General Message Flow Modulator

Ann E. Siebert
Donald I. Good

Technical Report #42 March 1984

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

Abstract

The general message flow modulator is a high level design of a family of mechanisms for controlling the flow of messages from a source to a destination. This family of mechanisms has a wide variety of computer security applications. The general modulator has been formally specified and proved, and one member of the family has been implemented as a running prototype to give a concrete illustration of the kinds of control that can be provided. The general modulator has been specified and implemented in Gypsy, and the proofs have been done mechanically within the Gypsy Verification Environment.

Acknowledgements

The development of the general message flow modulator was sponsored by the U. S. Naval Electronic Systems Command under Contract N00039-81-C-0074.

Table of Contents

1. Introduction

The general Message Flow Modulator (MFM) is a high level design for a family of programs which control the flow of messages from a source to a sink (or destination).

```
 ------------     -------------------------------     ------------
 |          |     |                             |     |          |
 |          |     |           Message           |     |          |
 |  Source  |---->|            Flow             |---->|   Sink   |
 |          |     |          Modulator          |     |          |
 |          |     |                             |     |          |
 ------------     -------------------------------     ------------
                       ^     |                |
                       |     |                |
                       |     v                v
                  -------------            -------
                  |Con-  |Dis- |           |     |
                  |sole  |play |           | Log |
                  |      |     |           |     |
                  -------------            -------
```

The flow is controlled by "modulating" each message received from the source. Modulating a message consists of the following steps:

```
In-msg          ---------- Pass   -----------         Out-msg
From    ------->| Filter |------->|Transform|------->To
Source          ----------        -----------         Sink
                    | Reject
                    V
```

1. A filter is applied to each message from the source to determine if it should be passed to the sink or rejected.

2. If the filter passes the input message, a transform function is applied to the message, and the transformed message is forwarded to the sink.

3. If the filter rejects the input message, nothing is forwarded to the sink.

Message flow is strictly in one direction, from source to sink. The modulator is controlled from an operator console with various information for the operator being shown on the display, and an audit trail of key events is recorded on the log.

The modulator operates in one of two modes, manual or automatic, which is selected by the operator. The modulator contains several predefined filters and transforms, which are defined when the modulator is built (compiled). In automatic mode, the modulator continuously modulates messages using the filter and transform that are currently selected by the operator. The operator may switch out of automatic mode at any time. In manual mode, the message filtering and transformation is done manually by the operator. The operator decides whether to pass or reject each message, and performs any desired transformation, possibly by editing or sanitizing the message. This manual mode provides much of the same capabilities as the LSI Guard [Craigen 82].

The general MFM is a design for an entire family of message flow modulators. The design has left pending the actual choices of input and output message format, the predefined filter and transform functions, operator console interaction, and content of the audit trail. (In the general MFM design, the number of predefined defined filters and transforms is limited to 5 each, but the limit can be changed easily, only being restricted by the amount of space needed for their object code.) Each particular choice of these pending items is one member of the family. The specifications, design and proof that are given are valid for every member of the family. What has been specified and proved for the family is that the appropriate filter and transform are applied to every incoming message. To complete the proof for a particular member of the family, the pending items would have to be specified, implemented and proved. One member of the family has been implemented (but not specified or proved) to serve as a running prototype to illustrate the basic capabilities of this family of modulators.

The members of the general modulator family can be applied to a variety of computer security applications. There are no restrictions on the choices of predefined filters and transforms. They may be as simple or as complex as desired. A transform, for example, might be an encryption algorithm. As another example, a transform might be used for changing the security level of input messages, either an upgrade or a downgrade. The security levels of the input and output messages could be either fixed or designated on each individual message.

In general, an instance of the general modulator performs both filtering and transformation. However, a modulator whose filter passes all messages operates strictly as a transformer, and a modulator whose transformation is the identity function operates strictly as a filter. The filter that passes all messages together with the identity transform gives a simple connector between source and sink. One modulator controls flow in one direction. If control of bidirectional flow is needed, two modulators can be run concurrently. Note, however, that there is no direct communication between the two modulators. They operate completely independently.

The general modulator has been developed for the United States Naval Electronic Systems Command (NAVELEX) to illustrate the present capabilities of formal specification and proof methods in the design of computer security software. The design and formal specifications were written in the Gypsy language [Good 78], and proofs that the design meets the specifications were accomplished in the Gypsy Verification Environment (GVE). In addition, a completed prototype, representing one member of the general MFM family, has been developed and runs on a DEC-20 under TOPS-20 at Utexas-20 on the ARPANET.

The MFM design described in this report follows from an initial modulator design described in [Good 81]. Modifications to the original design are based on experience gained while developing a modulator to monitor the flow of security sensitive message traffic from the Ocean Surveillance Information System (OSIS) of NAVELEX. The verified OSIS modulator is described in [Good 82].

2. Specifications of General MFM

The top level specification of the general MFM defines modulator output to the sink, log, and display as a function of input from the source and operator console.

```
function modulated_flow (      xs : a_timed_source_seq;
                               xc : a_timed_console_seq;
                             sink : a_sink_seq;
                          display : a_display_seq;
                              log : a_log_seq;
                              st0 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume result = some h: an_input_event_seq,
                          h = event_pack (merged_inputs(xs,xc), st0, t)
                        & sink = all_modulated (h, st0, t)
                        & log = initial_log (t) @ all_logged (h, st0, t)
                        & display = initial_display (t)
                                    @ all_displayed (h, st0, t));
end;
```

Here, "xs" is the time-stamped sequence of objects received from the source, "xc" the time-stamped sequence of objects received from the operator console. Time-stamps indicate the order of events; more recently received objects have larger time stamps than objects that were received earlier. "Sink" is the sequence of objects sent to the MFM destination, "display" is the sequence of objects displayed to the MFM operator, and "log" is the sequence of objects recorded on the audit trail. MFM output is a function of the initial modulator state ("st0"), any information supplied through the modulator constant parameter ("t"), and the sequence of input events ("h"). The modulator constant parameter is included in case some constant information needs to be supplied to the program. For example, in the OSIS modulator [Good 82], the table of patterns, which forms the basis of the message filtering mechanism, is supplied as a parameter.

2.1 Specification of Input

Each object received from the source or console represents an input event. Input events consist of partial messages, completed messages, partial commands, and completed commands. The nature of input objects and details of packing them into messages and commands are unspecified, making the MFM specifications independent of message and command formats and contents.

For example, a message might consist of four blocks - b1, b2, b3, and b4, giving rise to three partial-message events when b1, b2, and b3 are received and one completed-message event when b4 is received and the message is completed. Alternatively, a message might be a sequence of characters such as "ZCZC This is a msg. NNNN", giving rise to one partial-message event for each character except the last and one completed-message event when the last character is received and the message is completed.

The MFM input specification begins by merging the source and console input objects into one sequence in time-stamp order, that is in order of receipt. It is supposed that a common clock is used for assigning timestamps to source and console inputs.

```
function merged_inputs (s : a_timed_source_seq;
                        c : a_timed_console_seq) : an_input_seq =
begin
  exit (assume
    result =
      if s = null(a_timed_source_seq) & c = null(a_timed_console_seq) then
        null(an_input_seq)
      else if console_last (s,c) then
             merged_inputs (s, nonlast(c)) <: make_console_input (last(c))
           else
             merged_inputs (nonlast(s), c) <: make_source_input (last(s))
      fi   fi);
end;


function console_last (s : a_timed_source_seq;
                       c : a_timed_console_seq) : boolean =
begin
  exit (assume result = if c = null(a_timed_console_seq) then false
                        else if s = null(a_timed_source_seq) then true
                             else timestamp(last(s)) < timestamp(last(c))
                        fi   fi);
end;
```

Then the merged sequence ("inp") of input objects is converted to a sequence of input events.

```
function event_pack (inp : an_input_seq;
                     st0 : a_modulator_state;
                       t : a_modulator_constant) : an_input_event_seq =
begin
  exit (assume result =
                 if inp = null(an_input_seq) then
                   null(an_input_event_seq)
                 else event_pack (nonlast(inp), st0, t)
                        <: new_event (last(inp),
                                      event_pack (nonlast(inp),st0,t),
                                      st0, t)
                 fi);
end;
```

Objects received from the source become message events, and objects received from the console become command events.

```
function new_event (  i : an_input;
                      h : an_input_event_seq;
                    st0 : a_modulator_state;
                      t : a_modulator_constant) : an_input_event =
begin
  exit (assume result = if is_source_input (i) then
                          make_msg_event (source_part(i), partial_msg(h),
                                          flow_state (h, st0, t))
                        else {is_console_input(i)}
                          make_cmd_event (console_part(i), partial_cmd(h))
                        fi);
end;
```

A message event is either a partial message or a completed message. The message event consists of the object received from the source, source_part(i), the message or partial message that results when the message being assembled is updated with the new source object, and a tag to indicate whether the message is completed or not.

```
function make_msg_event ( x : a_source_object;
                          m : a_message;
                         st : a_modulator_state) : an_input_event =
begin
  exit (assume result = initial(an_input_event) with
                          (.source_object := x;
                           .message := updated_msg (x,m,st);
                           .tag := message_tag (x,m,st)));
end;
```

The message being assembled is extracted from the sequence of previous input events ("h").

```
function partial_msg (h : an_input_event_seq) : a_message =
begin
  exit (assume result =
                 if h = null(an_input_event_seq) then null(a_message)
                 else if is_message (last(h)) then null(a_message)
                      else if is_partial_message (last(h)) then
                             message (last(h))
                           else partial_msg (nonlast(h))
                 fi   fi   fi);
end;
```

Of course, if there have been no previous input events (h is null) or if the last message has been completed (is_message(last(h))), there is no partial message (that is, the partial message is null).

A command event is either a partial command or a completed command. Command packing is similar to message packing. See Appendix A for details.

2.2 Specification of Sink Output

Every message received from the source is modulated, and the result of the modulation is sent to the sink (message destination). Modulation results are sent out in exactly the same order as they are received.

```
function all_modulated ( h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_sink_seq =
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then
        null(a_sink_seq)
      else
          modulate_it (first(h), st, t, modulation_cmds(h,st,t))
        @ all_modulated (nonfirst(h), flow_transition(first(h),st,t), t)
      fi);
end;
```

Modulation of each message is with respect to the modulator constant parameter and the modulator state, which is a function of the initial modulator state and previous input from the source and console. Permitting the modulation to depend on console input is what allows the selection of operating mode (automatic or manual) and of various filters and transforms by the console operator. Manual modulation of a message is also with respect to commands that are received from the operator console after the message is received but before any other message or partial message is received.

```
function modulation_cmds ( h : an_input_event_seq;
                          st : a_modulator_state;
                           t : a_modulator_constant) : an_input_event_seq=
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then
        null(an_input_event_seq)
      else if need_mod_cmds (first(h), st) then
             mod_cmd_head (nonfirst(h), flow_transition(first(h),st,t), t)
           else null(an_input_event_seq)
      fi   fi);
end;


function need_mod_cmds ( e : an_input_event;
                        st : a_modulator_state) : boolean =
begin
  exit (assume result = [  is_message(e)
                         & not in_auto_mode (source_transition(e,st))]);
end;
```

```
function mod_cmd_head ( h : an_input_event_seq;
                       st : a_modulator_state;
                        t : a_modulator_constant) : an_input_event_seq =
begin
  exit (assume result = if need_all_cmds (h, st, t) then h
                        else mod_cmd_head (nonlast(h), st, t)
                        fi);
end;
```

```
function need_all_cmds ( h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = if h = null(an_input_event_seq) then true
             else   not is_msg_event (last(h))
                  & have_in_msg (flow_state (nonlast(h),st,t))
                  & need_all_cmds (nonlast(h), st, t)
             fi);
end;
```

The existence of a pending message is to be noted in the modulator state.

Each message is modulated either automatically or manually, according to the operating mode previously selected by the MFM operator and stored in the modulator state.

```
function modulate_it ( e : an_input_event;
                      st : a_modulator_state;
                       t : a_modulator_constant;
                       h : an_input_event_seq) : a_sink_seq =
begin
  exit (assume
    result =
      if not is_message(e) then null(a_sink_seq)
      else if in_auto_mode(source_transition(e,st)) then
             auto_modulation (message(e), source_transition(e,st), t)
           else
             manual_modulation (message(e), flow_transition(e,st,t), h, t)
      fi   fi);
end;
```

There is no sink output for input events that are not messages. Non-message input events are commands, partial commands, and partial messages.

When the MFM is running in automatic mode, the filter selected by the MFM operator is applied to each message. If the message passes the filter, the automatic transform selected by the MFM operator is applied and the resulting transformed message is sent to the sink. If the message does not pass the filter, nothing is sent to the sink.

```
function auto_modulation ( m : a_message;
                          st : a_modulator_state;
                           t : a_modulator_constant) : a_sink_seq =
begin
  exit (assume
    result = if auto_filter (m,st,t) = passed then
               sink_seq (auto_transform(m,auto_filter_transition(st,t),t))
             else null(a_sink_seq)
             fi);
end;


function auto_filter ( m : a_message;
                      st : a_modulator_state;
                       t : a_modulator_constant) : a_msg_disposition =
begin
  exit (assume result = if filter_selector(st) = 1 then
                          auto_filter_1 (m,st,t)
                        else if filter_selector(st) = 2 then
                          auto_filter_2 (m,st,t)
                        else if filter_selector(st) = 3 then
                          auto_filter_3 (m,st,t)
                        else if filter_selector(st) = 4 then
                          auto_filter_4 (m,st,t)
                        else auto_filter_5 (m,st,t)
                        fi fi fi fi);
end;


function auto_transform ( m : a_message;
                         st : a_modulator_state;
                          t : a_modulator_constant) : a_transformed_msg =
begin
  exit (assume result = if transform_selector(st) = 1 then
                          auto_transform_1 (m,st,t)
                        else if transform_selector(st) = 2 then
                          auto_transform_2 (m,st,t)
                        else if transform_selector(st) = 3 then
                          auto_transform_3 (m,st,t)
                        else if transform_selector(st) = 4 then
                          auto_transform_4 (m,st,t)
                        else auto_transform_5 (m,st,t)
                        fi fi fi fi);
end;
```

Details of the automatic filters and transforms are not defined, making the general MFM specifications independent of any particular filters and transforms.

When the MFM is running in manual mode, the filter function requires input from the operator console. Manual modulation is defined only when the sequence of input events ("h") contains a manual filter command, showing that the MFM operator has passed or rejected the current message.

```
function manual_modulation ( m : a_message;
                            st : a_modulator_state;
                             h : an_input_event_seq;
                             t : a_modulator_constant) : a_sink_seq =
begin
  exit (assume
       did_manual_modulation (h, m, st, t)
    ->
       result =
        if manual_filter(m,h,st,t) = passed then
         sink_seq (manual_transform (m, initial_transform(m,t), h, st, t))
        else null(a_sink_seq)
        fi);
end;


function did_manual_modulation ( h : an_input_event_seq;
                                 m : a_message;
                                st : a_modulator_state;
                                 t : a_modulator_constant) : boolean =
begin
  exit (assume result = [  have_in_msg(st) & in_msg(st) = m
                         & manual_filter(m,h,st,t) in [passed,rejected]]);
end;


function manual_filter ( m : a_message;
                         h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then undecided
      else if need_all_cmds(h,st,t) then
             manual_filter_result (m, last(h),
                                   flow_state(nonlast(h),st,t), t)
           else manual_filter (m, nonlast(h), st, t)
      fi   fi);
end;
```

If a message passes the manual filter, the manual transform of the message is sent to the sink. If the message does not pass the filter, nothing is sent to the sink. The manual transform function allows a default message transform in case no transform command is issued from the operator console.

```
function manual_transform ( m : a_message;
                           tm : a_transformed_msg;
                            h : an_input_event_seq;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg=
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then tm
      else if need_all_cmds(h,st,t) then
             msg_transform (m, manual_transform(m,tm,nonlast(h),st,t),
                            last(h), flow_state(nonlast(h),st,t), t)
           else manual_transform (m, tm, nonlast(h), st, t)
      fi   fi);
end;
```

The manual modulation specification depends (for reasonableness) on the existence of a manual filter command (or commands), but the MFM command structure is not otherwise restricted.

2.3 Specification of Log Output

All input events are audited by recording them on the "log". What is specified is that each input event is considered for auditing. Details of log format and content are left pending so that the MFM specifications and proof are valid for any type of audit trail.

```
function all_logged ( h : an_input_event_seq;
                     st : a_modulator_state;
                      t : a_modulator_constant) : a_log_seq =
begin
  exit (assume
    result = if h = null(an_input_event_seq) then
                null(a_log_seq)
             else all_logged (nonlast(h), st, t)
                    @ log_it (last(h), flow_state(nonlast(h),st,t), t)
             fi);
end;
```

The log output for each input event is a function of the event, the current modulator state, and the modulator constant parameter.

```
function log_it ( e : an_input_event;
                 st : a_modulator_state;
                  t : a_modulator_constant) : a_log_seq =
begin
  exit (assume result = if is_msg_event(e) then
                          message_log (e, st, t)
                        else {is_cmd_event(e)}
                          command_log (e, st, t)
                        fi);
end;
```

The audit trail for command events (Command_Log) is not further specified. Each complete command and each partial command, that is each object received from the console, is considered for auditing, but the content and format of log output is not restricted in any way.

The audit trail for message events, partial messages and complete messages, is further specified in Function Message_Log.

```
function message_log ( e : an_input_event;
                      st : a_modulator_state;
                       t : a_modulator_constant) : a_log_seq =
begin
  exit (assume result = if is_message(e)
                           & in_auto_mode(source_transition(e,st)) then
                          source_log(e,st)
                           @ auto_log (source_transition(e,st), t)
                        else source_log(e,st)
                        fi);
end;
```

Each object received from the source is considered for auditing (Source_Log). Each individual source object that is part of a message, only complete messages, or nothing at all may be recorded on the audit trail. What the audit trail for source input consists of depends on further refinement of the specification for each member of the general MFM family.

When the modulator is running in automatic mode, the modulation of each complete message also is considered for the audit trail.

```
function auto_log (st : a_modulator_state;
                    t : a_modulator_constant) : a_log_seq =
begin
  exit (assume
    result =
      if msg_disposition (auto_filter_transition(st,t)) = passed then
          auto_filter_log (st,t)
        @ auto_transform_log (auto_filter_transition(st,t), t)
        @ sink_log (auto_transform_transition
                      (auto_filter_transition(st,t),t))
      else auto_filter_log (st,t)
      fi);
end;
```

The message being logged is included in the modulator state ("st"). The log output consists of a report from the automatic filter (Auto_Filter_Log), and, if the message passes the filter, reports from the automatic transform (Auto_Transform_Log) and the sink handler (Sink_Log). Again, details of log output will be supplied by further refinement of the specification for each member of the general MFM family.

When the modulator is running in manual mode, output to the sink and the filter and transform functions depend on commands from the operator console. Thus, the audit trail for manual modulation of a message is to be defined in the specification of command audit when it is developed for each member of the general MFM family.

2.4 Specification of Display Output

The specification of display output follows the same form as specification of log output, which was described in the previous section. The display specification can be seen in Appendix A, beginning with Function All_Displayed of Scope Display_Specifications.

3. Detailed Design of General MFM

The design of the general Message Flow Modulator consists of higher level routines of the program written in the Gypsy language [Good 78]. Gypsy text for the design and specifications is given in Appendix A. The general structure of the design is the following:

```
                       --------------------
                       |     Modulator    |
                       --------------------
                                 |
                       --------------------
                       | Modulate_Message |
                       --------------------
                                 |
         -------------------------------------------------
         |                       |                       |
--------------------     -----------------      -------------------
| Receive_Message |      | Auto_Modulate |      | Manual_Modulate |
--------------------     -----------------      -------------------
         |                       |                       |
         |        --------------------------------   -----------
         |        |              |              |   |          |
         |    -----------    -------------    -----------      |
         |    | Filter_ |    | Transform |    |  Send_  |      |
         |    | Message |    | _Message  |    | to_Sink |      |
         |    -----------    -------------    -----------      |
         |                                          |          |
    ----------------------------------------------  |          |
    |                                            |  |          |
------------------                             ------------------
| Update_Message |                             | Follow_Command |
------------------                             ------------------
```

3.1 Top Levels

Procedure Modulator is the top level routine. It simply initializes the program state and then continuously modulates messages.

```
procedure modulator (var  source : a_source_buffer<input>;
                     var    sink : a_sink_buffer<output>;
                     var console : a_console_buffer<input>;
                     var display : a_display_buffer<output>;
                     var     log : a_log_buffer<output>;
                           table : a_modulator_constant)
        unless (incorrect_table) =
begin
  var state: a modulator_state;
  initialize (state, display, log, table) unless (incorrect_table);
  loop
    if stop_modulation (state) then leave
    else
      modulate_message (source, sink, console, display, log, state, table,
                        max_time_stamp (xinfrom(source,myid),
                                        xinfrom(console,myid)))
    end;
  end;
end;
```

So that the MFM proof can be based on exit specifications, Procedure Modulator supposes that modulation can be discontinued by setting a flag in the program state; this flag is set when stop_modulation(state) is true. Modulator has a parameter "table" in case some constant information needs to be supplied to the program. For example, in the OSIS modulator [Good 82], the table of patterns, which forms the basis of the message filtering mechanism, is supplied as a parameter.

At the next level of the design is the procedure, Modulate_Message, which modulates one message.

```
procedure modulate_message (var  source : a_source_buffer<input>;
                            var    sink : a_sink_buffer<output>;
                            var console : a_console_buffer<input>;
                            var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant;
                                   time : integer) =
begin
  cond stop_mod;
  receive_message (source, console, display, log, state, table)
        unless (stop_mod);
  if in_auto_mode (state) then
    auto_modulate (sink, console, display, log, state, table,
                   max_time_stamp(xinfrom(source,myid),
                                  xinfrom(console,myid)))
  else manual_modulate
         (sink, console, display, log, state, table,
          max_time_stamp(xinfrom(source,myid),xinfrom(console,myid)))
  end;
when is stop_mod:
end;
```

A message is received and either automatically or manually modulated, depending on whether the MFM is running in automatic or manual mode.

3.2 Message Reception

A message is assumed to be an entity that can be constructed from a sequence of source objects. The nature of the source objects and of the construction process is undefined, making the general MFM independent of message structure. Procedure Receive_Message packs source objects into messages:

```
procedure receive_message (var  source : a_source_buffer<input>;
                           var console : a_console_buffer<input>;
                           var display : a_display_buffer<output>;
                           var     log : a_log_buffer<output>;
                           var   state : a_modulator_state;
                                 table : a_modulator_constant)
        unless (stop_mod) =
begin
  entry no_pending_modulation(state) & correct_table(table);
  loop
    if have_in_msg(state) then leave
    elif stop_modulation(state) & no_in_msg(state) then
      signal stop_mod
    elif not empty(console) then
      follow_command
        (console, display, log, state, table,
         max_time_stamp (xinfrom(source,myid), xinfrom(console,myid)))
    elif not empty(source) then
      update_message
        (source, display, log, state,
         max_time_stamp (xinfrom(source,myid), xinfrom(console,myid)))
    end;
  end;
end;
```

The Receive_Message entry specification no_pending_modulation(state) requires that "state" contain no message or partial message when the procedure is called.

A message is accumulated in the modulator "state" by Procedure Update_Message, which also does log and display output required for source input. Each individual source object can be audited or the audit may be limited to complete messages.

Since receiving a message depends on input from an external source and so is subject to unlimited delay, Receive_Message also monitors the operator console for commands, which are executed as they are entered. It is through commands from the operator console that the MFM operating mode (automatic or manual) and automatic filter and transform functions are selected.

3.3 Automatic Modulation

When the modulator is running in automatic mode, the automatic filter previously selected by the MFM operator is applied to each message. If the message passes the filter, the automatic transform selected by the MFM operator is applied to the message and the transformed message is sent to the sink. If the message does not pass the filter, nothing is sent to the sink and the rejected message is discarded. Automatic modulation is done by the following procedure:

```
procedure auto_modulate (var    sink : a_sink_buffer<output>;
                         var console : a_console_buffer<input>;
                         var display : a_display_buffer<output>;
                         var     log : a_log_buffer<output>;
                         var   state : a_modulator_state;
                               table : a_modulator_constant;
                                time : integer) =
begin
  filter_message (display, log, state, table);
  if msg_disposition (state) = passed then
    transform_message (display, log, state, table);
    send_to_sink (sink, console, display, log, state, table, time)
  else remove_message (state)
  end;
end;
```

The message to be modulated is expected to be stored in "state", and the transformed message is also stored in "state".

The MFM operator's selection of a filter function to be used for automatic modulation is recorded in "state", and Procedure Filter_Message calls the appropriate function for the selected filter.

```
procedure filter_message (var display : a_display_buffer<output>;
                          var     log : a_log_buffer<output>;
                          var   state : a_modulator_state;
                                table : a_modulator_constant) =
begin
  case filter_selector(state)
    is 1:  do_auto_filter_1 (display, log, state, table);
    is 2:  do_auto_filter_2 (display, log, state, table);
    is 3:  do_auto_filter_3 (display, log, state, table);
    is 4:  do_auto_filter_4 (display, log, state, table);
    else:  do_auto_filter_5 (display, log, state, table);
  end;
end;
```

Likewise, Procedure Transform_Message selects the transform function previously set by the MFM operator and recorded in "state".

```
procedure transform_message (var display : a_display_buffer<output>;
                             var     log : a_log_buffer<output>;
                             var   state : a_modulator_state;
                                   table : a_modulator_constant) =
begin
  case transform_selector(state)
    is 1:  do_auto_transform_1 (display, log, state, table);
    is 2:  do_auto_transform_2 (display, log, state, table);
    is 3:  do_auto_transform_3 (display, log, state, table);
    is 4:  do_auto_transform_4 (display, log, state, table);
    else:  do_auto_transform_5 (display, log, state, table);
  end;
end;
```

The number of both filter and transform functions is limited to five, but is easily changeable. The audit trail and display output for the filter and transform parts of modulation are done by the individual filter and transform procedures.

3.4 Manual Modulation

When the modulator is operating in manual mode, the console operator decides whether to pass each message to the sink or to reject and discard it. The operator also does any needed transformations on the message.

```
procedure manual_modulate (var    sink : a_sink_buffer<output>;
                           var console : a_console_buffer<input>;
                           var display : a_display_buffer<output>;
                           var     log : a_log_buffer<output>;
                           var   state : a_modulator_state;
                                 table : a_modulator_constant;
                                  time : integer) =
begin
  initialize_manual_modulation (state, table);
  loop
    if msg_disposition(state) ne undecided then leave
    else
      follow_command (console, display, log, state, table,
                      max_time_stamp (null_source, xinfrom(console,myid)))
    end;
  end;
  if msg_disposition(state) = passed then
    send_to_sink (sink, console, display, log, state, table,
                  max_time_stamp (null_source, xinfrom(console,myid)))
  else remove_message (state)
  end;
end;
```

Manual modulation begins by noting that the message stored in "state" has been neither passed nor rejected and that no transformations have been applied to the message (Initialize_Manual_Modulation). Then commands, entered through the operator console, are executed until the message is either passed or rejected. If the message is passed, the output (transformed) message stored in "state" is sent to the sink. Otherwise, nothing is sent to the sink, and the message is discarded.

3.5 Message Output

Output to the message destination (sink) consists of the transformed messages that pass the security filter, the preselected automatic filter in automatic mode or the instruction of the console operator in manual mode. Procedure Send_to_Sink sends one of these messages to the sink.

```
procedure send_to_sink (var    sink : a_sink_buffer<output>;
                        var console : a_console_buffer<input>;
                        var display : a_display_buffer<output>;
                        var     log : a_log_buffer<output>;
                        var   state : a_modulator_state;
                              table : a_modulator_constant;
                               time : integer) =
begin
  var m: a_sink_seq;
  var i: integer := 1;
  prepare_sink_send (display, log, state, m);
  loop
    if i > size(m) then leave
    elif not empty(console) then
      follow_command (console, display, log, state, table,
                      max_time_stamp (null_source, xinfrom(console,myid)))
    elif not full(sink) then
      send m[i] to sink;
      i := i + 1
    end;
  end;
end;
```

Send_to_Sink first calls Prepare_Sink_Send, which constructs the audit trail (log output) and display output required for messages sent to the sink. Prepare_Sink_Send also converts the output (transformed) message stored in "state" to a sequence of sink objects (m). The sequence of sink objects may be identical to the transformed message (which may be identical to the input message), or it may be different. For example, sink output could be a sequence of segments or blocks constructed from an unsegmentized or unblocked message. After the sequence of sink objects is constructed, its elements are sent to the sink one by one until the whole message has been sent. Procedure Send_to_Sink also monitors the operator console for commands because output to the sink, an external device, is subject to unlimited delays. Commands are executed as they are received.

4. Proof of General MFM

The proof of the general MFM is a collection of proofs that each Gypsy procedure in the design meets its specifications. Formal specifications have been written for every Gypsy procedure in the design, and mechanical proofs for each procedure have been constructed in the GVE [Siebert 84a].

Constructing a proof in the GVE is a two stage process. First, each procedure and its specifications are fed to a verification condition (VC) generator. The VC generator traces all paths through the procedure and automatically constructs VC's, which are logical formulas sufficient to establish that execution of the procedure will always give the results described in the specifications. In the second stage, the interactive theorem prover in the GVE is used to prove the VC's by applying standard mathematical techniques.

The VC generator gave 51 verification conditions for the MFM. The VC's and 38 supporting lemmas were proved mechanically in the GVE. Of the 89 theorems, 16 were recognized to be true in the VC generator, and the remaining 73 were proved using the interactive theorem prover.

The general MFM proof rests on a basis of two assumed lemmas, is_source_ordered and is_console_ordered (see Appendix A, Scope GVE_Extension). These two lemmas state that the timestamps on histories of inputs from the source and console are correctly ordered. The ordering of timestamps is included in the Gypsy language definition but is not known in the GVE.

The following sections give the flavor and discussion of the general MFM proof. Details of the entire proof are available in [Siebert 84a], which contains logs of the mechanically constructed proofs.

4.1 Top Level Proof

The top level of the general MFM is Procedure Modulator. Its proof requires that the exit specification be established for two cases and that the loop assertion be proved.

```
procedure modulator (var  source : a_source_buffer<input>;
                     var    sink : a_sink_buffer<output>;
                     var console : a_console_buffer<input>;
                     var display : a_display_buffer<output>;
                     var     log : a_log_buffer<output>;
                           table : a_modulator_constant)
        unless (incorrect_table) =
begin
  exit case (
    is normal:
      modulated_flow (xinfrom(source,myid), xinfrom(console,myid),
                      outto(sink,myid), outto(display,myid),
                      outto(log,myid), initial_state(table), table);
    is incorrect_table:
      no_modulated_flow (infrom(source,myid), infrom(console,myid),
                         outto(sink,myid), outto(display,myid),
                         outto(log,myid), table));
  var state: a_modulator_state;
  initialize (state, display, log, table) unless (incorrect_table);
  loop
    assert   modulation (xinfrom(source,myid), xinfrom(console,myid),
                         outto(sink,myid), initial_display(table),
                         outto(display,myid), initial_log(table),
                         outto(log,myid), initial_state(table), state,
                         table)
           & no_pending_modulation (state) & correct_table(table);
    if stop_modulation (state) then leave
    else modulate_message (source, sink, console, display, log, state,
                           table, max_time_stamp (xinfrom(source,myid),
                                                  xinfrom(console,myid)))

    end;
  end;
end;
```

4.1.1 Incorrect_Table Exit

When the modulator constant parameter "table" is incorrect, the problem is detected in Procedure Initialize, which causes Modulator to exit signalling condition incorrect_table. In this case, the exit specification No_Modulated_Flow follows immediately from the definition of that function, the exit specification of Initialize for the incorrect_table case, and the fact that there has been no source or console input and no sink output.

4.1.2 Normal Exit

The other exit case to be established is the normal one. That the property Modulated_Flow holds on exit follows immediately from the loop assertion Modulation, given definitions of both functions.

4.1.3 Loop Assertion

The loop assertion is proved by induction on passes through the loop. The initial truth of the assertion is established by expanding definitions of the specification functions, given the normal exit specification of Procedure Initialize and the fact that there has been no source or console input and no sink output.

The induction step is a proof that the loop assertion is true after a call to Modulate_Message on the assumption that the assertion was true before the call. The entry specification of the assumption that the assertion was true before the call. That No-Modulate_Message follows immediately from the previous loop assertion. That No-Pending_Modulation is true after the call is given in the Modulate_Message exit specification. Correct_Table remains true because "table" is a constant.

The remaining property to be established for the induction step is Modulation, which is proved by recourse to the lemma Extend_Modulation.

```
    lemma extend_modulation (  sou1,sou2 : a_timed_source_seq;
                              cons1,cons2 : a_timed_console_seq;
                              sink1,sink2 : a_sink_seq;
                                 d0,d1,d2 : a_display_seq;
                                 l0,l1,l2 : a_log_seq;
                              st0,st1,st2 : a_modulator_state;
                                        t : a_modulator_constant) =
         [  modulation (sou1, cons1, sink1, d0, d1, l0, l1, st0, st1, t)
          & modulation (sou2, cons2, sink2, null(a_display_seq), d2,
                        null(a_log_seq), l2, st1, st2, t)
          & no_pending_modulation (st1)
          & input_times (sou2, cons2, max_time_stamp (sou1, cons1))
          & source_ordered (sou2)
          & console_ordered (cons2)]
      ->
         modulation (sou1 @ sou2, cons1 @ cons2, sink1 @ sink2, d0, d1 @ d2,
                     l0, l1 @ l2, st0, st2, t);
```

The first and third hypotheses of the lemma are given by the previous loop assertion, the second by the Modulate_Message exit specification. The Input_Times hypothesis is given by the assumed Modulate_Message exit specification, which introduces into the proof a statement that earlier events happen before later events (not generally known in the GVE). It is supposed that the source and console buffers have access to a common clock, which is used for assigning timestamps to Modulator inputs. The remaining two hypotheses give Gypsy-defined properties of buffer histories. See [Siebert 84a] for proof of Lemma Extend_Modulation.

4.2 Discussion of Proof

Proving that an exit specification holds when a procedure exits is meaningful only if it actually does (or can) exit. The GVE cannot yet do proofs of exit nor is it possible to specify in Gypsy that a procedure does (or can) exit.

Hence, further consideration needs to be given to the behavior of Procedure Modulator. Modulator will not exit if Stop_Modulation never becomes true or if one of its called routines fails to terminate.

While Stop_Modulation is not true, supposing that all the routines called by Modulator terminate normally, the loop code of Modulator is executed repeatedly. At the beginning of each pass through the loop, Modulated_Flow holds since it follows from the loop assertion Modulation.

Procedure Initialize and Function Stop_Modulation are pending routines and the question of their exit should be considered when they are defined for each member of the general MFM family.

However, at this level of the design, it can be seen that, if Initialize fails to exit, security will never be violated by sending unauthorized material to the sink because Initialize has access neither to the sink nor to the source. If Stop_Modulation fails to exit, the Modulated_Flow property will hold because it follows from the loop assertion Modulation. (Stop_Modulation, being a function, cannot change any of the Modulator variables.)

When Procedure Modulate_Message is called, Modulated_Flow holds for Modulator behavior prior to the call (following from the loop assertion Modulation). That is, only specified output has been sent to the sink, log, and display so far. Function Max_Time_Stamp, a parameter in the call to Modulate_Message, is a specification device only, a ghost "variable", so that the question of its termination does not arise. (In any case, Max_Time_Stamp, being a function, can do neither input nor output.) The termination of Modulate_Message depends on characteristics of routines that are not part of the general MFM design, routines that are to be supplied for each member of the general MFM family (details of automatic filters and transforms, for example). Thus, it cannot be established that Modulate_Message will exit. However, the general MFM design does establish that security will never be violated by sending unauthorized material to the sink. (Unauthorized material is any sink output other than that described in Modulated_Flow.) Following the text of the general MFM design (see Appendix A) from Procedure Modulate_Message down, one can see that nothing at all will have been sent to the sink by the current invocation of Modulate_Message except when the sink handler, Procedure Send_to_Sink, fails to terminate. Non-termination of Send_to_Sink gives a situation in which sink output is a head of the expected (specified) output for the current message. Thus, either nothing or a head of the correct (specified) output will have been sent to the sink if Modulate_Message does not terminate.

5. Running Prototype

One member of the general MFM family has been completely implemented to provide a running example. The prototype runs on a DEC-20 under TOPS-20 at Utexas-20 on the ARPANET. The MFM source, sink, and log are all simulated by text files. Instructions for running the program are given in [Siebert 84b].

5.1 Message Formats

The MFM prototype uses the input message format of [Good 82]. A well-formed input message is a sequence of no more than 7200 ASCII characters of the form

```
ZCZC . . . NNNN
```

where the ... part of the message does not contain an "NNNN" sequence. The "ZCZC" and "NNNN" are required to be in upper case, and they are included in the 7200 character maximum. Any characters that precede the first "ZCZC" or that appear between an "NNNN" and the next "ZCZC" are considered to be noise, and they are dropped. An ill-formed message is one that contains more than 7200 characters. If such a message is received, it is modulated in 7200 character segments.

An output message, sent to the sink, is a sequence of ASCII characters, the transformed message followed by the sequence <carriage_return, carriage_return, line_feed>.

5.2 Automatic Filters

The running prototype contains five automatic filters. Two of the filters are based on the pattern matcher described in [Good 82]. These are called "OSIS filter" and "Thesis filter". The names are derived from the origin of messages that have been used to test the program. Both of these filters are tables of patterns. If an input message contains a text string that matches any pattern in the selected filter, the filter rejects the message; otherwise, the filter passes the message. A third filter, called "Format filter", rejects the segments of ill-formed messages and passes all well-formed messages. A fourth filter rejects all messages. A fifth filter passes all messages.

5.3 Automatic Transforms

The prototype contains only two transforms. The first is the identity transform. The second is a simple encryption.

5.4 Console Interaction

Commands from the console are used by the MFM operator to control modulator operation. The commands that are available in the prototype are described in the following paragraphs.

5.4.1 Control of Operating Mode

COMMAND	PURPOSE
Mode := <setting>	Sets the mode of the modulator for automatic or manual modulation. Choices for <setting> are: Auto, Manual.
No quit	Counteracts the effect of the "Quit" command.
Quit (no restart)	Terminates modulator operation as soon as modulation of the current message is completed. The action of this command can be prevented if the "No quit" command is entered before processing of the current message is completed; otherwise, the termination is permanent.

5.4.2 Selection of Automatic Filter and Transform

COMMAND	PURPOSE
Filter for auto modulation := <filter>	Sets the filter to be used for automatic modulation. Choices for <filter> are: Format check, OSIS patterns, Pass all, Reject all, Thesis patterns. See Section 5.2.
Transform for auto modulation := <setting>	Sets the transform to be used for automatic modulation. Choices for <setting> are: Encrypt, Identity.

5.4.3 Manual Modulation Commands

COMMAND	PURPOSE
Auto modulate one msg	Used in manual mode to apply the currently selected automatic filter and transform to the pending input message. If none is pending, a notice is displayed for the MFM operator.
Out-msg := <msg>	Used to construct the manual transform of a message when the modulator is running in manual mode. Choices for <msg> are: In-msg (the identity transform), Transform <how> (used to apply one of the predefined transforms). This command could be extended to include a message editor as one of the choices for <msg>. The only predefined non-identity transform in the

	prototype is encrypt, so <how> is limited to: Encrypt In-msg or Encrypt Out-msg.
Pass out-msg to sink	Passes the current transformed message to the sink. The decision is final; it cannot be revoked. If there is no transformed message, the identity transform is assumed. If there is no message, an error note is displayed. This is one of the manual filter commands.
Reject in-msg	Rejects and discards the current input message if there is one (if not, an error note is displayed). The message is not sent to the sink. The decision is final; it cannot be revoked. This is one of the manual filter commands.

5.4.4 Display Control Commands

COMMAND -------	PURPOSE -------
Blank the screen	Clears the console screen and redisplays the information that is shown continuously at the top of the screen.
Console type := <setting>	Tells the modulator what type of terminal is being used for the operator console. The choices of <setting> are Teleray 1061, Datamedia Elite 2500 and Other.
Display of <what> := <flag>	Controls the amount of display to the MFM operator. Choices for <what> are: Filter result, In-msg, Out-msg. Choices for <flag> are: Complete, Bell, None. IN-MSG DISPLAY. Setting The In-msg display flag to COMPLETE causes each message from the source to be displayed as soon as it is received. Control characters are quoted with a preceding "!" so that they do not invoke terminal control functions (such as cursor control). If the In-msg display flag is BELL, the console bell is rung to notify the operator of each message received. If the In-msg flag is NONE, nothing is displayed on input from the source. OUT-MSG DISPLAY. An Out-msg display flag of COMPLETE causes each message sent to the sink to be displayed, with control characters quoted as for In-msg. If the Out-msg display flag is BELL, the console bell is rung each time a message is sent to the sink. If the Out-msg display flag is NONE, nothing is displayed for output to the sink. FILTER-RESULT DISPLAY. If the filter display flag is COMPLETE, the result of filtering each message is displayed with an indication of which filter was used,

	except when a message is manually passed or rejected. For the pattern matching filters, each matched pattern and the matching message text are also displayed. Both control characters and blanks in the message text are quoted with a preceding "!". If the filter display flag is BELL, the console bell is rung for each rejected message except manually rejected ones. If the filter display flag is NONE, no report of the message filtering operation is displayed.
Show <what>	Causes the selected information to be displayed to the operator. Choices for <what> are: In-msg, Out-msg, Auto filter, Filter - <which>. Auto filter is the currently selected filter for automatic modulation. Choices for <which> are the available automatic filters: Format check, OSIS patterns, Pass all, Reject all, Thesis patterns.
Window for <what> := <size>	Sets the size of display windows on the console screen. Choices for <what> are: Filter result, In-msg, Out-msg. Choices for <size> are: Full screen, Half screen.

5.4.5 Audit Control Commands

COMMAND	PURPOSE
Log of <what> := <flag>	Controls the amount of information recorded on the audit trail for each message. Choices for <what> and <flag> are the same as for display control, and their meanings differ in only two ways. First, results of manual modulation are also logged. Second, a COMPLETE log of filter-result for the pattern matching filters will also include the pattern table if it has not been logged previously or if it has been changed since it was logged.

5.4.6 Commands for Editing Pattern Tables

COMMAND	PURPOSE
Eliminate pattern from <pattern-list>	Allows a pattern to be removed from the table used by one of the pattern matching filters. Choices for <pattern-list> are: OSIS filter, Thesis filter. See Section 5.2.
Insert pattern into <pattern-list>	Allows a pattern to be added to the table used by one of the pattern matching filters. Choices for <pattern-list> are: OSIS filter, Thesis filter. See Section 5.2.

5.4.7 Help Commands

COMMAND	PURPOSE
Help	Gives a brief description of how to issue commands.
?	Gives the list of modulator commands.

5.5 Audit Trail

There are several options for audit trail content. These are indicated in Section 5.4. Message reception and modulation may be audited. Command audit is limited to reporting the results of manually modulating messages.

Appendix A
Gypsy Text

In the Gypsy listing of the general MFM, scopes are given in alphabetical order except that the top level scope, Flow_Modulator, is listed first. Within each scope, units are listed in the following order: procedures, functions, lemmas, constants, types. The units of each kind are listed alphabetically, except that the top level procedure, Modulator, is given first in Scope Flow_Modulator.

A.1 Scope Flow_Modulator

```
scope flow_modulator =
begin


procedure modulator (var  source : a_source_buffer<input>;
                     var    sink : a_sink_buffer<output>;
                     var console : a_console_buffer<input>;
                     var display : a_display_buffer<output>;
                     var     log : a_log_buffer<output>;
                           table : a_modulator_constant)
        unless (incorrect_table) =
begin
  exit case (
    is normal:
      modulated_flow (xinfrom(source,myid), xinfrom(console,myid),
                      outto(sink,myid), outto(display,myid),
                      outto(log,myid), initial_state(table), table);
    is incorrect_table:
      no_modulated_flow (infrom(source,myid), infrom(console,myid),
                         outto(sink,myid), outto(display,myid),
                         outto(log,myid), table));
  var state: a_modulator_state;
  initialize (state, display, log, table) unless (incorrect_table);
  loop
    assert   modulation (xinfrom(source,myid), xinfrom(console,myid),
                         outto(sink,myid), initial_display(table),
                         outto(display,myid), initial_log(table),
                         outto(log,myid), initial_state(table), state, table)
           & no_pending_modulation (state) & correct_table(table);
    if stop_modulation (state) then leave
    else modulate_message (source, sink, console, display, log, state, table,
                           max_time_stamp (xinfrom(source,myid),
                                           xinfrom(console,myid)))

    end;
  end;
end;



procedure auto_modulate (var    sink : a_sink_buffer<output>;
                         var console : a_console_buffer<input>;
                         var display : a_display_buffer<output>;
                         var     log : a_log_buffer<output>;
                         var   state : a_modulator_state;
                               table : a_modulator_constant;
                                time : integer) =
begin
  entry have_in_msg(state) & correct_table(table);
        {and TIME is earlier than the present}
  exit (prove auto_modulated (outto(sink,myid), xinfrom(console,myid),
                              outto(display,myid), outto(log,myid),
                              state', state, table);
        assume input_times (null(a_timed_source_seq), xinfrom(console,myid),
                            time));
  filter_message (display, log, state, table);
  if msg_disposition (state) = passed then
    transform_message (display, log, state, table);
    send_to_sink (sink, console, display, log, state, table, time)
  else remove_message (state)
  end;
end;
```

```
procedure do_auto_filter_1 (var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_filtered_1 (outto(display,myid), outto(log,myid),
                        state', state, table);
  pending;
end;


procedure do_auto_filter_2 (var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_filtered_2 (outto(display,myid), outto(log,myid),
                        state', state, table);
  pending;
end;


procedure do_auto_filter_3 (var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_filtered_3 (outto(display,myid), outto(log,myid),
                        state', state, table);
  pending;
end;


procedure do_auto_filter_4 (var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_filtered_4 (outto(display,myid), outto(log,myid),
                        state', state, table);
  pending;
end;
```

```
procedure do_auto_filter_5 (var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_filtered_5 (outto(display,myid), outto(log,myid),
                        state', state, table);
  pending;
end;


procedure do_auto_transform_1 (var display : a_display_buffer<output>;
                               var     log : a_log_buffer<output>;
                               var   state : a_modulator_state;
                                     table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_transformed_1 (outto(display,myid), outto(log,myid),
                           state', state, table);
  pending;
end;


procedure do_auto_transform_2 (var display : a_display_buffer<output>;
                               var     log : a_log_buffer<output>;
                               var   state : a_modulator_state;
                                     table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_transformed_2 (outto(display,myid), outto(log,myid),
                           state', state, table);
  pending;
end;


procedure do_auto_transform_3 (var display : a_display_buffer<output>;
                               var     log : a_log_buffer<output>;
                               var   state : a_modulator_state;
                                     table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_transformed_3 (outto(display,myid), outto(log,myid),
                           state', state, table);
  pending;
end;


procedure do_auto_transform_4 (var display : a_display_buffer<output>;
                               var     log : a_log_buffer<output>;
                               var   state : a_modulator_state;
                                     table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_transformed_4 (outto(display,myid), outto(log,myid),
                           state', state, table);
  pending;
end;
```

```
procedure do_auto_transform_5 (var display : a_display_buffer<output>;
                               var     log : a_log_buffer<output>;
                               var   state : a_modulator_state;
                                     table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_transformed_5 (outto(display,myid), outto(log,myid),
                           state', state, table);
  pending;
end;


procedure filter_message (var display : a_display_buffer<output>;
                          var     log : a_log_buffer<output>;
                          var   state : a_modulator_state;
                                table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_filtered (outto(display,myid), outto(log,myid),
                      state', state, table);
  case filter_selector(state)
    is 1:  do_auto_filter_1 (display, log, state, table);
    is 2:  do_auto_filter_2 (display, log, state, table);
    is 3:  do_auto_filter_3 (display, log, state, table);
    is 4:  do_auto_filter_4 (display, log, state, table);
    else:  do_auto_filter_5 (display, log, state, table);
  end;
end;


procedure initialize (var   state : a_modulator_state;
                      var display : a_display_buffer<output>;
                      var     log : a_log_buffer<output>;
                            table : a_modulator_constant)
        unless (incorrect_table) =
begin
  exit case (
    is normal:   state = initial_state (table)
               & no_pending_modulation (state)
               & outto(display,myid) = initial_display (table)
               & outto(log,myid) = initial_log (table)
               & correct_table (table);
    is incorrect_table:   outto(display,myid) = table_error_display(table)
                        & outto(log,myid) = table_error_log(table)
                        & not correct_table(table));
  pending;
end;
```

```
procedure manual_modulate (var    sink : a_sink_buffer<output>;
                           var console : a_console_buffer<input>;
                           var display : a_display_buffer<output>;
                           var     log : a_log_buffer<output>;
                           var   state : a_modulator_state;
                                 table : a_modulator_constant;
                                  time : integer) =
begin
  entry have_in_msg(state) & correct_table(table);
        {and TIME is earlier than the present}
  exit
    (prove manually_modulated (outto(sink,myid), xinfrom(console,myid),
                               outto(display,myid), outto(log,myid),
                               state', state, table);
     assume input_times (null(a_timed_source_seq), xinfrom(console,myid),
                         time));
  initialize_manual_modulation (state, table);
  loop
    assert   manually_modulating (outto(sink,myid), xinfrom(console,myid),
                                  outto(display,myid), outto(log,myid),
                                  state', state, table)
           & correct_table(table);
    if msg_disposition(state) ne undecided then leave
    else follow_command (console, display, log, state, table,
                         max_time_stamp (null_source, xinfrom(console,myid)))
    end;
  end;
  if msg_disposition(state) = passed then
    send_to_sink (sink, console, display, log, state, table,
                  max_time_stamp (null_source, xinfrom(console,myid)))
  else remove_message (state)
  end;
end;
```

```
procedure modulate_message (var  source : a_source_buffer<input>;
                            var    sink : a_sink_buffer<output>;
                            var console : a_console_buffer<input>;
                            var display : a_display_buffer<output>;
                            var     log : a_log_buffer<output>;
                            var   state : a_modulator_state;
                                  table : a_modulator_constant;
                                   time : integer) =
begin
  entry no_pending_modulation (state) & correct_table(table);
        {and TIME is earlier than the present}
  exit
    (prove   modulation (xinfrom(source,myid), xinfrom(console,myid),
                         outto(sink,myid), null(a_display_seq),
                         outto(display,myid), null(a_log_seq),
                         outto(log,myid), state', state, table)
           & no_pending_modulation (state);
     assume input_times (xinfrom(source,myid), xinfrom(console,myid), time));

  cond stop_mod;
  receive_message (source, console, display, log, state, table)
        unless (stop_mod);
  if in_auto_mode (state) then
    auto_modulate (sink, console, display, log, state, table,
                  max_time_stamp(xinfrom(source,myid),xinfrom(console,myid)))
  else manual_modulate
         (sink, console, display, log, state, table,
          max_time_stamp(xinfrom(source,myid),xinfrom(console,myid)))
  end;
when is stop_mod:
end;


procedure transform_message (var display : a_display_buffer<output>;
                             var     log : a_log_buffer<output>;
                             var   state : a_modulator_state;
                                   table : a_modulator_constant) =
begin
  entry have_in_msg(state) & correct_table(table);
  exit auto_transformed (outto(display,myid), outto(log,myid),
                         state', state, table);
  case transform_selector(state)
    is 1:  do_auto_transform_1 (display, log, state, table);
    is 2:  do_auto_transform_2 (display, log, state, table);
    is 3:  do_auto_transform_3 (display, log, state, table);
    is 4:  do_auto_transform_4 (display, log, state, table);
    else:  do_auto_transform_5 (display, log, state, table);
  end;
end;
```

```
function auto_filtered (        d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_filter_display (st1,t)
              & l = auto_filter_log (st1,t)
              & st2 = auto_filter_transition (st1,t)
              & msg_disposition(st2) = auto_filter (in_msg(st1), st1, t)
              & msg_integrity (st2,st1)]);
end;


function auto_filtered_1 (      d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_filter_display_1 (st1,t)
              & l = auto_filter_log_1 (st1,t)
              & st2 = auto_filter_transition_1 (st1,t)
              & msg_disposition(st2) = auto_filter_1 (in_msg(st1), st1, t)
              & msg_integrity (st2,st1)]);
end;


function auto_filtered_2 (      d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_filter_display_2 (st1,t)
              & l = auto_filter_log_2 (st1,t)
              & st2 = auto_filter_transition_2 (st1,t)
              & msg_disposition(st2) = auto_filter_2 (in_msg(st1), st1, t)
              & msg_integrity (st2,st1)]);
end;


function auto_filtered_3 (      d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_filter_display_3 (st1,t)
              & l = auto_filter_log_3 (st1,t)
              & st2 = auto_filter_transition_3 (st1,t)
              & msg_disposition(st2) = auto_filter_3 (in_msg(st1), st1, t)
              & msg_integrity (st2,st1)]);
end;
```

```
function auto_filtered_4 (      d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_filter_display_4 (st1,t)
              & l = auto_filter_log_4 (st1,t)
              & st2 = auto_filter_transition_4 (st1,t)
              & msg_disposition(st2) = auto_filter_4 (in_msg(st1), st1, t)
              & msg_integrity (st2,st1)]);
end;


function auto_filtered_5 (      d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_filter_display_5 (st1,t)
              & l = auto_filter_log_5 (st1,t)
              & st2 = auto_filter_transition_5 (st1,t)
              & msg_disposition(st2) = auto_filter_5 (in_msg(st1), st1, t)
              & msg_integrity (st2,st1)]);
end;


function auto_modulated (      s : a_sink_seq;
                              xc : a_timed_console_seq;
                               d : a_display_seq;
                               l : a_log_seq;
                         st1,st2 : a_modulator_state;
                               t : a_modulator_constant) : boolean =
begin
  exit (assume
    result =
      [  s = auto_modulation (in_msg(st1), st1, t)
       & some st: a_modulator_state, some h: an_input_event_seq,
           st = auto_transition (st1, t)
         & h = event_pack (merged_inputs(null(a_timed_source_seq),xc), st, t)
         & d = auto_display (st1, t) @ all_displayed (h, st, t)
         & l = auto_log (st1, t) @ all_logged (h, st, t)
         & st2 = flow_state (h, st, t)
         & no_pending_modulation(st2) & no_partial_cmd(h)]);
end;


function auto_transformed (      d : a_display_seq;
                                 l : a_log_seq;
                           st1,st2 : a_modulator_state;
                                 t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_transform_display (st1,t)
              & l = auto_transform_log (st1,t)
              & st2 = auto_transform_transition (st1,t)
              & out_msg(st2) = auto_transform (in_msg(st1), st1, t)]);
end;
```

```
function auto_transformed_1 (      d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_transform_display_1 (st1,t)
              & l = auto_transform_log_1 (st1,t)
              & st2 = auto_transform_transition_1 (st1,t)
              & out_msg(st2) = auto_transform_1 (in_msg(st1), st1, t)]);
end;


function auto_transformed_2 (      d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_transform_display_2 (st1,t)
              & l = auto_transform_log_2 (st1,t)
              & st2 = auto_transform_transition_2 (st1,t)
              & out_msg(st2) = auto_transform_2 (in_msg(st1), st1, t)]);
end;


function auto_transformed_3 (      d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_transform_display_3 (st1,t)
              & l = auto_transform_log_3 (st1,t)
              & st2 = auto_transform_transition_3 (st1,t)
              & out_msg(st2) = auto_transform_3 (in_msg(st1), st1, t)]);
end;


function auto_transformed_4 (      d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_transform_display_4 (st1,t)
              & l = auto_transform_log_4 (st1,t)
              & st2 = auto_transform_transition_4 (st1,t)
              & out_msg(st2) = auto_transform_4 (in_msg(st1), st1, t)]);
end;
```

```
function auto_transformed_5 (      d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  d = auto_transform_display_5 (st1,t)
              & l = auto_transform_log_5 (st1,t)
              & st2 = auto_transform_transition_5 (st1,t)
              & out_msg(st2) = auto_transform_5 (in_msg(st1), st1, t)]);
end;


function manually_modulated (      s : a_sink_seq;
                                  xc : a_timed_console_seq;
                                   d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result =
      some st: a_modulator_state, some h: an_input_event_seq,
          st = manual_mod_init(st1,t)
        & h = event_pack (merged_inputs(null(a_timed_source_seq),xc), st, t)
        & s = manual_modulation (in_msg(st1), st, mod_cmd_head(h,st,t), t)
        & d = all_displayed (h, st, t)
        & l = all_logged (h, st, t)
        & st2 = flow_state (h, st, t)
        & no_pending_modulation(st2) & no_partial_cmd(h));
end;


function manually_modulating (  sink : a_sink_seq;
                                  xc : a_timed_console_seq;
                                   d : a_display_seq;
                                   l : a_log_seq;
                             st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) : boolean =
begin
  exit (assume
    result =
      some st: a_modulator_state, some h: an_input_event_seq,
          st = manual_mod_init(st1,t)
        & h = event_pack (merged_inputs(null_source,xc), st, t)
        & sink = null(a_sink_seq)
        & manual_mod_display (d,st2) = all_displayed (h,st,t)
        & manual_mod_log (l, st2) = all_logged (h,st,t)
        & manual_mod_transition (st2) = flow_state (h,st,t)
        & manual_mod_record (st1,st,st2,h,t)
        & no_partial_cmd (h));
end;
```

```
function manual_mod_record (st1,st2,st3 : a_modulator_state;
                                      h : an_input_event_seq;
                                      t : a_modulator_constant) : boolean =
begin
  exit (assume
    result =
      [  msg_disposition(st3) = manual_filter (in_msg(st1), h, st2, t)
       & out_msg(st3) = manual_transform (in_msg(st1),
                                          initial_transform(in_msg(st1),t),
                                          h, st2, t)
       & msg_disposition(st3) in [undecided, passed, rejected]
       & need_all_cmds (h,st2,t) & have_in_msg(st3)
       & msg_integrity (st3,st1) & msg_integrity(st2,st1)]);
end;


function modulated_flow (      xs : a_timed_source_seq;
                               xc : a_timed_console_seq;
                             sink : a_sink_seq;
                          display : a_display_seq;
                              log : a_log_seq;
                              st0 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume result = some h: an_input_event_seq,
                          h = event_pack (merged_inputs(xs,xc), st0, t)
                        & sink = all_modulated (h, st0, t)
                        & log = initial_log (t) @ all_logged (h, st0, t)
                        & display = initial_display (t)
                                    @ all_displayed (h, st0, t));
end;


function modulation (     xs : a_timed_source_seq;
                          xc : a_timed_console_seq;
                        sink : a_sink_seq;
                        d0,d : a_display_seq;
                        l0,l : a_log_seq;
                     st0,st1 : a_modulator_state;
                           t : a_modulator_constant) : boolean =
begin
  exit (assume result = some h: an_input_event_seq,
                          h = event_pack (merged_inputs(xs,xc), st0, t)
                        & sink = all_modulated (h, st0, t)
                        & l = l0 @ all_logged (h, st0, t)
                        & d = d0 @ all_displayed (h, st0, t)
                        & st1 = flow_state (h, st0, t)
                        & no_partial_msg(h) & no_partial_cmd(h));
end;
```

```
function no_modulated_flow ( source : a_source_seq;
                            console : a_console_seq;
                               sink : a_sink_seq;
                            display : a_display_seq;
                                log : a_log_seq;
                              table : a_modulator_constant) : boolean =
begin
  exit (assume result = [  not correct_table (table)
                         & source = null(a_source_seq)
                         & console = null(a_console_seq)
                         & sink = null(a_sink_seq)
                         & display = table_error_display (table)
                         & log = table_error_log (table)]);
end;


lemma extend_modulation (  sou1,sou2 : a_timed_source_seq;
                         cons1,cons2 : a_timed_console_seq;
                         sink1,sink2 : a_sink_seq;
                            d0,d1,d2 : a_display_seq;
                            l0,l1,l2 : a_log_seq;
                         st0,st1,st2 : a_modulator_state;
                                   t : a_modulator_constant) =
     [  modulation (sou1, cons1, sink1, d0, d1, l0, l1, st0, st1, t)
      & modulation (sou2, cons2, sink2, null(a_display_seq), d2,
                    null(a_log_seq), l2, st1, st2, t)
      & no_pending_modulation (st1)
      & input_times (sou2, cons2, max_time_stamp (sou1, cons1))
      & source_ordered (sou2)
      & console_ordered (cons2)]
  ->
    modulation (sou1 @ sou2, cons1 @ cons2, sink1 @ sink2, d0, d1 @ d2,
                l0, l1 @ l2, st0, st2, t);
```

```
lemma msg_modulation (  h1,h2 : an_input_event_seq;
                           xs : a_timed_source_seq;
                      xc1,xc2 : a_timed_console_seq;
                         sink : a_sink_seq;
                            d : a_display_seq;
                            l : a_log_seq;
                      st1,st2 : a_modulator_state;
                            t : a_modulator_constant) =
     [  h1 = event_pack (merged_inputs(xs,xc1), st1, t)
      & h1 ne null(an_input_event_seq) & no_partial_cmd(h1)
      & no_message(nonlast(h1)) & is_message(last(h1))
      & h2 = event_pack (merged_inputs(null(a_timed_source_seq),xc2),
                         flow_state(h1,st1,t), t)
      & no_partial_cmd(h2) & console_ordered(xc2)
      & input_times (null(a_timed_source_seq), xc2, max_time_stamp(xs,xc1))
      & d =   all_displayed (nonlast(h1), st1, t)
            @ display_it (last(h1), flow_state(nonlast(h1),st1,t), t)
            @ all_displayed (h2, flow_state(h1,st1,t), t)
      & l =   all_logged (nonlast(h1), st1, t)
            @ log_it (last(h1), flow_state(nonlast(h1),st1,t), t)
            @ all_logged (h2, flow_state(h1,st1,t), t)
      & sink =
          modulate_it
            (last(h1), flow_state(nonlast(h1),st1,t), t,
             modulation_cmds(last(h1):>h2, flow_state(nonlast(h1),st1,t), t))
      & st2 = flow_state (h2, flow_state(h1,st1,t), t)]
  ->
     modulation (xs, xc1@xc2, sink, null(a_display_seq), d,
                 null(a_log_seq), l, st1, st2, t);


lemma null_modulation ( xsource : a_timed_source_seq;
                       xconsole : a_timed_console_seq;
                           sink : a_sink_seq;
                              t : a_modulator_constant) =
     [  xsource = null(a_timed_source_seq)
      & xconsole = null(a_timed_console_seq)
      & sink = null(a_sink_seq)]
  ->
     modulation (xsource, xconsole, sink, initial_display(t),
                 initial_display(t), initial_log(t), initial_log(t),
                 initial_state(t), initial_state(t), t);


const console_buffer_size : integer = pending;

const display_buffer_size : integer = pending;

const log_buffer_size : integer = pending;

const sink_buffer_size : integer = pending;

const source_buffer_size : integer = pending;


type a_console_buffer = buffer (console_buffer_size) of a_console_object;

type a_console_object = pending;
type a_console_seq = sequence of a_console_object;

type a_display_buffer = buffer (display_buffer_size) of a_display_object;

type a_display_object = pending;

type a_display_seq = sequence of a_display_object;

type a_log_buffer = buffer (log_buffer_size) of a_log_object;

type a_log_object = pending;

type a_log_seq = sequence of a_log_object;

type a_modulator_constant = pending;

type a_sink_buffer = buffer (sink_buffer_size) of a_sink_object;

type a_sink_object = pending;

type a_sink_seq = sequence of a_sink_object;

type a_source_buffer = buffer (source_buffer_size) of a_source_object;

type a_source_object = pending;

type a_source_seq = sequence of a_source_object;

type a_timed_console_object = record (message : a_console_object;
                                         time : integer);

type a_timed_console_seq = sequence of a_timed_console_object;

type a_timed_source_object = record (message : a_source_object;
                                        time : integer);

type a_timed_source_seq = sequence of a_timed_source_object;


name followed_command, follow_command, manual_mod_update, msg_disposition_ok
     from command_interpreter;

name all_displayed, auto_display, auto_filter_display, auto_filter_display_1,
     auto_filter_display_2, auto_filter_display_3, auto_filter_display_4,
     auto_filter_display_5, auto_transform_display, auto_transform_display_1,
     auto_transform_display_2, auto_transform_display_3,
     auto_transform_display_4, auto_transform_display_5, command_display,
     display_append, display_it, initial_display, manual_mod_display,
     message_display, sink_display, source_display, table_error_display
     from display_specifications;

name console_ordered, is_console_ordered, is_source_ordered, not_set_elmt,
     source_ordered
     from GVE_extension;
```

```
name an_input_event_seq, a_message, command, command_event_merge,
     command_event_pack, event_merge_append, event_pack,
     extend_no_partial_cmd, extend_no_partial_msg, input_times, is_command,
     is_message, is_msg_event, is_partial_message, max_time_stamp,
     merged_inputs, message, no_message, no_partial_cmd, no_partial_msg,
     null_source, partial_msg
     from input_specifications;

name all_logged, auto_filter_log, auto_filter_log_1, auto_filter_log_2,
     auto_filter_log_3, auto_filter_log_4, auto_filter_log_5, auto_log,
     auto_transform_log, auto_transform_log_1, auto_transform_log_2,
     auto_transform_log_3, auto_transform_log_4, auto_transform_log_5,
     command_log, initial_log, log_append, log_it, manual_mod_log,
     message_log, sink_log, source_log, table_error_log
     from log_specifications;

name all_modulated, auto_filter, auto_filter_1, auto_filter_2, auto_filter_3,
     auto_filter_4, auto_filter_5, auto_modulation, auto_transform,
     auto_transform_1, auto_transform_2, auto_transform_3, auto_transform_4,
     auto_transform_5, a_msg_disposition, contract_need_all_cmds,
     correct_table, did_manual_modulation, extend_need_all_cmds,
     initial_transform, manual_filter, manual_filter_eq,
     manual_filter_result, manual_modulation, manual_transform,
     manual_transform_eq, modulate_it, modulation_cmds, mod_cmd_head,
     msg_transform, need_all_cmds, need_mod_cmds, no_pending_modulation,
     null_sink, null_sink_head, sink_append
     from modulation_specifications;

name auto_filter_transition, auto_filter_transition_1,
     auto_filter_transition_2, auto_filter_transition_3,
     auto_filter_transition_4, auto_filter_transition_5,
     auto_transform_transition, auto_transform_transition_1,
     auto_transform_transition_2, auto_transform_transition_3,
     auto_transform_transition_4, auto_transform_transition_5,
     auto_transition, a_modulator_state, command_transition,
     extend_flow_state, filter_selector, flow_state, flow_transition,
     have_in_msg, initialize_manual_modulation, initial_state, in_auto_mode,
     in_msg, manual_mod_init, manual_mod_transition, message_transition,
     msg_disposition, msg_integrity, msg_removed, out_msg, remove_message,
     source_transition, stop_modulation, transform_selector
     from modulator_state;

name send_to_sink, sent_to_sink, sink_seq
     from sink_handler;

name received_msg, received_no_msg, receive_message
     from source_handler;

end; {scope flow_modulator}
```

A.2 Scope Command_Interpreter

```
scope command_interpreter =
begin


procedure follow_command (var console : a_console_buffer<input>;
                          var display : a_display_buffer<output>;
                          var     log : a_log_buffer<output>;
                          var   state : a_modulator_state;
                                table : a_modulator_constant;
                                 time : integer) =
begin
  entry correct_table(table);   {and TIME is earlier than the present}
  exit
    (prove followed_command (xinfrom(console,myid), outto(display,myid),
                             outto(log,myid), state', state, table);
     assume input_times (null_source, xinfrom(console,myid), time));
  pending;
end;



function followed_command (     xc : a_timed_console_seq;
                                 d : a_display_seq;
                                 l : a_log_seq;
                           st1,st2 : a_modulator_state;
                                 t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = some h: an_input_event_seq,
                 h = event_pack (merged_inputs(null_source,xc), st1, t)
               & manual_mod_display (d,st2) = all_displayed (h,st1,t)
               & manual_mod_log (l, st2) = all_logged (h,st1,t)
               & manual_mod_transition (st2) = flow_state (h,st1,t)
               & manual_mod_update (st2,st1,h,t)
               & msg_integrity (st2,st1)
               & no_partial_cmd (h));
end;



function manual_mod_update (st2,st1 : a_modulator_state;
                                  h : an_input_event_seq;
                                  t : a_modulator_constant) : boolean =
begin
  exit (assume
    result =
      (   [have_in_msg(st1) & msg_disposition(st1) = undecided]
       ->
         [  msg_disposition(st2) = manual_filter (in_msg(st1), h, st1, t)
          & out_msg(st2)
              = manual_transform (in_msg(st1), out_msg(st1), h, st1, t)
          & msg_disposition_ok(st2)
          & need_all_cmds (h, st1, t)]));
end;
```

```
function msg_disposition_ok (st : a_modulator_state) : boolean =
begin
  exit (assume result = if in_auto_mode(st) then
                          msg_disposition(st) in [passed, rejected]
                        else
                          msg_disposition(st) in [undecided,passed,rejected]
                        fi);
end;


name all_displayed, manual_mod_display
     from display_specifications;

name a_console_buffer, a_display_buffer, a_display_seq, a_log_buffer,
     a_log_seq, a_modulator_constant, a_timed_console_seq
     from flow_modulator;

name an_input_event_seq, event_pack, input_times, merged_inputs,
     no_partial_cmd, null_source
     from input_specifications;

name all_logged, manual_mod_log
     from log_specifications;

name a_msg_disposition, correct_table, manual_filter, manual_transform,
     need_all_cmds
     from modulation_specifications;

name a_modulator_state, flow_state, have_in_msg, in_auto_mode, in_msg,
     manual_mod_transition, msg_disposition, msg_integrity, out_msg
     from modulator_state;


end; {scope command_interpreter}
```

A.3 Scope Display_Specifications

```
scope display_specifications =
begin


function all_displayed ( h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_display_seq =
begin
  exit (assume
    result = if h = null(an_input_event_seq) then
                null(a_display_seq)
             else all_displayed (nonlast(h), st, t)
                    @ display_it (last(h), flow_state(nonlast(h),st,t), t)
             fi);
end;



function auto_display (st : a_modulator_state;
                        t : a_modulator_constant) : a_display_seq =
begin
  exit (assume
    result = if msg_disposition (auto_filter_transition(st,t)) = passed then
                 auto_filter_display (st,t)
               @ auto_transform_display (auto_filter_transition(st,t), t)
               @ sink_display (auto_transform_transition
                                  (auto_filter_transition(st,t),t))
             else auto_filter_display (st,t)
             fi);
end;



function auto_filter_display (st : a_modulator_state;
                               t : a_modulator_constant) : a_display_seq =
begin
  exit (assume result = if filter_selector(st) = 1 then
                          auto_filter_display_1 (st,t)
                        else if filter_selector(st) = 2 then
                          auto_filter_display_2 (st,t)
                        else if filter_selector(st) = 3 then
                          auto_filter_display_3 (st,t)
                        else if filter_selector(st) = 4 then
                          auto_filter_display_4 (st,t)
                        else auto_filter_display_5 (st,t)
                        fi fi fi fi);
end;



function auto_filter_display_1 (st : a_modulator_state;
                                 t : a_modulator_constant) : a_display_seq =
  pending;



function auto_filter_display_2 (st : a_modulator_state;
                                 t : a_modulator_constant) : a_display_seq =
  pending;
```

```
function auto_filter_display_3 (st : a_modulator_state;
                                t : a_modulator_constant) : a_display_seq =
  pending;


function auto_filter_display_4 (st : a_modulator_state;
                                t : a_modulator_constant) : a_display_seq =
  pending;


function auto_filter_display_5 (st : a_modulator_state;
                                t : a_modulator_constant) : a_display_seq =
  pending;


function auto_transform_display (st : a_modulator_state;
                                 t : a_modulator_constant) : a_display_seq =
begin
  exit (assume result = if transform_selector(st) = 1 then
                          auto_transform_display_1 (st,t)
                        else if transform_selector(st) = 2 then
                          auto_transform_display_2 (st,t)
                        else if transform_selector(st) = 3 then
                          auto_transform_display_3 (st,t)
                        else if transform_selector(st) = 4 then
                          auto_transform_display_4 (st,t)
                        else auto_transform_display_5 (st,t)
                        fi fi fi fi);
end;


function auto_transform_display_1 (st : a_modulator_state;
                                   t : a_modulator_constant) : a_display_seq=
  pending;


function auto_transform_display_2 (st : a_modulator_state;
                                   t : a_modulator_constant) : a_display_seq=
  pending;


function auto_transform_display_3 (st : a_modulator_state;
                                   t : a_modulator_constant) : a_display_seq=
  pending;


function auto_transform_display_4 (st : a_modulator_state;
                                   t : a_modulator_constant) : a_display_seq=
  pending;


function auto_transform_display_5 (st : a_modulator_state;
                                   t : a_modulator_constant) : a_display_seq=
  pending;
```

```
function command_display ( e : an_input_event;
                          st : a_modulator_state;
                           t : a_modulator_constant) : a_display_seq =
  pending;
```

```
function display_it ( e : an_input_event;
                     st : a_modulator_state;
                      t : a_modulator_constant) : a_display_seq =
begin
  exit (assume result = if is_msg_event(e) then
                          message_display (e, st, t)
                        else {is_cmd_event(e)}
                          command_display (e, st, t)
                        fi);
end;
```

```
function initial_display (table : a_modulator_constant) : a_display_seq =
  pending;
```

```
function manual_mod_display ( d : a_display_seq;
                             st : a_modulator_state) : a_display_seq =
begin
  exit (assume result = if have_in_msg(st) & msg_disposition(st) = passed then
                          d @ sink_display(st)
                        else d
                        fi);
end;
```

```
function message_display ( e : an_input_event;
                          st : a_modulator_state;
                           t : a_modulator_constant) : a_display_seq =
begin
  exit (assume result = if is_message(e)
                               & in_auto_mode(source_transition(e,st)) then
                          source_display(e,st)
                            @ auto_display (source_transition(e,st), t)
                        else source_display(e,st)
                        fi);
end;
```

```
function sink_display (st : a_modulator_state) : a_display_seq =
  pending;
```

```
function source_display ( e : an_input_event;
                         st : a_modulator_state) : a_display_seq =
  pending;
```

```
function table_error_display (table : a_modulator_constant) : a_display_seq =
  pending;


lemma display_append (h1,h2 : an_input_event_seq;
                         s0 : a_modulator_state;
                          t : a_modulator_constant) =
  all_displayed (h1@h2, s0, t)
    = all_displayed (h1,s0,t) @ all_displayed (h2, flow_state(h1,s0,t), t);


name a_display_seq, a_modulator_constant
     from flow_modulator;

name an_input_event, an_input_event_seq, is_cmd_event, is_message,
     is_msg_event
     from input_specifications;

name a_msg_disposition
     from modulation_specifications;

name auto_filter_transition, auto_transform_transition, a_modulator_state,
     extend_flow_state, filter_selector, flow_state, have_in_msg,
     in_auto_mode, msg_disposition, source_transition, transform_selector
     from modulator_state;


end; {scope display_specifications}
```

A.4 Scope GVE_Extension

```
scope GVE_extension =
begin


function console_ordered (c : a_timed_console_seq) : boolean =
begin
  exit (assume
    result = if c = null(a_timed_console_seq) then true
             else if nonlast(c) = null(a_timed_console_seq) then true
                  else   console_ordered (nonlast(c))
                       & timestamp(last(nonlast(c))) < timestamp(last(c))
             fi   fi);
end;


function source_ordered (s : a_timed_source_seq) : boolean =
begin
  exit (assume
    result = if s = null(a_timed_source_seq) then true
             else if nonlast(s) = null(a_timed_source_seq) then true
                  else   source_ordered (nonlast(s))
                       & timestamp(last(nonlast(s))) < timestamp(last(s))
             fi   fi);
end;


lemma first_eq (h1,h2 : an_input_event_seq) =
  h1 ne null(an_input_event_seq) -> (h1@h2)[1] = h1[1];


lemma is_console_ordered (  b : a_console_buffer;
                          aid : activationid) =
  console_ordered (xinfrom (b, aid));


lemma is_source_ordered (  b : a_source_buffer;
                         aid : activationid) =
  source_ordered (xinfrom (b, aid));


lemma nonfirst_eq (h1,h2 : an_input_event_seq) =
     h1 ne null(an_input_event_seq)
  ->
     (h1@h2)[2..size(h1) + size(h2)] = h1[2..size(h1)] @ h2;


lemma nonfirst_eq2 (h1,h2 : an_input_event_seq) =
     h1 ne null(an_input_event_seq)
  ->
     nonfirst (h1@h2) = nonfirst(h1) @ h2;
```

```
lemma nonlast_eq (h1,h2 : an_input_seq;
                      e : an_input) =
  h1 @ [seq:e] = h2 @ [seq:e] -> h1 = h2;


lemma not_set_elmt (x : integer) =
  (not [set:x] sub [set:1,2,3,4]) -> x ne 1 & x ne 2 & x ne 3 & x ne 4;


lemma size_neq_imp_neq (h1,h2 : an_input_event_seq) =
  size (h1) ne size(h2) -> h1 ne h2;


name a_console_buffer, a_source_buffer, a_timed_console_seq,
     a_timed_source_seq
     from flow_modulator;

name an_input, an_input_event_seq, an_input_seq
     from input_specifications;


end; {scope GVE_extension}
```

A.5 Scope Input_Specifications

```
scope input_specifications =
begin


function command (e : an_input_event) : a_command =
  pending;


function command_tag  (x : a_console_object;
                       c : a_command)       : an_input_event_tag =
begin
  exit (assume result = if completes_cmd (x,c) then complete_command
                        else partial_command
                        fi);
end;


function completes_cmd (x : a_console_object;
                        c : a_command)       : boolean =
  pending;


function completes_msg ( x : a_source_object;
                         m : a_message;
                        st : a_modulator_state) : boolean =
  pending;


function console_last (s : a_timed_source_seq;
                       c : a_timed_console_seq) : boolean =
begin
  exit (assume result = if c = null(a_timed_console_seq) then false
                        else if s = null(a_timed_source_seq) then true
                             else timestamp (last(s)) < timestamp (last(c))
                        fi   fi);
end;


function console_part (i : an_input) : a_console_object =
  pending;


function event_pack (inp : an_input_seq;
                     st0 : a_modulator_state;
                       t : a_modulator_constant) : an_input_event_seq =
begin
  exit (assume result = if inp = null(an_input_seq) then
                          null(an_input_event_seq)
                        else event_pack (nonlast(inp), st0, t)
                               <: new_event (last(inp),
                                             event_pack (nonlast(inp),st0,t),
                                             st0, t)
                        fi);
end;
```

```
function input_times (s : a_timed_source_seq;
                      c : a_timed_console_seq;
                      t : integer)              :  boolean =
begin
  exit (assume result = (  [all x: a_timed_source_object,
                              x in s -> t < timestamp(x)]
                         & [all y: a_timed_console_object,
                              y in c -> t < timestamp(y)]));
end;


function is_cmd_event (e : an_input_event) : boolean = pending;
  {is_command(e) or is_partial_command(e)}


function is_command (e : an_input_event) : boolean =
begin
  exit (assume result = [e.tag = complete_command]);
end;


function is_message (e : an_input_event) : boolean =
begin
  exit (assume result = [e.tag = complete_message]);
end;


function is_msg_event (e : an_input_event) : boolean =
begin
  exit (assume result = [is_message(e) or is_partial_message(e)]);
end;


function is_partial_command (e : an_input_event) : boolean =
begin
  exit (assume result = [e.tag = partial_command]);
end;


function is_partial_message (e : an_input_event) : boolean =
begin
  exit (assume result = [e.tag = partial_message]);
end;


function is_source_input (i : an_input) : boolean =
begin
  exit (assume result = [i.tag = source_input]);
end;
```

```
function make_cmd_event (x : a_console_object;
                         c : a_command)          : an_input_event =
begin
  exit (assume result = initial(an_input_event) with
                          (.console_object := x;
                           .command := updated_cmd (x,c);
                           .tag := command_tag (x,c)));
end;
```

```
function make_console_input (x : a_timed_console_object) : an_input =
begin
  exit (assume result = initial(an_input) with
                          (.tag := console_input;
                           .console_part := msg(x)));
end;
```

```
function make_msg_event ( x : a_source_object;
                          m : a_message;
                         st : a_modulator_state) : an_input_event =
begin
  exit (assume result = initial(an_input_event) with
                          (.source_object := x;
                           .message := updated_msg (x,m,st);
                           .tag := message_tag (x,m,st)));
end;
```

```
function make_source_input (x : a_timed_source_object) : an_input =
begin
  exit (assume result = initial(an_input) with
                          (.tag := source_input;
                           .source_part := msg(x)));
end;
```

```
function max_time_stamp (s : a_timed_source_seq;
                         c : a_timed_console_seq) : integer =
begin
  exit (assume   [all x: a_timed_source_object,
                     x in s -> timestamp(x) le result]
               & [all y: a_timed_console_object,
                     y in c -> timestamp(y) le result]);
end;
```

```
function merged_inputs (s : a_timed_source_seq;
                        c : a_timed_console_seq) : an_input_seq =
begin
  exit (assume
    result =
      if s = null(a_timed_source_seq) & c = null(a_timed_console_seq) then
        null(an_input_seq)
      else if console_last (s,c) then
             merged_inputs (s, nonlast(c)) <: make_console_input (last(c))
           else
             merged_inputs (nonlast(s), c) <: make_source_input (last(s))
      fi   fi);
end;


function message (e : an_input_event) : a_message =
  pending;


function message_tag ( x : a_source_object;
                       m : a_message;
                      st : a_modulator_state) : an_input_event_tag =
begin
  exit (assume result = if completes_msg (x,m,st) then complete_message
                        else partial_message
                        fi);
end;


function new_event (  i : an_input;
                      h : an_input_event_seq;
                    st0 : a_modulator_state;
                      t : a_modulator_constant) : an_input_event =
begin
  exit (assume result = if is_source_input (i) then
                          make_msg_event (source_part(i), partial_msg(h),
                                          flow_state (h, st0, t))
                        else {is_console_input(i)}
                          make_cmd_event (console_part(i), partial_cmd(h))
                        fi);
end;


function no_message (h : an_input_event_seq) : boolean =
begin
  exit (assume result = if h = null(an_input_event_seq) then true
                        else   no_message(nonlast(h))
                             & not is_message(last(h))
                        fi);
end;


function no_partial_cmd (c : an_input_event_seq) : boolean =
begin
  exit (assume result = [partial_cmd(c) = null(a_command)]);
end;
```

```
function no_partial_msg (s : an_input_event_seq) : boolean =
begin
  exit (assume result = [partial_msg(s) = null(a_message)]);
end;


function null_source : a_timed_source_seq =
begin
  exit (assume result = null(a_timed_source_seq));
end;


function partial_cmd (h : an_input_event_seq) : a_command =
begin
  exit (assume result = if h = null(an_input_event_seq) then null(a_command)
                        else if is_command (last(h)) then null(a_command)
                             else if is_partial_command (last(h)) then
                                    command (last(h))
                                  else partial_cmd (nonlast(h))
                        fi   fi   fi);
end;


function partial_msg (h : an_input_event_seq) : a_message =
begin
  exit (assume result = if h = null(an_input_event_seq) then null(a_message)
                        else if is_message (last(h)) then null(a_message)
                             else if is_partial_message (last(h)) then
                                    message (last(h))
                                  else partial_msg (nonlast(h))
                        fi   fi   fi);
end;


function source_part (i : an_input) : a_source_object =
begin
  exit (assume result = i.source_part);
end;


function updated_cmd (x : a_console_object;
                      c : a_command)        : a_command =
  pending;


function updated_msg ( x : a_source_object;
                       m : a_message;
                      st : a_modulator_state) : a_message =
  pending;
```

```
lemma add_msg_event (xs : a_timed_source_seq;
                     xc : a_timed_console_seq;
                      x : a_timed_source_object;
                      h : an_input_event_seq;
                      e : an_input_event;
                     st : a_modulator_state;
                      t : a_modulator_constant) =
     [  h = event_pack (merged_inputs(xs,xc), st, t)
      & e = make_msg_event (msg(x), partial_msg(h), flow_state(h,st,t))
      & input_times ([seq: x], null(a_timed_console_seq),
                     max_time_stamp(xs,xc))]
  ->
     h <: e = event_pack (merged_inputs (xs <: x, xc), st, t);


lemma command_event_append (    s : a_timed_source_seq;
                            c1,c2 : a_timed_console_seq;
                            h1,h2 : an_input_event_seq;
                              st0 : a_modulator_state;
                                t : a_modulator_constant) =
     [  h1 = event_pack (merged_inputs(s,c1), st0, t)
      & h2 = event_pack (merged_inputs(null_source,c2),
                         flow_state(h1,st0,t), t)
      & no_partial_cmd (h1) & console_ordered(c2)
      & input_times (null_source, c2, max_time_stamp(s,c1))]
  ->
     h1 @ h2 = event_pack (merged_inputs(s,c1@c2), st0, t);


lemma command_event_merge (c1,c2 : a_timed_console_seq;
                           h1,h2 : an_input_event_seq;
                              st : a_modulator_state;
                               t : a_modulator_constant) =
     [  h1 = event_pack (merged_inputs(null_source,c1), st, t)
      & h2 = event_pack (merged_inputs(null_source,c2),
                         flow_state(h1,st,t), t)
      & no_partial_cmd (h1)]
  ->
     h1 @ h2 = event_pack (merged_inputs(null_source,c1@c2), st, t);


lemma command_event_pack (xc : a_timed_console_seq;
                           h : an_input_event_seq;
                          st : a_modulator_state;
                           t : a_modulator_constant) =
     h = event_pack (merged_inputs(null(a_timed_source_seq),xc),
                     st, t)
  ->
     no_message(h) & no_partial_msg(h);
```

```
lemma event_merge_append (s1,s2 : a_timed_source_seq;
                          c1,c2 : a_timed_console_seq;
                          h1,h2 : an_input_event_seq;
                            st0 : a_modulator_state;
                              t : a_modulator_constant) =
     [  h1 = event_pack (merged_inputs(s1,c1), st0, t)
      & h2 = event_pack (merged_inputs(s2,c2), flow_state(h1,st0,t), t)
      & no_partial_msg (h1) & no_partial_cmd (h1)
      & source_ordered(s2) & console_ordered(c2)
      & input_times (s2, c2, max_time_stamp(s1,c1))]
  ->
     h1 @ h2 = event_pack (merged_inputs(s1@s2,c1@c2), st0, t);


lemma event_not_msg (h : an_input_event_seq;
                     i : integer) =
     [i in [1..size(h)] & no_message(h)]
  ->
     not is_message(h[i]);


lemma event_pack_append (i1,i2 : an_input_seq;
                           st0 : a_modulator_state;
                             t : a_modulator_constant) =
     [  no_partial_msg (event_pack(i1,st0,t))
      & no_partial_cmd (event_pack(i1,st0,t))]
  ->
     event_pack (i1@i2, st0, t)
       = event_pack(i1,st0,t)
          @ event_pack (i2, flow_state (event_pack(i1,st0,t), st0, t), t);


lemma extend_no_message (h1,h2 : an_input_event_seq) =
  [no_message(h1) & no_message(h2)] -> no_message(h1@h2);


lemma extend_no_partial_cmd (h1,h2 : an_input_event_seq) =
     [no_partial_cmd(h1) & no_partial_cmd(h2)]
  ->
     no_partial_cmd(h1@h2);


lemma extend_no_partial_msg (h1,h2 : an_input_event_seq) =
     [no_partial_msg (h1) & no_partial_msg (h2)]
  ->
     no_partial_msg(h1@h2);


lemma merged_inputs_append (s1,s2 : a_timed_source_seq;
                            c1,c2 : a_timed_console_seq) =
     [  source_ordered(s2) & console_ordered(c2)
      & input_times (s2, c2, max_time_stamp(s1,c1))]
  ->
     merged_inputs (s1@s2, c1@c2) = merged_inputs(s1,c1)
                                    @ merged_inputs(s2,c2);
```

```
lemma msg_event_is_msg_and_not_cmd ( e : an_input_event;
                                     x : a_source_object;
                                     m : a_message;
                                    st : a_modulator_state) =
      e = make_msg_event (x,m,st)
   ->
      [is_msg_event(e) & not is_command(e) & not is_partial_command(e)];


lemma partial_cmd_eq (h1,h2 : an_input_event_seq) =
      no_partial_cmd(h1)
   ->
      partial_cmd(h1@h2) = partial_cmd(h2);


lemma partial_msg_eq (h1,h2 : an_input_event_seq) =
      no_partial_msg (h1)
   ->
      partial_msg(h1@h2) = partial_msg(h2);


lemma unchanged_partial_msg (   xc : a_timed_console_seq;
                             h1,h2 : an_input_event_seq;
                                st : a_modulator_state;
                                 t : a_modulator_constant) =
      h2 = event_pack (merged_inputs(null_source,xc), flow_state(h1,st,t), t)
   ->
      partial_msg(h1@h2) = partial_msg(h1);


type an_input = record (         tag : an_input_tag;
                         source_part : a_source_object;
                        console_part : a_console_object);

type an_input_event = record (           tag : an_input_event_tag;
                                     message : a_message;
                                     command : a_command;
                               source_object : a_source_object;
                              console_object : a_console_object);

type an_input_event_seq = sequence of an_input_event;

type an_input_event_tag = (partial_message, partial_command,
                           complete_message, complete_command);

type an_input_seq = sequence of an_input;

type an_input_tag = (source_input, console_input);

type a_command = pending;

type a_message = pending;
```

```
name a_console_object, a_modulator_constant, a_source_object,
     a_timed_console_object, a_timed_console_seq, a_timed_source_object,
     a_timed_source_seq
     from flow_modulator;

name console_ordered, nonlast_eq, source_ordered
     from GVE_extension;

name a_modulator_state, extend_flow_state, flow_state
     from modulator_state;


end; {scope input_specifications}
```

A.6 Scope Log_Specifications

```
scope log_specifications =
begin


function all_logged ( h : an_input_event_seq;
                     st : a_modulator_state;
                      t : a_modulator_constant) : a_log_seq =
begin
  exit (assume
    result = if h = null(an_input_event_seq) then
               null(a_log_seq)
             else all_logged (nonlast(h), st, t)
                    @ log_it (last(h), flow_state(nonlast(h),st,t), t)
             fi);
end;


function auto_filter_log (st : a_modulator_state;
                           t : a_modulator_constant) : a_log_seq =
begin
  exit (assume result = if filter_selector(st) = 1 then
                          auto_filter_log_1 (st,t)
                        else if filter_selector(st) = 2 then
                          auto_filter_log_2 (st,t)
                        else if filter_selector(st) = 3 then
                          auto_filter_log_3 (st,t)
                        else if filter_selector(st) = 4 then
                          auto_filter_log_4 (st,t)
                        else auto_filter_log_5 (st,t)
                        fi fi fi fi);
end;


function auto_filter_log_1 (st : a_modulator_state;
                             t : a_modulator_constant) : a_log_seq =
  pending;


function auto_filter_log_2 (st : a_modulator_state;
                             t : a_modulator_constant) : a_log_seq =
  pending;


function auto_filter_log_3 (st : a_modulator_state;
                             t : a_modulator_constant) : a_log_seq =
  pending;


function auto_filter_log_4 (st : a_modulator_state;
                             t : a_modulator_constant) : a_log_seq =
  pending;
```

```
function auto_filter_log_5 (st : a_modulator_state;
                            t : a_modulator_constant) : a_log_seq =
  pending;


function auto_log (st : a_modulator_state;
                   t : a_modulator_constant) : a_log_seq =
begin
  exit (assume
    result = if msg_disposition (auto_filter_transition(st,t)) = passed then
                auto_filter_log (st,t)
               @ auto_transform_log (auto_filter_transition(st,t), t)
               @ sink_log (auto_transform_transition
                               (auto_filter_transition(st,t),t))
             else auto_filter_log (st,t)
             fi);
end;


function auto_transform_log (st : a_modulator_state;
                             t : a_modulator_constant) : a_log_seq =
begin
  exit (assume result = if transform_selector(st) = 1 then
                          auto_transform_log_1 (st,t)
                        else if transform_selector(st) = 2 then
                          auto_transform_log_2 (st,t)
                        else if transform_selector(st) = 3 then
                          auto_transform_log_3 (st,t)
                        else if transform_selector(st) = 4 then
                          auto_transform_log_4 (st,t)
                        else auto_transform_log_5 (st,t)
                        fi fi fi fi);
end;


function auto_transform_log_1 (st : a_modulator_state;
                               t : a_modulator_constant) : a_log_seq =
  pending;


function auto_transform_log_2 (st : a_modulator_state;
                               t : a_modulator_constant) : a_log_seq =
  pending;


function auto_transform_log_3 (st : a_modulator_state;
                               t : a_modulator_constant) : a_log_seq =
  pending;


function auto_transform_log_4 (st : a_modulator_state;
                               t : a_modulator_constant) : a_log_seq =
  pending;
```

```
function auto_transform_log_5 (st : a_modulator_state;
                               t : a_modulator_constant) : a_log_seq =
  pending;


function command_log ( e : an_input_event;
                      st : a_modulator_state;
                       t : a_modulator_constant) : a_log_seq =
  pending;


function initial_log (table : a_modulator_constant) : a_log_seq =
  pending;


function log_it ( e : an_input_event;
                 st : a_modulator_state;
                  t : a_modulator_constant) : a_log_seq =
begin
  exit (assume result = if is_msg_event(e) then
                          message_log (e, st, t)
                        else {is_cmd_event(e)}
                          command_log (e, st, t)
                        fi);
end;


function manual_mod_log ( l : a_log_seq;
                         st : a_modulator_state) : a_log_seq =
begin
  exit (assume result = if have_in_msg(st) & msg_disposition(st) = passed then
                          l @ sink_log(st)
                        else l
                        fi);
end;


function message_log ( e : an_input_event;
                      st : a_modulator_state;
                       t : a_modulator_constant) : a_log_seq =
begin
  exit (assume result = if is_message(e)
                              & in_auto_mode(source_transition(e,st)) then
                          source_log(e,st)
                            @ auto_log (source_transition(e,st), t)
                        else source_log(e,st)
                        fi);
end;


function sink_log (st : a_modulator_state) : a_log_seq =
  pending;
```

```
function source_log ( e : an_input_event;
                     st : a_modulator_state) : a_log_seq =
  pending;


function table_error_log (table : a_modulator_constant) : a_log_seq =
  pending;


lemma log_append (h1,h2 : an_input_event_seq;
                     s0 : a_modulator_state;
                      t : a_modulator_constant) =
  all_logged (h1@h2, s0, t)
    = all_logged (h1,s0,t) @ all_logged (h2, flow_state(h1,s0,t), t);


name a_log_seq, a_modulator_constant
     from flow_modulator;

name an_input_event, an_input_event_seq, is_cmd_event, is_message,
     is_msg_event
     from input_specifications;

name a_msg_disposition
     from modulation_specifications;

name auto_filter_transition, auto_transform_transition, a_modulator_state,
     extend_flow_state, filter_selector, flow_state, have_in_msg,
     in_auto_mode, msg_disposition, source_transition, transform_selector
     from modulator_state;


end; {scope log_specifications}
```

A.7 Scope Modulation_Specifications

```
scope modulation_specifications =
begin


function all_modulated ( h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_sink_seq =
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then
        null(a_sink_seq)
      else   modulate_it (first(h), st, t, modulation_cmds(h,st,t))
          @ all_modulated (nonfirst(h), flow_transition(first(h),st,t), t)
      fi);
end;



function auto_filter ( m : a_message;
                      st : a_modulator_state;
                       t : a_modulator_constant) : a_msg_disposition =
begin
  exit (assume result = if filter_selector(st) = 1 then
                          auto_filter_1 (m,st,t)
                        else if filter_selector(st) = 2 then
                          auto_filter_2 (m,st,t)
                        else if filter_selector(st) = 3 then
                          auto_filter_3 (m,st,t)
                        else if filter_selector(st) = 4 then
                          auto_filter_4 (m,st,t)
                        else auto_filter_5 (m,st,t)
                        fi fi fi fi);
end;



function auto_filter_1 ( m : a_message;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
  pending;



function auto_filter_2 ( m : a_message;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
  pending;



function auto_filter_3 ( m : a_message;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
  pending;
```

```
function auto_filter_4 ( m : a_message;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
  pending;


function auto_filter_5 ( m : a_message;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
  pending;


function auto_modulation ( m : a_message;
                          st : a_modulator_state;
                           t : a_modulator_constant) : a_sink_seq =
begin
  exit (assume
    result = if auto_filter (m,st,t) = passed then
               sink_seq (auto_transform (m, auto_filter_transition(st,t), t))
             else null(a_sink_seq)
             fi);
end;


function auto_transform ( m : a_message;
                         st : a_modulator_state;
                          t : a_modulator_constant) : a_transformed_msg =
begin
  exit (assume result = if transform_selector(st) = 1 then
                          auto_transform_1 (m,st,t)
                        else if transform_selector(st) = 2 then
                          auto_transform_2 (m,st,t)
                        else if transform_selector(st) = 3 then
                          auto_transform_3 (m,st,t)
                        else if transform_selector(st) = 4 then
                          auto_transform_4 (m,st,t)
                        else auto_transform_5 (m,st,t)
                        fi fi fi fi);
end;


function auto_transform_1 ( m : a_message;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg =
  pending;


function auto_transform_2 ( m : a_message;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg =
  pending;
```

```
function auto_transform_3 ( m : a_message;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg =
  pending;


function auto_transform_4 ( m : a_message;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg =
  pending;


function auto_transform_5 ( m : a_message;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg =
  pending;


function correct_table (table : a_modulator_constant) : boolean =
  pending;


function did_manual_modulation ( h : an_input_event_seq;
                                 m : a_message;
                                st : a_modulator_state;
                                 t : a_modulator_constant) : boolean =
begin
  exit (assume result = [  have_in_msg(st) & in_msg(st) = m
                         & manual_filter(m,h,st,t) in [passed,rejected]]);
end;


function initial_transform (m : a_message;
                            t : a_modulator_constant) : a_transformed_msg =
  pending;


function manual_filter ( m : a_message;
                         h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_msg_disposition =
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then undecided
      else if need_all_cmds(h,st,t) then
             manual_filter_result (m, last(h), flow_state(nonlast(h),st,t), t)
           else manual_filter (m, nonlast(h), st, t)
      fi   fi);
end;
```

```
function manual_filter_result ( m : a_message;
                                e : an_input_event;
                               st : a_modulator_state;
                                t : a_modulator_constant) : a_msg_disposition=
  pending;


function manual_modulation ( m : a_message;
                            st : a_modulator_state;
                             h : an_input_event_seq;
                             t : a_modulator_constant) : a_sink_seq =
begin
  exit (assume
       did_manual_modulation (h, m, st, t)
    ->
       result =
        if manual_filter(m,h,st,t) = passed then
          sink_seq (manual_transform (m, initial_transform(m,t), h, st, t))
        else null(a_sink_seq)
        fi);
end;


function manual_transform ( m : a_message;
                           tm : a_transformed_msg;
                            h : an_input_event_seq;
                           st : a_modulator_state;
                            t : a_modulator_constant) : a_transformed_msg =
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then tm
      else if need_all_cmds(h,st,t) then
             msg_transform (m, manual_transform(m,tm,nonlast(h),st,t),
                            last(h), flow_state(nonlast(h),st,t), t)
           else manual_transform (m, tm, nonlast(h), st, t)
      fi   fi);
end;


function modulate_it ( e : an_input_event;
                      st : a_modulator_state;
                       t : a_modulator_constant;
                       h : an_input_event_seq) : a_sink_seq =
begin
  exit (assume
    result =
      if not is_message(e) then null(a_sink_seq)
      else if in_auto_mode(source_transition(e,st)) then
             auto_modulation (message(e), source_transition(e,st), t)
           else manual_modulation (message(e), flow_transition(e,st,t), h, t)
      fi   fi);
end;
```

```
function modulation_cmds ( h : an_input_event_seq;
                          st : a_modulator_state;
                           t : a_modulator_constant) : an_input_event_seq =
begin
  exit (assume
    result =
      if h = null(an_input_event_seq) then
        null(an_input_event_seq)
      else if need_mod_cmds (first(h), st) then
            mod_cmd_head (nonfirst(h), flow_transition(first(h),st,t), t)
          else null(an_input_event_seq)
      fi   fi);
end;


function mod_cmd_head ( h : an_input_event_seq;
                       st : a_modulator_state;
                        t : a_modulator_constant) : an_input_event_seq =
begin
  exit (assume result = if need_all_cmds (h, st, t) then h
                        else mod_cmd_head (nonlast(h), st, t)
                        fi);
end;


function msg_transform ( m : a_message;
                        tm : a_transformed_msg;
                         e : an_input_event;
                        st : a_modulator_state;
                         t : a_modulator_constant) : a_transformed_msg =
  pending;


function need_all_cmds ( h : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = if h = null(an_input_event_seq) then true
             else   not is_msg_event (last(h))
                  & have_in_msg (flow_state (nonlast(h),st,t))
                  & need_all_cmds (nonlast(h), st, t)
             fi);
end;


function need_mod_cmds ( e : an_input_event;
                        st : a_modulator_state) : boolean =
begin
  exit (assume result = [  is_message(e)
                         & not in_auto_mode (source_transition(e,st))]);
end;
```

```
function no_pending_modulation (st : a_modulator_state) : boolean =
begin
  exit (assume result = [not have_in_msg(st) & in_msg(st) = null(a_message)]);
end;


lemma contract_need_all_cmds (h1,h2 : an_input_event_seq;
                                  st : a_modulator_state;
                                   t : a_modulator_constant) =
     need_all_cmds (h1@h2, st, t)
  ->
     need_all_cmds (h2, flow_state(h1,st,t), t);


lemma extend_need_all_cmds (h1,h2 : an_input_event_seq;
                                st : a_modulator_state;
                                 t : a_modulator_constant) =
     [  need_all_cmds (h1, st, t)
      & need_all_cmds (h2, flow_state(h1,st,t), t)]
  ->
     need_all_cmds (h1@h2, st, t);


lemma manual_filter_eq (     m : a_message;
                         h1,h2 : an_input_event_seq;
                            st : a_modulator_state;
                             t : a_modulator_constant) =
     [  manual_filter(m,h1,st,t) = undecided
      & need_all_cmds (h1@h2, st, t)]
  ->
     manual_filter (m, h1@h2, st, t)
       = manual_filter (m, h2, flow_state(h1,st,t), t);


lemma manual_transform_eq (     m : a_message;
                               tm : a_transformed_msg;
                            h1,h2 : an_input_event_seq;
                               st : a_modulator_state;
                                t : a_modulator_constant) =
     need_all_cmds (h1@h2, st, t)
  ->
     manual_transform (m, tm, h1@h2, st, t)
       = manual_transform (m, manual_transform(m,tm,h1,st,t), h2,
                           flow_state(h1,st,t), t);


lemma mod_cmd_eq (h1,h2 : an_input_event_seq;
                     st : a_modulator_state;
                      t : a_modulator_constant) =
     [  no_pending_modulation (flow_state(h1,st,t))
      & h1 ne null(an_input_event_seq)]
  ->
     modulation_cmds (h1,st,t) = modulation_cmds (h1@h2,st,t);
```

```
lemma mod_cmd_head_eq (h1,h2 : an_input_event_seq;
                         st : a_modulator_state;
                          t : a_modulator_constant) =
     no_pending_modulation (flow_state(h1,st,t))
  ->
     mod_cmd_head (h1 @ h2, st, t) = mod_cmd_head (h1, st, t);


lemma null_sink ( h : an_input_event_seq;
                 st : a_modulator_state;
                  t : a_modulator_constant) =
     no_message(h)
  ->
     all_modulated (h,st,t) = null(a_sink_seq);


lemma null_sink_head (h1,h2 : an_input_event_seq;
                        st : a_modulator_state;
                         t : a_modulator_constant) =
     no_message(h1)
  ->
     all_modulated (h1 @ h2, st, t)
       = all_modulated (h2, flow_state(h1,st,t), t);


lemma null_sink_subseq (h1,h2 : an_input_event_seq;
                            i : integer;
                           st : a_modulator_state;
                            t : a_modulator_constant) =
     [i in [0..size(h1)] & no_message(h1)]
  ->
     all_modulated (h2, flow_state(h1,st,t), t)
       = all_modulated (h1[size(h1) - i + 1..size(h1)] @ h2,
                        flow_state(h1[1..size(h1)-i],st,t), t);


lemma sink_append (h1,h2 : an_input_event_seq;
                     s0 : a_modulator_state;
                      t : a_modulator_constant) =
     no_pending_modulation (flow_state(h1,s0,t))
  ->
       all_modulated (h1@h2, s0, t)
     = all_modulated (h1,s0,t) @ all_modulated (h2, flow_state(h1,s0,t), t);


lemma sink_tail (h1,h2 : an_input_event_seq;
                     i : integer;
                   st0 : a_modulator_state;
                     t : a_modulator_constant) =
     [i in [0..size(h1)] & no_pending_modulation (flow_state(h1,st0,t))]
  ->
     all_modulated (h1[size(h1)-i+1..size(h1)] @ h2,
                    flow_state (h1[1..size(h1)-i], st0, t), t)
       = all_modulated (h1[size(h1)-i+1..size(h1)],
                        flow_state (h1[1..size(h1)-i], st0, t), t)
           @ all_modulated (h2, flow_state (h1, st0, t), t);
```

```
type a_filter_selector = integer[1..5];

type a_msg_disposition = (undecided, rejected, passed);

type a_transformed_msg = pending;

type a_transform_selector = integer[1..5];


name a_modulator_constant, a_sink_seq from flow_modulator;

name first_eq, nonfirst_eq, size_neq_imp_neq
     from GVE_extension;

name an_input_event, an_input_event_seq, a_message, event_not_msg,
     is_message, is_msg_event, message, no_message
     from input_specifications;

name auto_filter_transition, a_modulator_state, extend_flow_state,
     filter_selector, flow_state, flow_transition, have_in_msg, in_auto_mode,
     in_msg, source_transition, transform_selector
     from modulator_state;

name sink_seq from sink_handler;

end; {scope modulation_specifications}
```

A.8 Scope Modulator_State

```
scope modulator_state =
begin


procedure initialize_manual_modulation (var state : a_modulator_state;
                                            table : a_modulator_constant) =
begin
  entry correct_table (table);
  exit   state = manual_mod_init(state',table) & msg_integrity(state,state')
       & msg_disposition(state) = undecided
       & out_msg(state) = initial_transform (in_msg(state'), table);
  pending;
end;



procedure remove_message (var st : a_modulator_state) =
begin
  exit st = msg_removed(st') & no_pending_modulation(st);
  pending;
end;



function auto_filter_transition
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
begin
  exit (assume result = if filter_selector(st) = 1 then
                          auto_filter_transition_1 (st,t)
                        else if filter_selector(st) = 2 then
                          auto_filter_transition_2 (st,t)
                        else if filter_selector(st) = 3 then
                          auto_filter_transition_3 (st,t)
                        else if filter_selector(st) = 4 then
                          auto_filter_transition_4 (st,t)
                        else auto_filter_transition_5 (st,t)
                        fi fi fi fi);
end;


function auto_filter_transition_1
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_filter_transition_2
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_filter_transition_3
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;
```

```
function auto_filter_transition_4
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_filter_transition_5
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_transform_transition
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
begin
  exit (assume result = if transform_selector(st) = 1 then
                          auto_transform_transition_1 (st,t)
                        else if transform_selector(st) = 2 then
                          auto_transform_transition_2 (st,t)
                        else if transform_selector(st) = 3 then
                          auto_transform_transition_3 (st,t)
                        else if transform_selector(st) = 4 then
                          auto_transform_transition_4 (st,t)
                        else auto_transform_transition_5 (st,t)
                        fi fi fi fi);
end;


function auto_transform_transition_1
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_transform_transition_2
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_transform_transition_3
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_transform_transition_4
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;
```

```
function auto_transform_transition_5
                (st : a_modulator_state;
                  t : a_modulator_constant) : a_modulator_state =
  pending;


function auto_transition (st : a_modulator_state;
                           t : a_modulator_constant) : a_modulator_state =
begin
  exit (assume
    result = if msg_disposition (auto_filter_transition(st,t)) = passed then
                msg_removed (auto_transform_transition
                                (auto_filter_transition(st,t), t))
             else msg_removed (auto_filter_transition(st,t))
             fi);
end;


function command_transition ( e : an_input_event;
                              st : a_modulator_state;
                               t : a_modulator_constant) : a_modulator_state=
  pending;


function console_transition ( e : an_input_event;
                              st : a_modulator_state;
                               t : a_modulator_constant) : a_modulator_state=
  pending;


function filter_selector (st : a_modulator_state) : a_filter_selector =
  pending;


function flow_state ( h : an_input_event_seq;
                      st : a_modulator_state;
                       t : a_modulator_constant) : a_modulator_state =
begin
  exit (assume
    result = if h = null(an_input_event_seq) then st
             else
               flow_transition (last(h), flow_state(nonlast(h),st,t), t)
             fi);
end;
```

```
function flow_transition ( e : an_input_event;
                          st : a_modulator_state;
                           t : a_modulator_constant) : a_modulator_state =
begin
  exit (assume
    result = if is_message(e) then
                message_transition (e, st, t)
             else if is_partial_message(e) then
                    source_transition (e, st)
                  else if is_command(e) then
                         command_transition (e, st, t)
                       else console_transition (e, st, t)
             fi   fi   fi);
end;


function have_in_msg (st : a_modulator_state) : boolean =
  pending;


function initial_state (table : a_modulator_constant) : a_modulator_state =
  pending;


function in_auto_mode (st : a_modulator_state) : boolean =
  pending;


function in_msg (st : a_modulator_state) : a_message =
  pending;


function manual_mod_init (st : a_modulator_state;
                           t : a_modulator_constant) : a_modulator_state =
  pending;


function manual_mod_transition (st : a_modulator_state) : a_modulator_state =
begin
  exit (assume
    result = if have_in_msg(st) & msg_disposition(st) ne undecided then
                msg_removed(st)
             else st
             fi);
end;
```

```
function message_transition ( e : an_input_event;
                             st : a_modulator_state;
                              t : a_modulator_constant) : a_modulator_state=
begin
  exit (assume result = if in_auto_mode (source_transition(e,st)) then
                          auto_transition (source_transition(e,st), t)
                        else manual_mod_init (source_transition(e,st), t)
                        fi);
end;


function msg_disposition (st : a_modulator_state) : a_msg_disposition =
  pending;


function msg_integrity (st2,st1 : a_modulator_state) : boolean =
begin
  exit (assume result = [  in_msg(st2) = in_msg(st1)
                         & (have_in_msg(st2) = have_in_msg(st1))]);
end;


function msg_removed (st : a_modulator_state) : a_modulator_state =
  pending;


function no_in_msg (st : a_modulator_state) : boolean =
begin
  exit result = [in_msg(st) = null(a_message)];
  pending;
end;


function out_msg (st : a_modulator_state) : a_transformed_msg =
  pending;


function source_transition ( e : an_input_event;
                            st : a_modulator_state) : a_modulator_state =
  pending;


function stop_modulation (state : a_modulator_state) : boolean =
  pending;


function transform_selector (st : a_modulator_state) : a_transform_selector =
  pending;
```

```
lemma extend_flow_state (h1,h2 : an_input_event_seq;
                             s0 : a_modulator_state;
                              t : a_modulator_constant) =
  flow_state (h1@h2, s0, t) = flow_state (h2, flow_state(h1,s0,t), t);


type a_modulator_state = pending;


name a_modulator_constant from flow_modulator;

name an_input_event, an_input_event_seq, a_command, a_message, command,
     is_command, is_message, is_partial_message
     from input_specifications;

name a_filter_selector, a_msg_disposition, a_transformed_msg,
     a_transform_selector, correct_table, initial_transform,
     no_pending_modulation
     from modulation_specifications;


end; {scope modulator_state}
```

A.9 Scope Sink _ Handler

```
scope sink_handler =
begin


procedure prepare_sink_send (var display : a_display_buffer<output>;
                             var     log : a_log_buffer<output>;
                             var   state : a_modulator_state;
                             var       m : a_sink_seq) =
begin
  exit   m = sink_seq(out_msg(state')) & state = msg_removed(state')
       & no_pending_modulation(state) & outto(log,myid) = sink_log(state')
       & outto(display,myid) = sink_display(state');
  pending;
end;



procedure send_to_sink (var    sink : a_sink_buffer<output>;
                        var console : a_console_buffer<input>;
                        var display : a_display_buffer<output>;
                        var     log : a_log_buffer<output>;
                        var   state : a_modulator_state;
                              table : a_modulator_constant;
                               time : integer) =
begin
  entry correct_table(table);   {and TIME is earlier than the present}
  exit (prove sent_to_sink (outto(sink,myid), xinfrom(console,myid),
                            outto(display,myid), outto(log,myid),
                            state', state, table);
        assume input_times (null_source, xinfrom(console,myid), time));
  var m: a_sink_seq;
  var i: integer := 1;
  prepare_sink_send (display, log, state, m);
  loop
    assert   sending_to_sink (m, i, outto(sink,myid), xinfrom(console,myid),
                              outto(display,myid), outto(log,myid),
                              state', state, table)
           & correct_table(table);
    if i > size(m) then leave
    elif not empty(console) then
      follow_command (console, display, log, state, table,
                      max_time_stamp (null_source, xinfrom(console,myid)))
    elif not full(sink) then
      send m[i] to sink;
      i := i + 1
    end;
  end;
end;
```

```
function sending_to_sink (      m : a_sink_seq;
                                i : integer;
                             sink : a_sink_seq;
                               xc : a_timed_console_seq;
                                d : a_display_seq;
                                l : a_log_seq;
                          st1,st2 : a_modulator_state;
                                t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  sink = m[1..i-1] & i in [1..size(m) + 1]
              & m = sink_seq (out_msg(st1))
              & some st: a_modulator_state, some h: an_input_event_seq,
                    st = msg_removed (st1) & no_pending_modulation(st)
                & h = event_pack(merged_inputs(null_source,xc), st, t)
                & d = sink_display (st1) @ all_displayed (h,st,t)
                & l = sink_log (st1) @ all_logged (h,st,t)
                & st2 = flow_state (h,st,t)
                & no_pending_modulation (st2)
                & no_partial_cmd (h)]);
end;


function sent_to_sink (   sink : a_sink_seq;
                            xc : a_timed_console_seq;
                             d : a_display_seq;
                             l : a_log_seq;
                       st1,st2 : a_modulator_state;
                             t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = [  sink = sink_seq (out_msg(st1))
              & some st: a_modulator_state, some h: an_input_event_seq,
                    st = msg_removed (st1) & no_pending_modulation(st)
                & h = event_pack(merged_inputs(null_source,xc), st, t)
                & d = sink_display (st1) @ all_displayed (h,st,t)
                & l = sink_log (st1) @ all_logged (h,st,t)
                & st2 = flow_state (h,st,t)
                & no_pending_modulation (st2)
                & no_partial_cmd (h)]);
end;


function sink_seq (m : a_transformed_msg) : a_sink_seq =
  pending;


name followed_command, follow_command, manual_mod_update
     from command_interpreter;

name all_displayed, display_append, manual_mod_display, sink_display
     from display_specifications;

name a_console_buffer, a_display_buffer, a_display_seq, a_log_buffer,
     a_log_seq, a_modulator_constant, a_sink_buffer, a_sink_seq,
     a_timed_console_seq, a_timed_source_seq
     from flow_modulator;
```

```
name an_input_event_seq, a_message, command_event_merge, event_pack,
     extend_no_partial_cmd, input_times, max_time_stamp, merged_inputs,
     no_partial_cmd, null_source
     from input_specifications;

name all_logged, log_append, manual_mod_log, sink_log
     from log_specifications;

name a_msg_disposition, a_transformed_msg, correct_table,
     no_pending_modulation
     from modulation_specifications;

name a_modulator_state, extend_flow_state, flow_state, have_in_msg, in_msg,
     manual_mod_transition, msg_disposition, msg_integrity, msg_removed,
     out_msg
     from modulator_state;


end; {scope sink_handler}
```

A.10 Scope Source_Handler

```
scope source_handler =
begin


procedure receive_message (var  source  : a_source_buffer<input>;
                           var console  : a_console_buffer<input>;
                           var display  : a_display_buffer<output>;
                           var     log  : a_log_buffer<output>;
                           var   state  : a_modulator_state;
                                 table  : a_modulator_constant)
        unless (stop_mod) =
begin
  entry no_pending_modulation(state) & correct_table(table);
  exit case (is normal:
               received_msg (xinfrom(source,myid), xinfrom(console,myid),
                             outto(display,myid), outto(log,myid),
                             state', state, table);
             is stop_mod:
               received_no_msg (xinfrom(source,myid), xinfrom(console,myid),
                                outto(display,myid), outto(log,myid),
                                state', state, table));
  loop
    assert   receiving_msg (xinfrom(source,myid), xinfrom(console,myid),
                            outto(display,myid), outto(log,myid),
                            state', state, table)
           & correct_table(table);
    if have_in_msg(state) then leave
    elif stop_modulation(state) & no_in_msg(state) then
      signal stop_mod
    elif not empty(console) then
      follow_command
        (console, display, log, state, table,
         max_time_stamp (xinfrom(source,myid), xinfrom(console,myid)))
    elif not empty(source) then
      update_message
        (source, display, log, state,
         max_time_stamp (xinfrom(source,myid), xinfrom(console,myid)))
    end;
   end;
 end;


procedure update_message (var  source  : a_source_buffer<input>;
                          var display  : a_display_buffer<output>;
                          var     log  : a_log_buffer<output>;
                          var   state  : a_modulator_state;
                                 time  : integer) =
begin
  entry not have_in_msg(state); {and TIME is earlier than the present}
  exit (prove updated_message (xinfrom(source,myid), outto(display,myid),
                               outto(log,myid), state', state);
        assume input_times (xinfrom(source,myid), null(a_timed_console_seq),
                            time));
  pending;
end;
```

```
function received_msg (      xs : a_timed_source_seq;
                             xc : a_timed_console_seq;
                              d : a_display_seq;
                              l : a_log_seq;
                        st1,st2 : a_modulator_state;
                              t : a_modulator_constant) : boolean =
begin
  exit (assume
    result =
      some h: an_input_event_seq,
          h = event_pack (merged_inputs(xs,xc), st1, t)
        & h ne null(an_input_event_seq)
        & no_message (nonlast(h)) & is_message (last(h))
        & no_partial_cmd(h)
        & d =   all_displayed (nonlast(h), st1, t)
             @ source_display (last(h), flow_state(nonlast(h),st1,t))
        & l =   all_logged (nonlast(h), st1, t)
             @ source_log (last(h), flow_state(nonlast(h),st1,t))
        & st2 = source_transition(last(h),flow_state(nonlast(h),st1,t))
        & in_msg(st2) = message(last(h)) & have_in_msg(st2));
end;


function received_no_msg (      xs : a_timed_source_seq;
                                xc : a_timed_console_seq;
                                 d : a_display_seq;
                                 l : a_log_seq;
                           st1,st2 : a_modulator_state;
                                 t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = some h: an_input_event_seq,
                h = event_pack (merged_inputs(xs,xc), st1, t)
              & d =   all_displayed (h, st1, t)
              & l =   all_logged (h, st1, t)
              & st2 = flow_state (h,st1,t)
              & stop_modulation (st2) & no_pending_modulation(st2)
              & no_message (h) & no_partial_msg (h) & no_partial_cmd(h));
end;
```

```
function receiving_msg (      xs : a_timed_source_seq;
                              xc : a_timed_console_seq;
                               d : a_display_seq;
                               l : a_log_seq;
                         st1,st2 : a_modulator_state;
                               t : a_modulator_constant) : boolean =
begin
  exit (assume
    result = if have_in_msg(st2) then
                received_msg (xs,xc,d,l,st1,st2,t)
             else
                some h: an_input_event_seq,
                    h = event_pack (merged_inputs(xs,xc), st1, t)
                  & no_message(h) & no_partial_cmd(h)
                  & d = all_displayed (h,st1,t)
                  & l = all_logged (h,st1,t)
                  & st2 = flow_state (h,st1,t)
                  & in_msg(st2) = partial_msg(h)
             fi);
end;


function updated_message (      xs : a_timed_source_seq;
                                 d : a_display_seq;
                                 l : a_log_seq;
                           st1,st2 : a_modulator_state) : boolean =
begin
  exit (assume
    result = some x: a_timed_source_object, some e: an_input_event,
                  xs = [seq: x] & e = make_msg_event (msg(x), in_msg(st1), st1)
                & d = source_display (e,st1)
                & l = source_log (e,st1)
                & st2 = source_transition (e,st1)
                & in_msg(st2) = message(e)
                & [is_message(e) iff have_in_msg(st2)]);
end;


name followed_command, follow_command, manual_mod_update
     from command_interpreter;

name all_displayed, display_append, manual_mod_display, source_display
     from display_specifications;

name a_console_buffer, a_display_buffer, a_display_seq, a_log_buffer,
     a_log_seq, a_modulator_constant, a_source_buffer, a_timed_console_seq,
     a_timed_source_object, a_timed_source_seq
     from flow_modulator;

name console_ordered, is_console_ordered
     from GVE_extension;
```

```
name add_msg_event, an_input_event, an_input_event_seq, a_command, a_message,
     command, command_event_append, command_event_pack, event_pack,
     extend_no_message, extend_no_partial_cmd, input_times, is_command,
     is_message, is_msg_event, is_partial_command, is_partial_message,
     make_msg_event, max_time_stamp, merged_inputs, message,
     msg_event_is_msg_and_not_cmd, no_message, no_partial_cmd,
     no_partial_msg, null_source, partial_cmd, partial_msg,
     unchanged_partial_msg
     from input_specifications;

name all_logged, log_append, manual_mod_log, source_log
     from log_specifications;

name correct_table, no_pending_modulation
     from modulation_specifications;

name a_modulator_state, extend_flow_state, flow_state, have_in_msg, in_msg,
     manual_mod_transition, msg_integrity, no_in_msg, source_transition,
     stop_modulation
     from modulator_state;


end; {scope source_handler}
```

References

[Craigen 82] Dan Craigen.
A Formal Specification Report of the LSI Guard.
I. P. Sharp Tech Report TR-5031-82-2, August 1982.

[Good 78] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

[Good 81] Donald I. Good.
Message Flow Modulator Status Report.
Internal Note #15, Institute for Computing Science, The University of Texas at Austin.

[Good 82] Donald I. Good, Ann E. Siebert, Lawrence M. Smith.
Message Flow Modulator - Final Report.
Technical Report #34, Institute for Computing Science, The University of Texas at Austin, December, 1982.

[Siebert 84a] Ann E. Siebert.
General Message Flow Modulator - Proof Logs.
Internal Note #128, Institute for Computing Science, The University of Texas at Austin, March 1984.

[Siebert 84b] Ann E. Siebert.
General Message Flow Modulator - Instructions for Running One Member of the Family.
Internal Note #132, Institute for Computing Science, The University of Texas at Austin, March 1984.