A MECHANICAL VERIFICATION OF THE
ALTERNATING BIT PROTOCOL

Benedetto L. DiVito

ICSCA-CMP-21          15 June 1981

Institute For Computing Science And Computer Applications
Certifiable Minicomputer Project
The University of Texas at Austin
Austin, Texas 78712
# Table of Contents

1 Introduction ........................................ 1
2 Problem Description ................................ 1
   2.1 Model ............................................ 1
   2.2 Service Specification ....................... 3
   2.3 Protocol Specification ...................... 4
   2.4 Medium Specification ....................... 6
3 Verification Conditions .......................... 6
4 Proof Description ................................ 8
   4.1 Decomposition .................................. 8
   4.2 Gypsy-Affirm Interface ..................... 10
5 Conclusions ........................................ 11
Appendix I ........................................... 13
Appendix II .......................................... 23
Appendix III ......................................... 26
Appendix IV .......................................... 28
ABSTRACT

The Alternating Bit Protocol has been modeled via a straightforward application of the Gypsy methodology. A safety property was stated for its service specification and a procedural protocol specification was written using Gypsy procedure definitions. Mechanical verification was carried out, including proofs of the supporting lemmas. A unique aspect of this verification effort is the cooperative proof strategy that was employed, making use of two separate verification systems. The combined capabilities of both the Gypsy system and the Affirm system were utilized to achieve this result.
1 Introduction

The world has yet another verification of the Alternating Bit Protocol. A brief description of this latest addition is presented. The protocol was modeled as an abstract program using the Gypsy verification methodology. A fully mechanical proof of a safety property was obtained. What is perhaps more interesting is that the proof was performed with the combined help of two separate verification systems: the Gypsy system [Good 77, Good 78a] and the Affirm system [Musser 80, Gerhart 80].

The modeling and specification effort was a more or less straightforward application of the Gypsy methodology for concurrent programming. In addition, the Alternating Bit Protocol is by now a very well known example problem. Therefore, the following discussion will not dwell on those aspects of the effort. A bit more emphasis will be placed on the discussion of the proof approach, since it is this area which forms the more unique part of this work.

A complete listing of the formal model is presented in Appendix I. Further supporting information can be found in the other appendices. Complete transcripts of the proofs and other verifier output have been compiled into a separate (rather lengthy) report [DiVito 81].

2 Problem Description

The Alternating Bit Protocol [Bartlett 69] is an example used widely to illustrate the application of formal methods to protocols. It is assumed that the reader has some familiarity with this protocol so we will go directly to our particular model of it. The version being used is that described in [Bochnann 77]. It is concerned with one-way data transfer only. The Gypsy model used here evolved from the one originally presented in [Sunshine 79a].

2.1 Model

The protocol is viewed as providing a virtual communication medium for reliable data transfer. This virtual medium is modeled as a Gypsy concurrent process. It is in turn composed of a set of sequential and concurrent processes which communicate through message buffers. The model for the Alternating Bit Protocol is depicted below.
This structure is rendered in Gypsy by the following process definition. For full information on Gypsy notation consult the language report [Good 78b].

```
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;
```

There are five subprocesses which operate concurrently to realize the parent process "ab_protocol." The buffers which interconnect
there are local variables of the parent. The buffers which connect to the external environment (higher level protocol) are formal parameters of the parent.

In the subsequent discussion, we follow the general architectural model for specifying protocols that is outlined in [Sunshine 79b]. Its basic features include the use of a layered protocol structure as well as the separation of specifications into two classes: service specifications and protocol specifications. Thus the basic verification problem in these terms is to show that the service specification is satisfied by its corresponding protocol specification, assuming the service specification of the next lower layer is met.

2.2 Service Specification

The service provided by this protocol is sequenced, reliable data transfer from the source to the sink. This is the main property to be proved about the process just defined. This property is expressed as a normal Gypsy blockage specification on the "ab_protocol" process. As such it is an assertion over the buffer histories of the formal parameters of the process. It must hold whenever the process is blocked waiting to send or receive a message. A history based assertion offers a nonprocedural way of stating the service specification.

The exact property which has been proved is stated as

\[ \text{msg\_lag}(\text{outto}(\text{sink}, \text{myid}), \text{infrom}(\text{source}, \text{myid}), 1) \]

The "outto" and "infrom" functions are buffer history functions for the formal parameters. The "msg\_lag" function is a user defined specification function which states that

1. the sequence of messages sent to the sink is an initial subsequence of that received from the source,

2. the sink is no more than one message behind the source.

The second item is not strictly necessary to state for the usual conception of a safety property, but it can be obtained without much additional work and yields a stronger service specification.
The formal definition of the "msg_lag" function is given below.

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 =
begin
exit (assume result iff some s : msg_seq, u @ s = v);
end;

2.3 Protocol Specification

A protocol specification must describe the behavior required of the interacting entities, in this case the "sender" and "receiver" processes. Typically, this is done by writing procedures which perform the necessary functions of the sender and receiver. In the present context, normal Gypsy executable statements can be used to construct such procedures. This results in a procedural form of specification for these entities.

The following procedure definitions represent this style of protocol specification applied to the Alternating Bit Protocol.

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
    var pack, ack : packet;
    var b : boolean;
    var next : bit := one;

    loop
      receive pack.mssg from source;
      pack.seqno := next;
      send pack to pkt_send;
      send true to start;
      loop
        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 {await}
end; {loop}
next := comp (next)
end {loop}
end;

procedure receiver (var sink : msg_buf <output>;
  var pkt_rcv : pkt_buf <input>;
  var ack_rcv : pkt_buf <output>)
begin
  var exp : bit := one;
loop
    receive pack from pkt_rcv;
    if exp = pack.seqno
      then send pack.assg to sink;
    exp := comp (exp)
  end;
  send pack to ack_rcv
end

Both the sender and receiver are modeled as nonterminating sequential processes. The send and receive primitives of Gypsy are used to communicate through the message buffers. The "await" statement of the sender represents a parallel wait operation, being satisfied by the first receive which becomes ready. Other features of these processes should be clear. Note that the versions shown above are void of any assertions needed for verification. These are included in the Gypsy text displayed in Appendix I.

The timer process will not be described any further. It can be thought of as a process which receives a signal to start timing and sends back a tick some time later. Its only purpose is for the sender to set retransmission timeouts.
2.4 Medium Specification

The transmission medium is simply the virtual communication medium of the next lower protocol layer. Hence, it is none other than the service specification for that particular protocol. Thus it should be stated as before in a nonprocedural form as the Gypsy blockage specification of the "medium" processes. The actual process is not defined any further.

The characteristics of the medium that are assumed are that it provides unreliable but sequenced delivery of messages, the usual assumptions about a data link. In other words, messages may be lost or corrupted but not delivered out of order. This is conveniently expressed in Gypsy with the built in subsequence operator. It is a boolean operator which is true when one sequence is a (noncontiguous) subsequence of another. With this in mind the "medium" process is defined as follows.

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;

3 Verification Conditions

The foregoing discussion has sketched out the essential parts of the specifications needed for the protocol and the service it provides. However, in order to make use of the Gypsy verification methods, several gaps must be filled in. Specifically, these are the external and internal specifications of the sender and receiver processes when viewed as normal bodies of executing code. The external specifications are just assertions over the buffer histories, much the same as was described for the "ab_protocol" process. The internal specifications are simply the loop invariants required by the inductive assertion method for proving the sequential code of the sender and receiver. On the other hand, the timer process requires no additional specifications since it does not affect the safety property being proved.

These additional assertions in turn require additional specification function definitions for expressing the various relations among the buffer histories. These assertions and their
function definitions are presented in full in Appendix I. Also, an informal description of these functions can be found in Appendix II. A short summary of the entire set is presented below.

<table>
<thead>
<tr>
<th></th>
<th>ab_protocol</th>
<th>sender</th>
<th>receiver</th>
<th>medium</th>
<th>timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>External specs</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Internal specs</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Total specification functions: 11

The function "comp" is counted as a specification function although it is also used as an executable function by the sender and receiver processes.

The basic structure of the overall proof can now be represented by Figure 1.

![Diagram](image)

FIG. 1.

In Gypsy terms this corresponds to basically three proofs, one for each of the routines ab_protocol, sender, and receiver.

As the first step toward proving these routines, the verification conditions were generated by the Gypsy verifier.
The principles for generating VCs are based on the proof methods described in [Good 79]. Mechanical VC generation resulted in the following breakdown of VCs to be proved.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>9</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab.protocol</td>
<td>sender</td>
<td>receiver</td>
<td></td>
</tr>
</tbody>
</table>

Verification conditions

Actually, the receiver yielded 8 VCs but two of them are identical. Transcripts of VC generation are contained in the companion report [DiVito 81].

As is usually the case, the cobegin VC (ab protocol) is far more difficult to prove than the sequential VCs (sender, receiver). However, the sequential VCs are not easily outdone, and always manage to appear in far greater numbers.

4 Proof Description

Carrying out the verification means proving all of the verification conditions just described. Ordinarily, to obtain a fully mechanical verification from this point, one would simply charge ahead and push the VCs through the Gypsy interactive theorem prover. Unfortunately, the proof of the cobegin VC required doing some induction proofs which are not directly supported by the Gypsy prover. After enlisting the help of the Affirm verification system, it was found that the proof could in fact be completed mechanically. Although there was some hand translation of information across verification system boundaries, the verification that resulted can be regarded for all practical purposes as a fully mechanical proof.

4.1 Decomposition

Proving a set of nontrivial verification conditions typically requires an effective proof decomposition based on a set of supporting lemmas. Having a general strategy in mind will naturally make this decomposition a little more orderly. In fact, there is an analogous decomposition problem for specification functions since they control the level of detail which finds its way into the VCs. What was found to be useful for organizing these lemmas and functions is the following conceptual framework.

Specification functions and lemmas are viewed as falling into three classes which form a hierarchy of decreasing
generality. The first category could be termed "type-specific" and includes those concepts which deal with basic data types and structures in a generic way. An example would be the concept of initial subsequence for sequences of arbitrary types. Next are the "domain-specific" concepts which involve things found over a broad problem domain. Examples include concepts for sequence numbered messages in the domain of protocols, or more specifically link level protocols. The third class may be called "assertion-specific" for functions and "VC-specific" for lemmas. These are the least general and are devised intentionally for particular assertions or VCs.

If the categories are viewed as forming a pair of parallel hierarchies, they can be depicted as in Figure 2.

FIG. 2.

```
      Assertions -----> VCs
           |        |
           +-------+------+
        assertion-specific -----> VC-specific
                          |
                          |
      functions ----> domain-specific
                          |
                          |
                          |
                          type-specific ----->

Roughly speaking, an arrow denotes the relationship "is used by." Of course, not all of the functions and lemmas can be thrown into one group or another, but the scheme seems to be helpful for organizing things. Also, it provides a way of summarizing a proof's organization after the fact.

Returning to the problem at hand, it is now possible to give a summary for the decomposition of the proof, shown in Table 1. Of the lemmas enumerated above, all but four were proved mechanically with the Gypsy and Affirm theorem provers. Three of these had previously been proved and placed in the Affirm type library. The remaining lemma, which had to be assumed and thus forms the entire basis of the proof, is shown below.
Lemma bit_cases (b: bit) = b = zero or b = one

This is a property about the scalar type "bit" which should be regarded as an axiom about such types but is currently unknown to the Gypsy prover or simplifier.

<table>
<thead>
<tr>
<th>Specification functions</th>
<th>Lemma Gypsy</th>
<th>Lemma Affirm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion/VC-specific</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Domain-specific</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Type-specific</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Totals</td>
<td>11</td>
<td>28</td>
</tr>
</tbody>
</table>

4.2 Gypsy-Affirm Interface

Normal application of the Gypsy methods for verifying concurrent programs involves the proof of theorems containing various relations over buffer histories. Since these histories are instances of the sequence data type, often the heart of these proofs is showing that certain properties of specific sequence types hold. Many of these properties are best proved by structural induction, a proof method not currently supported by the Gypsy prover. The Affirm system, on the other hand, directly supports this kind of proof. Several lemmas of this type were required to complete the proof of the Alternating Bit Protocol so an attempt was made to apply the Affirm prover to this task.

The lemmas arose from the concurrent part of the proof, i.e. the verification condition for "ab_protocol." In the Gypsy proof, four lemmas were used and assumed and subsequently carried over to the Affirm system for proof. One of these can be thought of as the key lemma which makes the Alternating Bit Protocol work; the others are lesser theorems. In the proof of this key lemma, additional supporting Affirm lemmas were put forth and proved.

The procedure for combining proofs on the two systems was as follows.
1. The first step was to arrive at a common concept of sequences. A subset of the Gypsy sequence operators was chosen and mapped into the corresponding operators in the Affirm standard type specification for sequence.

2. This basic sequence type was then instantiated to get sequences with appropriate element types. The domain-specific and type-specific functions expressed in Gypsy were then mapped onto these type specifications as additional operators. The function definitions were introduced as axioms so they could be applied as rewrite rules.

3. Finally, the required lemmas were expressed in terms of these new types and functions and proofs were carried out with the Affirm prover. Intermediate lemmas were further introduced as needed.

Details of the translation of formalisms from Gypsy to Affirm are given in Appendix III. The Affirm type specifications which resulted from this mapping are displayed in Appendix IV. Complete documentation on Affirm notation and the use of the Affirm system is contained in [Thompson 81].

5 Conclusions

The Alternating Bit Protocol has been modeled using the Gypsy methodology for concurrent programming. A comprehensive safety property was stated and the actions of the protocol modules were specified in a procedural manner. A mechanical verification was performed which involved the use of both the Gypsy and the Affirm verification systems.

Two observations can be made regarding this effort. First, notice the relatively large size of the proof as reflected in the number of required lemmas. This is due in part to the nature of the problem and in part to the fact that it was a mechanical proof. Many of those things which we tend to dismiss as "obvious" facts when doing a hand proof turn out to require lengthy proofs when spelled out in detail. This can be considered as more evidence to support the need for developing high level theories for verification.

Second, the attempt to perform a combined Gypsy-Affirm proof was highly successful. Taking advantage of the specialized
capabilities of different verification systems makes good sense from a practical point of view. The principal drawback at this point is the need to do hand translation of concepts from one system to the other. In any event, it should be concluded that future combined work will undoubtedly be synergetic.

Experience with this example has proven to be extremely valuable. Comparison of one's methods with those of others is greatly facilitated when all attempt a common problem. Many strengths and weaknesses can be readily discerned from such trial applications. Large proofs of small problems give some cause for concern and provokes the search for improved and more specialized methods.

The work reported herein was based on the direct application of existing Gypsy methods and tools (and to a lesser extent those of Affirm). Current and future work is aimed at developing a specialized methodology for the problem of protocol specification and verification. It will be based on the Gypsy model but will contain adaptations to avoid the use of procedural forms of specification and to reduce the total amount of proof necessary. Nevertheless, the major goal is still the same, namely to obtain fully mechanical proofs.
APPENDIX I

TEXT OF GYPsy DESCRIPTION AND SUPPORTING LEMMAS

The model of the Alternating Bit Protocol as expressed in Gypsy is now presented in full. This text includes type declarations, procedure (process) definitions, all the necessary assertions for specification, function definitions used for expressing the assertions and all the supporting lemmas used in the Gypsy part of the proof. Refer to the Gypsy Report [Good 78b] for details on the syntax and semantics of the notation.

```
scope alt_bit_protocol =
begin

    type bit = (zero, one);
    type message = sequence of character;
    type packet = record (msg : 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 (in from (source, myid),
                                outto (pkt_send, myid),
                                in from (ack_send, myid), 1);
    exit false;
    var pack, ack : packet;
    var b : boolean;
    var next : bit := one;
    loop
        assert proper_transmission (in from (source, myid),
                                    outto (pkt_send, myid),
                                    in from (ack_send, myid), 0)
        & next = next_seqnum (outto (pkt_send, myid))
        & next = next_seqnum (in from (ack_send, myid));
        receive pack.msg from source;
        pack.seqno := next;
        send pack to pkt_send;
        send true to start;
        loop
            assert proper_transmission (in from (source, myid),
                                        outto (pkt_send, myid),
                                        in from (ack_send, myid), 1)
            & in from (source, myid)
            = unique_msg (outto (pkt_send, myid))
            & size (unique_msg (outto (pkt_send, myid)))
            = size (unique_msg (in from (ack_send, myid))) + 1
            & outto (pkt_send, myid) ne null (pkt_seq)
            & pack = last (outto (pkt_send, myid))
            & next = next_seqnum (in from (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
next := comp (next)
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;

{ Assertion-specific functions }

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;

{ Domain-specific functions }

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))
                fi
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;

{ Type-specific functions }

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 =
begin
  exit (assume result iff some s : msg_seq, u @ s = v);
end;

end;

scope lemmas =
begin

{ VC-specific lemmas }

lemma prop_trans_l (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)
\( \text{proper\_reception\ (u, x, y, l);} \)

\text{lemma prop\_rec\_4\ (u: msg\_seq; x, y: pkt\_seq; p: packet; m: message) =}
\[ p.\text{seqno} = \text{next\_seqnum\ (x)} \]
& \[ p.\text{seqno} = \text{next\_seqnum\ (y)} \]
& \[ \text{proper\_reception\ (u, x, y, 0) & p.mssg = m} \]
\( \rightarrow \text{proper\_reception\ (u <: m, x <: p, y <: p, 0);} \)

\text{lemma prop\_rec\_5\ (u: msg\_seq; x, y: pkt\_seq; p: packet) =}
\[ p.\text{seqno} \neq \text{next\_seqnum\ (x)} \]
& \[ p.\text{seqno} \neq \text{next\_seqnum\ (y)} \]
& \[ \text{proper\_reception\ (u, x, y, 0)} \]
\( \rightarrow \text{proper\_reception\ (u, x <: p, y <: p, 0);} \)

\text{lemma abp\_l\ (u, v: msg\_seq; w, x, y, z: pkt\_seq) =}
\[ \text{proper\_transmission\ (u, w, z, l)} \]
& \[ \text{proper\_reception\ (v, x, y, l)} \]
& \[ x \text{ sub } w \text{ & } z \text{ sub } y \]
\( \rightarrow \text{msg\_lag\ (v, u, l);} \)

{ Domain-specific lemmas }

\text{lemma main\_lemma\ (s: bit\_seq; x, y: pkt\_seq) =}
\[ s \text{ sub seqnums\ (x)} \text{ & } x \text{ sub } y \text{ & } \text{repeats\ (y)} \]
& \[ \text{nchanges\ (seqnums\ (y))} - \text{nchanges\ (s) in [0..1]} \]
\( \rightarrow \text{msg\_lag\ (unique\_msg\ (x), unique\_msg\ (y), 1);} \)

\text{lemma interpolate\ (s: bit\_seq; x, y: pkt\_seq) =}
\[ s \text{ sub seqnums\ (x)} \text{ & } x \text{ sub } y \text{ & } \text{repeats\ (y)} \]
& \[ \text{nchanges\ (seqnums\ (y))} - \text{nchanges\ (seqnums\ (x)) in [0..1]} \]
\( \rightarrow \text{nchanges\ (seqnums\ (y))} \)

\text{lemma next\_comp\ (x: pkt\_seq; p: packet) =}
\[ \text{next\_seqnum\ (x <: p)} = \text{comp\ (p.seqno)} \)

\text{lemma ne\_next\ (x: pkt\_seq; p: packet) =}
\[ p.\text{seqno} \neq \text{next\_seqnum\ (x)} \]
\( \rightarrow \text{next\_seqnum\ (x <: p)} = \text{next\_seqnum\ (x);} \)

\text{lemma last\_next\ (x: pkt\_seq; p: packet) =}
\[ x \text{ ne null\ (pkt\_seq) & p = last\ (x)} \]
\( \rightarrow p.\text{seqno} \neq \text{next\_seqnum\ (x);} \)
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);

{ Type-specific lemmas }

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);

{ Lemmas proved under Affirm }

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;
APPENDIX II

INFORMAL DESCRIPTION OF SPECIFICATION FUNCTIONS

As an aid to understanding the formal definitions, a brief informal description of each specification function is presented.

function proper_transmission (source : msg_seq;
   pkt_send, ack_send : pkt_seq;
   n : integer) : boolean

Describes the correct relations maintained by the sender. Loosely speaking, it expresses the following.

1) The "unique messages" (duplicates removed) sent toward the receiver lag those received from the source.

2) The difference in the number of bit changes between the packets sent and the acks received is bounded by n.

3) The difference between the number of bit changes in the acks and the number of messages received from the source is bounded by n.

4) Adjacent packets sent to the receiver having the same sequence number have the same message field.

function proper_reception (sink : msg_seq;
   pkt_rcv, ack_rcv : pkt_seq;
   n : integer) : boolean

Describes the correct relations maintained by the receiver.

1) Messages delivered to the sink lag the unique messages received from the sender.

2) The acks returned to the sender form a subsequence of the packets received.

3) The difference between the number of messages delivered and the bit changes in the acks is
bounded by \( n \).

```plaintext
function next_seqnum (ps : pkt_seq) : bit

Returns the bit value of the sequence number which should logically be the next one after those in ps.
```

```plaintext
function last_bit (bs : bit_seq) : bit

An extension of the "last" function meant to be defined even when the bit sequence is null.
```

```plaintext
function seqnums (ps : pkt_seq) : bit_seq

Extracts the sequence number fields from ps.
```

```plaintext
function nchanges (bs : bit_seq) : integer

Counts the number of times adjacent bits in the sequence are different.
```

```plaintext
function unique_msg (ps : pkt_seq) : msg_seq

Yields the "unique messages" contained in a packet sequence, those which coincide with changes in the sequence numbers.
```

```plaintext
function repeats (ps : pkt_seq) : boolean

True if each adjacent pair of packets with the same sequence number also have the same message field.
```

```plaintext
function msg_lag (s, t : msg_seq; n : integer) : boolean

True if \( s \) is an initial subsequence of \( t \) and their lengths differ by at most \( n \).
```

```plaintext
function initial_subseq (u, v : msg_seq) : boolean

Holds if \( u \) is an initial (contiguous) subsequence of \( v \).
```
function comp (b: bit): bit

Evaluates to the complement of the bit.
APPENDIX III
DETAILS OF GYPSY TO AFFIRM TRANSLATION

Some details of the correspondence between the Gypsy and Affirm notations are now presented. First of all, we itemize the sequence operators selected as the working subset and the equivalent notations used to represent them.

<table>
<thead>
<tr>
<th>Gypsy</th>
<th>Affirm</th>
</tr>
</thead>
<tbody>
<tr>
<td>type seq_type =</td>
<td>type SequenceOfElemType</td>
</tr>
<tr>
<td>sequence of elem_type</td>
<td></td>
</tr>
<tr>
<td>null(seq_type)</td>
<td>NewSequenceOfElemType</td>
</tr>
<tr>
<td>[seq: e]</td>
<td>seq(e)</td>
</tr>
<tr>
<td>s &lt;: e</td>
<td>s apr e</td>
</tr>
<tr>
<td>e :&gt; s</td>
<td>e apl1 s</td>
</tr>
<tr>
<td>sl @ s2</td>
<td>sl join s2</td>
</tr>
<tr>
<td>first(s)</td>
<td>First(s)</td>
</tr>
<tr>
<td>last(s)</td>
<td>Last(s)</td>
</tr>
<tr>
<td>nonfirst(s)</td>
<td>LessFirst(s)</td>
</tr>
<tr>
<td>nonlast(s)</td>
<td>LessLast(s)</td>
</tr>
<tr>
<td>e in s</td>
<td>e in s</td>
</tr>
<tr>
<td>sl sub s2</td>
<td>sl subseq s2</td>
</tr>
<tr>
<td>size(s)</td>
<td>Length(s)</td>
</tr>
</tbody>
</table>

Next, we detail the specific types and specification functions used for this Alternating Bit problem.

<table>
<thead>
<tr>
<th>Gypsy</th>
<th>Affirm</th>
</tr>
</thead>
<tbody>
<tr>
<td>type packet =</td>
<td>type Packet</td>
</tr>
<tr>
<td>record</td>
<td></td>
</tr>
<tr>
<td>(msg: message;</td>
<td></td>
</tr>
<tr>
<td>seqno: bit)</td>
<td></td>
</tr>
</tbody>
</table>
Finally, we present a brief outline of the proof interface between the Gypsy and Affirm parts of the overall proof. This forms a loose description of the connections between the two separate proof forests.

1. The proof of \texttt{main lemma} begins in the Gypsy prover. It is proved in terms of the lemmas \texttt{interpolate}, \texttt{sub_to_lag}, and \texttt{nchanges_unique}.

2. A proof of \texttt{interpolate} is done with the Gypsy prover, using \texttt{sub_seqnum} and \texttt{sub_nchanges}.

3. The lemmas \texttt{sub_to_lag}, \texttt{sub_seqnum}, \texttt{sub_nchanges}, and \texttt{nchanges_unique} are carried over to the Affirm system for proof.

4. The proof of \texttt{sub_to_lag} is decomposed through a number of lemmas introduced for the Affirm part of the proof.

5. Thus the proof forest of the Affirm part of the proof can be viewed as an extension of the overall proof which has its roots in the Gypsy portion.
APPENDIX IV
AFFIRM DEFINITIONS AND LEMMAS

Following is a list of the Affirm type definitions and lemmas. Several of the type specifications have many axioms in common, so in the interest of brevity they are listed only once. Much of the axiomatization is taken from the Affirm type libraries and extended for its application to this problem. See the Affirm manuals [Thompson 81] for an explanation of the notation.

type ElemType;
declare dummy: ElemType;
axiom dummy=dummy == TRUE;
end {ElemType} ;

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} ;
type Packet;

needs types Bit, ElemType;

declare dummy, p: Packet;

interface msg(p): ElemType;

interface seqno(p): Bit;

axiom dummy=dummy == TRUE;

end {Packet};

type SequenceOfElemType;

needs types Integer, ElemType;

declare dummy, ss, s, sl, s2, s3, s4, s5: SequenceOfElemType;

declare k, k1, k2: Integer;

declare i1, i, ii, i2, j: ElemType;

interfaces NewSequenceOfElemType, s apr i, i apl s, seq(i),
   sl 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, sl 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 = sl apr il == ((s=sl) and (i=il));

axioms i apl NewSequenceOfElemType ==
   NewSequenceOfElemType apr i,
   i apl (s apr il) == (i apl s) apr il;
axiom seq(i) == NewSequenceOfElementType apr i;

axioms NewSequenceOfElementType join s == s,  
    (s apr i) join sl == s join (i apr sl);

axiom LessFirst(s apr i)  
    == if s = NewSequenceOfElementType  
        then NewSequenceOfElementType  
        else LessFirst(s) apr i;

axiom LessLast(s apr i) == s;

axiom isNewSequenceOfElementType(s) ==  
    (s = NewSequenceOfElementType);

axioms sl subseq (s apr i)  
    == (  
            (sl = NewSequenceOfElementType) or sl subseq s  
            or LessLast(sl) subseq s and (Last(sl) = i)),  
    s subseq NewSequenceOfElementType ==  
        (s = NewSequenceOfElementType);

axioms i in NewSequenceOfElementType == FALSE,  
    i in (s apr il) == (i in s or (i=il));

axioms s iss NewSequenceOfElementType ==  
    (s = NewSequenceOfElementType),  
    sl iss (s2 apr i)  
    == (  
            (sl = NewSequenceOfElementType) or sl iss s2  
            or (LessLast(sl) = s2) and (Last(sl) = i));

axioms Length(NewSequenceOfElementType) == 0,  
    Length(s apr i) == Length(s) + 1;

axiom First(s apr i) == if s = NewSequenceOfElementType  
    then i  
    else First(s);

axiom Last(s apr i) == i;

rulelemmas NewSequenceOfElementType = i apr s == FALSE,  
    i apr s = NewSequenceOfElementType == FALSE;

rulelemmas s join (sl apr i) == (s join sl) apr i,  
    s join NewSequenceOfElementType == s,  
    (i apr sl) join s2 == i apr (sl join s2),  
    (s join (i apr sl)) join s2  
    == s join (i apr (sl 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 Last(s);

rulelemmas NewSequenceOfElemType subseq s == TRUE, s subseq s == TRUE;

rulelemma i in (il apl s) == (i in s or (i = il));

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(iii
            apl ss)),
    Induction(s) == cases(Prop(NewSequenceOfElemType),
        all ss, ii
        (    IH(ss)
            imp Prop(ii
                apr ii)),
    NormalForm(s) == cases(Prop(NewSequenceOfElemType),
        all ss, ii (Prop(ii
            ss
            apr ii)))
)

end {SequenceOfElemType} ;

type SequenceOfBit;
needs type Bit;

declare dummy, ss, s, s1, s2, s3, s4, s5: SequenceOfBit;
declare k, kl, k2: Integer;
declare ii, i, il, i2, j: Bit;

interfaces NewSequenceOfBit, s apr i, i apl s, seq(i),
sl join s2, LessFirst(s), LessLast(s):
SequenceOfBit;

infix join, apl, apr;

interfaces isNewSequenceOfBit(s), sl subseq s2,
FirstInduction(s), Induction(s), NormalForm(s),
i in s, sl iss s2: Boolean;

infix in, subseq, iss;

interfaces Nchanges(s), Length(s): Integer;

interfaces First(s), Last(s), LastBit(s): Bit;

{ All axioms and rulelemmas from SequenceOfElemType are
  included in this type; only the additional ones are
  listed here. }

axioms Nchanges(NewSequenceOfBit) == 0,
    Nchanges(s apr i)
    == if LastBit(s) = i
      then Nchanges(s)
      else Nchanges(s) + 1;

axioms LastBit(NewSequenceOfBit) == zero,
    LastBit(s apr i) == i;

end {SequenceOfBit} ;

type SequenceOfPacket;

needs types Integer, Packet, SequenceOfBit, SequenceOfElemType;

declare dummy, ss, s, s1, s2, s3, s4, s5: SequenceOfPacket;
declare k, kl, k2: Integer;
declare ii, i, il, i2, j: Packet;
interfaces NewSequenceOfPacket, s apr i, i apr s, seq(i),
   sl join s2, LessFirst(s), LessLast(s):
   SequenceOfPacket;

infix join, apr, ap1;

interfaces isNewSequenceOfPacket(s), sl subseq s2,
   FirstInduction(s), Induction(s), NormalForm(s),
   i in s, sl 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;

{ All axioms and rulelemmas from SequenceOfElemType are
  included in this type; only the additional ones are
  listed here. }

axioms Repeats(NewSequenceOfPacket) == TRUE,
   Repeats(s apr i)
   == (   Repeats(s)
       and seqo(i) == seqo(Last(s))
       imp i == Last(s));

axioms Seqnums(NewSequenceOfPacket) == NewSequenceOfBit,
   Seqnums(s apr i) == Seqnums(s) apr seqo(i);

axioms UniqueMsg(NewSequenceOfPacket) ==
   NewSequenceOfElemType,
   UniqueMsg(s apr i)
   == if seqo(i) == LastBit(Seqnums(s))
       then UniqueMsg(s)
       else UniqueMsg(s) apr dmsg(i);

end {SequenceOfPacket} ;

type AebpContext;
needs types Bit, Integer, Packet, SequenceOfElemType, SequenceOfBit, SequenceOfPacket, ElemType;

declare dummy: AbpContext;
declare b, bl, b2: Bit;
declare i, i1, i2, j, k, k1, k2: Integer;
declare p, pl, p2, iS: Packet;
declare m, ml, m2: SequenceOfElemType;
declare n, nl, n2: SequenceOfBit;
declare s, sl, s2: SequenceOfPacket;
declare e, el, e2: ElemType;

interfaces Bounded(k1, k2, j), MsgLag(ml, 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(n, m, 1) == TRUE,
       MsgLag(n, m apr e, 1) == TRUE,
       MsgLag(n apr e, m, 1) == FALSE;

define Bounded(k1, k2, j)
       == ((k1 <= k2) and (k2 <= j+k1)),
           MsgLag(ml, m2, j)
       == (ml iss m2 and Bounded(Length(ml),
                        Length(m2), j));

end {AbpContext} ;

{ Domain-specific lemmas }

theoren SubToLag, sl subseq s2 and Repeats(s2)
       and Bounded(Nchanges(Seqnums(s1)),
                        Nchanges(Seqnums(s2)), 1)
       imp UniqueMsg(s1) iss UniqueMsg(s2);

theoren PktSubBound, sl 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, nl subseq n2 and Bounded(Nchanges(n1), Nchanges(n2), 1) 
imp LastBit(n1) = LastBit(n2) eqv Nchanges(n1) = Nchanges(n2);

theorem SubLess, nl subseq n2 imp Nchanges(n1) <= Nchanges(n2);

theorem SameLastBit, nl subseq n2 and b = LastBit(n2) and \(\neg((nl\lross b)\ subseq \ n2)\) 
imp LastBit(n1) = LastBit(n2);

theorem SameLastBit2, 
sl subseq s2 and p = Last(s2) and \(\neg((sl\lross p)\ subseq \ s2)\) 
imp LastBit(Seqnums(sl)) = LastBit(Seqnums(s2));

theorem SubSeqnum, sl subseq s2 
imp Seqnums(sl) 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;
{ Type-specific lemmas }

theorem BoundedByl, Bounded(k1, k2, l) 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 =\equiv b1) and (b =\equiv b2) imp b1=b2;
REFERENCES

A Note on Reliable Full Duplex Transmission over
Half Duplex Links.

A Verified Method for the Specification and
Verification of Protocols.

[DiVito 81] B. L. DiVito.
Transcripts for the Proof of the Alternating Bit
Protocol.
Technical Report ICSCA-CMP-22, The University of
Texas at Austin, June, 1981.

An Overview of Affirm: A Specification and
Verification System.
In Proc. IFIP. IFIP, October, 1980.

Constructing Verified and Reliable Communications
Processing Systems.
ACM Sigsoft Software Eng. Notes 2(5), October,
1977.

Verifiable Communications Processing in Gypsy.

[Good 78b] D. I. Good, R. M. Cohen, C. G. Hoch, L. W. Hunter,
D. F. Hare.
Technical Report ICSCA-CMP-10, University of Texas
at Austin, September, 1978.

Principles of Proving Concurrent Programs in
Gypsy.
Technical Report ICSCA-CMP-15, University of Texas
at Austin, January, 1979.
Abstract Data Type Specification in the Affirm System.


Formal Techniques for Protocol Specification and Verification.

The Affirm Reference Library
USC-ISI, 1981.
Five volume set.