elemtype;
(elemtype => ElemType)
type ElemType;
declare dummy: ElemType;
axiom dummy=dummy == TRUE;
end {ElemType} ;
3 U: print type bit;
(bit => Bit)
type Bit;
declare dummy, b: Bit;
interfaces zero, one, comp(b): Bit;
interface NormalForm(b): Boolean;
axioms dummy=dummy == TRUE,
zero = one == FALSE,
one = zero == FALSE;
axioms comp(zero) == one,
comp(one) == zero;
schema NormalForm(b) == cases(Prop(zero), Prop(one));
end {Bit} ;
4 U: print type packet;
(packet => Packet)
type Packet;
needs types Bit, ElemType;
declare dummy, p: Packet;
interface mssg(p): ElemType;
interface seqno(p): Bit;
axiom dummy=dummy == TRUE;
end {Packet} ;
5 U: print type sequenceofelemtype;
(sequenceofelemtype => SequenceOfElemType)
type SequenceOfElemType;
needs types Integer, ElemType;
declare dummy, ss, s, s1, s2, s3, s4, s5: SequenceOfElemType;
declare k, k1, k2: Integer;
declare ii, i, i1, i2, j: ElemType;
interfaces NewSequenceOfElemType, s apr i, i apl s, seq(i),

```
s1 join s2, LessFirst(s), LessLast(s): SequenceOfElemType;
infix join, apl, apr;
interfaces isNewSequenceOfElemType(s), s1 subseq s2, FirstInduction(s),
Induction(s), NormalForm(s), i in s, s1 iss s2: Boolean;
infix in, subseq, iss;
interface Length(s): Integer;
interfaces First(s), Last(s): ElemType;
axioms dummy=dummy == TRUE,
NewSequenceOfElemType = s apr i == FALSE,
s apr i = NewSequenceOfElemType == FALSE,
s apr i = s1 apr i1 == ((s=s1) and (i=i1));
axioms i apl NewSequenceOfElemType == NewSequenceOfElemType apr i,
i apl (s apr i1) == (i apl s) apr i1;
axiom seq(i) == NewSequenceOfElemType apr i;
axioms NewSequenceOfElemType join s == s,
(s apr i) join s1 == s join (i apl s1);
axiom LessFirst(s apr i)
== if s = NewSequenceOfElemType
then NewSequenceOfElemType
else LessFirst(s) apr i;
axiom LessLast(s apr i) == s;
axiom isNewSequenceOfElemType(s) == (s = NewSequenceOfElemType);
axioms s1 subseq (s apr i)
== ( (s1 = NewSequenceOfElemType) or s1 subseq s
or LessLast(s1) subseq s and (Last(s1) = i)),
s subseq NewSequenceOfElemType == (s = NewSequenceOfElemType);
axioms i in NewSequenceOfElemType == FALSE,
i in (s apr i1) == (i in s or (i=i1));
axioms s iss NewSequenceOfElemType == (s = NewSequenceOfElemType),
s1 iss (s2 apr i)
== ( (s1 = NewSequenceOfElemType) or s1 iss s2
or (LessLast(s1) = s2) and (Last(s1) = i));
axioms Length(NewSequenceOfElemType) == 0,
Length(s apr i) == Length(s) + 1;
axiom First(s apr i) == if s = NewSequenceOfElemType
then i
else First(s);
axiom Last(s apr i) == i;
rulelemmas NewSequenceOfElemType = i apl s == FALSE,
i apl s = NewSequenceOfElemType == FALSE;
rulelemmas s join (s1 apr i) == (s join s1) apr i,
s join NewSequenceOfElemType == s,
```

```
(i apl s1) join s2 == i apl (s1 join s2),
(s join (i apl s1)) join s2
== s join (i apl (s1 join s2)),
s join (s1 join s2) == (s join s1) join s2;
rulelemma LessFirst(i apl s) == s;
rulelemma LessLast(i apl s)
== if s = NewSequenceOfElemType
then NewSequenceOfElemType
else i apl LessLast(s);
rulelemmas NewSequenceOfElemType subseq s == TRUE,
s subseq s == TRUE;
rulelemma i in (i1 apl s) == (i in s or (i=i1));
rulelemmas NewSequenceOfElemType iss s == TRUE,
s iss s == TRUE;
rulelemma First(i apl s) == i;
rulelemma Last(i apl s) == if s = NewSequenceOfElemType
then i
else Last(s);
schemas FirstInduction(s)
== cases(Prop(NewSequenceOfElemType), all ss, ii
( IH(ss)
imp Prop( ii
apl ss))),
Induction(s)
== cases(Prop(NewSequenceOfElemType), all ss, ii
( IH(ss)
imp Prop( ss
apr ii))),
NormalForm(s)
== cases(Prop(NewSequenceOfElemType), all ss, ii (Prop( ss
apr ii)))
;
end {SequenceOfElemType} ;
6 U: print type sequenceofbit;
(sequenceofbit => SequenceOfBit)
type SequenceOfBit;
needs type Bit;
declare dummy, ss, s, s1, s2, s3, s4, s5: SequenceOfBit;
declare k, k1, k2: Integer;
declare ii, i, i1, i2, j: Bit;
interfaces NewSequenceOfBit, s apr i, i apl s, seq(i), s1 join s2,
LessFirst(s), LessLast(s): SequenceOfBit;
```

```
infix join, apl, apr;
interfaces isNewSequenceOfBit(s), s1 subseq s2, FirstInduction(s),
Induction(s), NormalForm(s), i in s, s1 iss s2: Boolean;
infix in, subseq, iss;
interfaces Nchanges(s), Length(s): Integer;
interfaces First(s), Last(s), LastBit(s): Bit;
axioms dummy=dummy == TRUE,
NewSequenceOfBit = s apr i == FALSE,
s apr i = NewSequenceOfBit == FALSE,
s apr i = s1 apr i1 == ((s=s1) and (i=i1));
axioms i apl NewSequenceOfBit == NewSequenceOfBit apr i,
i apl (s apr i1) == (i apl s) apr i1;
axiom seq(i) == NewSequenceOfBit apr i;
axioms NewSequenceOfBit join s == s,
(s apr i) join s1 == s join (i apl s1);
axiom LessFirst(s apr i)
== if s = NewSequenceOfBit
then NewSequenceOfBit
else LessFirst(s) apr i;
axiom LessLast(s apr i) == s;
axiom isNewSequenceOfBit(s) == (s = NewSequenceOfBit);
axioms s1 subseq (s apr i)
== ( (s1 = NewSequenceOfBit) or s1 subseq s
or LessLast(s1) subseq s and (Last(s1) = i)),
s subseq NewSequenceOfBit == (s = NewSequenceOfBit);
axioms i in NewSequenceOfBit == FALSE,
i in (s apr i1) == (i in s or (i=i1));
axioms s iss NewSequenceOfBit == (s = NewSequenceOfBit),
s1 iss (s2 apr i)
== ( (s1 = NewSequenceOfBit) or s1 iss s2
or (LessLast(s1) = s2) and (Last(s1) = i));
axioms Nchanges(NewSequenceOfBit) == 0,
Nchanges(s apr i)
== if LastBit(s) = i
then Nchanges(s)
else Nchanges(s) + 1;
axioms Length(NewSequenceOfBit) == 0,
Length(s apr i) == Length(s) + 1;
axiom First(s apr i) == if s = NewSequenceOfBit
then i
else First(s);
axiom Last(s apr i) == i;
```

```
axioms LastBit(NewSequenceOfBit) == zero,
LastBit(s apr i) == i;
rulelemmas NewSequenceOfBit = i apl s == FALSE,
i apl s = NewSequenceOfBit == FALSE;
rulelemmas s join (s1 apr i) == (s join s1) apr i,
s join NewSequenceOfBit == s,
(i apl s1) join s2 == i apl (s1 join s2),
(s join (i apl s1)) join s2
== s join (i apl (s1 join s2)),
s join (s1 join s2) == (s join s1) join s2;
rulelemma LessFirst(i apl s) == s;
rulelemma LessLast(i apl s)
== if s = NewSequenceOfBit
then NewSequenceOfBit
else i apl LessLast(s);
rulelemmas NewSequenceOfBit subseq s == TRUE,
s subseq s == TRUE;
rulelemma i in (i1 apl s) == (i in s or (i=i1));
rulelemmas NewSequenceOfBit iss s == TRUE,
s iss s == TRUE;
rulelemma First(i apl s) == i;
rulelemma Last(i apl s) == if s = NewSequenceOfBit
then i
else Last(s);
schemas FirstInduction(s)
== cases(Prop(NewSequenceOfBit), all ss, ii
( IH(ss)
imp Prop(ii apl ss))),
Induction(s)
== cases(Prop(NewSequenceOfBit), all ss, ii
( IH(ss)
imp Prop(ss apr ii))),
NormalForm(s)
== cases(Prop(NewSequenceOfBit), all ss, ii (Prop(ss apr ii)));
end {SequenceOfBit} ;
7 U: print type sequenceofpacket;
(sequenceofpacket => SequenceOfPacket)
type SequenceOfPacket;
needs types Integer, Packet, SequenceOfBit, SequenceOfElemType;
declare dummy, ss, s, s1, s2, s3, s4, s5: SequenceOfPacket;
declare k, k1, k2: Integer;
declare ii, i, i1, i2, j: Packet;
```

```
interfaces NewSequenceOfPacket, s apr i, i apl s, seq(i), s1 join s2,
LessFirst(s), LessLast(s): SequenceOfPacket;
infix join, apl, apr;
interfaces isNewSequenceOfPacket(s), s1 subseq s2, FirstInduction(s),
Induction(s), NormalForm(s), i in s, s1 iss s2, Repeats(s): Boolean;
infix in, subseq, iss;
interface Seqnums(s): SequenceOfBit;
interface UniqueMsg(s): SequenceOfElemType;
interface Length(s): Integer;
interfaces First(s), Last(s): Packet;
axioms dummy=dummy == TRUE,
NewSequenceOfPacket = s apr i == FALSE,
s apr i = NewSequenceOfPacket == FALSE,
s apr i = s1 apr i1 == ((s=s1) and (i=i1));
axioms i apl NewSequenceOfPacket == NewSequenceOfPacket apr i,
i apl (s apr i1) == (i apl s) apr i1;
axiom seq(i) == NewSequenceOfPacket apr i;
axioms NewSequenceOfPacket join s == s,
(s apr i) join s1 == s join (i apl s1);
axiom LessFirst(s apr i)
== if s = NewSequenceOfPacket
then NewSequenceOfPacket
else LessFirst(s) apr i;
axiom LessLast(s apr i) == s;
axiom isNewSequenceOfPacket(s) == (s = NewSequenceOfPacket);
axioms s1 subseq (s apr i)
== ( (s1 = NewSequenceOfPacket) or s1 subseq s
or LessLast(s1) subseq s and (Last(s1) = i)),
s subseq NewSequenceOfPacket == (s = NewSequenceOfPacket);
axioms i in NewSequenceOfPacket == FALSE,
i in (s apr i1) == (i in s or (i=i1));
axioms s iss NewSequenceOfPacket == (s = NewSequenceOfPacket),
s1 iss (s2 apr i)
== ( (s1 = NewSequenceOfPacket) or s1 iss s2
or (LessLast(s1) = s2) and (Last(s1) = i));
axioms Repeats(NewSequenceOfPacket) == TRUE,
Repeats(s apr i)
== ( Repeats(s)
and s ~= NewSequenceOfPacket
and seqno(i) = seqno(Last(s))
imp i = Last(s));
axioms Seqnums(NewSequenceOfPacket) == NewSequenceOfBit,
```

```
Seqnums(s apr i) == Seqnums(s) apr seqno(i);
axioms UniqueMsg(NewSequenceOfPacket) == NewSequenceOfElemType,
UniqueMsg(s apr i)
== if seqno(i) = LastBit(Seqnums(s))
then UniqueMsg(s)
else UniqueMsg(s) apr mssg(i);
axioms Length(NewSequenceOfPacket) == 0,
Length(s apr i) == Length(s) + 1;
axiom First(s apr i) == if s = NewSequenceOfPacket
then i
else First(s);
axiom Last(s apr i) == i;
rulelemmas NewSequenceOfPacket = i apl s == FALSE,
i apl s = NewSequenceOfPacket == FALSE;
rulelemmas s join (s1 apr i) == (s join s1) apr i,
s join NewSequenceOfPacket == s,
(i apl s1) join s2 == i apl (s1 join s2),
(s join (i apl s1)) join s2
== s join (i apl (s1 join s2)),
s join (s1 join s2) == (s join s1) join s2;
rulelemma LessFirst(i apl s) == s;
rulelemma LessLast(i apl s)
== if s = NewSequenceOfPacket
then NewSequenceOfPacket
else i apl LessLast(s);
rulelemmas NewSequenceOfPacket subseq s == TRUE,
s subseq s == TRUE;
rulelemma i in (i1 apl s) == (i in s or (i=i1));
rulelemmas NewSequenceOfPacket iss s == TRUE,
s iss s == TRUE;
rulelemma First(i apl s) == i;
rulelemma Last(i apl s) == if s = NewSequenceOfPacket
then i
else Last(s);
schemas FirstInduction(s)
== cases(Prop(NewSequenceOfPacket), all ss, ii
( IH(ss)
imp Prop(ii apl ss))),
Induction(s)
== cases(Prop(NewSequenceOfPacket), all ss, ii
( IH(ss)
imp Prop(ss apr ii))),
```

```
NormalForm(s)
== cases(Prop(NewSequenceOfPacket), all ss, ii (Prop( ss
apr ii)));
end {SequenceOfPacket} ;
8 U: print type abpcontext;
(abpcontext => AbpContext)
type AbpContext;
needs types Bit, Integer, Packet, SequenceOfElemType, SequenceOfBit,
SequenceOfPacket, ElemType;
declare dummy: AbpContext;
declare b, b1, b2: Bit;
declare i, i1, i2, j, k, k1, k2: Integer;
declare p, p1, p2, i$: Packet;
declare m, m1, m2: SequenceOfElemType;
declare n, n1, n2: SequenceOfBit;
declare s, s1, s2: SequenceOfPacket;
declare e, e1, e2: ElemType;
interfaces Bounded(k1, k2, j), MsgLag(m1, m2, j): Boolean;
axiom dummy=dummy == TRUE;
rulelemmas Bounded(k, k, 1) == TRUE,
Bounded(k, k+1, 1) == TRUE,
Bounded(k+1, k, 1) == FALSE,
Bounded(k1+1, k2+1, 1) == Bounded(k1, k2, 1);
rulelemmas MsgLag(m, m, 1) == TRUE,
MsgLag(m, m apr e, 1) == TRUE,
MsgLag(m apr e, m, 1) == FALSE;
define Bounded(k1, k2, j)
== ((k1 <= k2) and (k2 <= j+k1)),
MsgLag(m1, m2, j)
== (m1 iss m2 and Bounded(Length(m1), Length(m2), j));
end {AbpContext} ;
9 U: read abpcontext.theorems;
(Reading AFFIRM commands from <DIVITO>ABPCONTEXT.THEOREMS.13)
New environment:
dummy: AbpContext
b, b1, b2: Bit
i, i1, i2, j, k, k1, k2: Integer
p, p1, p2, i$: Packet
m, m1, m2: SequenceOfElemType
n, n1, n2: SequenceOfBit
s, s1, s2: SequenceOfPacket
e, e1, e2: ElemType
```

```
theorem SubToLag, s1 subseq s2
and Repeats(s2)
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)), 1)
imp UniqueMsg(s1) iss UniqueMsg(s2);
theorem PktSubBound, s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)) + 1, 1)
imp Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2));
theorem BitSubBound, n1 subseq n2
and Bounded(Nchanges(n1),
Nchanges(n2) + 1, 1)
imp Bounded(Nchanges(n1), Nchanges(n2), 1)
and Nchanges(n1) = Nchanges(n2);
theorem UniqueEq, UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)), 1)
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2);
theorem LastEq, n1 subseq n2
and Bounded(Nchanges(n1), Nchanges(n2), 1)
imp LastBit(n1) = LastBit(n2) eqv Nchanges(n1) = Nchanges(n2);
theorem SubLess, n1 subseq n2 imp Nchanges(n1) <= Nchanges(n2);
theorem SameLastBit, n1 subseq n2
and b = LastBit(n2)
and ~((n1 apr b) subseq n2)
imp LastBit(n1) = LastBit(n2);
theorem SameLastBit2, s1 subseq s2
and p = Last(s2)
and ~((s1 apr p) subseq s2)
imp LastBit(Seqnums(s1)) = LastBit(Seqnums(s2));
theorem SubSeqnum, s1 subseq s2 imp Seqnums(s1) subseq Seqnums(s2);
theorem NcUnique, Nchanges(Seqnums(s)) = Length(UniqueMsg(s));
theorem LastSeq, s ~= NewSequenceOfPacket
imp seqno(Last(s)) = LastBit(Seqnums(s));
theorem NcNonneg, Nchanges(n) >= 0;
theorem BoundedBy1, Bounded(k1, k2, 1)
eqv (k1=k2) or (k1+1 = k2);
theorem IssLenEq, m1 iss m2 and (Length(m1) = Length(m2))
```

```
eqv m1=m2;
theorem IssLenLe, m1 iss m2 imp Length(m1) <= Length(m2);
theorem BiValued, (b ~= b1) and (b ~= b2)
imp b1=b2;
10 U: print status;
The untried theorems are BitSubBound, BiValued, BoundedBy1, IssLenEq, IssLenLe
, LastEq, LastSeq, NcNonneg, NcUnique, PktSubBound, SameLastBit, SameLastBit2,
SubLess, SubSeqnum, SubToLag, and UniqueEq.
No theorems are tried.
No theorems are assumed.
No theorems are awaiting lemma proof.
No theorems are proved.
11 U: try SubToLag;
SubToLag is untried.
all s1, s2
( s1 subseq s2 and Repeats(s2)
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
imp UniqueMsg(s1) iss UniqueMsg(s2))
12 U: employ Induction(s2);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfPacket:)
all s1
( (s1 = NewSequenceOfPacket) and Bounded(Nchanges(Seqnums(s1)),
0, 1)
imp UniqueMsg(s1) = NewSequenceOfElemType)
13 U: replace;
TRUE
Going to leaf apr:.
all ss', ii', s1
( IH(ss', 1 {SubToLag})
and (s1 = NewSequenceOfPacket) or s1 subseq ss'
or LessLast(s1) subseq ss' and (Last(s1) = ii')
and Repeats(ss')
and (ss' ~= NewSequenceOfPacket) and seqno(ii')
= seqno(Last(ss'))
imp ii' = Last(ss')
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
imp UniqueMsg(s1) iss UniqueMsg(ss')
else Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
```

imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
14 U: split;
(first:)
all ss', ii', s1
(IH(ss', 1 {SubToLag}) and (s1 = NewSequenceOfPacket) and Repeats(
ss')
and (ss' ~= NewSequenceOfPacket) and seqno(ii')
= seqno(Last(ss'))
imp ii' = Last(ss')
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
imp UniqueMsg(s1) iss UniqueMsg(ss')
else Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
15 U: replace;
TRUE
Going to leaf second:.
all ss', ii', s1
(IH(ss', 1 {SubToLag}) and (s1 ~= NewSequenceOfPacket)
and s1 subseq ss' or LessLast(s1) subseq ss' and (Last(s1) = ii')
and Repeats(ss')
and (ss' ~= NewSequenceOfPacket) and seqno(ii')
= seqno(Last(ss'))
imp ii' = Last(ss')
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
imp UniqueMsg(s1) iss UniqueMsg(ss')
else Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
16 U: suppose LastBit(Seqnums(ss')) = seqno(ii');
(yes:)

all ss', ii', s1
(LastBit(Seqnums(ss')) = seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and s1 subseq ss'
or LessLast(s1) subseq ss'
and Last(s1) = ii'
and Repeats(ss')
and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
imp UniqueMsg(s1) iss UniqueMsg(ss'))
17 U: spllit;
(spllit => split)
(first:)
all ss', ii', s1
(LastBit(Seqnums(ss')) = seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and s1 subseq ss'
and Repeats(ss')
and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
imp UniqueMsg(s1) iss UniqueMsg(ss'))
18 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1: s1' = s1
Proved by chaining and narrowing
using the substitution
s1' = s1
TRUE
Going to leaf second:.
all ss', ii', s1
(LastBit(Seqnums(ss')) = seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and ~(s1 subseq ss')
and LessLast(s1) subseq ss'
and Last(s1) = ii'

and Repeats(ss')
and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
imp UniqueMsg(s1) iss UniqueMsg(ss'))
19 U: employ NormalForm(s1);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii', ss, ii
(LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)),
1)
imp UniqueMsg(ss') iss UniqueMsg(ss)
else Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii')) iss UniqueMsg(ss))
20 U: suppose;
(first:)
all ss', ii', ss, ii
(LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss = NewSequenceOfPacket
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)),
1)
imp UniqueMsg(ss') iss UniqueMsg(ss)
else Bounded(Nchanges(Seqnums(ss')) + 1,

Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii')) iss UniqueMsg(ss))
21 U: undo;
suppose undone.
22 U: suppose LastBit(Seqnums(ss')) = seqno(ii');
(yes:)
all ss', ii', ss, ii
(LastBit(Seqnums(ss')) = seqno(ii')
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)), 1)
imp UniqueMsg(ss') iss UniqueMsg(ss))
23 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2: s1 = ss' apr ii'
2/2: s1 = ss'
Proved by chaining and narrowing
using the substitution
s1 = ss'
TRUE
Going to leaf no:.
all ss', ii', ss, ii
(LastBit(Seqnums(ss')) ~= seqno(ii')
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii')) iss UniqueMsg(ss))

```
24 U: replace ii';
all ss', ii', ss, ii
( LastBit(Seqnums(ss')) ~= seqno(ii)
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss))
25 U: apply LastSeq;
some s ((s = NewSequenceOfPacket) or (seqno(Last(s)) = LastBit(Seqnums(s))))
Automatically search for instantiation? no [confirm]
26 U: put s=ss;
all ss', ii', ss, ii
(if ss = NewSequenceOfPacket
then LastBit(Seqnums(ss')) ~= seqno(ii)
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss)
else seqno(Last(ss)) = LastBit(Seqnums(ss))
and LastBit(Seqnums(ss')) ~= seqno(ii)
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
```

Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss))
27 U: suppose;
(first:)
all ss', ii', ss, ii
(ss = NewSequenceOfPacket
and LastBit(Seqnums(ss')) ~= seqno(ii)
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss))
28 U: replace ss;
all ss', ii', ss, ii
(ss = NewSequenceOfPacket
and LastBit(Seqnums(ss')) ~= seqno(ii)
and zero = seqno(ii)
and IH(NewSequenceOfPacket, 1 {SubToLag})
and ss' = NewSequenceOfPacket
and ii' = ii
imp ~Bounded(Nchanges(Seqnums(ss')) + 1, 0, 1))
29 U: replace ss';
TRUE
Going to leaf second:.
all ss', ii', ss, ii
(ss ~= NewSequenceOfPacket
and seqno(Last(ss)) = LastBit(Seqnums(ss))
and LastBit(Seqnums(ss')) ~= seqno(ii)
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)

imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss))
30 U: replace seqno(ii);
all ss', ii', ss, ii
(ss ~= NewSequenceOfPacket
and seqno(Last(ss)) = LastBit(Seqnums(ss))
and LastBit(Seqnums(ss')) ~= LastBit(Seqnums(ss))
and LastBit(Seqnums(ss)) = seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')) + 1,
Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss))
31 U: apply SameLastBit2;
some s1, s2, p
(s1 subseq s2 and (p = Last(s2))
imp (s1 apr p) subseq s2 or (LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))))
Automatically search for instantiation? yes [confirm]
1/4: (s2 = ss) and (s1 = ss' apr ii)
2/4: (s2 = ss) and (s1 = ss')
1/1: p = ii
Proved by chaining and narrowing
using the substitution
(s2 = ss) and (s1 = ss') and (p = ii)
TRUE
32 U: nxt;
(nxt => next)
Going to leaf no:.
all ss', ii', s1
(LastBit(Seqnums(ss')) ~= seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and s1 subseq ss'
or LessLast(s1) subseq ss'
and Last(s1) = ii'
and Repeats(ss')
and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')

```
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
33 U: split;
(first:)
all ss', ii', s1
( LastBit(Seqnums(ss')) ~= seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and s1 subseq ss'
and Repeats(ss')
and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
34 U: apply PktSubBound;
some s1', s2
( s1' subseq s2
and Bounded(Nchanges(Seqnums(s1')), Nchanges(Seqnums(s2)) + 1,
1)
imp Bounded(Nchanges(Seqnums(s1')), Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1')) = Nchanges(Seqnums(s2)))
Automatically search for instantiation? no [confirm]
35 U: put s1'=s1, s2=ss';
all ss', ii', s1
( s1 subseq ss'
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(ss')), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(ss'))
and LastBit(Seqnums(ss')) ~= seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and Repeats(ss')
```

and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')
imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
36 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1: s1' = s1
Proved by chaining and narrowing
using the substitution
s1' = s1
TRUE
Going to leaf second:.
all ss', ii', s1
(LastBit(Seqnums(ss')) ~= seqno(ii')
and IH(ss', 1 {SubToLag})
and s1 ~= NewSequenceOfPacket
and ~(s1 subseq ss')
and LessLast(s1) subseq ss'
and Last(s1) = ii'
and Repeats(ss')
and ss' ~= NewSequenceOfPacket
and seqno(ii') = seqno(Last(ss'))
imp ii' = Last(ss')
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(ss')) + 1, 1)
imp UniqueMsg(s1) = NewSequenceOfElemType
or UniqueMsg(s1) iss UniqueMsg(ss')
or LessLast(UniqueMsg(s1)) = UniqueMsg(ss')
and Last(UniqueMsg(s1)) = mssg(ii'))
37 U: employ NormalForm(s1);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii', ss, ii
(LastBit(Seqnums(ss)) ~= seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii

and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Bounded(Nchanges(Seqnums(ss')),
Nchanges(Seqnums(ss)) + 1, 1)
imp UniqueMsg(ss') = NewSequenceOfElemType
or UniqueMsg(ss') iss UniqueMsg(ss)
or LessLast(UniqueMsg(ss')) = UniqueMsg(ss)
and Last(UniqueMsg(ss')) = mssg(ii)
else Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)),
1)
imp (UniqueMsg(ss') apr mssg(ii')) iss UniqueMsg(ss)
or UniqueMsg(ss') = UniqueMsg(ss)
and mssg(ii') = mssg(ii))
38 U: suppose LastBit(Seqnums(ss')) = seqno(ii');
(yes:)
all ss', ii', ss, ii
(LastBit(Seqnums(ss')) = seqno(ii')
and LastBit(Seqnums(ss)) ~= seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')),
Nchanges(Seqnums(ss)) + 1, 1)
imp UniqueMsg(ss') = NewSequenceOfElemType
or UniqueMsg(ss') iss UniqueMsg(ss)
or LessLast(UniqueMsg(ss')) = UniqueMsg(ss)
and Last(UniqueMsg(ss')) = mssg(ii))
39 U: apply PktSubBound;
some s1, s2
(s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)) + 1, 1)
imp Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2)))
Automatically search for instantiation? no [confirm]

40 U: put s1=ss', s2=ss;
all ss', ii', ss, ii
(ss' subseq ss
and Bounded(Nchanges(Seqnums(ss')),
Nchanges(Seqnums(ss)) + 1, 1)
and Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)), 1)
and Nchanges(Seqnums(ss')) = Nchanges(Seqnums(ss))
and LastBit(Seqnums(ss')) = seqno(ii')
and LastBit(Seqnums(ss)) ~= seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
imp UniqueMsg(ss') = NewSequenceOfElemType
or UniqueMsg(ss') iss UniqueMsg(ss)
or LessLast(UniqueMsg(ss')) = UniqueMsg(ss)
and Last(UniqueMsg(ss')) = mssg(ii))
41 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2: s1 = ss'
Proved by chaining and narrowing
using the substitution
s1 = ss'
TRUE
Going to leaf no:.
all ss', ii', ss, ii
(LastBit(Seqnums(ss')) ~= seqno(ii')
and LastBit(Seqnums(ss)) ~= seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii') subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii')) iss UniqueMsg(ss)
or UniqueMsg(ss') = UniqueMsg(ss)

and mssg(ii') = mssg(ii))
42 U: repalce;
(repalce => replace)
all ss', ii', ss, ii
(LastBit(Seqnums(ss')) ~= seqno(ii)
and LastBit(Seqnums(ss)) ~= seqno(ii)
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss)
or UniqueMsg(ss') = UniqueMsg(ss))
43 U: apply bivalued;
(bivalued => BiValued)
some b, b1, b2 ((b=b1) or (b=b2) or (b1=b2))
Automatically search for instantiation? no [confirm]
44 U: put b=seqno(ii), b1=LastBit(Seqnums(ss')), b2=LastBit(Seqnums(ss));
all ss', ii', ss, ii
(seqno(ii) ~= LastBit(Seqnums(ss'))
and seqno(ii) ~= LastBit(Seqnums(ss))
and LastBit(Seqnums(ss')) = LastBit(Seqnums(ss))
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss)
or UniqueMsg(ss') = UniqueMsg(ss))
45 U: apply UniqueEq;
some s1, s2
(UniqueMsg(s1) iss UniqueMsg(s2) and s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))

```
imp UniqueMsg(s1) = UniqueMsg(s2))
Automatically search for instantiation? no [confirm]
46 U: put s1=ss', s2=ss;
all ss', ii', ss, ii
( ~(UniqueMsg(ss') iss UniqueMsg(ss))
and seqno(ii) ~= LastBit(Seqnums(ss'))
and seqno(ii) ~= LastBit(Seqnums(ss))
and LastBit(Seqnums(ss')) = LastBit(Seqnums(ss))
and IH(ss, 1 {SubToLag})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
and Repeats(ss)
and ss ~= NewSequenceOfPacket
and seqno(ii) = seqno(Last(ss))
imp ii = Last(ss)
and Bounded(Nchanges(Seqnums(ss')), Nchanges(Seqnums(ss)), 1)
imp (UniqueMsg(ss') apr mssg(ii)) iss UniqueMsg(ss)
or UniqueMsg(ss') = UniqueMsg(ss))
47 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2:  s1 = ss'
Proved by chaining and narrowing
using the substitution
s1 = ss'
TRUE
SubToLag is awaiting the proof of lemmas LastSeq, SameLastBit2, PktSubBound,
BiValued, and UniqueEq.
Going to lemma PktSubBound.
PktSubBound is untried.
all s1, s2
( s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)) + 1, 1)
imp Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2)))
48 U: note The hump has been passed but there still are
a few nontrivial lemmas left.  ;
49 U: freeze abp.frz;
Writing file ... <DIVITO>ABP.FRZ.1
50 U: quit;
Automatically summarize the proof attempts? no [confirm]
Type CONTINUE to return to AFFIRM.
```

Transcript file <DIVITO>AFFLOG.12-FEB-81.1
is open in the AFFIRM system <AFFIRM>AFFIRM.EXE.120
(<DIVITO>ABP.FRZ.1 . <AFFIRM>AFFIRM.EXE.120)
50 U: print status;
The untried theorems are BitSubBound, BiValued, BoundedBy1, IssLenEq, IssLenLe
, LastEq, LastSeq, NcNonneg, NcUnique, PktSubBound, SameLastBit, SameLastBit2,
SubLess, SubSeqnum, and UniqueEq.
No theorems are tried.
No theorems are assumed.
The awaiting lemma proof theorem is SubToLag.
No theorems are proved.
51 U: print result;
all s1, s2
(s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)) + 1, 1)
imp Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2)))
52 U: apply SubSeqnum;
some s1', s2' (s1' subseq s2' imp Seqnums(s1') subseq Seqnums(s2'))
Automatically search for instantiation? no [confirm]
53 U: put s1'=s1, s2'=s2;
all s1, s2
(s1 subseq s2 and Seqnums(s1) subseq Seqnums(s2)
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)) + 1, 1)
imp Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2)))
54 U: invoke Bounded |all|;
all s1, s2
(s1 subseq s2 and Seqnums(s1) subseq Seqnums(s2)
and Nchanges(Seqnums(s1)) <= Nchanges(Seqnums(s2)) + 1
and Nchanges(Seqnums(s2)) <= Nchanges(Seqnums(s1))
imp Nchanges(Seqnums(s1)) <= Nchanges(Seqnums(s2)))
55 U: apply SubLess;
some n1, n2 (n1 subseq n2 imp Nchanges(n1) <= Nchanges(n2))
Automatically search for instantiation? yes [confirm]
1/2: (n2 = Seqnums(s2)) and (n1 = Seqnums(s1))
Proved by chaining and narrowing
using the substitution
(n2 = Seqnums(s2)) and (n1 = Seqnums(s1))
TRUE
PktSubBound is awaiting the proof of lemmas SubSeqnum and SubLess.

56 U: next;
Going to lemma SubLess.
SubLess is untried.
all n1, n2 (n1 subseq n2 imp Nchanges(n1) <= Nchanges(n2))
57 U: employ Induction(n2);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfBit:)
all n1 (n1 = NewSequenceOfBit imp Nchanges(n1) <= 0)
58 U: replace;
TRUE
Going to leaf apr:.
all ss$, ii$, n1
(IH(ss$, 6 {SubLess})
and (n1 = NewSequenceOfBit) or n1 subseq ss$
or LessLast(n1) subseq ss$ and (Last(n1) = ii$)
imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1)
59 U: split;
(first:)
all ss$, ii$, n1
(IH(ss$, 6 {SubLess}) and (n1 = NewSequenceOfBit)
imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1)
60 U: apply NcNonneg;
some n (0 <= Nchanges(n))
Automatically search for instantiation? no [confirm]
61 U: put n=ss$;
all ss$, ii$, n1
((0 <= Nchanges(ss$)) and IH(ss$, 6 {SubLess})
and n1 = NewSequenceOfBit
imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1)
62 U: replace;
TRUE
Going to leaf second:.
all ss$, ii$, n1
(IH(ss$, 6 {SubLess}) and (n1 ~= NewSequenceOfBit)
and n1 subseq ss$ or LessLast(n1) subseq ss$ and (Last(n1) = ii$)

imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1)
63 U: split;
(first:)
all ss$, ii$, n1
(IH(ss$, 6 {SubLess}) and (n1 ~= NewSequenceOfBit) and n1 subseq ss$
imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1)
64 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1: n1' = n1
Unsuccessful.
all ss$, ii$, n1 (some n1'
(n1' subseq ss$ imp Nchanges(n1') <= Nchanges(ss$)
and n1 ~= NewSequenceOfBit
and n1 subseq ss$
imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1))
65 U: put n1'=n1;
TRUE
Going to leaf second:.
all ss$, ii$, n1
(IH(ss$, 6 {SubLess}) and (n1 ~= NewSequenceOfBit)
and ~(n1 subseq ss$)
and LessLast(n1) subseq ss$
and Last(n1) = ii$
imp if LastBit(ss$) = ii$
then Nchanges(n1) <= Nchanges(ss$)
else Nchanges(n1) <= Nchanges(ss$) + 1)
66 U: employ NormalForm(n1);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss$, ii$, ss$', ii$'
(IH(ss$', 6 {SubLess}) and ~((ss$ apr ii$) subseq ss$')
and ss$ subseq ss$'
and ii$=ii$'
imp if LastBit(ss$') = ii$'
then if LastBit(ss$) = ii$

then Nchanges(ss$) <= Nchanges(ss$')
else Nchanges(ss$) < Nchanges(ss$')
else if LastBit(ss$) = ii$
then Nchanges(ss$) <= Nchanges(ss$') + 1
else Nchanges(ss$) <= Nchanges(ss$'))
67 U: invoke IH;
Automatically search for instantiation? no [confirm]
all ss$, ii$, ss$', ii$' (some n1
(n1 subseq ss$' imp Nchanges(n1) <= Nchanges(ss$')
and ~((ss$ apr ii$) subseq ss$')
and ss$ subseq ss$'
and ii$=ii$'
imp if LastBit(ss$') = ii$'
then if LastBit(ss$) = ii$
then Nchanges(ss$) <= Nchanges(ss$')
else Nchanges(ss$) < Nchanges(ss$')
else if LastBit(ss$) = ii$
then Nchanges(ss$) <= Nchanges(ss$') + 1
else Nchanges(ss$) <= Nchanges(ss$')))
68 U: put n1=ss$;
all ss$, ii$, ss$', ii$'
(ss$ subseq ss$' and (Nchanges(ss$) <= Nchanges(ss$'))
and ~((ss$ apr ii$) subseq ss$')
and ii$ = ii$'
and LastBit(ss$') = ii$'
imp (LastBit(ss$) = ii$) or (Nchanges(ss$) < Nchanges(ss$')))
69 U: replace;
all ss$, ii$, ss$', ii$'
(ss$ subseq ss$' and (Nchanges(ss$) <= Nchanges(ss$'))
and ~((ss$ apr ii$') subseq ss$')
and ii$ = ii$'
and LastBit(ss$') = ii$'
imp (LastBit(ss$) = ii$') or (Nchanges(ss$) < Nchanges(ss$')))
70 U: apply SameLastBit;
some n1, n2, b
(n1 subseq n2 and (b = LastBit(n2))
imp (n1 apr b) subseq n2 or (LastBit(n1) = LastBit(n2)))
Automatically search for instantiation? yes [confirm]
1/4: (n2 = ss$') and (n1 = ss$)
1/2: b = LastBit(ss$)
2/2: b = ii$'
2/4: (n2 = ss$') and (n1 = ss$ apr ii$')

3/4: b = LastBit(n1)
4/4: (n2 = ss$') and (b = ii$') and (n1 = ss$)
Unsuccessful.
71 U: choose 4;
4/4: (n2 = ss$') and (b = ii$') and (n1 = ss$)
all ss$, ii$, ss$', ii$'
(ss$ subseq ss$' and (ii$' = LastBit(ss$'))
and ~((ss$ apr ii$') subseq ss$')
and LastBit(ss$) = LastBit(ss$')
and Nchanges(ss$) <= Nchanges(ss$')
and ii$ = ii$'
imp (LastBit(ss$) = ii$') or (Nchanges(ss$) < Nchanges(ss$')))
72 U: replce;
(replce => replace)
SubLess is awaiting the proof of lemmas NcNonneg and SameLastBit.
TRUE
Going to lemma SameLastBit.
SameLastBit is untried.
all n1, n2, b
(n1 subseq n2 and (b = LastBit(n2))
imp (n1 apr b) subseq n2 or (LastBit(n1) = LastBit(n2)))
73 U: employ NormalForm(n2);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) remains to be shown.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(NewSequenceOfBit:)
all n1, b ((n1 = NewSequenceOfBit) and (b = zero)
imp LastBit(n1) = zero)
74 U: replace;
TRUE
Going to leaf apr:.
all ss$, ii$, n1, b
(if n1 = NewSequenceOfBit
then b=ii$
imp (n1 apr b) subseq ss$ or n1 subseq ss$
or LastBit(n1) = ii$
else ~(n1 subseq ss$) and LessLast(n1) subseq ss$
and Last(n1) = ii$
and b=ii$
imp (n1 apr b) subseq ss$ or (LastBit(n1) = ii$))
75 U: employ NormalForm(n1);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) proven.
Case apr: all ss, ii (Prop(ss apr ii)) proven.

SameLastBit proved.
TRUE
Going to unproven ancestor NcNonneg.
NcNonneg is untried.
all n (0 <= Nchanges(n))
76 U: employ Induction(n);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) proven.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(apr:)
all ss$, ii$
(IH(ss$, 12 {NcNonneg})
imp if LastBit(ss$) = ii$
then 0 <= Nchanges(ss$)
else 0 <= Nchanges(ss$) + 1)
77 U: invoke IH;
NcNonneg proved.
SubLess proved.
TRUE
Going to unproven ancestor SubSeqnum.
SubSeqnum is untried.
all s1, s2 (s1 subseq s2 imp Seqnums(s1) subseq Seqnums(s2))
78 U: employ Induction(s2);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfPacket:)
all s1 (s1 = NewSequenceOfPacket imp Seqnums(s1) = NewSequenceOfBit)
79 U: repalce;
(repalce => replace)
TRUE
Going to leaf apr:.
all ss', ii', s1
(IH(ss', 9 {SubSeqnum})
and (s1 = NewSequenceOfPacket) or s1 subseq ss'
or LessLast(s1) subseq ss' and (Last(s1) = ii')
imp (Seqnums(s1) = NewSequenceOfBit) or Seqnums(s1) subseq Seqnums(ss')
or LessLast(Seqnums(s1)) subseq Seqnums(ss') and Last(Seqnums(s1))
= seqno(ii'))
80 U: split;
(first:)
all ss', ii', s1
(IH(ss', 9 {SubSeqnum}) and (s1 = NewSequenceOfPacket)
imp (Seqnums(s1) = NewSequenceOfBit) or Seqnums(s1) subseq Seqnums(ss')

or LessLast(Seqnums(s1)) subseq Seqnums(ss') and Last(Seqnums(s1))
= seqno(ii'))
81 U: replace;
TRUE
Going to leaf second:.
all ss', ii', s1
(IH(ss', 9 {SubSeqnum}) and (s1 ~= NewSequenceOfPacket)
and s1 subseq ss' or LessLast(s1) subseq ss' and (Last(s1) = ii')
imp (Seqnums(s1) = NewSequenceOfBit) or Seqnums(s1) subseq Seqnums(ss')
or LessLast(Seqnums(s1)) subseq Seqnums(ss') and Last(Seqnums(s1))
= seqno(ii'))
82 U: split;
(first:)
all ss', ii', s1
(IH(ss', 9 {SubSeqnum}) and (s1 ~= NewSequenceOfPacket) and s1
subseq ss'
imp (Seqnums(s1) = NewSequenceOfBit) or Seqnums(s1) subseq Seqnums(ss')
or LessLast(Seqnums(s1)) subseq Seqnums(ss') and Last(Seqnums(s1))
= seqno(ii'))
83 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1: s1' = s1
Proved by chaining and narrowing
using the substitution
s1' = s1
TRUE
Going to leaf second:.
all ss', ii', s1
(IH(ss', 9 {SubSeqnum}) and (s1 ~= NewSequenceOfPacket)
and ~(s1 subseq ss')
and LessLast(s1) subseq ss'
and Last(s1) = ii'
imp (Seqnums(s1) = NewSequenceOfBit) or Seqnums(s1) subseq Seqnums(ss')
or LessLast(Seqnums(s1)) subseq Seqnums(ss') and Last(Seqnums(s1))
= seqno(ii'))
84 U: employ NormalForm(s1);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii', ss, ii
(IH(ss, 9 {SubSeqnum}) and ~((ss' apr ii') subseq ss)
and ss' subseq ss

and ii'=ii
imp (Seqnums(ss') apr seqno(ii')) subseq Seqnums(ss)
or Seqnums(ss') subseq Seqnums(ss) and (seqno(ii') = seqno(ii)))
85 U: replace
;
all ss', ii', ss, ii
(IH(ss, 9 {SubSeqnum})
and ~((ss' apr ii) subseq ss)
and ss' subseq ss
and ii' = ii
imp (Seqnums(ss') apr seqno(ii)) subseq Seqnums(ss)
or Seqnums(ss') subseq Seqnums(ss))
86 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2: s1 = ss' apr ii
2/2: s1 = ss'
Proved by chaining and narrowing
using the substitution
s1 = ss'
TRUE
SubSeqnum proved.
PktSubBound proved.
There's more than one unproven ancestor. You may pick one of UniqueEq,
SameLastBit2, LastSeq, or BiValued.
87 U: note The toughest lemmas remain in the path which
begins with UniqueEq. ;
88 U: try uniqueeq;
(uniqueeq => UniqueEq)
UniqueEq is untried.
all s1, s2
(UniqueMsg(s1) iss UniqueMsg(s2) and s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
89 U: apply BoundedBy1;
some k1, k2
(Bounded(k1, k2, 1)
eqv (k1=k2) or (k1+1 = k2))
Automatically search for instantiation? yes [confirm]
1/1: (k2 = Nchanges(Seqnums(s2))) and (k1 = Nchanges(Seqnums(s1)))
Unsuccessful.
90 U: choose 1;

1/1: (k2 = Nchanges(Seqnums(s2))) and (k1 = Nchanges(Seqnums(s1)))
all s1, s2
(Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2))
or Nchanges(Seqnums(s1)) < Nchanges(Seqnums(s2))
and Nchanges(Seqnums(s1)) + 1 = Nchanges(Seqnums(s2))
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
91 U: split;
(first:)
all s1, s2
(Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2))
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
92 U: apply ncunique;
(ncunique => NcUnique)
some s (Nchanges(Seqnums(s)) = Length(UniqueMsg(s)))
Automatically search for instantiation? no [confirm]
93 U: apply ncunique;
(ncunique => NcUnique)
some s' (Nchanges(Seqnums(s')) = Length(UniqueMsg(s')))
Automatically search for instantiation? no [confirm]
94 U: put s=s1, s'=s2;
all s1, s2
(Nchanges(Seqnums(s2)) = Length(UniqueMsg(s2))
and Nchanges(Seqnums(s1)) = Length(UniqueMsg(s1))
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2))
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
95 U: replace;
all s1, s2
(Nchanges(Seqnums(s2)) = Length(UniqueMsg(s2))
and Nchanges(Seqnums(s1)) = Length(UniqueMsg(s2))
and Length(UniqueMsg(s1)) = Length(UniqueMsg(s2))

and UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
96 U: apply IssLenEq;
some m1, m2 (m1 iss m2 and (Length(m1) = Length(m2)) eqv m1=m2)
Automatically search for instantiation? yes [confirm]
1/1: (m2 = UniqueMsg(s2)) and (m1 = UniqueMsg(s1))
Proved by chaining and narrowing
using the substitution
(m2 = UniqueMsg(s2)) and (m1 = UniqueMsg(s1))
TRUE
97 U: next;
Going to leaf second:.
all s1, s2
(Nchanges(Seqnums(s1)) ~= Nchanges(Seqnums(s2))
and Nchanges(Seqnums(s1)) < Nchanges(Seqnums(s2))
and Nchanges(Seqnums(s1)) + 1 = Nchanges(Seqnums(s2))
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
98 U: apply SubSeqnum;
some s1', s2' (s1' subseq s2' imp Seqnums(s1') subseq Seqnums(s2'))
Automatically search for instantiation? no [confirm]
99 U: put s1'=s1, s2'=s2;
all s1, s2
(s1 subseq s2 and Seqnums(s1) subseq Seqnums(s2)
and Nchanges(Seqnums(s1)) ~= Nchanges(Seqnums(s2))
and Nchanges(Seqnums(s1)) < Nchanges(Seqnums(s2))
and Nchanges(Seqnums(s1)) + 1 = Nchanges(Seqnums(s2))
and Bounded(Nchanges(Seqnums(s1)), Nchanges(Seqnums(s2)), 1)
and UniqueMsg(s1) iss UniqueMsg(s2)
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2))
100 U: apply LastEq;
some n1, n2
(n1 subseq n2 and Bounded(Nchanges(n1), Nchanges(n2), 1)
imp LastBit(n1) = LastBit(n2) eqv Nchanges(n1) = Nchanges(n2))
Automatically search for instantiation? yes [confirm]
1/1: (n2 = Seqnums(s2)) and (n1 = Seqnums(s1))

Proved by chaining and narrowing
using the substitution
(n2 = Seqnums(s2)) and (n1 = Seqnums(s1))
TRUE
UniqueEq is awaiting the proof of lemmas BoundedBy1, NcUnique, IssLenEq
, and LastEq.
101 U: note LastEq continues the "hard line" of lemmas. ;
102 U: tyr LastEq;
what? Please correct tyr using the list: try or type.
Please correct tyr: try [confirm]
LastEq;
LastEq is untried.
all n1, n2
(n1 subseq n2 and Bounded(Nchanges(n1), Nchanges(n2), 1)
imp LastBit(n1) = LastBit(n2) eqv Nchanges(n1) = Nchanges(n2))
103 U: annotate Stating the right form of this lemma was one of
the most crucial parts of the overall proof.;
104 U: employ Induction(n2);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfBit:)
all n1
((n1 = NewSequenceOfBit) and Bounded(Nchanges(n1), 0, 1)
imp LastBit(n1) = zero eqv Nchanges(n1) = 0)
105 U: replace;
TRUE
Going to leaf apr:.
all ss$, ii$, n1
(IH(ss$, 5 {LastEq})
and (n1 = NewSequenceOfBit) or n1 subseq ss$
or LessLast(n1) subseq ss$ and (Last(n1) = ii$)
imp if LastBit(ss$) = ii$
then Bounded(Nchanges(n1), Nchanges(ss$), 1)
imp LastBit(n1) = ii$ eqv Nchanges(n1) = Nchanges(ss$)
else Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
imp LastBit(n1) = ii$
eqv Nchanges(n1) = Nchanges(ss$) + 1)
106 U: suppose LastBit(ss$) = ii$;
(yes:)
all ss$, ii$, n1
((LastBit(ss$) = ii$) and IH(ss$, 5 {LastEq})
and (n1 = NewSequenceOfBit) or n1 subseq ss$

or LessLast(n1) subseq ss$
and Last(n1) = ii$
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
imp LastBit(n1) = ii$ eqv Nchanges(n1) = Nchanges(ss$))
107 U: replace ii$;
all ss$, ii$, n1
((LastBit(ss$) = ii$) and IH(ss$, 5 {LastEq})
and (n1 = NewSequenceOfBit) or n1 subseq ss$
or LessLast(n1) subseq ss$
and Last(n1) = LastBit(ss$)
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
imp LastBit(n1) = LastBit(ss$)
eqv Nchanges(n1) = Nchanges(ss$))
108 U: suppose n1=NewSequenceOfBit or n1 subseq ss$;
(yes:)
all ss$, ii$, n1
((n1 = NewSequenceOfBit) or n1 subseq ss$
and LastBit(ss$) = ii$
and IH(ss$, 5 {LastEq})
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
imp LastBit(n1) = LastBit(ss$)
eqv Nchanges(n1) = Nchanges(ss$))
109 U: augment n1 subseq ss$;
(main:)
all ss$, ii$, n1
(n1 subseq ss$ and (LastBit(ss$) = ii$)
and IH(ss$, 5 {LastEq})
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
imp LastBit(n1) = LastBit(ss$)
eqv Nchanges(n1) = Nchanges(ss$))
110 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1: n1’ = n1
Proved by chaining and narrowing
using the substitution
n1’ = n1
TRUE
Going to leaf thesis:.
all ss$, ii$, n1
((n1 = NewSequenceOfBit) and (LastBit(ss$) = ii$)
and IH(ss$, 5 {LastEq})
and Bounded(Nchanges(n1), Nchanges(ss$), 1)

imp n1 subseq ss$)
111 U: replace n1;
TRUE
Going to leaf no:.
all ss$, ii$, n1
((n1 ~= NewSequenceOfBit) and ~(n1 subseq ss$)
and LastBit(ss$) = ii$
and IH(ss$, 5 {LastEq})
and LessLast(n1) subseq ss$
and Last(n1) = LastBit(ss$)
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
imp LastBit(n1) = LastBit(ss$)
eqv Nchanges(n1) = Nchanges(ss$))
112 U: employ NormalForm(n1);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') = ii$'
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = LastBit(ss$')
imp if LastBit(ss$) = ii$
then Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) = Nchanges(ss$')
else Bounded(Nchanges(ss$) + 1, Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$'))
113 U: suppose;
(first:)
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') = ii$'
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = LastBit(ss$')
and LastBit(ss$) = ii$
and Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) = Nchanges(ss$'))
114 U: replace;
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$') subseq ss$')

and LastBit(ss$') = ii$'
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = ii$'
and LastBit(ss$) = ii$'
and Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) = Nchanges(ss$'))
115 U: undo;
replace undone.
116 U: replace ii$;
all ss$, ii$, ss$', ii$'
(~((ss$ apr LastBit(ss$')) subseq ss$')
and LastBit(ss$') = ii$'
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = LastBit(ss$')
and LastBit(ss$) = LastBit(ss$')
and Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) = Nchanges(ss$'))
117 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2: n1 = ss$ apr LastBit(ss$')
2/2: n1 = ss$
Proved by chaining and narrowing
using the substitution
n1 = ss$
TRUE
Going to leaf second:.
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') = ii$'
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = LastBit(ss$')
and LastBit(ss$) ~= ii$
and Bounded(Nchanges(ss$) + 1, Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$'))
118 U: replace ii$;
all ss$, ii$, ss$', ii$'
(~((ss$ apr LastBit(ss$')) subseq ss$')
and LastBit(ss$') = ii$'
and IH(ss$', 5 {LastEq})

and ss$ subseq ss$'
and ii$ = LastBit(ss$')
and LastBit(ss$) ~= LastBit(ss$')
and Bounded(Nchanges(ss$) + 1, Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$'))
119 U: apply SameLastBit;
some n1, n2, b
(n1 subseq n2 and (b = LastBit(n2))
imp (n1 apr b) subseq n2 or (LastBit(n1) = LastBit(n2)))
Automatically search for instantiation? yes [confirm]
1/6: (n2 = ss$') and (n1 = ss$ apr LastBit(ss$'))
2/6: (n2 = ss$') and (n1 = ss$)
1/3: b = ii$
2/3: b = LastBit(ss$)
3/3: b = LastBit(ss$')
Proved by chaining and narrowing
using the substitution
(n2 = ss$') and (n1 = ss$) and (b = LastBit(ss$'))
TRUE
120 U: next;
Going to leaf no:.
all ss$, ii$, n1
((LastBit(ss$) ~= ii$) and IH(ss$, 5 {LastEq})
and (n1 = NewSequenceOfBit) or n1 subseq ss$
or LessLast(n1) subseq ss$
and Last(n1) = ii$
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
imp LastBit(n1) = ii$
eqv Nchanges(n1) = Nchanges(ss$) + 1)
121 U: suppose n1=NewSequenceOfBit or n1 subseq ss4;
(... suppose aborted: effects undone)
122 U: suppose n1=NewSequenceOfBit or n1 subseq ss$;
(yes:)
all ss$, ii$, n1
((n1 = NewSequenceOfBit) or n1 subseq ss$
and LastBit(ss$) ~= ii$
and IH(ss$, 5 {LastEq})
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
imp LastBit(n1) = ii$
eqv Nchanges(n1) = Nchanges(ss$) + 1)
123 U: augment n1 subseq ss$;
(main:)

```
all ss$, ii$, n1
( n1 subseq ss$ and (LastBit(ss$) ~= ii$)
and IH(ss$, 5 {LastEq})
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
imp LastBit(n1) = ii$
eqv Nchanges(n1) = Nchanges(ss$) + 1)
124 U: apply BitSubBound;
some n1', n2
( n1' subseq n2 and Bounded(Nchanges(n1'), Nchanges(n2) + 1, 1)
imp Bounded(Nchanges(n1'), Nchanges(n2), 1) and (Nchanges(n1') = Nchanges(
n2)))
Automatically search for instantiation? no [confirm]
125 U: put n1'=n1, n2=ss$;
all ss$, ii$, n1
( n1 subseq ss$
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
and Nchanges(n1) = Nchanges(ss$)
and LastBit(ss$) ~= ii$
and IH(ss$, 5 {LastEq})
imp LastBit(n1) ~= ii$)
126 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1:  n1' = n1
Unsuccessful.
all ss$, ii$, n1 (some n1'
( n1 subseq ss$
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
and Nchanges(n1) = Nchanges(ss$)
imp LastBit(ss$) = ii$
or if n1' subseq ss$
and Bounded(Nchanges(n1'), Nchanges(ss$), 1)
then if Nchanges(n1') = Nchanges(ss$)
then LastBit(n1') = LastBit(ss$)
imp LastBit(n1) ~= ii$
else LastBit(n1') = LastBit(ss$)
or LastBit(n1) ~= ii$
else LastBit(n1) ~= ii$))
127 U: put n1'=n1;
all ss$, ii$, n1
( n1 subseq ss$
```

and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
and Bounded(Nchanges(n1), Nchanges(ss$), 1)
and Nchanges(n1) = Nchanges(ss$)
and LastBit(ss$) ~= ii$
and LastBit(n1) = LastBit(ss$)
imp LastBit(n1) ~= ii$)
128 U: replace;
TRUE
Going to leaf thesis:.
all ss$, ii$, n1
((n1 = NewSequenceOfBit) and (LastBit(ss$) ~= ii$)
and IH(ss$, 5 {LastEq})
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
imp n1 subseq ss$)
129 U: replace n1;
TRUE
Going to leaf no:.
all ss$, ii$, n1
((n1 ~= NewSequenceOfBit) and ~(n1 subseq ss$)
and LastBit(ss$) ~= ii$
and IH(ss$, 5 {LastEq})
and LessLast(n1) subseq ss$
and Last(n1) = ii$
and Bounded(Nchanges(n1), Nchanges(ss$) + 1, 1)
imp LastBit(n1) = ii$
eqv Nchanges(n1) = Nchanges(ss$) + 1)
130 U: employ NormalForm(n1);
Case NewSequenceOfBit: Prop(NewSequenceOfBit) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') ~= ii$'
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = ii$'
imp if LastBit(ss$) = ii$
then Bounded(Nchanges(ss$), Nchanges(ss$') + 1, 1)
imp Nchanges(ss$) = Nchanges(ss$') + 1
else Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$') + 1)
131 U: replace ii$';

all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') ~= ii$
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = ii$'
imp if LastBit(ss$) = ii$
then Bounded(Nchanges(ss$),
Nchanges(ss$') + 1, 1)
imp Nchanges(ss$) = Nchanges(ss$') + 1
else Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$') + 1)
132 U: suppose;
(first:)
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') ~= ii$
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = ii$'
and LastBit(ss$) = ii$
and Bounded(Nchanges(ss$), Nchanges(ss$') + 1, 1)
imp Nchanges(ss$) = Nchanges(ss$') + 1)
133 U: swap |all|;
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and ii$ ~= LastBit(ss$')
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$' = ii$
and ii$ = LastBit(ss$)
and Bounded(Nchanges(ss$), Nchanges(ss$') + 1, 1)
imp Nchanges(ss$') + 1 = Nchanges(ss$))
134 U: replace;
all ss$, ii$, ss$', ii$'
(~((ss$ apr LastBit(ss$)) subseq ss$')
and LastBit(ss$) ~= LastBit(ss$')
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$' = LastBit(ss$)
and ii$ = LastBit(ss$)
and Bounded(Nchanges(ss$), Nchanges(ss$') + 1, 1)

imp Nchanges(ss$') + 1 = Nchanges(ss$))
135 U: apply BitSubBound;
some n1, n2
(n1 subseq n2 and Bounded(Nchanges(n1), Nchanges(n2) + 1, 1)
imp Bounded(Nchanges(n1), Nchanges(n2), 1) and (Nchanges(n1) = Nchanges(n2)
))
Automatically search for instantiation? no [confirm]
136 U: put n1=ss$, n2=ss$';
all ss$, ii$, ss$', ii$'
(ss$ subseq ss$'
and Bounded(Nchanges(ss$), Nchanges(ss$') + 1, 1)
and Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
and Nchanges(ss$) = Nchanges(ss$')
and ~((ss$ apr LastBit(ss$)) subseq ss$')
and LastBit(ss$) ~= LastBit(ss$')
and IH(ss$', 5 {LastEq})
and ii$' = LastBit(ss$)
imp ii$ ~= LastBit(ss$))
137 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2: n1 = ss$
Proved by chaining and narrowing
using the substitution
n1 = ss$
TRUE
Going to leaf second:.
all ss$, ii$, ss$', ii$'
(~((ss$ apr ii$) subseq ss$')
and LastBit(ss$') ~= ii$
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = ii$'
and LastBit(ss$) ~= ii$
and Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$') + 1)
138 U: apply bivallued;
(bivallued => BiValued)
some b, b1, b2 ((b=b1) or (b=b2) or (b1=b2))
Automatically search for instantiation? no [confirm]
139 U: put b=ii$, b1=LastBit(ss$), b2=LastBit(ss$');
all ss$, ii$, ss$', ii$'
(ii$ ~= LastBit(ss$)

```
and ii$ ~= LastBit(ss$')
and LastBit(ss$) = LastBit(ss$')
and ~((ss$ apr ii$) subseq ss$')
and IH(ss$', 5 {LastEq})
and ss$ subseq ss$'
and ii$ = ii$'
and Bounded(Nchanges(ss$), Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1 = Nchanges(ss$') + 1)
140 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/2:  n1 = ss$
2/2:  n1 = ss$ apr ii$
Unsuccessful.
all ss$, ii$, ss$', ii$' (some n1
( ii$ ~= LastBit(ss$)
and ii$ ~= LastBit(ss$')
and LastBit(ss$) = LastBit(ss$')
imp (ss$ apr ii$) subseq ss$'
or if n1 subseq ss$'
and Bounded(Nchanges(n1), Nchanges(ss$'), 1)
then if Nchanges(n1) = Nchanges(ss$')
then LastBit(n1) = LastBit(ss$')
and ss$ subseq ss$'
and ii$ = ii$'
and Bounded(Nchanges(ss$),
Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1
= Nchanges(ss$') + 1
else LastBit(n1) ~= LastBit(ss$')
and ss$ subseq ss$'
and ii$ = ii$'
and Bounded(Nchanges(ss$),
Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1
= Nchanges(ss$') + 1
else ss$ subseq ss$'
and ii$ = ii$'
and Bounded(Nchanges(ss$),
Nchanges(ss$'), 1)
imp Nchanges(ss$) + 1
= Nchanges(ss$') + 1))
141 U: choose 1;
```

1/2: n1 = ss$
LastEq is awaiting the proof of lemmas BitSubBound and BiValued.
TRUE
Going to lemma BitSubBound.
BitSubBound is untried.
all n1, n2
(n1 subseq n2 and Bounded(Nchanges(n1), Nchanges(n2) + 1, 1)
imp Bounded(Nchanges(n1), Nchanges(n2), 1) and (Nchanges(n1) = Nchanges(n2)
))
142 U: note The hump has been passed and the rest should
be clear sailing. ;
143 U: apply SubLess;
some n1', n2' (n1' subseq n2' imp Nchanges(n1') <= Nchanges(n2'))
Automatically search for instantiation? no [confirm]
144 U: put n1'=n1, n2'=n2;
all n1, n2
(n1 subseq n2 and (Nchanges(n1) <= Nchanges(n2))
and Bounded(Nchanges(n1), Nchanges(n2) + 1, 1)
imp Bounded(Nchanges(n1), Nchanges(n2), 1)
and Nchanges(n1) = Nchanges(n2))
145 U: invoke Bounded |all|;
BitSubBound proved.
TRUE
Going to unproven ancestor BiValued.
BiValued is untried.
all b, b1, b2 ((b=b1) or (b=b2) or (b1=b2))
146 U: employ NormalForm(b);
Case zero: Prop(zero) remains to be shown.
Case one: Prop(one) remains to be shown.
(zero:)
all b1, b2 ((zero = b1) or (zero = b2) or (b1=b2))
147 U: employ NormalForm(b1);
Case zero: Prop(zero) proven.
Case one: Prop(one) remains to be shown.
(one:)
all b2 ((zero = b2) or (one = b2))
148 U: employ NormalForm(b2);
Case zero: Prop(zero) proven.
Case one: Prop(one) proven.
TRUE
Going to leaf one:.
all b1, b2 ((one = b1) or (one = b2) or (b1=b2))

149 U: employ NormalForm(b1);
Case zero: Prop(zero) remains to be shown.
Case one: Prop(one) proven.
(zero:)
all b2 ((one = b2) or (zero = b2))
150 U: employ NormalForm(b2);
Case zero: Prop(zero) proven.
Case one: Prop(one) proven.
BiValued proved.
LastEq proved.
TRUE
There's more than one unproven ancestor. You may pick one of NcUnique,
BoundedBy1, IssLenEq, SameLastBit2, or LastSeq.
151 U: try IssLenEq;
IssLenEq is untried.
all m1, m2 (m1 iss m2 and (Length(m1) = Length(m2)) eqv m1=m2)
152 U: employ Induction(m2);
Case NewSequenceOfElemType: Prop(NewSequenceOfElemType) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfElemType:)
all m1 (m1 = NewSequenceOfElemType imp Length(m1) = 0)
153 U: replace;
TRUE
Going to leaf apr:.
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq})
imp (m1 = NewSequenceOfElemType) or m1 iss ss$$
or (LessLast(m1) = ss$$) and (Last(m1) = ii$$)
and Length(m1) = Length(ss$$) + 1
eqv m1 = ss$$ apr ii$$)
154 U: replace;
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq})
and m1 ~= ss$$ apr ii$$
and (m1 = NewSequenceOfElemType) or m1 iss ss$$
or LessLast(m1) = ss$$
and Last(m1) = ii$$
imp Length(m1) ~= Length(ss$$) + 1)
155 U: split;
(first:)
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq})

and m1 ~= ss$$ apr ii$$
and m1 = NewSequenceOfElemType
imp Length(m1) ~= Length(ss$$) + 1)
156 U: replace;
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq}) and (m1 = NewSequenceOfElemType)
imp 0 ~= Length(ss$$) + 1)
157 U: apply LenNonneg, Length(m) >= 0;
some m (0 <= Length(m))
Automatically search for instantiation? yes [confirm]
Unsuccessful.
158 U: put m=ss$$;
TRUE
Going to leaf second:.
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq})
and m1 ~= ss$$ apr ii$$
and m1 ~= NewSequenceOfElemType
and m1 iss ss$$
or LessLast(m1) = ss$$
and Last(m1) = ii$$
imp Length(m1) ~= Length(ss$$) + 1)
159 U: split;
(first:)
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq})
and m1 ~= ss$$ apr ii$$
and m1 ~= NewSequenceOfElemType
and m1 iss ss$$
imp Length(m1) ~= Length(ss$$) + 1)
160 U: apply IssLenLe;
some m1', m2 (m1' iss m2 imp Length(m1') <= Length(m2))
Automatically search for instantiation? no [confirm]
161 U: put m1'=m1, m2=ss$$;
all ss$$, ii$$, m1
(m1 iss ss$$ and (Length(m1) <= Length(ss$$))
and IH(ss$$, 14 {IssLenEq})
imp m1 = ss$$ apr ii$$
or m1 = NewSequenceOfElemType
or Length(m1) ~= Length(ss$$) + 1)
162 U: apply IntFact1, k1 <= k2 imp k1 ~= k2 + 1;
some k1, k2 (k1 <= k2 imp k1 ~= k2+1)

Automatically search for instantiation? yes [confirm]
1/1: (k2 = Length(ss$$)) and (k1 = Length(m1))
Proved by chaining and narrowing
using the substitution
(k2 = Length(ss$$)) and (k1 = Length(m1))
TRUE
163 U: next;
Going to leaf second:.
all ss$$, ii$$, m1
(IH(ss$$, 14 {IssLenEq})
and m1 ~= ss$$ apr ii$$
and m1 ~= NewSequenceOfElemType
and ~(m1 iss ss$$)
and LessLast(m1) = ss$$
and Last(m1) = ii$$
imp Length(m1) ~= Length(ss$$) + 1)
164 U: employ NormalForm(m1);
Case NewSequenceOfElemType: Prop(NewSequenceOfElemType) proven.
Case apr: all ss, ii (Prop(ss apr ii)) proven.
IssLenEq is awaiting the proof of lemmas LenNonneg, IssLenLe, and IntFact1.
TRUE
Going to lemma IssLenLe.
IssLenLe is untried.
all m1, m2 (m1 iss m2 imp Length(m1) <= Length(m2))
165 U: employ Induction(m2);
Case NewSequenceOfElemType: Prop(NewSequenceOfElemType) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfElemType:)
all m1 (m1 = NewSequenceOfElemType imp Length(m1) <= 0)
166 U: replace;
TRUE
Going to leaf apr:.
all ss$$, ii$$, m1
(IH(ss$$, 15 {IssLenLe})
and (m1 = NewSequenceOfElemType) or m1 iss ss$$
or (LessLast(m1) = ss$$) and (Last(m1) = ii$$)
imp Length(m1) <= Length(ss$$) + 1)
167 U: suppose m1=NewSequenceOfElemType or m1 iss ss$$;
(yes:)
all ss$$, ii$$, m1
((m1 = NewSequenceOfElemType) or m1 iss ss$$
and IH(ss$$, 15 {IssLenLe})

imp Length(m1) <= Length(ss$$) + 1)
168 U: augment m1 iss ss$$;
(main:)
all ss$$, ii$$, m1
(m1 iss ss$$ and IH(ss$$, 15 {IssLenLe})
imp Length(m1) <= Length(ss$$) + 1)
169 U: invoke IH;
Automatically search for instantiation? yes [confirm]
1/1: m1' = m1
Unsuccessful.
all ss$$, ii$$, m1 (some m1'
(m1 iss ss$$
and m1' iss ss$$ imp Length(m1') <= Length(ss$$)
imp Length(m1) <= Length(ss$$) + 1))
170 U: put m1'=m1;
all ss$$, ii$$, m1
(m1 iss ss$$ and (Length(m1) <= Length(ss$$))
imp Length(m1) <= Length(ss$$) + 1)
171 U: apply IntFact2, k1 <= k2 imp k1 <= k2 + 1;
some k1, k2 (k1 <= k2 imp k1 <= k2+1)
Automatically search for instantiation? yes [confirm]
1/2: (k2 = Length(ss$$)) and (k1 = Length(m1))
Proved by chaining and narrowing
using the substitution
(k2 = Length(ss$$)) and (k1 = Length(m1))
TRUE
172 U: next;
Going to leaf thesis:.
all ss$$, ii$$, m1
((m1 = NewSequenceOfElemType) and IH(ss$$, 15 {IssLenLe})
imp m1 iss ss$$)
173 U: replace;
TRUE
Going to leaf no:.
all ss$$, ii$$, m1
((m1 ~= NewSequenceOfElemType) and ~(m1 iss ss$$)
and IH(ss$$, 15 {IssLenLe})
and LessLast(m1) = ss$$
and Last(m1) = ii$$
imp Length(m1) <= Length(ss$$) + 1)
174 U: employ NormalForm(m1);
Case NewSequenceOfElemType: Prop(NewSequenceOfElemType) proven.

Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss$$, ii$$, ss$$', ii$$'
(~((ss$$ apr ii$$) iss ss$$') and IH(ss$$', 15 {IssLenLe})
and ss$$ = ss$$'
and ii$$ = ii$$'
imp Length(ss$$) <= Length(ss$$'))
175 U: repace;
what? Please correct repace using the list: readp or replace.
Please correct repace: replace [confirm]
;
IssLenLe is awaiting the proof of lemma IntFact2.
TRUE
Going to lemma IntFact2.
IntFact2 is untried.
IntFact2 proved.
IssLenLe proved.
TRUE
There's more than one unproven ancestor. You may pick one of LenNonneg
or IntFact1.
176 U: try IntFact1;
IntFact1 is untried.
IntFact1 proved.
TRUE
Going to unproven ancestor LenNonneg.
LenNonneg is untried.
all m (0 <= Length(m))
177 U: annotate Proved in Affirm type library. ;
178 U: assume LenNonneg;
IssLenEq proved.
179 U: next;
There's more than one unproven ancestor. You may pick one of NcUnique
or BoundedBy1.
180 U: try ncunique;
(ncunique => NcUnique)
NcUnique is untried.
all s (Nchanges(Seqnums(s)) = Length(UniqueMsg(s)))
181 U: employ Induction(s);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) proven.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii'

(IH(ss', 10 {NcUnique})
imp if LastBit(Seqnums(ss')) = seqno(ii')
then Nchanges(Seqnums(ss')) = Length(UniqueMsg(ss'))
else Nchanges(Seqnums(ss')) + 1 = Length(UniqueMsg(ss')) + 1)
182 U: invoke IH;
NcUnique proved.
TRUE
Going to unproven ancestor BoundedBy1.
BoundedBy1 is untried.
all k1, k2
(Bounded(k1, k2, 1)
eqv (k1=k2) or (k1<k2) and (k1+1 = k2))
183 U: invoke Bounded;
BoundedBy1 proved.
UniqueEq proved.
TRUE
There's more than one unproven ancestor. You may pick one of SameLastBit2
or LastSeq.
184 U: try SameLastBit2;
SameLastBit2 is untried.
all s1, s2, p
(s1 subseq s2 and (p = Last(s2))
imp (s1 apr p) subseq s2 or (LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))))
185 U: employ NormalForm(s2);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) remains to be shown.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(NewSequenceOfPacket:)
all s1, p
((s1 = NewSequenceOfPacket) and (p = Last(NewSequenceOfPacket))
imp LastBit(Seqnums(s1)) = zero)
186 U: replace;
TRUE
Going to leaf apr:.
all ss', ii', s1, p
(if s1 = NewSequenceOfPacket
then p=ii'
imp (s1 apr p) subseq ss' or s1 subseq ss'
or LastBit(Seqnums(s1)) = seqno(ii')
else ~(s1 subseq ss') and LessLast(s1) subseq ss'
and Last(s1) = ii'
and p=ii'
imp (s1 apr p) subseq ss' or (LastBit(Seqnums(s1)) = seqno(ii')))

187 U: employ NormalForm(s1);
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii', ss, ii, p
(~((ss' apr ii') subseq ss) and ss' subseq ss
and ii'=ii
and p=ii
imp ((ss' apr ii') apr p) subseq ss or (seqno(ii') = seqno(ii)))
188 U: replace;
SameLastBit2 proved.
TRUE
Going to unproven ancestor LastSeq.
LastSeq is untried.
all s ((s = NewSequenceOfPacket) or (seqno(Last(s)) = LastBit(Seqnums(s))))
189 U: emploloy NormalForm(s);
(emploloy => employ)
Case NewSequenceOfPacket: Prop(NewSequenceOfPacket) proven.
Case apr: all ss, ii (Prop(ss apr ii)) proven.
LastSeq proved.
SubToLag proved.
TRUE
The proof of this part is finished.
190 U: print status;
No theorems are untried.
No theorems are tried.
The assumed theorem is LenNonneg.
No theorems are awaiting lemma proof.
The proved theorems are BitSubBound, BiValued, BoundedBy1, IntFact1, IntFact2,
IssLenEq, IssLenLe, LastEq, LastSeq, NcNonneg, NcUnique, PktSubBound,
SameLastBit, SameLastBit2, SubLess, SubSeqnum, SubToLag, and UniqueEq.
191 U: quit;
Automatically summarize the proof attempts? yes [confirm]
theorem LastSeq, s ~= NewSequenceOfPacket
imp seqno(Last(s)) = LastBit(Seqnums(s));
proof tree:
189:! LastSeq
employ NormalForm(s) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
-> NewSequenceOfPacket:
Immediate

-> apr:
Immediate
theorem SameLastBit2, s1 subseq s2
and p = Last(s2)
and ~((s1 apr p) subseq s2)
imp LastBit(Seqnums(s1)) = LastBit(Seqnums(s2));
proof tree:
185:! SameLastBit2
employ NormalForm(s2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
186: NewSequenceOfPacket:
209 replace
-> (proven!)
187: apr:
210 employ NormalForm(s1)
-> NewSequenceOfPacket:
Immediate
188: apr:
211 replace
-> (proven!)
theorem BoundedBy1, Bounded(k1, k2, 1)
eqv (k1=k2) or (k1+1 = k2);
proof tree:
182:! BoundedBy1
normint {proved by Divito using AFFIRM 120 on 12-Feb-81 in
transcript <DIVITO>AFFLOG.12-FEB-81.1}
183: 207 invoke Bounded
183: 208 normint
-> (proven!)
theorem NcUnique, Nchanges(Seqnums(s)) = Length(UniqueMsg(s));
proof tree:
181:! NcUnique
employ Induction(s) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
-> NewSequenceOfPacket:
Immediate
181: apr:
204 cases
182: 205 invoke IH
182: 206 normint

-> (proven!)
theorem LenNonneg, Length(m) >= 0;
theorem IntFact1, k1 <= k2 imp k1 ~= k2+1;
proof tree:
176:! IntFact1
normint {proved by Divito using AFFIRM 120 on 12-Feb-81 in
transcript <DIVITO>AFFLOG.12-FEB-81.1}
-> (proven!)
theorem IntFact2, k1 <= k2 imp k1 <= k2+1;
proof tree:
175:! IntFact2
normint {proved by Divito using AFFIRM 120 on 12-Feb-81 in
transcript <DIVITO>AFFLOG.12-FEB-81.1}
-> (proven!)
theorem IssLenLe, m1 iss m2 imp Length(m1) <= Length(m2);
IssLenLe uses IntFact2!.
proof tree:
165:! IssLenLe
employ Induction(m2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
166: NewSequenceOfElemType:
192 replace
-> (proven!)
167: apr:
193 suppose m1 = NewSequenceOfElemType
or m1 iss ss$$
168: yes:
194 augment m1 iss ss$$
169: main:
196 invoke IH
170: 198 put m1'=m1
171: 199 apply IntFact2
171: 201 put k2 = Length(ss$$)
and k1 = Length(m1) {search}
171: (proven!)
173: thesis:{IssLenLe, apr:, yes:}
197 replace
-> (proven!)
174: no:{IssLenLe, apr:}
195 employ NormalForm(m1)
-> NewSequenceOfElemType:

Immediate
175: apr:
203 replace
-> (proven!)
theorem IssLenEq, m1 iss m2 and (Length(m1) = Length(m2))
eqv m1=m2;
IssLenEq uses LenNonneg%, IssLenLe!, and IntFact1!.
proof tree:
152:! IssLenEq
employ Induction(m2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
153: NewSequenceOfElemType:
176 replace
-> (proven!)
154: apr:
177 replace
155: 178 split
156: first:
179 replace
157: 181 apply LenNonneg
158: 183 put m=ss$$
158: 184 normint
-> (proven!)
159: second:{IssLenEq, apr:}
180 split
160: first:
185 apply IssLenLe
161: 187 put (m1'=m1) and (m2=ss$$)
162: 188 apply IntFact1
162: 190 put k2 = Length(ss$$)
and k1 = Length(m1) {search}
162: (proven!)
164: second:{IssLenEq, apr:, second:}
186 employ NormalForm(m1)
-> NewSequenceOfElemType:
Immediate
-> apr:
Immediate
theorem BiValued, (b ~= b1) and (b ~= b2)
imp b1=b2;
proof tree:

146:! BiValued
employ NormalForm(b) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
147: zero:
172 employ NormalForm(b1)
-> zero:
Immediate
148: one:
174 employ NormalForm(b2)
-> zero:
Immediate
-> one:
Immediate
149: one:{BiValued}
173 employ NormalForm(b1)
150: zero:
175 employ NormalForm(b2)
-> zero:
Immediate
-> one:
Immediate
-> one:
Immediate
theorem BitSubBound, n1 subseq n2
and Bounded(Nchanges(n1),
Nchanges(n2) + 1, 1)
imp Bounded(Nchanges(n1), Nchanges(n2), 1)
and Nchanges(n1) = Nchanges(n2);
BitSubBound uses SubLess!.
proof tree:
143:! BitSubBound
apply SubLess {proved by Divito using AFFIRM 120 on 12-Feb-81
in transcript <DIVITO>AFFLOG.12-FEB-81.1}
144: 169 put (n1'=n1) and (n2'=n2)
145: 170 invoke Bounded | all |
145: 171 normint
-> (proven!)
theorem LastEq, n1 subseq n2
and Bounded(Nchanges(n1), Nchanges(n2), 1)
imp LastBit(n1) = LastBit(n2) eqv Nchanges(n1) = Nchanges(n2);
LastEq uses SameLastBit!, BitSubBound!, and BiValued!.

proof tree:
104:! LastEq
employ Induction(n2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
105: NewSequenceOfBit:
122 replace
-> (proven!)
105: apr:
123 cases
106: 124 suppose LastBit(ss$) = ii$
107: yes:
125 replace ii$
108: 127 suppose n1 = NewSequenceOfBit
or n1 subseq ss$
109: yes:
128 augment n1 subseq ss$
110: main:
130 invoke IH
110: 132 put n1' = n1 {search}
110: (proven!)
111: thesis:
131 replace n1
-> (proven!)
112: no:{LastEq, apr:, yes:}
129 employ NormalForm(n1)
-> NewSequenceOfBit:
Immediate
112: apr:
134 cases
113: 135 split
116: first:
136 replace ii$
117: 138 invoke IH
117: 139 put n1 = ss$ {search}
117: (proven!)
118: second:{LastEq, apr:, yes:, no:, apr:}
137 replace ii$
119: 141 apply SameLastBit
119: 142 put n2 = ss$'
and n1 = ss$
and b = LastBit(ss$') {search}

119: (proven!)
122: no:{LastEq, apr:}
126 suppose n1 = NewSequenceOfBit
or n1 subseq ss$
123: yes:
144 augment n1 subseq ss$
124: main:
146 apply BitSubBound
125: 148 put (n1'=n1) and (n2=ss$)
125: 149 normint
126: 150 invoke IH
127: 151 put n1'=n1
128: 152 replace
-> (proven!)
129: thesis:{LastEq, apr:, no:, yes:}
147 replace n1
-> (proven!)
130: no:{LastEq, apr:, no:}
145 employ NormalForm(n1)
-> NewSequenceOfBit:
Immediate
130: apr:
153 cases
131: 154 replace ii$'
132: 155 split
133: first:
156 swap | all |
134: 158 replace
135: 159 apply BitSubBound
136: 160 put (n1=ss$) and (n2=ss$')
136: 161 normint
137: 162 invoke IH
137: 163 put n1 = ss$ {search}
137: (proven!)
138: second:{LastEq, apr:, no:, no:, apr:}
157 apply BiValued
139: 165 put b=ii$
and b1 = LastBit(ss$)
and b2 = LastBit(ss$')
140: 166 invoke IH
141: 167 put n1 = ss$ {choose}
141: 168 normint

-> (proven!)
theorem UniqueEq, UniqueMsg(s1) iss UniqueMsg(s2)
and s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)), 1)
and LastBit(Seqnums(s1)) = LastBit(Seqnums(s2))
imp UniqueMsg(s1) = UniqueMsg(s2);
UniqueEq uses BoundedBy1!, NcUnique!, IssLenEq!, SubSeqnum!, and LastEq!.
proof tree:
89:! UniqueEq
apply BoundedBy1 {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
90: 107 put k2 = Nchanges(Seqnums(s2))
and k1 = Nchanges(Seqnums(s1)) {choose}
90: 108 normint
91: 109 split
92: first:
110 apply NcUnique
93: 112 apply NcUnique
94: 113 put (s=s1) and (s'=s2)
95: 114 replace
96: 115 apply IssLenEq
96: 116 put (m2 = UniqueMsg(s2)) and (m1 = UniqueMsg(s1)) {search}
96: (proven!)
98: second:{UniqueEq}
111 apply SubSeqnum
99: 118 put (s1'=s1) and (s2'=s2)
100: 119 apply LastEq
100: 120 put (n2 = Seqnums(s2)) and (n1 = Seqnums(s1)) {search}
100: (proven!)
theorem SubSeqnum, s1 subseq s2 imp Seqnums(s1) subseq Seqnums(s2);
proof tree:
78:! SubSeqnum
employ Induction(s2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
79: NewSequenceOfPacket:
95 replace
-> (proven!)
80: apr:
96 split

```
81:  first:
97 replace
-> (proven!)
82:  second:
98 split
83:  first:
99 invoke IH
83:  101 put s1' = s1 {search}
83:  (proven!)
84:  second:
100 employ NormalForm(s1)
-> NewSequenceOfPacket:
Immediate
85:  apr:
103 replace
86:  104 invoke IH
86:  105 put s1 = ss' {search}
86:  (proven!)
theorem NcNonneg, Nchanges(n) >= 0;
proof tree:
76:!  NcNonneg
employ Induction(n) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
-> NewSequenceOfBit:
Immediate
76:  apr:
92 cases
77:  93 invoke IH
77:  94 normint
-> (proven!)
theorem SameLastBit, n1 subseq n2
and b = LastBit(n2)
and ~((n1 apr b) subseq n2)
imp LastBit(n1) = LastBit(n2);
proof tree:
73:!  SameLastBit
employ NormalForm(n2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
74:  NewSequenceOfBit:
90 replace
```

-> (proven!)
75: apr:
91 employ NormalForm(n1)
-> NewSequenceOfBit:
Immediate
-> apr:
Immediate
theorem SubLess, n1 subseq n2 imp Nchanges(n1) <= Nchanges(n2);
SubLess uses NcNonneg! and SameLastBit!.
proof tree:
57:! SubLess
employ Induction(n2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
58: NewSequenceOfBit:
70 replace
-> (proven!)
58: apr:
71 cases
59: 72 split
60: first:
73 apply NcNonneg
61: 75 put n=ss$
62: 76 replace
62: 77 normint
-> (proven!)
63: second:{SubLess, apr:}
74 split
64: first:
78 invoke IH
65: 80 put n1'=n1
65: 81 normint
-> (proven!)
66: second:{SubLess, apr:, second:}
79 employ NormalForm(n1)
-> NewSequenceOfBit:
Immediate
66: apr:
82 cases
67: 83 invoke IH
68: 84 put n1=ss$
68: 85 normint

69: 86 replace
70: 87 apply SameLastBit
71: 88 put n2 = ss$'
and b = ii$'
and n1 = ss$ {choose}
72: 89 replace
-> (proven!)
theorem PktSubBound, s1 subseq s2
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)) + 1, 1)
imp Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)), 1)
and Nchanges(Seqnums(s1)) = Nchanges(Seqnums(s2));
PktSubBound uses SubSeqnum! and SubLess!.
proof tree:
52:! PktSubBound
apply SubSeqnum {proved by Divito using AFFIRM 120 on 12-Feb-81
in transcript <DIVITO>AFFLOG.12-FEB-81.1}
53: 64 put (s1'=s1) and (s2'=s2)
54: 65 invoke Bounded | all |
54: 66 normint
55: 67 apply SubLess
55: 68 put (n2 = Seqnums(s2)) and (n1 = Seqnums(s1)) {search}
55: (proven!)
theorem SubToLag, s1 subseq s2
and Repeats(s2)
and Bounded(Nchanges(Seqnums(s1)),
Nchanges(Seqnums(s2)), 1)
imp UniqueMsg(s1) iss UniqueMsg(s2);
SubToLag uses LastSeq!, SameLastBit2!, PktSubBound!, BiValued!, and UniqueEq!.
proof tree:
12:! SubToLag
employ Induction(s2) {proved by Divito using AFFIRM 120 on
12-Feb-81 in transcript
<DIVITO>AFFLOG.12-FEB-81.1}
13: NewSequenceOfPacket:
17 replace
-> (proven!)
13: apr:
18 cases
14: 19 split
15: first:

20 replace
-> (proven!)
16: second:
21 suppose LastBit(Seqnums(ss')) = seqno(ii')
17: yes:
22 split
18: first:
24 invoke IH
18: 26 put s1' = s1 {search}
18: (proven!)
19: second:
25 employ NormalForm(s1)
-> NewSequenceOfPacket:
Immediate
19: apr:
28 cases
22: 29 suppose LastBit(Seqnums(ss'))
= seqno(ii')
23: yes:
30 invoke IH
23: 32 put s1 = ss' {search}
23: (proven!)
24: no:31 replace ii'
25: 34 apply LastSeq
26: 35 put s=ss
27: 36 split
28: first:
37 replace ss
29: 39 replace ss'
-> (proven!)
30: second:
38 replace seqno(ii)
31: 40 apply SameLastBit2
31: 41 put s2 = ss
and s1 = ss'
and p = ii {search}
31: (proven!)
33: no:{SubToLag, apr:, second:}
23 split
34: first:
43 apply PktSubBound
35: 45 put (s1'=s1) and (s2=ss')

36: 46 invoke IH
36: 47 put s1' = s1 {search}
36: (proven!)
37: second:{SubToLag, apr:, second:, no:}
44 employ NormalForm(s1)
-> NewSequenceOfPacket:
Immediate
37: apr:
49 cases
38: 50 suppose LastBit(Seqnums(ss'))
= seqno(ii')
39: yes:
51 apply PktSubBound
40: 53 put (s1=ss') and (s2=ss)
41: 54 invoke IH
41: 55 put s1 = ss' {search}
41: (proven!)
42: no:{SubToLag, apr:, second:, no:, second:, apr:}
52 replace
43: 57 apply BiValued
44: 58 put b = seqno(ii)
and b1 = LastBit(Seqnums(ss'))
and b2 = LastBit(Seqnums(ss))
45: 59 apply UniqueEq
46: 60 put (s1=ss') and (s2=ss)
47: 61 invoke IH
47: 62 put s1 = ss' {search}
47: (proven!)
Type CONTINUE to return to AFFIRM.
192 U: print uses;
Theorem Uses: Used by:
LastSeq, proved - SubToLag
SameLastBit2, proved - SubToLag
BoundedBy1, proved - UniqueEq
NcUnique, proved - UniqueEq
LenNonneg, assumed - IssLenEq
IntFact1, proved - IssLenEq
IntFact2, proved - IssLenLe
IssLenLe, proved IntFact2 IssLenEq
IssLenEq, proved LenNonneg UniqueEq
IssLenLe
IntFact1

BiValued, proved - LastEq
SubToLag
BitSubBound, proved SubLess LastEq
LastEq, proved SameLastBit UniqueEq
BitSubBound
BiValued
UniqueEq, proved BoundedBy1 SubToLag
NcUnique
IssLenEq
SubSeqnum
LastEq
SubSeqnum, proved - UniqueEq
PktSubBound
NcNonneg, proved - SubLess
SameLastBit, proved - LastEq
SubLess
SubLess, proved NcNonneg BitSubBound
SameLastBit PktSubBound
PktSubBound, proved SubSeqnum SubToLag
SubLess
SubToLag, proved LastSeq -
SameLastBit2
PktSubBound
BiValued
UniqueEq
193 U: print assumptions;
LenNonneg is used by IssLenEq.
assume LenNonneg, Length(m) >= 0;
194 U: quit;
Automatically summarize the proof attempts? no [confirm]
Type CONTINUE to return to AFFIRM.