Message Flow Modulator
Final Report

Donald I. Good
Ann E. Siebert
Lawrence M. Smith

Technical Report 34 December 1982

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

Abstract

The message flow modulator is a formally specified and proved filter program that is applied continuously to a stream of messages flowing from one computer system to another. Messages that pass the filter are passed to their destination. Messages that do not are logged on an audit trail. The modulator has been designed specifically to monitor the flow of security sensitive message traffic from the Ocean Surveillance Information System of the United States Naval Electronic Systems Command.

The modulator has been designed, specified, and implemented in the Gypsy language. All of the modulator, from the highest level of design to the lowest level of coding, has been formally specified and mechanically proved with the Gypsy Verification Environment. The modulator is specifically designed and intended for use in actual field operation. It has been tested in a simulated operational environment at the Patuxent River Naval Air Test Center with scenarios developed by an independent, external group. Without any modification, the proved modulator passed all of these tests on the first attempt.

Table of Contents

Acknowledgements

The development of the proved message flow modulator was sponsored by the United States Naval Electronic Systems Command (NAVELEX) through Contract N00039-81-C-0074. The contributions of Gene Moy, Ken Alonge, H. O. Lubbes and Dan Gamble to the design and development of the flow modulator are gratefully acknowledged. We also wish to thank Eilene Neely, Dave Mokry, Sharon Spence, Jim McIntyre, Ray Randall, Barkley Shepherd, Bruce Levy, Eleanor Belknap and Barbara Howard for their assistance in testing and demonstrating the flow modulator at the Patuxent River Naval Air Test Center.

Chapter 1
MESSAGE FLOW MODULATOR

The Message Flow Modulator (MFM) has been developed as an example of using formal specification and program proof methods in developing computer security software. The formal methods have been used effectively at various stages in the software development cycle, and the software that has been produced has performed extremely well in all tests.

1.1 Function

The MFM is a formally specified and proved filter program that is applied continuously to a stream of messages flowing from one computer system to another. Messages that pass the filter are passed to their destination. Messages that do not are logged on an audit trail.

```
                                   -----------          ---------------
---------------                   |           | Passed |               |
|             |     Messages      |           |        |               |
|   Source    |------------------>|    MFM    |------->|  Destination  |
|             |                   |           |        |               |
---------------                    -----------          ---------------
                                        | Rejected
                                        V
                                   -----------
                                   |   Log   |
                                   -----------
```

The modulator has been designed specifically to monitor the flow of security sensitive message traffic from the Ocean Surveillance Information System (OSIS) of the United States Naval Electronic Systems Command. The filter applies pre-defined security criteria to each message. Those messages that satisfy the security criteria are passed to the destination. All others are rejected and logged on an audit trail.

The primary purpose of developing the proved MFM has been to demonstrate the use of formal specification and program proof methods in developing a small software system that could be used in actual operation. The proved MFM has been developed completely within the framework of the Gypsy methodology [Good 78]. The Gypsy methodology has been used throughout the development of the MFM, from the initial stages of its design through the final stages of coding. Formal specifications and proofs have been given in Gypsy at all levels of the MFM software.

The MFM is the second of two major demonstrations of applying formal specification and program proof methods to all levels of computer security software. The other application is outlined in [Good 82a]. Although the other application was developed to the running prototype stage and successfully demonstrated, the MFM is the first mechanically proved software system that is intended for actual operational use.

In order to demonstrate the use of the formal specification and proof methods on a software system that could be used in actual operation, the MFM was developed specifically to be a security filter with the Ocean Surveillance Information System (OSIS). Although the higher levels of the MFM design and proof are fairly general, the lower levels contain message formats and a filter that are specific to OSIS. Formal specifications and proofs have been given at all of these levels.

The OSIS MFM filter is a simple pattern matcher. The MFM filter is a table of patterns that defines security sensitive key words and how they can be punctuated. The MFM filter searches each message completely for all occurrences of every pattern in the table. A message is passed to its destination only if it does not contain any occurrence of any pattern; otherwise, it is logged as a security violation.

1.2 Proof

In general, the correct operation of computer security software is a major concern. The MFM has been developed as an example of how formal specification and proof methods can be used to provide a high level of confidence in the correct operation of security software. The following steps have been taken:

1. Formal specifications have been given for all levels of the MFM software, and formal, mechanical proofs that the software always runs according to those specifications have been constructed.
2. The formally proved MFM has been tested to see that its formal specifications adequately describe what is expected of MFM operation and that it runs with adequate efficiency.
3. The MFM runs on LSI-11 hardware that is connected to OSIS and its destination system only by standard RS-232 input/output interfaces. This hardware separation protects the proved software from inadvertent or malicious damage by either OSIS or its destination system.

As further protection, before the proved MFM is placed into actual operation, its software and its security filter table will be burned into PROM. Collectively, these steps provide a very high degree of confidence that the messages passed to the OSIS destination will not contain any of the security-sensitive patterns contained in the MFM filter.

It is important to note, however, that even all of these steps provide no guarantee of absolute security. It is tempting to believe that a formally specified and proved program should be absolutely correct, but there are several reasons why a proved program may not behave exactly as expected.

1. The formal specifications may not describe exactly what is expected of program behavior.
2. Every proof is based on certain assumptions, and some of these may be invalid.
3. The verification tools that support the formal methods may malfunction.
4. The compiler that compiles the proved program may malfunction.
5. The run-time support for the proved program may malfunction.
6. The hardware that runs the proved program may malfunction.

Some of these simply are due to the current early stage of development of formal methods. Others are unavoidable. The first two of these potential sources of error are unavoidable because they are subjective and involve human judgment. The last four, in principle, could be minimized by specifying and proving the tools that support the formal methods, the compiler, the run-time support, and to some degree, the hardware. These proofs of this magnitude and complexity, however, are well beyond our current capabilities.

For the reasons listed above, formal specification and proof, in general, do not offer either absolute correctness or absolute security. But, in spite of these potential sources of error, formal specification and proof provide a much higher degree of confidence that a program will function as expected than can be attained by other means. This is simply because the use of formal specification and proof makes a much larger part of the software development process an objective activity rather than a subjective one.

1.3 Development

The proved MFM has been developed entirely within the framework of the Gypsy methodology for building formally specified and proved software systems. First, specifications for the MFM as a total system were given. Then, the total system was structured into its major subcomponents. Specifications were formulated for these, and a proof of the total system in terms of this subcomponent structure was given. This high level system specification and design is quite straight forward. Basically, it says the the MFM applies its filter to all messages and sends them to the right place. This level of abstraction, however, omits all of the details of what the filter actually does and all of the details about input and output. At this stage in the MFM development, the expectations about these details of the modulator were not well defined.

Therefore, the next major stage was the development of a sequence of running prototypes. These prototypes were used to clarify, evaluate, and evolve the specific expectations of MFM behavior. These running prototypes also were used to test the actual input/output interfaces with OSIS, to evaluate MFM performance and to help select a pattern matching procedure that gave adequate efficiency. Once the more detailed MFM expectations stabilized and the performance issues were resolved, formal specification and proofs were constructed for all levels of the MFM implementation. This led to some additional evolution of the MFM. Finally, the proved MFM was tested with a test plan that was developed independently by System Development Corporation [Neely 82]. These tests were conducted on the OSIS system at the Patuxent River Naval Air Test Center, and without any modification, the proved MFM passed all tests on the first attempt.

The final proved MFM consists of 1283 lines of formal specifications, and 556 lines of executable Gypsy program which compiles into 3849 words of LSI-11 machine code. (The 556 lines counts only those lines of Gypsy that actually produce LSI-11 machine code. It does not, for example, include type declarations.) The proof of the MFM consists of proving 348 theorems in the Gypsy Verification Environment. All of these theorems were constructed automatically, and proved mechanically with the interactive theorem prover. The complete MFM effort (not including the development of the external test plan) required an estimated 286 working days for a net result of 1.94 lines of proved executable Gypsy code per working day. The effort used an estimated 220 CPU hours on a DEC 2060 computer system for a net result of 2.53 lines of proved executable code per CPU hour. Also, approximately 45,000 page-months of disk storage were used.

Chapter 2
DESIGN

The fundamental requirement of reliable software is simplicity, and the basic design of the MFM is very simple. It receives one sequence of messages as input, and separates it into two output sequences, a sequence of passed messages and a sequence of rejected ones.

```
Input Messages ---> MFM ---> Passed Messages
   (Source)          |          (Destination)
                     V
            Rejected Messages
                  (Log)
```

The passed messages are passed to the destination, and the rejected ones are recorded on a log along with their specific security violations.

2.1 Message Security

In the MFM, a message is secure if and only if it contains no security sensitive pattern.

2.1.1 Pattern Matching

A message is a sequence of ASCII characters that begins with the four characters "ZCZC" and ends with an "NNNN" sequence. For purposes of pattern matching, a message is regarded as being composed of just three kinds of characters: letters, digits and delimiters. A delimiter is any ASCII character except a letter or digit. This includes blanks, carriage returns, line feeds, etc. Also, the opening "ZCZC" is regarded as a single delimiter, and so is the closing "NNNN" sequence.

A pattern is defined by a sequence of upper case letters, digits, dots(.) and stars(*). In essence, the string of letters and digits defines a security sensitive key word, and the dots and stars indicate where delimiters may or must appear. Thus, the dots and stars define a variety of ways in which the word may be "punctuated" with delimiters and still be considered a security violation.

Specifically, the characters in a pattern are matched as follows:

1. A letter matches either the upper or lower case of that letter.

2. A digit matches itself.

3. A dot(.) matches any one delimiter.

4. A star(*) matches the longest continuous string of zero or more delimiters.

Thus, a dot requires some delimiter to appear at that point in the key word, and a star allows one or more delimiters to appear. Typically, dots are used to require that the key word be embedded in delimiters, and stars are used to allow delimiters to appear within the key word.

There are several consequences of these pattern matching rules that should be noted. First, a pattern that contains a "*." sequence never matches because the star matches the longest continuous string of delimiters that appear at that point, and there is none left for the dot to match. Second, some patterns that are expressed differently actually may be equivalent in that they match exactly the same character sequences. For example, if p and q are strings of just letters and digits, then the pattern *p* is equivalent to p, .*p.* is equivalent to .p., and p**q is equivalent to p*q. In general, a shorter pattern is matched more efficiently than an equivalent longer one.

The MFM security filter consists of a table of patterns. A message is secure only if it does not contain any match of any pattern in the table. Currently, the maximum pattern size is 26 characters, and the maximum table size is 200 patterns. These limits, however, are restricted only by available storage capacity.

The following are examples of pattern matches:

```
PATTERN       MESSAGE

              ZCZCHigh: Blue-Fin was highly successful.NNNN
HIGH              xxxx               xxxx
.HIGH.           xxxxxx
.H*I*G*H.        xxxxxx
BLUE*FIN                xxxxxxxx
.BLUE*FIN.             xxxxxxxxxx

              ZCZC[H.I.G.H] Blue-Fin was highly successful.NNNN
HIGH                                     xxxx
.HIGH.
.H*I*G*H.         xxxxxxxxx
BLUE*FIN                    xxxxxxxx
.BLUE*FIN.                 xxxxxxxxxx

              ZCZC Low: Up high, it became blue finally.NNNN
HIGH                       xxxx
.HIGH.                    xxxxxx
.H*I*G*H.                 xxxxxx
BLUE*FIN                                 xxxxxxxx
.BLUE*FIN.
```

This very simple pattern matching scheme has some obvious deficiencies in detecting actual security violations in messages that are composed of arbitrary text strings. For example, if a security sensitive key word is misspelled in the message, it may not be detected. Also, the pattern match may detect a number of false alarms – the use of a security sensitive key word in a non-sensitive way. The pattern matcher, however, has not been designed to detect all security violations in arbitrary, manually composed text strings. It has been designed to be applied primarily to message headers that are constructed mechanically by a computer in a highly constrained format. Currently, however, the entire message, including both its header and its text, is filtered.

2.1.2 Pattern Logging

Although a message is rejected if it contains any security sensitive pattern, the MFM matches and logs all patterns that occur in a message. The message is viewed as a sequence of characters C1, C2, ... on a tape, and the filter table is viewed as a sensor.

```
         -------------------------------------------------
MESSAGE  | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | . . .
         -------------------------------------------------

         ------------------
FILTER   | P1 | P2 | ... |
TABLE    ------------------
         | Q1 | Q2 | ... |
         ------------------
         | . . . .       |
         ------------------
```

The scan for patterns is started with the sensor positioned under character C1 at the beginning of the tape. The sequence of message characters beginning with C1 is compared against each pattern in the filter table. Any pattern that is matched is logged when it is discovered, and the message is marked for rejection. Then, the sensor is moved one character position to the right so that it is positioned under C2, and the process is repeated until the sensor reaches the end of the tape. The scan does not stop when the first match is discovered. It continues to find all matches in the message.

2.2 Input Messages

The messages received from OSIS are expected to be in a certain format. The MFM, however, also must be able to process ill-formed messages.

2.2.1 Well-formed Messages

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.

2.2.2 Ill-formed Messages

An ill-formed messages is one that contains more than 7200 characters. If such a message is received, it is filtered in 7200 character segments, and all of the segments are logged and rejected even if no pattern match is found. The MFM filter is applied independently to each segment, and therefore, patterns that cross segment boundaries are not detected. The segments are filtered independently because an ill-formed message may be arbitrarily long, and therefore, it may exceed the available storage capacity. The substring matched by a pattern also may be arbitrarily long if the pattern contains a star.

If OSIS goes down and restarts, the modulator also may receive a message of normal size but with one or more embedded "ZCZC" strings,

```
ZCZC . . . ZCZC . . . NNNN
```

The MFM has no special provisions to detect these messages. If desired, they can be detected and not passed to the destination by putting a "ZCZC" pattern in the filter table.

2.3 Output Messages

Every message that is received is either passed to the destination or logged as a security violation.

2.3.1 Messages Passed

If a message passes the security filter, it is passed to the destination exactly as it was received (including the opening "ZCZC" and the closing "NNNN"). Then, an additional sequence <CR><CR><LF> is sent to the destination. Messages that pass the filter are not logged.

2.3.2 Messages Rejected

When a pattern match is discovered, the matching substring is recorded on the logging device. Then, the whole message is sent to the logging device, and it is not passed to the destination. Ill-formed messages are logged in segments regardless of their content, and no segment is passed to the destination.

In order to display the contents of the rejected message precisely, the "!" is used to quote characters that normally would print as blanks or invoke various control functions on the logging device; the exclamation point also is quoted. For example, "!M" denotes a carriage return, "!J" denotes a line feed, "! " denotes a blank, and "!!" denotes an exclamation point. In displaying the matched substring, blanks are quoted, but they are not quoted in the display of the message. This displays precisely what substring was matched, but still keeps the message fairly readable.

The message logger also introduces new lines at the places suggested by the text of the message and also when the length of a line exceeds a pre-defined terminal width. The appearance of a new line on the log is never part of the message.

Examples of log output for rejected messages are shown below. The following is an example of rejecting a well-formed message:

```
Rejected Text ----------------

!M!M!Jrepresentation! ! !

! Limitations!

-LIMITATIONS.


Message ----------------------
ZCZC!M!M!J
  The  following  sections  present  a  brief  discussion of!M!M!J
representation   in   general,   a   description   of    the!M!M!J
representation system used, the procedural motivation behind!M!M!J
its  various  aspects,  and  the four major divisions of our!M!M!J
representation space. Limitations of our representation  are!M!M!J
discussed in SYSTEM-LIMITATIONS.!M!M!J
!M!M!J
NNNN
-----------------------------
```

The following is an example of logging an ill-formed message in segments:

```
Message Too Long --------------
Message Segment ---------------
ZCZC!M!M!J
3.1 Levels of representation!M!M!J
...
the adequacy of theories of n
-------------------------------


Rejected Text -----------------

!M!M!Jrepresentation.!M!M


Message Too Long --------------
Message Segment ---------------
f natural language processing. One!M!M!J
classic difficulty, for example, is the attempt  to  specify!M!M!J
...
we    describe    require    certain    properties    of    the!M!M!J
representation.!M!M!J
NNNN
-------------------------------
```

Chapter 3 FORMAL SPECIFICATION AND PROOF

All levels of the MFM were formally specified and proved using the Gypsy methodology [Good 78]. The specifications, describe what the program is supposed to do, and the implementation describes how it does it. Both the specifications and the implementation of the MFM were written in the Gypsy language. Then, the Gypsy Verification Environment (GVE) was used to construct mechanical proofs that the implementation satisfies its specifications for all possible input. This is very much different from conventional testing in which an implementation is demonstrated to perform correctly only on certain test cases.

The following sections outline the formal Gypsy specifications and implementation for the top levels of the MFM. These sections contain excerpts from the actual Gypsy text. The complete Gypsy text for the specifications and implementation of the MFM are in [Siebert 82a].

3.1 Formal Specification

3.1.1 Top Level Modulator

The top level Gypsy specification for the MFM is

```
procedure start_modulator (var source : a_source_buffer<input>;
                           var   sink : a_sink_buffer<output>;
                           var    log : a_log_buffer<output>;
                               filter : a_filter) =

begin
  block correct_modulation (infrom(source,myid),
                            outto(sink,myid),
                            outto(log,myid), filter);
  exit no_modulation (infrom(source,myid), outto(sink,myid),
                      outto(log,myid), filter);
end;
```

START_MODULATOR has access to exactly four external data objects. SOURCE is its input buffer; SINK is the buffer to its destination; LOG is the buffer to its audit trail. Each of these is a buffer of ASCII characters. FILTER is its filter table. START_MODULATOR may modify the value of SOURCE, SINK and LOG, but it may not modify the value of FILTER.

BLOCK CORRECT_MODULATION(...) states that the CORRECT_MODULATION relation must be satisfied among the streams of characters coming in from the SOURCE and going out to the SINK and LOG. This relation must be satisfied whenever START_MODULATOR is ready to send or receive any character from any of its three buffers. EXIT NO_MODULATION(...) states the relation that START_MODULATOR must satisfy when it terminates normally. (These specifications make no statement about what must be true if START_MODULATOR terminates abnormally by signalling a condition. However, if it does terminate abnormally, the BLOCK specification will still be true.)

CORRECT_MODULATION specifies one of two relations depending on the validity of the FILTER which is provided as a parameter. In order for the discriminated linear search to work properly, the filter table must have a valid form. But, because the table is prepared independently and provided to the MFM as a parameter, it is possible that it may be invalid.

```
function correct_modulation (source : a_char_seq;
                               sink : a_char_seq;
                                log : a_char_seq;
                             filter : a_filter)    :  boolean =

begin
  exit (assume
    result =
      if correct_filter (filter) then
        modulated_flow (source, sink, log, filter)
      else   source = null(a_char_seq)
           & sink = null(a_char_seq)
           & partial_string (log, bad_filter_report)
      fi);
end;
```

Within CORRECT_MODULATION, SOURCE refers to the entire sequence of characters received from the source buffer, SINK refers to the sequence of characters sent to the destination buffer, and LOG refers to the sequence of characters sent to the log buffer. In essence, CORRECT_MODULATION says that either FILTER is valid and the modulator modulates the flow of messages or FILTER is invalid and the modulator prints an error message on the LOG.

If FILTER is invalid, then nothing is received from SOURCE, nothing is sent to SINK, and an error message, BAD_FILTER_REPORT, is sent to LOG. PARTIAL_STRING allows the log to be unfinished with respect to the error message. However, whatever has been sent out must be some initial part of the expected full output.

```
function partial_string (s2,s1 : a_char_seq) : boolean =
begin
  exit (assume
    result = some i: integer,
               s2 = s1[1..i] & i in [0..size(s1)]);
end;
```

If the modulator terminates normally, the NO_MODULATION relation is satisfied.

```
function no_modulation (source : a_char_seq;
                          sink : a_char_seq;
                           log : a_char_seq;
                        filter : a_filter)    :  boolean =

begin
  exit (assume result = [  not correct_filter (filter)
                         & source = null(a_char_seq)
                         & sink = null(a_char_seq)
                         & log = bad_filter_report]);
end;
```

FILTER is invalid, no characters have been received from SOURCE, none have been sent to SINK, and the printing of the error message on the LOG has been completed.

3.1.2 Filter Validity

What it means for the filter to be valid is specified by CORRECT_FILTER.

```
function correct_filter (filter : a_filter) : boolean =
begin
  exit (assume result = [  is_ordered(filter)
                         & no_bad_chars (filter)
                         & no_star_dot (filter)]);
end;
```

IS_ORDERED(FILTER) means that FILTER is ordered on the first character of each pattern (a null pattern is treated as though it were "*"). IS_ORDERED aspect of filter correctness is involved in showing that the discriminated linear search satisfies the pattern matcher specification. IS_ORDERED is the property of FILTER that is required by the method used to search the pattern table. (See Section 4.2.2.) NO_BAD_CHARS(FILTER) says that all of the characters in patterns are upper case letters, digits, dots, or stars. NO_STAR_DOT(filter) states that the sequence "*." does not appear in any pattern.

3.1.3 Modulated Flow

If the filter is valid, the modulator must satisfy the MODULATED_FLOW specification. MODULATED_FLOW specifies the output of messages to the SINK and LOG in terms of input of messages from the SOURCE and the FILTER. Within MODULATED_FLOW, SEGS represents all of the completed messages or segments that have been received from the source. If the last source message is incomplete and the current segment is shorter than 7200 characters, it is excluded from SEGS.

```
function modulated_flow (source : a_char_seq;
                           sink : a_char_seq;
                            log : a_char_seq;
                         filter : a_filter)   :  boolean =
begin
  exit (assume
    result =
      some segs: a_msg_seq,
          segs = completed_segments (source_pack(source))
        & partial_string (sink, all_passed (segs, filter))
        & partial_string (log, all_audited (segs, filter)));
end;
```

PARTIAL_STRING allows the output to SINK and LOG to be unfinished with respect to all of the messages and segments received from the source. However, whatever has been sent out must be some initial part of the expected full output. The use of PARTIAL_STRING is necessary because the output is sent one character at a time.

3.1.4 Messages Passed

Output to the sink is specified to be all of the messages that pass the security filter. Each message is followed by two carriage returns and a line feed, <CR><CR><LF>.

```
function all_passed (in_msgs : a_msg_seq;
                     filter : a_filter)   :  a_char_seq =
begin
  exit (assume
    result =
      if in_msgs = null(a_msg_seq) then
        null(a_char_seq)
      else  all_passed (nonlast(in_msgs), filter)
          @ passed_msg (last(in_msgs),
                        last_at_eom (nonlast(in_msgs)),
                        filter)
      fi);
end;


function passed_msg (     m : a_msg;
                      at_bom : boolean;
                      filter : a_filter) : a_char_seq =
begin
  exit (assume
    result = if msg_filter (m, at_bom, filter) then
                 out_msg(m)
             else null(a_char_seq)
             fi);
end;


function out_msg (m : a_msg) : a_char_seq =
begin
  exit (assume result = m @ [seq: carriage_return,
                                  carriage_return,
                                  line_feed]);
end;
```

When a message is filtered, it is necessary to know if the previous message ended with the closing "NNNN" of the message (LAST_AT_EOM). Otherwise, it may not be possible to recognize that the current message is a segment and should be rejected.

3.1.5 Messages Rejected

Output to the log is specified by the following four specification functions.

```
function all_audited (in_msgs : a_msg_seq;
                      filter : a_filter)   :  a_char_seq =
begin
  exit (assume
    result = if in_msgs = null(a_msg_seq) then
                 null(a_char_seq)
             else  all_audited (nonlast(in_msgs), filter)
                 @ audit_report (last(in_msgs),
                                 last_at_eom(nonlast(in_msgs)),
                                 filter)
             fi);
end;
```

```
function audit_report (      m : a_msg;
                        at_bom : boolean;
                        filter : a_filter) : a_char_seq =
begin
  exit (assume
    result = filter_report (m, at_bom, filter)
               @ msg_audit (m, at_bom, filter));
end;


function filter_report (      m : a_msg;
                         at_bom : boolean;
                         filter : a_filter) : a_char_seq =
begin
  exit (assume
    result = if pattern_filter (m, at_bom, filter) then
               null(a_char_seq)
             else rejection_log (m, at_bom, filter, true)
             fi);
end;


function msg_audit (      m : a_msg;
                     at_bom : boolean;
                     filter : a_filter) : a_char_seq =
begin
  exit (assume
    result = if msg_filter (m, at_bom, filter) then
               null(a_char_seq)
             else msg_log (m, is_full_msg(m,at_bom))
             fi);
end;
```

FILTER_REPORT is rejected text (message substrings that match patterns in the filter table) if there is any. For any message that does not pass the security filter, MSG_AUDIT is mainly the complete message (along with formatting specifications that call for output like the examples in Section 2.3.2). There is no log output for messages that pass the security filter.

3.1.6 Message Filtering

The security filter specification is specified by the function MSG_FILTER.

```
function msg_filter (      m : a_msg;
                      at_bom : boolean;
                      filter : a_filter) : boolean =
begin
  exit (assume
    result = [  pattern_filter (m, at_bom, filter)
              & is_full_msg (m, at_bom)]);
end;
```

MSG_FILTER being true of a message means that the message passes the security filter. The pattern matcher specification PATTERN_FILTER is true when no substring of a message matches any pattern in the filter table. IS_FULL_MSG is true for whole messages, all messages except those that have been segmented because they are longer than 7200 characters.

The pattern matcher works by setting a pointer to the first character of a message and checking to see that none of the patterns in the filter match the message substring that begins with the first character. Then it advances the pointer by one character and checks to see that no pattern matches the substring beginning with the second character. (The opening "ZCZC" and the closing "NNNN" of a message are each considered to be one character by the pattern matcher.) The process of advancing the pointer by one and checking for pattern matches continues until the pointer falls off the end of the message. Specifications for these operations are given below.

```
function pattern_filter (     m : a_msg;
                         at_bom : boolean;
                         filter : a_filter) : boolean =
begin
  exit (assume
    result =
      if m = null(a_msg) then true
      else  no_matches (m, at_bom, filter)
          & pattern_filter (non_first(m,at_bom), false, filter)
      fi);
end;


function no_matches (     m : a_msg;
                     at_bom : boolean;
                     filter : a_filter) : boolean =
begin
  exit (assume
    result = all i: integer,
                 i in [1..size(filter)]
              ->
                 not match (filter[i], m, at_bom));
end;
```

AT_BOM (being true) means that the character sequence "ZCZC" on the beginning of message M should be interpreted as one delimiter. (The "ZCZC" denotes the beginning of a message.) Not AT_BOM (AT_BOM is FALSE) means that "ZCZC" on the beginning of M should be interpreted as four letters.

Specification of the actual pattern match is given by the function MATCH. MATCH is stated in terms of the pattern matching rules. It is independent of any specific patterns that could be placed in the filter table and of any particular input messages.

```
function match (      p : a_pattern;
                      m : a_msg;
                 at_bom : boolean)   :  boolean =
begin
  exit (assume
    result =
      if p = null(a_pattern) then true
      else if first(p) = '* then
             match (nonfirst(p),
                    no_leading_delimiters(m,at_bom), false)
           else if m = null(a_msg) then false
                else   first(p) = first(normal_form(m,at_bom))
                     & match (nonfirst(p),
                              non_first(m,at_bom), false)
      fi   fi   fi);
end;
```

NORMAL_FORM(M,AT_BOM) is a form of message M that permits a simple check for equality between the pattern character and the message character. In the normal form of M, lower case letters have been converted to upper case and delimiters have been converted to dots.

3.2 Implementation

The following sections give the top two levels of the MFM implementation along with their formal specifications. Every Gypsy program is implemented by a collection of procedures and functions.

3.2.1 Start Modulator

The highest level of MFM implementation is the Gypsy procedure START_MODULATOR. As specified, it checks the validity of FILTER and then either modulates messages or prints an error message.

```
procedure start_modulator (var source : a_source_buffer<input>;
                           var   sink : a_sink_buffer<output>;
                           var    log : a_log_buffer<output>;
                               filter : a_filter) =
begin
  block correct_modulation (infrom(source,myid), outto(sink,myid),
                            outto(log,myid), filter);
  exit no_modulation (infrom(source,myid), outto(sink,myid),
                      outto(log,myid), filter);
  var hp: an_int;
  if valid_filter (filter) then
    modulator (source, sink, log, filter)
  else new_lines (5, hp, log);
       send_line (bad_filter, no_quote_blank, hp, log)
  end;
end;
```

3.2.2 Modulator

START_MODULATOR is implemented in terms of several other procedures and functions. The major one is procedure MODULATOR. Its specifications state that it assumes that FILTER is valid, and then continues to maintain the MODULATED_FLOW property immediately before the sending and receiving of every character on the SOURCE, SINK and LOG buffers. The EXIT FALSE specifies that MODULATOR is not to terminate normally. (It may, however, terminate by signalling a condition.) The implementation of MODULATOR initializes the index tables for the discriminated linear search, and then proceeds to read characters from SOURCE and do the main work of the MFM.

```
procedure modulator (var source : a_source_buffer<input>;
                     var   sink : a_sink_buffer<output>;
                     var    log : a_log_buffer<output>;
                         filter : a_filter) =

begin
  entry correct_filter (filter);
  block modulated_flow (infrom(source,myid), outto(sink,myid),
                        outto(log,myid), filter);

  exit false;

  var ch: character;
  var m: a_msg;
  var at_bom, at_eom: boolean;
  var fc: a_filter_index;

  set_filter_index (fc, filter);
  at_bom := true; m := null(a_msg);
  loop
    assert   modulation (infrom(source,myid), outto(sink,myid),
                         outto(log,myid), filter, m, at_bom)
           & correct_filter_index (fc, filter)
           & msg_ok (m, at_bom) & size(m) < max_msg_size;
    receive ch from source;
    add_msg_char (m, ch, at_bom);
    at_eom := msg_completed (m, at_bom);
    if at_eom or size(m) ge max_msg_size then
      process_msg (m, at_bom, at_eom, sink, log, filter, fc);
      reset_msg (m, at_bom, at_eom);
    end;
  end;
end;
```

3.2.3 Process Msg

The main procedure of MODULATOR is PROCESS_MSG which is implemented as follows:

```
procedure process_msg (        m : a_msg;
                          at_bom : boolean;
                          at_eom : boolean;
                      var   sink : a_sink_buffer<output>;
                      var    log : a_log_buffer<output>;
                          filter : a_filter;
                              fc : a_filter_index) =

begin
  entry   correct_filter_index (fc, filter)
        & correct_msg (m, at_bom, at_eom);
  block   partial_string (outto(sink,myid), passed_msg(m,at_bom,filter))
        & partial_string (outto(log,myid), audit_report(m,at_bom,filter));
  exit   outto(sink,myid) = passed_msg (m, at_bom, filter)
       & outto(log,myid) = audit_report (m, at_bom, filter);
  var pass_it, full_msg: boolean;
  filter_msg (m, at_bom, at_eom, filter, fc, pass_it, log);
  full_msg := at_bom and at_eom;
  if pass_it and full_msg then send_msg (m, sink)
  else log_msg (m, full_msg, log)
  end;
end;
```

3.3 Proof

The proof of the MFM is a collection of proofs that each Gypsy procedure or function in the implementation of the MFM meets its specifications. Formal specifications and mechanical proofs have been constructed for every Gypsy function and procedure in the MFM.

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

The VC generator gave 304 verification conditions for the MFM. The VCs and 44 supporting lemmas were proved mechanically in the GVE. Of the 348 theorems, 150 were recognized to be true in the VC generator, and the remaining 198 were proved using the interactive theorem prover. The proof rests on a basis of four assumed lemmas. All of these lemmas state simple properties of the ASCII character set, properties that are clearly true but unknown to the GVE.

Details of the MFM proofs can be seen in [Siebert 82b] through [Siebert 82zi]. The basis lemmas and an index to the proof logs are given in [Siebert 82a] along with the Gypsy listing of the whole program. The general idea behind the proof is very simple: if each message is correctly received and processed, then all of the messages are correctly received and processed.

3.4 Testing

The MFM has been tested successfully in several configurations. The MFM was tested in isolation from OSIS at the University of Texas (UT) at various stages in its development. The MFM also was tested with OSIS at the Patuxent River Naval Air Test Center (PAX) with test scenarios developed by an independent contractor, Systems Development Corporation (SDC) [Neely 82]. The proved MFM passed all of the SDC tests correctly on the first attempt.

The tests at UT were done in the following two basic configurations. Most of the test data for these runs were text from the draft of a PhD thesis.

```
                                  -------------                       ---------------
 -------------                   |     MFM     |                     | Destination  |
| Source      |                  |(PDP 11/70   |                     | (Unix File)  |
|(Unix File)  |----------->      |   Unix)     |----------->         |              |
|             |                   -------------                       ---------------
 -------------                          |
                                        V
                                  -------------
                                 |     Log     |
                                 |(Unix File)  |
                                  -------------
 ----------------               ------------              ---------------
|    Source      |             |    MFM     |            | Destination   |
|(TOPS-20 File   |------------>| (LSI-11)   |----------->|    (CRT)      |
|   DEC 2060)    |             |            |            |               |
 ----------------               ------------              ---------------
                                     |
                                     V
                                ------------
                               |    Log     |
                               |(Hard Copy  |
                               |  LA-120)   |
                                ------------
```

The testing at PAX was done in the following configuration. OSIS was used as the source, and terminals were used for the destination and log. The MFM was not connected to any real destination system. The tests at PAX were the tests developed independently by SDC [Neely 82]. In this configuration, the proved MFM performed correctly in all of the SDC tests on the first attempt.

```
 ----------------               ------------              ---------------
|    Source      |             |    MFM     |            | Destination   |
|(OSIS           |------------>| (LSI-11)   |----------->| (TI Silent    |
|   PDP 11/70)   |             |            |            |     700)      |
 ----------------               ------------              ---------------
                                     |
                                     V
                                ------------
                               |    Log     |
                               |(Hard Copy  |
                               |  LA-36)    |
                                ------------
```

Chapter 4 PERFORMANCE

The installation of the MFM in a communication line between the source and destination systems will of necessity at least double the elapsed time that it takes for a message to flow from the one system to the other. However, if the filter is fast enough, the rate at which messages flow across a modulated connection will be the same as for direct (unmodulated) connection. The filtering rate will vary dramatically depending on the size and content of a particular filter and message. Performance tests have produced filter rates from 4.6 to 4417 baud. The 4.6 baud case is an extreme worst case that is very unlikely to be encountered in practice. It contains over 1.4 million pattern matches! The tests that represent the kinds of data that most likely will be encountered in practice range from 918 baud with a large filter to 2318 baud with a small one. Therefore, in many cases, we would expect that the filter is fast enough to support a 1200 baud data rate.

4.1 System Performance

4.1.1 Message Transfer Time

Without the MFM, the source is connected directly to its destination by an asynchronous communication line.

```
---------------              ---------------
|             |  Messages   |             |
|   Source    |------------>| Destination |
|             |             |             |
---------------              ---------------
```

A message is sent character by character from source to destination, and therefore, the amount of time it takes a message to flow from source to destination is just the time needed for its characters to move across the communication line.

The MFM, however, must filter an entire message before it can pass any part of it to the destination. To do this, the MFM first reads an entire message, then filters it, and then passes it to the destination. This requires moving the message first across an input line and then across an output line rather than across just one line as in the direct connection. Therefore, when the MFM is installed in the line, the message transfer time is twice that of the direct connection plus whatever time is needed to filter the message.

4.1.2 Message Delivery Rate

With adequate space for buffering messages and a sufficiently fast filter, the rate at which messages are delivered to the destination can be as fast as for a direct connection. (As shown in Section 4.2.2, the actual speed of the filter depends on the size and content of the filter and the messages being scanned.)

Internally, the MFM consists of four processes running concurrently.

```
-------------              -------------              -------------
|  Source   | ---------    |           |  ---------  |Destination|
|  Input    |-|c|...|c|->|   Filter   |-|c|...|c|->|  Output   |
|           | ---------    |           |  ---------  |           |
-------------   (7400)     -------------   (7400)    -------------
                                 |
                                ---
                                |c|
                                ---   (100)
                                |c|
                                ---
                                 |
                                 V
                           -------------
                           |    Log    |
                           |  Output   |
                           -------------
```

The numbers in parentheses are the maximum number of characters that can be in each buffer. With this structure, if most messages are passed to the destination, the rate at which messages flow through the MFM is determined by the processing rate of the slowest process. Thus, if the filtering process can keep up with the input and output processes, the message delivery rate is roughly the same for a modulated connection as for a direct connection.

4.2 Filter Performance

The dominant part of the filter performance is the pattern matching, and several steps have been taken to make it adequately efficient. The message is converted to a normal form which reduces the total number of comparisons needed, and a discriminated linear search is used to find matching patterns in the filter table.

4.2.1 Normal Form

The first step in pattern matching is to convert the message to a normal form which reduces the number of comparisons that are needed to detect a pattern match. The conversion is done as follows:

1. All lower case letters are converted to upper case.
2. The opening "ZCZC" string is replaced by a dot.
3. The closing "NNNN" string is replaced by a dot.
4. All delimiter characters are replaced with dots.

For example, the normal form of the message

```
        ZCZCThis is red.<CR><CR><LF>NNNN
```

is

```
        .THIS.IS.RED.....
```

Thus, a letter, a digit, or a dot in a pattern matches only that exact same character in the normal form, and a star is the only pattern character that is a special case.

4.2.2 Discriminated Linear Search

The dominant part of the filtering process is the pattern matching. Several approaches were studied both with respect to performance and to ease of verification. The results of the performance tests are summarized in Section 4.2.3.

Based on these studies, the pattern matcher was implemented as a discriminated linear search. The discriminated linear search requires that the table be ordered on the first character of the patterns. The MFM does not do this arrangement, but simply checks to see first characters of the patterns are in order. Star comes first, then dot, then 0 through 9 and finally A through Z.

A simple linear search of the filter table would compare the substring being matched to each of the patterns in the table. As an estimate of the number of comparisons, consider a table of N patterns each of length P, and a message of length M, and assume that the patterns do not contain stars. The filter table is applied at M-P+1 positions in scanning the message, and each position might require as many as N*P comparisons. Thus, a simple linear search could require as many as N*P*(M-P+1) comparisons, and if M is considerably greater than P, this is approximately N*P*M.

Rather than search the entire table, the discriminated linear searches only two parts of the table.

1. A linear search is made of the patterns that begin with the first character of the substring being matched.
2. A linear search also is made of the patterns that begin with a star.

For example, if the first character of the substring being matched is "A", the discriminated linear search does a linear search only of the "A" and the "*" parts of the filter table. (It should be noted that it is never necessary to begin a pattern with a star. Any pattern that begins with a star is equivalent to one that does not. However, for general robustness, patterns that begin with a star are allowed.)

To get an estimate of the number of comparisons required in the discriminated linear search, let us first assume that there are no patterns that begin with a star. With this simplification, the maximum number of comparisons at each position of the filter table is C*P where C is the number of patterns that begin with the same character as the substring being matched (rather than N*P where N is the total number of patterns). Notice that, in many cases, C may be zero, and the filter table can be moved immediately to its next scan position. The total number of comparisons for the entire message is

```
    C[1]*P + ... + C[M-P+1]*P
```

where C[i] is the number of patterns that begin with the first character of the substring to be matched in filter position i. If we let, L be the largest C[i], then an upper bound for the number of comparisons is L*P*(M-P+1), and if M is much larger than P, this is approximately L*P*M.

4.2.3 Testing

Several versions of the pattern matcher were written and evaluated for filtering speed. The results of these performance studies led to the selection of the discriminated linear search. Simplicity and ease of verification also favored the discriminated linear search (although none of the pattern matchers were ruled out on the grounds of difficulty of verification).

The first pattern matcher was a simple linear search on the unnormalized message, and it was used quite effectively in some of the early prototypes. This pattern matcher, however, required 51.5 minutes to filter a 7200 character message when the filter table contained 200 patterns. This was unacceptably slow, and therefore, the following pattern matchers also were built and tested:

1. Linear 1. This is an improved version of the simple linear search matcher. The improvement consisted mainly of converting the message to normal form and then searching the normal form for patterns.

2. Linear 2. This is a discriminated linear search very similar to the one used in the proved MFM.

3. Binary 1. This matcher is based on a modified binary search of the columns of the filter table. (The ordinary binary search must be modified because of the stars that can occur in patterns and the requirement for finding multiple matches.)

4. Binary 2. This is a second version of the binary search matcher that uses a fair amount of pre-computed indexing information. These pre-computed indices were expected to speed things up considerably, but as the results show, they did not.

The preceding pattern matchers were implemented and run on an LSI-11. The time required for filtering messages was measured using the following test cases:

1. Worst Case. This is a 7200 character message consisting of all X's except for the opening "ZCZC" and the closing "NNNN" sequence. The filter consists of 200 patterns. Each pattern has 25 characters all of which are X's and stars. This filter applied to this message produces about 1.4 million matches! Basically every pattern matches at every position in the message.

2. Best Case for Largest Message and Filter. This is a 7200 character message containing all O's except for the opening "ZCZC" and closing "NNNN" sequence. The filter table consists of 200 identical patterns. Each pattern is a string of 25 X's. In this case, it can be determined that there is no match by examining only the first character of every pattern. This is a best case for the largest messages and tables handled by the MFM.

3. Thesis Messages with Large Filter. For this test, 158 messages were selected from a draft of a PhD thesis. The lengths of the messages range from 74 to 7200 characters with an average of 537 characters. Filter times for the discriminated linear search (Linear 2) were measured for only seven of these messages (average length 3759 characters, length range 380 to 7200 characters). The filter contains 200 random words giving a total of 2 matches in the entire set of 158 messages.

4. Thesis Messages with Small Filter. This is the same set of 158 messages as in the preceding test. The filter, however, contains 7 random words that produce 130 matches.

The test results are summarized in the following tables (results for the proved MFM are given for comparison):

SUMMARY.

	Worst Case	Best Case, Large Msg and Filter	Thesis, Large Filter*	Thesis, Small Filter*
Linear 1	2.13 hr.	682.3 sec.	39.1 sec.	2.3 sec.
Linear 2	--	13.1 sec.	--	--
Binary 1	8.67 hr.	32.7 sec.	3.9 sec.	2.0 sec.
Binary 2	9.89 hr.	21.8 sec.	3.3 sec.	2.2 sec.
Proved MFM	4.35 hr.	16.3 sec.	5.9 sec.	2.3 sec.
Effective baud rate for Proved MFM	4.6	4417	918	2388

* Average time for 158 messages

THESIS MESSAGES, LARGE FILTER

	158 Msg Total	158 Msg Average	7 Msg Average	Longest Time
Linear 1	102.9 min.	39.1 sec.	293.2 sec.	612.5 sec.
Linear 2	--	--	29.3 sec.	57.6 sec.
Binary 1	10.3 min.	3.9 sec.	28.7 sec.	59.4 sec.
Binary 2	8.8 min.	3.3 sec.	24.4 sec.	50.5 sec.
Proved MFM	15.4 min.	5.9 sec.	42.0 sec.	82.4 sec.

THESIS MESSAGES, SMALL FILTER

	Total	Shortest	Longest	Average
Linear 1	6.17 min.	0.2 sec.	34.1 sec.	2.3 sec.
Binary 1	5.34 min.	0.2 sec.	29.1 sec.	2.0 sec.
Binary 2	5.82 min.	0.2 sec.	32.0 sec.	2.2 sec.
Proved MFM	5.92 min.	0.3 sec.	30.5 sec.	2.3 sec.

These tests are informative, but it is not possible to draw any strong conclusions about how the MFM will perform in practice. Let us consider the test cases one by one.

The worst case tests give an estimated upper bound on the running times. Unfortunately, these bounds are unacceptably large rather than acceptably small, so this test is not conclusive. We can only observe that the worst case is a really extreme one requiring matching about 1.4 million patterns of 25 characters each. Presumably, nothing even approaching this extreme will ever be encountered in practice.

The best case for large messages and filters is an important one. It gives the amount of time required to filter a message when all matches fail immediately. We would expect that many situations similar to this would arise in practice. For a large table, the time for the simple linear search (Linear 1) is too long, about 11 minutes. All of the other search methods filter the message in times that we would expect to be more than adequate. (It was in this test that the early pattern matcher used 51.5 minutes to filter the message.)

The tests on the thesis data were intended to be a simulation of real world data. In these tests, the simple linear search was adequate for most cases, but became uncomfortably slow for large messages and filter tables. The other search methods gave acceptable results throughout.

The discriminated linear search was chosen for the proved MFM pattern matcher. The best case test and the longest times observed on the thesis data indicate that the simple linear search based matcher is too slow for large messages and filter tables. All of the other search methods seem quite adequate for these cases. The discriminated linear search was preferred over a binary search because it seemed easier to prove.

The filter times for the proved MFM are somewhat longer than the times for the Linear 2 search because some inefficiencies were introduced into the Gypsy code for convenience in verification and because the Gypsy compiler has been modified. The Gypsy inefficiencies, while not strictly necessary, gave a cleaner proof for the pattern matcher. Compiler changes were made to generalize the translation of Gypsy features that had been treated as special cases. The filter times for the proved MFM remain considerably less than the anticipated input and output times, and so they are considered adequate.

Chapter 5
RUNNING THE MODULATOR

The following sections describe what currently is required to install and run the MFM.

5.1 Making a Filter

The filter table is a parameter to the main program of the modulator, and a table must be created prior to running the modulator. A filter table can be created with a Gypsy program that runs on a PDP 11/70 Unix system [Siebert 82zj]. To make the filter available as the appropriate parameter to the main modulator program, the Macro-11 output of the filter construction program is assembled by Macn11, and the filter object file is linked with the flow modulator object code to give a standard load file.

When the MFM begins running, the runtime support checks that the filter table is a well-formed Gypsy structure, and the MFM initialization checks that the table and patterns conform to expectations of the security filter (see Section 3.1.2). If any of these checks fail, the MFM will not run.

5.2 Instrumented Modulator

In addition to the proved MFM, an instrumented MFM that provides an additional monitoring and control console also has been built. Because of its increased monitoring and control facilities, it is recommended that the instrumented modulator be installed prior to installing the proved MFM.

```
                      ---------------          ---------------
----------           |Instrumented|          |               |
|        |           |    MFM      |          |               |
| Source |---------->|             |--------->| Destination   |
|        |           |             |          |               |
----------            ---------------          ---------------
                          |      |
                          |      V
                    -----------  ---------
                    | Console |  |  Log  |
                    -----------  ---------
```

5.2.1 Differences

The most obvious difference is that the instrumented MFM has a console and the proved MFM does not. The console performs various control functions and message and filter display. It also allows insertion and deletion of patterns in the filter table.

Another difference is in output to the rejection log. The instrumented MFM logs the filter table when it is first used and when it has been changed since the previous use. Also when a pattern is matched, both the pattern and the message text it matches are logged. In the proved MFM, only the offending substring is logged.

```
Rejection Patterns ------------

Pattern ...REPRESENTATION...
Matches !M!M!Jrepresentation! ! !

Pattern .LIMITATIONS.
Matches ! Limitations!

Pattern .LIMITATIONS.
Matches -LIMITATIONS.



Message ------------------------
ZCZC!M!M!J
  The  following  sections  present  a  brief  discussion of!M!M!J
representation   in   general,   a   description   of    the!M!M!J
representation system used, the procedural motivation behind!M!M!J
its  various  aspects,  and  the four major divisions of our!M!M!J
representation space. Limitations of our representation  are!M!M!J
discussed in SYSTEM-LIMITATIONS.!M!M!J
!M!M!J
NNNN
--------------------------------
```

The most important difference is that the proofs of the proved MFM are not valid for the instrumented modulator because of slight functional differences. The separation of the message stream into passed messages and rejected messages, however, is the same. The instrumented modulator also has an "operational" mode which actually runs the proved MFM. Once the the instrumented MFM is placed in operational mode, the proofs are valid and there is to no way to switch the MFM out of operational mode except to restart it.

Another noticeable difference between the two is that the buffers in the instrumented MFM are considerably smaller to accommodate the increased amount of code needed to drive the console. Therefore, the flow rates of messages through the two modulators may be different.

5.2.2 Console Commands

The following commands give information about the status of the modulator:

HELP	Gives a brief description of how to issue commands.
?	Gives the list of modulator commands.

WHAT ARE YOU DOING?	Gives the current modulator activity and also tells whether the automatic message display is on or off.

The following commands control the flow of messages through the modulator.

RUN	Causes the modulator to either begin modulating messages when the system is first brought up or to resume modulating messages after a PAUSE command.
PAUSE	Stops modulation immediately even if only part of a message has been received. Modulation can be resumed with the RUN command.
QUIT	Terminates the modulator immediately. It may have to be reloaded to restart.
OPERATIONAL MODE	Runs the operational (production) version of the modulator, which has no console or display functionality and restricted log output. If a message is being modulated when the command is issued, processing of the message is completed before switching to the operational modulator.

The following commands control the display option:

AUTOMATIC MESSAGE DISPLAY	The complete message is displayed on the console before it is filtered. Control characters are quoted with a preceding "!" so that they do not invoke terminal control functions (such as cursor control). After the message is filtered, a note indicating whether the message was passed or rejected is displayed.
NO AUTOMATIC MESSAGE DISPLAY	Nothing is displayed. No display gives faster modulator throughput.

The following commands are used for display independent of the automatic display:

MESSAGE	Displays the current message.
FILTER	Displays the rejection pattern table.

The modulator receives messages one character at a time, and it must receive a complete message before deciding to pass or reject the message. The automatic display of the message is done only after all of it is received. The MESSAGE command can be invoked at any stage in the processing of a message, even if only part of the message has been received.

The following commands are used for editing the table of rejection patterns used for filtering messages:

INSERT FILTER PATTERN	Allows a rejection pattern to be added to the filter table.
DELETE FILTER PATTERN	Allows a rejection pattern to be removed from the filter table.

5.3 Hardware

The MFM runs on a DEC LSI-11 microcomputer system. The following table shows the DEC modules required for the MFM. Those marked with the designator "(P)" are only required for the production MFM. Those marked with "(D)" are only required for the instrumented MFM. These modules plus miscellaneous cabling, terminals, etc, make up the digital processor on which the MFM will run.

KD11/HA	LSI-11/2 CPU
MXV11	(two required) 32kw RAM, 2 I/O ports
KEV11	EIS/FIS Extended Instruction Set ROM
BA11-VA	box
DLV11-J	4-port serial interface (D)
TU58-KB	Dectape 2 tape system (D)
-----	Boot ROM for MXV11
MRV11-A	PROM memory board (P)

The DLV11-J and the TU58 DECTAPE 2 tape system are necessary for the current MFM. The MFM program is loaded into the LSI-11 memory from the TU58 tape. A ROM chip plugs into the MXV11 board and provides a power-up boot program from the TU58. The extra DLV11-J is necessary to provide an interface to the TU58 and a terminal used as a debugging console.

The production version of the MFM will be burned into ROM.

5.4 Loading

Presently, the modulator program is loaded into the LSI-11 from a TU58 cartridge tape drive [tu58 79, Smith 82a]. Ultimately, it is intended that both the modulator and its filter table be burned into PROM [Smith 82b] and loading will not be required.

5.4.1 Input/Output Interfaces

OSIS communicates with the MFM over a standard two-way RS232 connection. The message output from the PDP 11/70 OSIS is through one port of a DH11 interface. Input to the LSI-11 MFM is through a DLV11 or one of the equivalent multi-purpose boards. The LSI-11 used at PAX has a DLV11-J four-port interface. Standard DEC cabling is used. (See Appendix A for a detailed description of the cabling used in the tests at PAX.)

The baud rates of the source and MFM interfaces must match. Due to the software limitations mentioned in Section 5.4.2, the data rate from the source cannot exceed 1200 baud. The baud rate of the DH11 can be set by software. The LSI-11 interfaces require restrapping - the DLV11 requires soldering jumpers; the DLV11-J and the MXV11-A require a wire wrap gun.

Several of the interfaces, in particular the Silent 700 terminals, require the Data Terminal Ready (DTR) signal to be high. This can be accomplished by strapping the DTR signal to the Clear-to-Send signal in the cable connected to the interface.

5.4.2 Input/Output Drivers

The MFM input/output drivers are capable of receiving input at 1200 baud and simultaneously sending it to three output ports at 9600 baud. (The MFM has only two output ports, but the input/output drivers have been tested with three.) Attempting to read at higher than 1200 baud results in lost characters. The input/output drivers and interrupt handlers are implemented in run-time support. They are coded in a straightforward way in BLISS-16, which is the normal target language of the Gypsy compiler. By de-modularizing the code and optimizing it in assembly language, it should be possible to obtain a 4800 baud input rate.

The MFM to source half of the connection is used only to control the data rate of the messages from the source using DEC's usual control-S/control-Q protocol. When the source receives a control-S character from the MFM, it immediately suspends output until it receives a control-Q character. The MFM will send no other data to the source.

In the same manner, the MFM recognizes the control-S/control-Q protocol from the console, log, and destination. For example, this allows the destination to be simulated by a LA-120 printer at 9600 baud. All other input from the log or destination is ignored by the MFM.

5.5 Military Standard Documentation

Military standard documentation [United States 78] for the MFM is given in [Siebert 82zk, Smith 82c, Smith 82d, Smith 82e, Siebert 82zl, Siebert 82zm]. These are the documents described in Data Item Descriptions DI-E-2136A (Program Performance Specification), DI-E-2135 (Interface Design Specification), DI-E-2138 (Program Design Specification), and DI-S-2139 (Program Description Document).

Chapter 6 DEVELOPMENT HISTORY

The use of formal specification and proof methods in the software development cycle is still in the very experimental stages. Therefore, the way in which the formal methods are integrated into the software cycle is often of considerable interest.

In the earliest stages, Gypsy was used to specify and prove the high level design of the modulator. Then, the development of the modulator went through a rather lengthy period of building and evaluating prototypes. These prototypes were implemented in Gypsy, and they were used to help define and stabilize the MFM specifications and to evaluate the performance of various pattern matchers. Finally, once the specifications stabilized and a pattern matching procedure was selected, the modulator was fully specified and mechanical proofs were constructed.

The following chronology summarizes the key events in the history of the MFM development. One of the main characteristics of the history is that it took a fairly long time to stabilize the MFM specifications. Another important characteristic is the use of a number of running prototypes to help stabilize the specifications and to evaluate the impact of design decisions on MFM performance. The resources used in the MFM development are summarized in the final section.

6.1 Chronology

Start Up, 11/3/80

The project was started with the general purpose of demonstrating the use of the Gypsy methodology in formally specifying and proving security software that moves classified information between systems of different security levels. No specific applications were identified at the outset. They were to be selected in consultation with the project sponsor.

Prototype General Modulator, 4/6/81

The original message flow modulator is what has become known as the "general modulator" (as opposed to the "OSIS modulator" which is the main subject of this report). The general modulator evolved from an initial assessment of the general requirements for software that moves classified information between systems of different security levels. In general, sometimes information must not be allowed to flow from one system to another, sometimes the information can be moved from one system to another if it is transformed in some way, and sometimes it can be moved without transformation.

The structure of the general modulator is similar to the OSIS modulator. The function of the general modulator, however, is more complex.

```
 ---------------              -----------               ---------------
|               |   Input    | General |  Output      |               |
|    Source     |----------->|   MFM   |------------->|  Destination  |
|               |  Messages  |         |  Messages    |               |
 ---------------              -----------               ---------------
                                ^|        |
                                |V        V
                           -----------  -----------
                           | Console |  |   Log   |
                           -----------  -----------
```

As in the OSIS modulator, the general modulator begins by applying a filter to a message. However, in the general modulator, if the message passes the filter, the message is transformed before it is passed to the destination.

```
Input             ----------- Pass  -----------            Output
Message ------->| Filter   |------->|Transform|------->Message
                 -----------         -----------
                      | Reject
                      V
                     Log
```

The general modulator also may contain several different pre-defined filters and transforms. Which pair is applied is selected dynamically from the operator console. The general modulator also has both an automatic and a manual mode of operation. In automatic mode, the selected filter and transform are applied continuously to the stream of incoming messages. In manual mode, the filter also may be a manual review of the message on the console, and the message also may be transformed manually (edited, sanitized) from the console. By appropriate selection of filters and transforms, the general modulator can perform a variety of security functions.

By April 6, 1981 a running prototype of the general modulator was built. To implement a running prototype, a simple message format and a number of simple filters and transforms were chosen primarily on the basis of ease of implementation. The prototype did not keep an audit trail. The top several levels of the prototype were formally specified and proved [Good 81a]. These top levels expressed the basic design of the modulator.

For ease of evaluation and demonstration, the prototype was built on ARPANET host UTEXAS-11 which is a PDP 11/70 running Unix. The prototype served several purposes.

1. It provided a running demonstration of the kind of thing that a general modulator could do.

2. It demonstrated that both message filtering and message transformation can be integrated smoothly into the same program so that the general modulator could be used for a wide variety of applications.

3. It demonstrated that both an automatic and manual mode of modulator operation could be supported conveniently in the same program.

4. It illustrated an effective and flexible implementation of the interaction between the modulator and its operator through a simple, yet powerful and flexible command scanner.

5. It was a basis for investigating the specification, implementation and proof of the basic design of a modulator.

OSIS Application Selected, 5/21/81

The prototype general modulator was built, demonstrated and reviewed prior to the selection of an actual target application. However, one of the primary goals of the modulator project was to develop and prove a system that could be used in actual operation.

On May 21, 1981, OSIS was selected as the general target for the modulator application. The OSIS system runs on a PDP 11/70 and the MFM is to run on a separate LSI-11. This determines the input message formats for the modulator, and a study of the hardware and software interface issues is begun.

It also was decided that the OSIS modulator would act primarily as a filter. The filtering would be done by a "rejection table" of about 100 key words of approximately no more than 20 characters each. An incoming message would be "compacted" by removing blanks and control characters, and then scanned for any occurrence of any word in the table. If any occurrence was found, the message would be rejected, but the entire message should be scanned for all words in the table.

It also was decided that the modulator should be tested and demonstrated on the OSIS system at the Patuxent River Naval Air Test Center (PAX). Work was initiated on acquiring the hardware required to run the modulator at PAX and on the logistics of developing software at the University of Texas and running it at PAX.

Flow Controller, 12/23/81

The MFM is similar in function to the LSI guard being developed by I. P. Sharp [Craigen 82]. The key difference is that the LSI guard is intended to review and sanitize messages manually. An attempt was made to design a high level flow controller system that could be refined into either the LSI guard or the MFM. A high level flow controller was specified, implemented and proved through its first level of refinement [Good 81b, Good 81c, Good 81d]. However, it was not possible to refine the flow controller as we had hoped, primarily because the specifications of both the LSI guard and the MFM had not yet stabilized. Therefore, the flow controller was abandoned.

The specifications and proofs of the flow controller did, however, serve a very useful purpose in developing the MFM. The proofs dealt primarily with moving characters across the I/O interfaces and grouping them into whole messages. Two different approaches were specified and proved. The cleanest approach for proof was one in which each I/O channel was served by a Gypsy process that actually transformed strings of characters into messages and vice versa. These driver processes ran concurrently with the main flow controller process which operated only on whole messages. The other approach had no concurrency and the main controller was connected directly to the I/O channels at the character level. This later approach requires a more complex proof, but it was adopted in the MFM anyway because it did not require concurrency. This decision was made because concurrency was not implemented in the Gypsy compiler, and we chose not to make the success of the MFM depend on a hurried implementation of concurrency.

Status, 1/23/82

[Good 82b] describes the status of the MFM on January 23, 1982. At this point, there is a running prototype general modulator, a development plan for the OSIS modulator and a high level flow controller that has been specified and proved (but not run). Also, a Unix prototype for the OSIS modulator is partially implemented. From this point on, the OSIS modulator will receive all of the remaining effort.

It has been decided that the OSIS modulator is to be strictly an automatic filter. The operator console will be used to start and stop the modulator, but except for that, it can be used only to control various kinds of displays of the message traffic. This restriction of the console is a considerable loss of generality over the general modulator, but it eliminates the possibility of operator error, and it permits a simpler specification and proof.

A line-based message format has been adopted. Each message begins with "VZCZC" and ends with "NNNN." Each message contains at most 100 lines. Each line has at most 69 characters of which the last three are <CR><CR><LF>.

Message filtering is to be composed of several stages. First, each message is to be checked for conformity with the preceding format. Second, checks for I/O transmission faults are to be made. Third, messages are packed into a normal form. All null, blank and digit characters are to be omitted, and all lower case is to be converted to upper case. The normal form then is to be compared to the table of rejection words. Each rejection word is defined by a simple string of characters.

An audit trail that records all messages arriving from OSIS and their disposition is planned. The audit trail is to be kept on a cassette tape on a TU58 tape drive. The amount of storage available on a single tape is a matter of some concern.

Evolution, 1/25/82

After reviewing the January 23 status report with the contract sponsor, it was decided that the modulator should be simplified even further to its present configuration [Good 82c]. The console would be eliminated altogether. It is to be retained only in the instrumented modulator for development and demonstration purposes. The audit trail simply would be recorded on a hard copy printer rather than the TU58 tape drive, and only rejected messages would be recorded. The message format is to be modified slightly to begin with a "ZCZC" instead of "VZCZC." It also is decided that a straight key word pattern match is too crude, and a simple pattern matcher similar to the final one is planned. It also is decided that the operational modulator should be burned into PROM. The idea was to make the entire modulator an automatic black box that could be plugged in between two different systems.

Status, 4/20/82

[Good 82d] summarizes the status of the modulator at this time. A running prototype of the instrumented OSIS modulator is complete. It has been run successfully on a Unix system and on an LSI-11 at The University of Texas. The top level of the modulator and the message reception components have been formally specified and proved.

First PAX Test, 5/4/82

The first test of the instrumented prototype coupled directly to OSIS was done at PAX on May 5, 1982 [Good 82e]. After getting the equipment connected properly, no problems were encountered with the MFM software.

Evolution, 5/4/82

As a result of an April 21-23 project review and the initial OSIS test at PAX, several modifications were made. The treatment of ill-formed messages was revised to its final form. It is decided that the operational modulator should log the offending substring that matches a pattern, but it should not display the actual pattern.

Also, at this stage, the prototype modulator contained a simple linear search pattern matcher, and the performance observed in the PAX test caused some concern over the speed of the modulator in some cases. Therefore, it is decided that a more thorough performance analysis is necessary.

Performance Analysis, 6/8/82

Several aspects of MFM performance were analyzed [Good 82f]. This included a simple mathematical analysis of message flow through the modulator, and it included implementing and timing of several approaches to a linear and a binary search based pattern matcher. (See Section 4.2.3.) The discriminated linear search was developed and tested after the simple linear and binary searches.

Second PAX Test, 6/11/82

The instrumented prototype MFM was run again at PAX [Smith 82f]. This time it contained an improved (but not the discriminated) linear search based pattern matcher, and it was run on the test scenarios developed by SDC [Neely 82]. After some considerable adjustment of the filter table and the format of the OSIS generated messages, the MFM passed all of the SDC tests. The adjustments did not include any modification to the MFM software. Only the contents of the filter table were modified.

At this time, the MFM requirements were declared to be frozen so that formal specification and verification could proceed. It was agreed that any of the pattern matching methods would give adequate performance in the normal case, and the choice was left to the contractor. The discriminated linear search was adopted because it gave the best combination of performance and verifiability.

Filter Making Program, 6/21/82

The filter table is prepared separately and provided to the MFM as a parameter to the main modulator procedure. To reduce the probability of error in this process, a Gypsy program for making filter tables was written [Siebert 82zj]. This program evolved rather simply from the instrumented prototype modulator, and the first running prototype was completed on June 21.

Status 6/29/82

The status of the MFM on June 29 is described in [Good 82g]. Because of evolving requirements, new specifications and proofs need to be written for the entire MFM. The previous specifications and proofs were useful guides, but they did not carry over to the final version.

Formal Specification and Proof, 8/23/82

The specifications and proofs of the MFM were completed on August 23. The verified MFM was modified only slightly from the prototype. The verified MFM was re-integrated into the instrumented MFM and both were running on August 28.

Final PAX Test, 9/3/82

The verified MFM was run with OSIS at PAX on September 1-3. It passed all of its tests on the first attempt.

6.2 Resources Used

The size of the MFM and estimates of the resources used in its development are shown below. The resource usage includes all of the activities described in the preceding chronology.

Lines of Gypsy specifications	1283	
Lines of Gypsy that produce executable code	556	
Words of LSI-11 code	3849	
Verification conditions	304	
Supporting lemmas	44	
VCS and lemmas requiring interactive theorem prover	198	
Lemmas assumed without proof	4	
Person-Months	13	
DEC 2060 CPU hours	220	
Page-months of file storage	45,000	
Lines of specified, proved, executable Gypsy per working day		1.94
Lines of specified, proved, executable Gypsy per DEC-2060 CPU hour		2.53

Of the thirteen months, approximately nine were used in defining MFM requirements. This includes building and testing the prototypes as well as formally specifying and proving parts of some of them. The remaining four months were devoted to performance studies and design refinement and to completion of formal specifications and proofs for the MFM.

Appendix A
MFM Cable Connections used at PAX

```
21-Sep-82 15:51:26-CDT,5878;000000000001
Mail-from: ARPANET site NRL-CSS rcvd at 21-Sep-82 1548-CDT
Date: 21 Sep 1982 16:33:04-EDT
From: moy at NRL-CSS (Gene Moy)
To: LSmith at Utexas
Subject: LSI and other information

...

                                        Elex 8141
                                        Ser 131
                                        17 September

Memorandum

Subject:  Message Flow Modulator Cable Connections
```

1. For the most part cable connections to the LSI-11 are straight forward. They follow the conventions for connecting terminals and/or modems to computers. For direct connection to the LSI-11, the sink and log terminals each requires a null modem. The source may or may not need null modem depending on the cable coming from the OBS PDP-11/70 (In our last attempt, we did have a null modem in the circuit). The console, by DEC's convention, does not require the null modem. The TU58 tape drive has its own special cable for connecting to the DLV11-J.

2. If a Texas Instruments Silent 700 series terminal is used, a special cable will need to be made. This terminal needs pins 4 (RTS) and 8 (CD) jumpered together before the terminal can communicate with the LSI11.

3. Details of the cable connections are shown in the diagram on the following page.

MESSAGE FLOW MODULATOR Cable Connections

```
 |
 |              ------------------
 |              | Final Filter: |
 |              |               |                   ------------
 |              |               |                   | Sink:    |
 |              |               [3]<--n-->[]<--s-->|          |
[1]<--a-->[]<--n-->[2]  LSI-11/03 |                 |  TI765   |
 |              |               |                   ------------
 |              |               |                   ------------
 |        ---->[6]              |                   | Log:     |
 |        |     |               [4]<--n-->[]<--a-->|          |
 |        t     |               |                   |  LA36    |
 |        |     |               |                   ------------
 |        |     --------[5]-------
 |   -----------         ^
 |   | Boot:   |         |
 |   |         |         c
 |   |  TU-58  |         |
 |   -----------         |
                         v
                   -------------
                   | Console:  |
                   | Tektronix |
                   |   4014    |
                   -------------
```

[1] DH11-AE from OBS PDP-11/70

[2] port 0 of 2nd DLV11-J

[3] port 1 of 1st DLV11-J

[4] port 2 of 1st DLV11-J

[5] port 3 of 1st DLV11-J

[6] port 0 of 1st DLV11-J

<a> male to male RS232 adapter cable

<c> BC21B, RS232 to DLV11-J cable

<n> BC20N/BC24C, RS232 to DLV11-J with a built in null modem

<s> modified male to male RS232 adapter cable
with pins 4 and 8 jumpered

<t> special cable for connecting TU58 and DLV11-J

* Note: wherever a BC20N and a male to male adapter is used together, they can be substituted for by a BC21B and a null

modem extension combination.

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 81a] Donald I. Good.
Message Flow Modulator Status Report.
Internal Note #15, Institute for Computing Science, The University of Texas at Austin, December 1981.

[Good 81b] Donald I. Good.
Flow Controller.
Internal Note #17, Institute for Computing Science, The University of Texas at Austin, December 1981.

[Good 81c] Donald I. Good.
Message Level IO Drivers.
Internal Note #18, Institute for Computing Science, The University of Texas at Austin, December 1981.

[Good 81d] Donald I. Good.
Flow Controller with Character IO Interface.
Internal Note #19, Institute for Computing Science, The University of Texas at Austin, December 1981.

[Good 82a] Donald I. Good.
The Proof of a Distributed System in Gypsy.
In *Proceedings of the 15th IBM/University of Newcastle upon Tyne Joint Seminar: Formal Specifications.* September, 1982.
Also Technical Report #30, Institute for Computing Science, The University of Texas at Austin.

[Good 82b] Donald I. Good.
Message Flow Modulator Status Report.
Internal Note #20, Institute for Computing Science, The University of Texas at Austin, January 1982.

[Good 82c] Donald I. Good.
OSIS Modulator Meeting - January 25, 1982.
Internal Note #23, Institute for Computing Science, The University of Texas at Austin, January 1982.

[Good 82d] Donald I. Good, Ann E. Siebert, Lawrence M. Smith.
OSIS Message Flow Modulator - Status Report.
Internal Note #36, Institute for Computing Science, The University of Texas at Austin, April 1982.

[Good 82e] Donald I. Good.
OSIS Modulator - Initial OSIS Test and Future Directions.
Internal Note #38, Institute for Computing Science, The University of Texas at Austin, May 1982.

[Good 82f] Donald I. Good, Ann E. Siebert.
MFM Performance.
Internal Note #39, Institute for Computing Science, The University of Texas at Austin, June 1982.

[Good 82g] Donald I. Good, Ann E. Siebert, Lawrence M. Smith.
OSIS Message Flow Modulator - Status Report.
Internal Note #36-A, Institute for Computing Science, The University of Texas at Austin, June 1982.

[Neely 82] Eileen Neely.
Security Test for OSIS Flow Modulator.
Technical Report TM-WD-8068/451/01, System Development Corporation, McLean, Virginia, May, 1982.

[Siebert 82a] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Gypsy Text and Proof Index.
Internal Note #90, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82b] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Modulator.
Internal Note #57, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82c] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Message Packing Lemmas.
Internal Note #58, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82d] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Process_Msg.
Internal Note #59, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82e] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Filter_Msg.
Internal Note #60, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82f] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Match_All_Patterns.
Internal Note #61, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82g] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Match_Pattern.
Internal Note #62, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82h] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Lemmas for Accelerated Linear Search.
Internal Note #63, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82i] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Procedure Remove_Leading_Delimiters.
Internal Note #64, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82j] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Procedure Next_Char.
Internal Note #65, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82k] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Normalize_Msg.
Internal Note #66, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82l] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Pattern Matcher Lemmas.
Internal Note #67, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82m] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Function Nondelimiter & Character Lemmas.
Internal Note #68, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82n] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Add_Msg_Char.
Internal Note #69, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82o] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Function Msg_Completed.
Internal Note #70, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82p] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Reset_Msg.
Internal Note #71, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82q] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Message Lemmas and Functions.
Internal Note #72, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82r] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Procedure Append_String.
Internal Note #73, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82s] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Function Seq_Eq_Subseq.
Internal Note #74, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82t] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Function Subseq_Select.
Internal Note #75, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82u] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Utility Lemmas and Functions.
Internal Note #76, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82v] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Send_Msg.
Internal Note #77, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82w] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Log_Pattern_Match.
Internal Note #78, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82x] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure New_Lines.
Internal Note #79, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82y] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Send_Char.
Internal Note #80, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82za] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Next_Char_Position.
Internal Note #81, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zb] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Log_Msg.
Internal Note #82, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zc] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Log_Rejection_Header.
Internal Note #83, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zd] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Send_Line.
Internal Note #84, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82ze] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Send_String.
Internal Note #85, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zf] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Procedure Log_Msg_Text.
Internal Note #86, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zg] Ann E. Siebert.
Message Flow Modulator - Proof Logs - Function Is_Msg_Line.
Internal Note #87, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zh] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Pattern Table Index.
Internal Note #88, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zi] Ann E. Siebert, Donald I. Good.
Message Flow Modulator - Proof Logs - Pattern Table Validity.
Internal Note #89, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zj] Ann E. Siebert.
Filter Constructor for Message Flow Modulator.
Internal Note #99, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zk] Ann E. Siebert, Lawrence M. Smith, Donald I. Good.
Message Flow Modulator - Program Performance Specification.
Internal Note #93, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zl] Ann E. Siebert, Lawrence M. Smith, Donald I. Good.
Message Flow Modulator - Program Design Specification.
Internal Note #97, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Siebert 82zm] Ann E. Siebert, Lawrence M. Smith, Donald I. Good.
Message Flow Modulator - Program Description Document.
Internal Note #98, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Smith 82a] Lawrence M. Smith.
Booting from the TU58 Tape Drive.
Internal Note #27, Institute for Computing Science, The University of Texas at Austin, March 1982.

[Smith 82b] Lawrence M. Smith.
Notes on Programming PROMS.
Internal Note #40, Institute for Computing Science, The University of Texas at Austin, June 1982.

[Smith 82c] Lawrence M. Smith.
OSIS to Message Flow Modulator - Interface Design Description.
Internal Note #94, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Smith 82d] Lawrence M. Smith.
Message Flow Modulator Output - Interface Design Description.
Internal Note #95, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Smith 82e] Lawrence M. Smith.
Message Flow Modulator to Logging Device - Interface Design Description.
Internal Note #96, Institute for Computing Science, The University of Texas at Austin, December 1982.

[Smith 82f] Lawrence M. Smith.
PAX Flow Modulator Demo, June 10-11, 1982.
Internal Note #41, Institute for Computing Science, The University of Texas at Austin, June 1982.

[tu58 79] *TU58 DECtape II Technical Manual*
Digital Equipment Corporation, 1979.
Order No. EK-0TU58-TM-001.

[United States 78]
United States Department of Defense.
Military Standard - Weapon System Software Development.
MIL-STD-1679 (Navy), 1 December 1978.