DOCUMENTATION OF CONCURRENT PROGRAMS

DEBORAH A. BOEHM-DAVIS
ANDREW M. FREGLY

Software Management Research & Ada Development
Data & Information Systems
General Electric Company
1755 Jefferson Davis Highway
Arlington, Virginia 22202

Copy available to DTIC does not permit fully legible reproduction

July 1983
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
# Documentation of Concurrent Programs

**Authors:** Deborah A. Boehm-Davis & Andrew M. Fregly

**Performing Organization:**
General Electric Company
1755 Jefferson Davis Hwy., Arlington, VA 22202

**Contract or Grant Number:**
N00014-79-C-0595

**Program Element, Project, Task, & Work Unit Numbers:**
61153N 42; RR04209; RR0420901; NR 196-160

**Report Date:**
July 1983

**Number of Pages:**
59

**Distribution Statement:**
Approved for public release; distribution unlimited.

**Supplementary Notes:**
Technical Monitor: Dr. John J. O'Hare

**Key Words:**
Software engineering, Software experiments, Modern programming practices, Software documentation, Petri nets, Resource diagrams, Program design languages (PDLs), Software human factors

**Abstract:**
Previous research on the effectiveness of documentation for sequential programs has suggested that the most effective documentation aids are those which provide clear control-flow information. The current research extends this work into the domain of concurrent processing programs to determine whether the documentation for these programs requires additional information regarding interprocess communications. In this research, programmer performance was examined on a modification task, where modifications were made.
to either the data structure or control flow of the program. Taken as a whole, the data suggest that the most appropriate type of documentation for concurrent processing may be different than the most appropriate type of documentation for strictly sequential processing. For modifications to concurrent processing programs, at least for simple programs and simple modifications, it is not crucial whether interprocess communications or control-flow information is highlighted in the documentation format. For more complex problems, it would appear that control-flow information is not necessary, and, in fact, may interfere with making the modification. These data are especially interesting at this time, when PDLs are becoming a de facto standard in the software industry. Further, they suggest that industry may be preparing to adopt, as a standard, a documentation format which will not necessarily provide them with the greatest possible benefit.
DOCUMENTATION OF CONCURRENT PROGRAMS

DEBORAH A. BOEHM-DAVIS
ANDREW M. FREGLY

Software Management Research & Ada Development
Data & Information Systems
General Electric Company
1755 Jefferson Davis Highway
Arlington, Virginia 22202

Submitted to:

Office of Naval Research
Engineering Psychology Group
Arlington, Virginia

Contract: N00014-79-C-0595
Work Unit: NR 196-160

JULY 1983

Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>METHOD</td>
<td>5</td>
</tr>
<tr>
<td>Materials</td>
<td>5</td>
</tr>
<tr>
<td>Problems</td>
<td>5</td>
</tr>
<tr>
<td>Modifications</td>
<td>5</td>
</tr>
<tr>
<td>Documentation Formats</td>
<td>5</td>
</tr>
<tr>
<td>Supplemental Materials</td>
<td>6</td>
</tr>
<tr>
<td>Design</td>
<td>6</td>
</tr>
<tr>
<td>Participants</td>
<td>7</td>
</tr>
<tr>
<td>Procedure</td>
<td>7</td>
</tr>
<tr>
<td>RESULTS</td>
<td>9</td>
</tr>
<tr>
<td>Modification Time</td>
<td>9</td>
</tr>
<tr>
<td>Errors</td>
<td>10</td>
</tr>
<tr>
<td>Preferences for Documentation Format</td>
<td>11</td>
</tr>
<tr>
<td>Experiential Factors</td>
<td>13</td>
</tr>
<tr>
<td>DISCUSSION</td>
<td>14</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>16</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>17</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>19</td>
</tr>
<tr>
<td>TECHNICAL REPORTS DISTRIBUTION LIST</td>
<td>56</td>
</tr>
</tbody>
</table>
INTRODUCTION

A complete software package always includes documentation. Although its importance is often overlooked, documentation may be the only source of program design information. Major tasks in the software life cycle, such as design, coding, testing and maintenance, are often performed by different individuals. Lientz and Swanson (1979) found that, typically, only about half of a software system's maintenance personnel had been involved in its development. Poor documentation techniques can, therefore, dramatically increase labor costs throughout the labor intensive software life cycle by making both development and maintenance tasks more difficult.

Recent research in this area (Boehm-Davis, Sheppard, & Bailey, 1982; Sheppard, Kruesi, & Bailey, in press; Sheppard, Kruesi, & Curtis, 1981) has been directed toward determining performance on a set of software tasks as a function of the type of documentation. In these studies, programmer performance was examined on comprehension, coding, debugging, and modification tasks as a function of the type of documentation provided. The documentation formats were constructed from the factorial combination of three types of symbology with three types of spatial arrangement. These formats were chosen because they represent the primary dimensions for categorizing the way in which available documentation aids configure the information they present to programmers (Jones, 1979). The three types of symbology in which information was presented consisted of normal English, abbreviated English (such as program design language), and ideograms. The spatial arrangements of the information used in these experiments were sequential, branching, and hierarchical. While each of the four tasks pursued in this research produced slightly different results, there was a general trend towards the superiority of succinct symbology and a branching spatial arrangement in each.

The current research extends the previous investigations on purely sequential programs into the domain of concurrent programming
by examining performance on a modification task. Concurrent processing refers to the simultaneous processing of two (or more) portions of the same program. Concurrent processing may be carried out by separate processors in a single computer, separate processors in several computers (distributed processing), or it may be simulated by time-sharing within one processor of a computer. The use of concurrent processing in a program presents a problem in representing those processes in the documentation. Most current documentation formats were designed for sequential program representation, and may not be suitable for the representation of parallel processing. It is especially important to represent this parallelism because, when a task is split into parallel parts, two or more of these paths may need to access the same resources. The documentation should, therefore, provide explicit information on the relationships between processes. If more than one process requires access to the same piece of information, protection of the data may be required to assure its integrity. Thus, programs using concurrent processing must be constructed and documented carefully to ensure orderly access to and sharing of resources.

The investigation of documentation for concurrent processing is especially important since this form of processing is generally considered to be more complex than strictly sequential processing and it is used extensively in embedded computer systems which can monitor and control a number of hardware interfaces simultaneously. Examples of embedded applications include systems for missile guidance, aircraft flight control, and multiplexing of communication channels. The current research will investigate the usefulness of different forms of documentation for this kind of processing.

The task chosen for this experiment was a modification task. Recent reports have asserted that almost 70% of costs associated with software are sustained after the product is delivered. These costs generally are spent in modifying the original program due to changing requirements and correcting errors, and these figures suggest that even small improvements in program maintainability
could be translated into substantial time and cost savings. For this reason, it is important to investigate modification performance.

Also, making a modification to an existing program requires several kinds of software skills: an understanding of how the program works; the ability to generate the code required to make changes; and the ability to debug these changes. Thus, it is important to study the modification task; it encompasses more general skills that are required for other software-related tasks.

The previous research suggested that the display of control flow was important in the documentation of sequential programs. While the display of control flow should remain important in documenting concurrent processing, it may be equally important to document the resource sharing among processes. The forms of documentation used in this experiment highlight these different types of information. While all of the documentation formats contain both control-flow and resource-sharing information, the two types of information are differentially emphasized. The first form of documentation is a standard program design language (PDL). The emphasis in PDLs is on the control flow rather than on the resource sharing of a program and the PDLs use abbreviated English in a sequential arrangement. The second form of documentation is a resource diagram, where the emphasis is on providing information about the sharing of resources rather than on control flow. Resource diagrams use abbreviated English in the communication circles and natural language in the process boxes; their spatial arrangement is most similar to the branching arrangement used in our earlier research. The third form of documentation combines both types of information by using Petri nets. Petri nets allow an equal emphasis on control flow and resource sharing. The nodes in the diagram show which resources are required for a task while the constrained language descriptions contain control-flow information. The Petri nets also use a spatial arrangement most similar to our branching arrangement.

The structure of the problem solutions was also manipulated in this research. Different design methodologies currently in use take
different approaches to structuring programs. While some methodologies tend to focus on data structures in decomposing problems, others focus on functional decomposition. This may have an impact on the effectiveness of different documentation formats. The research described here examined the effectiveness of different documentation formats using problems which were structured to represent solutions which might be produced by commonly-used design methodologies.
METHOD

Materials

Problems. Three experimental problems and one practice problem were created for use in this experiment. The experimental problems were a message distribution system, an air traffic display, and a text search problem. The practice problem was a message encryption system. The algorithms used to solve the problems were chosen such that they each represented approximately the same overall level of control-flow complexity (as indicated by the McCabe (1976) metric). Each problem was coded in three ways. One version coded the problem such that it had a complex data structure and a simple control flow; one version coded the problem such that it had a simple data structure and a complex control flow; and for one version, the data structure and control flow each carried an intermediate level of complexity.

Modifications. Two modifications were constructed for each problem. One involved a change in the data structure of the problem; the other involved a change in the control flow of the problem. For example, the data-structure modification for the message distribution program (shown in the appendix) required the programmers to change the length of the message. The control-flow modification for the same problem required programmers to change the algorithm so that when a message was entered with a particular message code, all of the readers would receive the message.

Documentation formats. Three documentation formats were created for use in this experiment: Petri nets, resource diagrams, and PDLs. Examples of each of these forms of documentation are shown for all of the problems in the appendix. In the Petri nets (based on ideas in Peterson, 1981), each large box represents a process in the system. The circles represent conditions which must be satisfied before processing can continue. Information listed on the lines between circles represent actions that are being carried out or information that is being passed between processes. In the
resource diagrams (based on ideas in Shaw, 1974), the boxes represent processes. The circles represent information which is being passed between processes, and the arrows indicate the direction in which information is being passed. The PDLs use standard notation, except for the use of "send" and "accept" which were the terms used to represent the passing and receiving of communications between and from processes.

Supplemental Materials. Each program was accompanied by four supplemental materials: a program overview, a data dictionary, a program listing, and a listing of the expected output from the program. The program overview contained the requirements, a general description of the program design, and the modification to be performed for each program. The data dictionary contained the variable names, an English description of the variables, and the data type for each variable. The program listing was a paper printout of the FORTRAN code which was identical to the code presented on the CRT screen. The listing of the expected output provided the programmers with the output expected from a correct run of the program; this allowed them to determine where they had gone wrong if their modification to the program did not run correctly.

Design

The experimental design used in this experiment was a 3x3x3x2 split-plot partially confounded design (based on Davies, 1956; Winer, 1971). The within-subject factors were type of documentation (Petri net, resource diagram, PDL), problem (text search, air traffic display, message distribution), and problem structure (complex data structure, complex control flow, intermediate). Type of modification (data structure, control flow) was a between-subjects variable. Each programmer modified three of the twenty-seven possible combinations of documentation, problem, and problem structure; each programmer made three modifications of the same type. For example, a programmer might modify the data-structure version of the text search program using a Petri net, the control-flow version of the air traffic display program using a resource
diagram, and the intermediate version of the message distribution program using a PDL. The order in which the programmers were observed under each treatment condition was randomized independently for each programmer.

Participants

The participants in this experiment were 72 professional programmers from four different locations. All were General Electric Company employees. The programmers averaged 8.4 years of programming experience and were familiar with an average of 5.7 programming languages. All of the programmers had previous experience with FORTRAN.

Procedure

Prior to the experiment, the participants were given a one-hour training session in which they were shown examples of each type of documentation format. The experimenter also described the procedure for using the text editor to modify the programs during this session.

Experimental sessions were conducted at CRT terminals on a VAX 11/780. Each participant modified all three of the programs, which were written in FORTRAN-77, using only one of the documentation formats for each. The participants were first asked to enter the changes from the practice problem which was used during the training session to familiarize them with the operation of the experimental system and its editor. Following the practice program, the three experimental programs were presented.

For each program, the participants were asked to first indicate, on the documentation format, the locations in the program where changes needed to be made and then to actually make the modifications using the editor. An interactive data collection system prompted the participants throughout the session. The system recorded each call for an editor command (e.g. ADD, CHANGE, LIST, or DELETE). From these, the overall time to modify and debug the
programs was calculated by summing the times from the individual editing sessions; the number of errors made was also calculated. The time required for compiling, linking, and executing the programs was not included in these measures. The programmers were required to continue working on a program until it was completed successfully. The programmers were allowed to take breaks between programs.

Following the experiment, the programmers completed a questionnaire about their previous programming experience. The information requested included number of years of experience and number of programming languages known. The participants were also asked to choose which documentation format they liked most and least, and to rate how much they relied on each documentation format.
RESULTS

Modification Time

The participants required an average of 23 minutes to modify each program. This represents the amount of time studying the program, deciding on the appropriate changes to make the modification, and using the text editor (i.e., the total time spent at the terminal less the time for compiling linking, and executing the program).

<table>
<thead>
<tr>
<th>MODIFICATION</th>
<th>PROBLEM</th>
<th>DOCUMENTATION FORMAT</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RESOURCE</td>
<td>PDL</td>
</tr>
<tr>
<td>CONTROL FLOW</td>
<td>MESSAGE DISTRIBUTION</td>
<td>19.8</td>
<td>22.1</td>
</tr>
<tr>
<td></td>
<td>AIR TRAFFIC</td>
<td>21.3</td>
<td>25.3</td>
</tr>
<tr>
<td></td>
<td>TEXT SEARCH</td>
<td>28.9</td>
<td>30.1</td>
</tr>
<tr>
<td>DATA STRUCTURE</td>
<td>MESSAGE DISTRIBUTION</td>
<td>13.0</td>
<td>12.2</td>
</tr>
<tr>
<td></td>
<td>AIR TRAFFIC</td>
<td>21.0</td>
<td>23.3</td>
</tr>
<tr>
<td></td>
<td>TEXT SEARCH</td>
<td>20.9</td>
<td>22.8</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>20.9</td>
<td>22.7</td>
</tr>
</tbody>
</table>

Table 1. Mean Time to Complete Modification Task (in Minutes)

Table 1 shows the mean times for each combination of documentation format, program, and type of modification. An analysis of variance showed that, overall, it took programmers less time to make a data-structure modification (21 minutes) than it did to make a control-flow modification (26 minutes) ($F(2,64) = 12.64, p < .001$). This analysis also showed that, overall, resource diagrams required the least amount of time (21 minutes), PDLs required an intermediate amount of time (23 minutes), and Petri nets required the greatest amount of time (26 minutes) ($F(2,95) = 7.31, p < .001$). A significant interaction was also found between problem and documentation format ($F(4,95) = 2.74, p < .05$). An examination of the data suggests that for the message distribution and air traffic display...
problems, there were no significant differences in modification
times for resource diagrams versus PDLs or for PDLs versus Petri
nets. There does appear to be a significant difference between
resource diagrams and Petri nets for both problems, however. For
the text search problem, the differences between pairs of
documentation formats all appear to be significant.

There were also large differences in the amount of time required
to modify the programs (control flow and data structure). The
message distribution program required the least amount of time to
modify (17 minutes), the air traffic display program required an
intermediate amount of time (24 minutes), and the text search
program required the greatest amount of time (29 minutes). The
analysis of variance supported this conclusion ($F(2,95) = 32.30,
p < .001$). This pattern of results mirrors the complexity ratings of
the programs, as measured by the McCabe metric. While the programs
were chosen to be roughly equal in overall complexity, there were
some differences among their ratings, which followed the pattern of
the time data; the message distribution program had an overall
complexity rating of 14, the air traffic display program had an
average complexity rating of 15, and the text search program had an
average complexity rating of 23.

There was no effect of the structure of the programs (simple
control-flow with a complex data structure, intermediate control
flow and data structure, or simple data-structure with complex
control-flow) on modification time ($F(2,95) < 1$), and it did not
interact with any of the other variables.

Errors

For programs that did not compile or run successfully on the
first submission, the programmers' editing activities for subsequent
submissions were analyzed to determine the number of errors. Table 2
shows the mean number of errors for each combination of documenta-
tion format and type of modification. The number of errors was low;
in addition, the majority of the errors (63%) were syntax errors.
rather than semantic errors. (For this analysis, misspellings of variable names, starting a line in the wrong column, and other such errors were categorized as syntax errors.) Due to the low number of semantic errors, no further analysis of these data was carried out.

<table>
<thead>
<tr>
<th>MODIFICATION</th>
<th>PROBLEM</th>
<th>DOCUMENTATION FORMAT</th>
<th>RESOURCE</th>
<th>PDL</th>
<th>PETRI</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL FLOW</td>
<td>MESSAGE DISTRIBUTION</td>
<td>.8 .9 .7 .8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AIR TRAFFIC</td>
<td>1.2 1.3 .8 1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TEXT SEARCH</td>
<td>1.1 1.4 1.7 1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA STRUCTURE</td>
<td>MESSAGE DISTRIBUTION</td>
<td>.1 0 .1 .1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AIR TRAFFIC</td>
<td>.4 1.1 .6 .7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TEXT SEARCH</td>
<td>.4 .7 .6 .6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>.7 .9 .8 .8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Mean Number of Errors

Preferences for Documentation Format

Across the three problems, the programmers received each type of documentation format. On the questionnaire, they were asked to state which documentation format was easiest to use and which was hardest to use. They were also asked to rate how much they relied on each version of documentation format on a seven-point scale (from 0 = not at all to 6 = constantly throughout). Tables 3 and 4 show the number of people choosing each documentation format as easiest or hardest to use as a function of type of modification made. In the control-flow group, two programmers failed to indicate which format had been easiest to use; a third programmer failed to indicate which format had been hardest to use. Overall, seventy-one percent of the programmers chose the PDL format as the easiest to use; 18% chose the Petri net; and 14% chose the resource diagram. The programmers were also asked if they had previously used any of the documentation formats. Eighty-three percent of the programmers making a control-flow modification indicated that they had
previously used a PDL; only 53% of the programmers making a data-structure modification had previously used a PDL. Three of the programmers indicated that they had previously used a form of resource diagram; four of the programmers had previously used a form of Petri net. Table 5 shows the mean rating of how much they relied on documentation format for each type of modification. For both types of modifications, the programmers stated they relied most heavily on the PDLs, and less so on the resource diagrams and Petri nets.

<table>
<thead>
<tr>
<th>MODIFICATION</th>
<th>DOCUMENTATION FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCE</td>
</tr>
<tr>
<td>CONTROL FLOW</td>
<td>5</td>
</tr>
<tr>
<td>DATA STRUCTURE</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3. Number of Times Documentation Chosen as Easiest to Use

<table>
<thead>
<tr>
<th>MODIFICATION</th>
<th>DOCUMENTATION FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCE</td>
</tr>
<tr>
<td>CONTROL FLOW</td>
<td>11</td>
</tr>
<tr>
<td>DATA STRUCTURE</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 4. Number of Times Documentation Chosen as Hardest to Use

<table>
<thead>
<tr>
<th>MODIFICATION</th>
<th>DOCUMENTATION FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCE</td>
</tr>
<tr>
<td>CONTROL FLOW</td>
<td>2.4</td>
</tr>
<tr>
<td>DATA STRUCTURE</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 5. Mean Ratings of Reliance Upon Each Documentation
Experiential Factors

The participants were asked the number of years they had been programming and the number of programming languages they knew. No correlation was found between years of programming experience and modification time. A low negative correlation ($r = -0.23, p < .05$) was found between number of programming languages known and modification time.
DISCUSSION

Substantial differences in completion time were observed among the three types of documentation formats. For both kinds of modification (control flow or data structure), the resource diagrams led to the best performance while Petri nets led to the poorest performance. This suggested that, unlike sequential processes where control-flow information was required, concurrent processing requires information about interprocess communications. Because data structures are often used to pass information between processes, the resource diagrams, which highlight information about communications between processes, also highlight data structures. Both kinds of modifications required locating the particular data structures that needed to be changed; this probably accounts for the fact that it was easier to locate and make modifications when resource diagrams were used. Two things should be noted, though. First, the data suggest that the differences among documentation formats are not very pronounced for all cases; the text search program provided the most striking differences. Second, the modifications used in this experiment were simple and did not require many control-flow changes; this will not always be the case with modifications. This suggests that, at least for simple programs and simple modifications, it is not crucial whether interprocess communications or control-flow information is highlighted in the documentation format. For more complex problems, the longer times required by the Petri nets and PDLs suggest that when modifications are made, detailed control-flow information is not necessary, and, in fact, may interfere with making the modification.

Differences were also observed among the three problem types used in this experiment. The message distribution problem was associated with the shortest times, the text search problem resulted in the longest times, and the air traffic display problem was in-between. This result parallels our past experiences in finding differences across problems. While the programs were roughly equated in terms of a common measure of complexity, they did have
slightly different complexity ratings, as measured by the McCabe metric. The amount of time required to make modifications was found to be longer for the problems with a higher complexity metric, suggesting that control-flow complexity may indeed provide a good measure of psychological complexity.

Diversity of experience, in terms of the number of languages used, was a better predictor of performance than years of experience. This result replicates results from our earlier research (Sheppard, Kruesi, & Bailey, in press; Sheppard, Kruesi, & Curtis, 1981; Sheppard, Milliman, & Curtis, 1979) and highlights the importance of ensuring that programmers have an opportunity to gain broad applications experience as part of their professional development.

The participants' choices for the easiest to use documentation format and their previous familiarity with one of the documentation formats lead to an interesting observation. Although, overall, 68% of the programmers had used PDLs before this experiment and 71% of them chose it as the easiest to use, the time required to make the modifications with the PDLs was in between the other documentation formats, for the two types of task modification.

Taken as a whole, the data suggest that the most appropriate type of documentation for concurrent processing (resource diagram) is different than the most appropriate type of documentation for strictly sequential processing (PDL). For modifications to concurrent processing programs, at least for simple programs and simple modifications, it is not crucial whether interprocess communications or control-flow information is highlighted in the documentation format. For more complex problems, it would appear that detailed control-flow information is not necessary, and, in fact, may interfere with making the modification. These data are especially interesting at this time, when PDLs are becoming a de facto standard in the software industry. Further, they suggest that industry may be preparing to adopt, as a standard, a documentation format which will not necessarily provide them with the greatest possible benefit.
ACKNOWLEDGEMENTS

The authors would like to thank Sue Hannan of GE in Lanham, Ron Boehmke of GE in Valley Forge, Bill Carrens and Jackie Pickard of GE in Springfield, and Roger Collins, John Ivers, Dave Morris, and Lou Oliver of GE in Arlington for providing participants and facilities; Dr. John O'Hare for advice; Don Wittig for graphic support in preparing the documentation formats, and Tom McDonald for preparing the supplemental materials and statistical analyses.
REFERENCES


APPENDIX - DOCUMENTATION FORMATS

RESOURCE DIAGRAMS

PROGRAM DESIGN LANGUAGES (PDLs)

PETRI NETS
PROGRAM DESIGN LANGUAGES (PDLs)
program EXAMPLE

declare
ID1_CEID0:10,2_CEID2: "COMMUNICATION_FLAG"

test PROCESS_1
declare
IN_LINE STRING; 60;
/ - INTEGER

begin
do forever
read (IN_LINE) from terminal
if (end of file read) then exit do
end if
for I = 1 to (location of last non-blank character in IN_LINE)
SET_F I/O(1,1)
WAIT_FLG(SEND,1)
CLEAR_FLG(SEND,1)
send (IN_LINE[1:I]) to MONITOR
end do
SET_F I/O(1,1)
when PROCESS_1 terminates, the end of file this generates will notify
MONITOR that PROCESS_1 is terminating.

and PROCESS_1

test PROCESS_2
declare
IN_LINE STRING; 60;
/ - INTEGER

begin
do forever
read (IN_LINE) from terminal
if (end of file read) then exit do
end if
for I = 1 to (location of last non-blank character in IN_LINE)
SET_F I/O(1,1)
WAIT_FLG(SEND,2)
CLEAR_FLG(SEND,2)
send (IN_LINE[1:I]) to MONITOR
end do
SET_F I/O(1,1)
when PROCESS_2 terminates, the end of file this generates will notify
MONITOR that PROCESS_2 is terminating.

and PROCESS_2

test MONITOR
declare
ONE TDWO 10 INTEGER
ID1_READ TD2_READ FLAG_TRUE
PROC_1_ALIVE PROC_2_ALIVE LOGICAL = TRUE
IN_CHAR CHARACTER

begin
prompt (Operator to continue) to terminal
CREATE(PROCESS_1)
CREATE(PROCESS_2)
do while (PROC_1_ALIVE or PROC_2_ALIVE)
WAIT_OK_OF_FLAGS

READ_FLG(10,1) TEN_1 READ:
if (ID1 READY) then
CLEAR_FLG(10,1)
SET_FLG(SEND,1)
accept (IN_CHAR) from PROCESS_1
if (data ok) then
write (IN_CHAR) to terminal

else
PROC_1_ALIVE = false
CLEAR_FLG(SEND,1)
end if
end if

READ_FLG(10,2) TEN_2 READ:
if (ID2 READY) then
CLEAR_FLG(10,2)
SET_FLG(SEND,1)
accept (IN_CHAR) from PROCESS_2
if (data ok) then
write (IN_CHAR) to terminal
else
PROC_2_ALIVE = false
CLEAR_FLG(SEND,2)
end if
end if
end do
end MONITOR

begin
start MONITOR
end
program MESSAGE DISTRIBUTION
declare
  type SIGNAL is (NEW_MESSAGE, READ_MESSAGE, MESSAGE_EXEC)
  NEW_MESSAGE_FLAG, NEW_MESSAGE_FLAG_2, STOP_EVT, COMMUNICATION_FLAG
  NEWS_CODE STRING(16)
  MESSAGE STRING(16)

task MESSAGE PRODUCER
declare
  EXEC_ID integer
  NEWS_CODE STRING(16)
  MESSAGE STRING(16)
begin
  CREATE(NEWS_CODE, EXEC_ID)
end

begin
  IF (NEWS_CODE = NEW_MESSAGE) THEN
    MESSAGE = CREATE(MESSAGE, EXEC_ID)
    BEGIN
      IF (MESSAGE < NEW_MESSAGE) THEN
        IF (MESSAGE < NEW_MESSAGE_FLAG) THEN
          SET_PROC(NEWS_CODE, EXEC_ID)
          IF (MESSAGE < NEW_MESSAGE_FLAG_2) THEN
            IF (MESSAGE < NEW_MESSAGE_FLAG_2) THEN
              WRITE(MESSAGE, EXEC_ID)
              READ_EVT_TRUE(MESSAGE, EXEC_ID)
            END IF
          END IF
        ELSE
          WRITE(MESSAGE, EXEC_ID)
          READ_EVT_TRUE(MESSAGE, EXEC_ID)
        END IF
      ELSE
        WRITE(MESSAGE, EXEC_ID)
        READ_EVT_TRUE(MESSAGE, EXEC_ID)
      END IF
    END BEGIN
  ELSE
    WRITE(MESSAGE, EXEC_ID)
    READ_EVT_TRUE(MESSAGE, EXEC_ID)
  END IF
end MESSAGE PRODUCER

task MESSAGE EXEC
declare
  REQUEST SIGNAL
  CURRENT_FLAG, COMMUNICATION_FLAG, NEW_MESSAGE_FLAG STRING(16)
  MESSAGE STRING(16)
begin
  IF (REQUEST = NEW_MESSAGE) THEN
    IF (CURRENT_FLAG = NEW_MESSAGE_FLAG) THEN
      IF (CURRENT_FLAG = NEW_MESSAGE_FLAG_2) THEN
        IF (CURRENT_FLAG = NEW_MESSAGE_FLAG_2) THEN
          WRITE(MESSAGE, EXEC_ID)
          READ_EVT_TRUE(MESSAGE, EXEC_ID)
        END IF
      ELSE
        WRITE(MESSAGE, EXEC_ID)
        READ_EVT_TRUE(MESSAGE, EXEC_ID)
      END IF
    ELSE
      WRITE(MESSAGE, EXEC_ID)
      READ_EVT_TRUE(MESSAGE, EXEC_ID)
    END IF
  ELSE
    WRITE(MESSAGE, EXEC_ID)
    READ_EVT_TRUE(MESSAGE, EXEC_ID)
  END IF
end MESSAGE EXEC

task MESSAGE READER
declare
  CURRENT_FLAG, COMMUNICATION_FLAG STRING(16)
  TERMINATED_FLAG STATUS
  READER_CODE, NEWS_CODE STRING(16)
  MESSAGE STRING(16)
begin
  WHILE TRUE DO
    IF (CURRENT_FLAG = NEW_MESSAGE_FLAG) THEN
      IF (CURRENT_FLAG = NEW_MESSAGE_FLAG_2) THEN
        IF (CURRENT_FLAG = NEW_MESSAGE_FLAG_2) THEN
          WRITE(MESSAGE, EXEC_ID)
          READ_EVT_TRUE(MESSAGE, EXEC_ID)
        END IF
      ELSE
        WRITE(MESSAGE, EXEC_ID)
        READ_EVT_TRUE(MESSAGE, EXEC_ID)
      END IF
    ELSE
      WRITE(MESSAGE, EXEC_ID)
      READ_EVT_TRUE(MESSAGE, EXEC_ID)
    END IF
  END WHILE
end MESSAGE READER
end MESSAGE DISTRIBUTION

MESSAGE DISTRIBUTION (C)
program MESSAGE_DISTRIBUTION
declare
  type SIGNAL is (NEW_MESSAGE, READ_MESSAGE, NEW_RECEIVER);

task MESSAGE_PRODUCER
declare
  EXEC_ID INTEGER;
  MSG_CODE STRING(5);
  MESSAGE STRING(10); begin
  create(MESSAGE_EXEC, exec_id);
  do while (not stepped by operator)
    prompt (operator for MSG_CODE) to terminal;
    prompt (operator for MESSAGE) to terminal;
    send (NEW_MESSAGE) to MESSAGE_EXEC;
    send (MSG_CODE, MESSAGE) to MESSAGE_EXEC;
  end do;
end task MESSAGE_PRODUCER;

task MESSAGE_EXEC
declare
  REQUEST SIGNAL;
  ID INTEGER;
  MSG_CODE STRING(5);
  MESSAGE STRING(10); begin
  if (not all MESSAGE_RECEIVERS have been terminated) then
    accept (REQUEST) from MESSAGE_PRODUCER or MESSAGE_READER;
    if (REQUEST = NEW_MESSAGE) then
      accept (MSG_CODE, MESSAGE) from MESSAGE_PRODUCER;
      ID = (new message identifier);
      case (MESSAGE_PRODUCER sends system terminated by checking MSG_CODE value);
      else if (REQUEST = READ_MESSAGE) then
        if (last terminating MESSAGE_RECEIVER processed) then
          send (ID, MSG_CODE, MESSAGE) to MESSAGE_RECEIVER;
        else
          send (ID, special termination MSG_CODE) MESSAGE to MESSAGE_RECEIVER;
        end if;
      end if;
      else if (REQUEST = NEW_RECEIVER) then
        send (message that another MESSAGE_RECEIVER is active)
      end if;
  end do;
end task MESSAGE_EXEC;

task MESSAGE_READER
declare
  ID INTEGER;
  READER_CODE, MSG_CODE STRING(5);
  MESSAGE STRING(10); begin
  send (NEW_RECEIVER) to MESSAGE_EXEC;
  prompt (operator for new READER_CODE) to terminal;
  do while (termination has not been requested by MESSAGE_EXEC) generate
    accept (REQUEST) to MESSAGE_EXEC;
    accept (ID, MSG_CODE, MESSAGE) from MESSAGE_EXEC;
    if (first terminating requested by checking MSG_CODE value) then
      write (MESSAGE) to terminal;
    end if;
  end do;
end task MESSAGE_READER;

begin
  start MESSAGE_PRODUCER;
  (Operating system will allow people to get into the distribution system by running the MESSAGE_READER task);
end begin;
end MESSAGE_DISTRIBUTION;
program MESSAGE_DISTRIBUTION

begin
accept SIGNAL to NEW_MESSAGE. NEW_READER
end MESSAGE_PRODUCER

begin
read MESSAGE_STRING(1);
EXEC(6) INTO INTEGER
MESS_CODE STRING(1) := 5.
end MESSAGE_PRODUCER

create MESSAGE_EXEC:
begin
while (not staged by operator)
begin
accept (operator for MESS_CODE) to terminal.
send (MESS_CODE) to MESSAGE_EXEC
end while
end MESSAGE_EXEC

begin
MESSAGE_STRING(1) := 7.
EXEC(6) INTO INTEGER
MESS_CODE STRING(1) := 5.
end MESSAGE_PRODUCER

begin
FOREVER
accept REQUEST, from MESSAGE_PRODUCER to MESSAGE_READER
if REQUEST = NEW_MESSAGE then
accept MESS_CODE MESSAGE from MESSAGE_PRODUCER
else MESS_CODE = special termination value.
end do for i := 1 to NUM_READERS
if MESS_CODE = READER_CODE then
SET_FLSF:COM_FLSF :=
send (MESS_CODE MESSAGE) to MESSAGE_READER
end if
end do
end if
end do
end if
else if REQUEST = NEW_READER then
NUM_READERS := NUM_READERS + 1;
accept (READER_CODE, NEW_READER) from MESSAGE_READER
send (read_message_element or COM_FLSF) to MESSAGE_READER
end if
end do
end MESSAGE_EXEC

begin
MESSAGE_READER
begin
accept COM_FLSF from MESSAGE_EXEC
begin
accept (READER_CODE MESSAGE) to terminal.
send (READER_CODE MESSAGE) to MESSAGE_EXEC
end message
end do
end while
end MESSAGE_READER
end MESSAGE_DISTRIBUTION

MESSAGE DISTRIBUTION (D)
program AIR_TRAFFIC_DISPLAY
declare
type OBJECT_DESCRIPTOR_RECORD is record
   ID : INTEGER
   ALTITUDE : INTEGER
   ROW : INTEGER
   COLUMN : INTEGER
   ALTITUDE_CHANGE_INDICATOR : INTEGER
   HAZARD_INDICATOR : INTEGER
   OLD_ALT : INTEGER
end record
SYNC_SIGNAL_TO_RADAR_MONITOR : COMMUNICATION_FLAG

task CONTROL
<starts up the other two processes in the system and allows the operator to terminate the system.>
end CONTROL

task RADAR_MONITOR
<periodically sends a set of OBJECT_DESCRIPTOR_RECORDs to SCREEN_UPDATE so that it can update the air traffic display and also notifies the SCREEN_UPDATE process at the time it should terminate that it should terminate>
end RADAR_MONITOR

task SCREEN_UPDATE
declare
   OBJECTS(20) : OBJECT_DESCRIPTOR_RECORD
   NUM_OBJECTS INTEGER
begin
do forever
   SET_FLAG(SYNC_SIGNAL_TO_RADAR_MONITOR)
   accept (NUM_OBJECTS) from RADAR_MONITOR
   if ((end of file found instead of NUM_OBJECTS) then
      exit do end if
   do for I = 1 to NUM_OBJECTS
      accept (OBJECTS(I)) from RADAR_MONITOR
      if ((object disappeared from screen) then
      <clear image of object from screen>
      end if
   end do
   do for I = 1 to NUM_OBJECTS
      if ((new object on screen) then
      <initialize record OBJECTS(I)>
   else
      <save indicator of altitude change of object in record OBJECTS(I)>
   end if
   do for each object described by OBJECTS, update the object display on the CRT
end do
end SCREEN_UPDATE
begin
   start CONTROL
end
program AIR_TRAFFIC_DISPLAY
begin
end program AIR_TRAFFIC_DISPLAY

declare
  type OBJECT_DESCRIPTOR_RECORD is record
  ID : INTEGER
  ALTITUDE : INTEGER
  ROW : INTEGER
  COLUMN : INTEGER
  ALTITUDE_CHANGE_INDICATOR : INTEGER
  HAZARD_INDICATOR : INTEGER
  end record

SYNC_SIGNAL_TO_RADAR_MONITOR : COMMUNICATION_FLAG

task CONTROL
  (starts up the other two processes in the system and allows the operator to terminate the system.)
end task

end begin

task RADAR_MONITOR
  (periodically sends a set of OBJECT_DESCRIPTOR_RECORDs to SCREEN_UPDATE so that it can update the air traffic display - also notifies the SCREEN_UPDATE process at the appropriate time that it should terminate)
end task

end begin

task SCREEN_UPDATE
begin
  start CONTROL
end task
begin

AIR_TRAFFICDISPLAY (II)

-37-
program AIR_TRAFFIC_DISPLAY

declare

type OBJECT_DESCRIPTOR_RECORD is record
  ID : INTEGER
  ALTITUDE : INTEGER
  ROW : INTEGER
  COLUMN : INTEGER
  ALTITUDE_CHANGE_INDICATOR : INTEGER
  HAZARD_INDICATOR : INTEGER
end record

SYNC_SIGNAL_TO_RADAR_MONITOR : COMMUNICATION_FLAG

end AIR_TRAFFIC_DISPLAY

begin
  start CONTROL
end
program TEXT_SEARCH is
  declare
    type SIGNAL is PROCEDURE; FINISHED, SEARCH_DONE:
  end
  REQUEST_HANDLER
  declare
    I:NUR_KEY SEARCH_ID INTEGER
    KEYS: STRING(1 50)
  begin
    write (description of program) to terminal
    prompt (operator to continue) to terminal
    create (SEARCH, SEARCH_ID):
      accept (PROCEED) from SEARCH
      prompt (operator to continue program) to terminal:
      if (end of file received) then
        exit do
        end if
      end do
    for every search created, accept (FINISHED) from SEARCH
    and REQUEST_HANDLER
  end
  SEARCH
  declare
    NUR_KEY, KEYS_LENGTH, PRINT_ID, DATA_BASE_CHOICE INTEGER
    KEYS(5), STRING(1 50)
  begin
    NUR_KEY := 0
    do forever
      prompt (operator for KEYS, NUR_KEY) to terminal
      if (end of file received) then
        exit do
        end if
      end do
      if (NUR_KEY > NUR_KEY) then
        exit do
      end do
      prompt (operator to enter his DATA_BASE_CHOICE) to terminal
      end do
      send (PROCEED) to REQUEST_HANDLER
      exit do
      end do
    for i in 1..NUR_KEY
      if (print file) then
        send (KEYS(i)) to request Handler
        KEYS_LENGTH(i) := last_char_loc(KEYS(i));
        end do
      end do
      send ("STOP") to PRINT_FILE
      open specified data base direction, file
      do forever
        read (FILE_NAME) from directory
        if (end of file received) then
          exit do
          end if
        if (key_in_file(FILE_NAME, KEYS, KEYS_LENGTH) then
          send (FILE_NAME) to PRINT_FILE
          and if
          end do
        send ("STOP") to PRINT_FILE
        (if necessary, notify next search that this one is terminating)
        send (FINISHED) to REQUEST_HANDLER
        exit do
      end SEARCH
  end
  PRINT_FILE
  declare
    KEYS STRING(1 50) := null
    FILE_NAME STRING(1 50) := null
  begin
    accept (PROCEED) from SEARCH
    create output file "TEXT_MATCH_DAT"
    do while (KEYS := "STOP")
      accept (KEY) from SEARCH
      write (KEY) to "TEXT_MATCH_DAT"
      end do
      do while (FILE_NAME := "STOP")
      accept (FILE_NAME) from SEARCH
      write (FILE_NAME) to "TEXT_MATCH_DAT"
      end do
      end PRINT_FILE
  end
  print request HANDLER
  and TEXT_SEARCH
program TEXT_SEARCH
declare
  type SIGNAL is (ENQUEUE, DEQUEUE);
  type KEY, NAME is string(1..50).
  type NUM_KEYS is integer range 0..50.
  type DATA_BASE_CHOICE is integer range 1..10.
  type FILE_NAME is string(1..50).
begin
  write (description of program) to terminal.
  prompt (operator to continue) to terminal.
  num_keys := 0;
  do forever
    num_keys := 0;
    do forever
      prompt (operator to enter key to search) to terminal.
      if (end of file received) then
        exit do
      else
        num_keys := num_keys + 1;
      end if;
      do forever
        prompt (operator to enter file name) to terminal.
        if (end of file received) then
          exit do
        end if;
        create (search_it, num_keys) to search.
        create (print_it, num_keys) to print.
        prompt (operator to continue program) to terminal.
        if (end of file received) then
          exit do
        end if;
      end do;
    end do;
    request_handler:
    declare
      request: SIGNAL;
      print_id: num_keys, num_files: key_length: integer
      file_name: string(1..50);
    begin
      do forever
        accept (request) from SEARCH;
        if request = ENQUEUE then
          create (print_it, num_keys) to print;
          create (search_it, num_keys) to search;
          create (print_it, num_keys) to print;
        end if;
        if request = DEQUEUE then
          delete (print_it, num_keys) from print;
          delete (search_it, num_keys) from search;
          delete (print_it, num_keys) from print;
        end if;
      end do;
    end request_handler;
end TEXT_SEARCH.
begin
    accept REQUEST from REQUEST_HANDLER or SEARCH
    if REQUEST = START_SEARCH then
        CREATE SEARCH_ID = NUMBER
        ACCEPT SQLite_DB CHOICE, NUMBER KEYS, and the set of KEYS from REQUEST_HANDLER
        and save SEARCH_ID, DATA_BASE_CHOICE, NUMBER KEYS and the set of KEYS to the SEARCH process
        NUM_KEYS = (number of keys)
        NUM_INDEXES = NUM_KEYS (key list received)
    else if REQUEST = SEARCH_DONE then
        ACCEPT SEARCH_ID, NUMBER_FILES from SEARCH
        save SEARCH ID, INDEX, and the set of KEYS extracted at INDEX to the
        SQL_DATABASE process and send NUMBER_FILES and the set of FILE_NAMES to the SQL_DATABASE process
    end if
end
end REQUEST_HANDLER

end QUEUE_MANAGER

begin
    ACCEPT SQLite_DB CHOICE, NUMBER KEYS, and the set of KEYS from REQUEST_MANAGER
    open specific file data set at directory files
    NUM_FILES = 0
end
end廣QUE Manager

begin
    ACCEPT SQL DATABASE, NUMBER_FILES, and the set of FILE NAMES to REQUEST_MANAGER
end
end SEARCH

begin
    ACCEPT SQL DATABASE, NUMBER_FILES, and the set of FILE NAMES to REQUEST_MANAGER
end
end PRINT_FILE

begin
    ACCEPT REQUEST_HANDLER
end
end TEXT_SEARCH
PETRI NETS
TECHNICAL REPORTS DISTRIBUTION LIST
### OFFICE OF NAVAL RESEARCH

Engineering Psychology Group

#### TECHNICAL REPORTS DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>OSD</th>
<th>Department of the Navy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPT Paul R. Chatelier</td>
<td>Tactical Development &amp; Evaluation Support Programs</td>
</tr>
<tr>
<td>Office of the Deputy Under Secretary of Defense</td>
<td>Code 230</td>
</tr>
<tr>
<td>OUSDRE (E&amp;LS)</td>
<td>Office of Naval Research</td>
</tr>
<tr>
<td>Pentagon, Room 3D129</td>
<td>800 North Quincy Street</td>
</tr>
<tr>
<td>Washington, DC 20301</td>
<td>Arlington, VA 22217</td>
</tr>
<tr>
<td>Dr. Dennis Leedom</td>
<td>Manpower, Personnel &amp; Training Programs</td>
</tr>
<tr>
<td>Office of the Deputy Under Secretary of Defense (C3I)</td>
<td>Code 270</td>
</tr>
<tr>
<td>Pentagon</td>
<td>Office of Naval Research</td>
</tr>
<tr>
<td>Washington, DC 20301</td>
<td>800 North Quincy Street</td>
</tr>
<tr>
<td>Department of the Navy</td>
<td>Arlington, VA 22217</td>
</tr>
<tr>
<td>Engineering Psychology Group</td>
<td>Special Assistant for Marine Corps Matters</td>
</tr>
<tr>
<td>Office of Naval Research</td>
<td>Code 100M</td>
</tr>
<tr>
<td>Code 442 EP</td>
<td>Office of Naval Research</td>
</tr>
<tr>
<td>Arlington, VA 22217 (2 cys.)</td>
<td>800 North Quincy Street</td>
</tr>
<tr>
<td>Department of the Navy</td>
<td>Arlington, VA 22217</td>
</tr>
<tr>
<td>Aviation &amp; Aerospace Technology Programs</td>
<td>CDR James Offutt, Officer-in-Charge</td>
</tr>
<tr>
<td>Code 210</td>
<td>ONR Detachment</td>
</tr>
<tr>
<td>Office of Naval Research</td>
<td>1030 East Green Street</td>
</tr>
<tr>
<td>800 North Quincy Street</td>
<td>Pasadena, CA 91106</td>
</tr>
<tr>
<td>Arlington, VA 22217</td>
<td>Director</td>
</tr>
<tr>
<td>Communication &amp; Computer Technology Programs</td>
<td>Naval Research Laboratory</td>
</tr>
<tr>
<td>Code 240</td>
<td>Technical Information Division</td>
</tr>
<tr>
<td>Office of Naval Research</td>
<td>Code 2627</td>
</tr>
<tr>
<td>800 North Quincy Street</td>
<td>Washington, DC 20375</td>
</tr>
<tr>
<td>Arlington, VA 22217</td>
<td>Dr. Michael Melich</td>
</tr>
<tr>
<td>Information Sciences Division</td>
<td>Communications Sciences Division</td>
</tr>
<tr>
<td>Code 433</td>
<td>Code 7500</td>
</tr>
<tr>
<td>Office of Naval Research</td>
<td>Naval Research Laboratory</td>
</tr>
<tr>
<td>800 North Quincy Street</td>
<td>Washington, DC 20375</td>
</tr>
<tr>
<td>Arlington, VA 22217</td>
<td>Dr. Robert E. Conley</td>
</tr>
<tr>
<td>Department of the Navy</td>
<td>Office of Chief of Naval Operations Command and Control</td>
</tr>
<tr>
<td>Special Assistant for Marine Corps Matters</td>
<td>OP-094H</td>
</tr>
<tr>
<td>Office of Naval Research</td>
<td>Washington, DC 20350</td>
</tr>
</tbody>
</table>

**-57-**
Department of the Army

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

Department of the Air Force

U. S. Air Force Office of Scientific Research
Life Sciences Directorate, NL Bolling Air Force Base
Washington, DC 20332

AFHRL/LRS TDC
ATTN: Susan Ewing
Wright-Patterson AFB, OH 45433

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HEE
Wright-Patterson AFB, OH 45433

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks AFB, TX 78235

Foreign Addresses

Director, Human Factors Wing
Defence & Civil Institute of Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

Department of the Army

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

Department of the Air Force

U. S. Air Force Office of Scientific Research
Life Sciences Directorate, NL Bolling Air Force Base
Washington, DC 20332

AFHRL/LRS TDC
ATTN: Susan Ewing
Wright-Patterson AFB, OH 45433

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HEE
Wright-Patterson AFB, OH 45433

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks AFB, TX 78235

Foreign Addresses

Director, Human Factors Wing
Defence & Civil Institute of Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

Other Government Agencies

Dr. M. Montemerlo
Human Factors & Simulation Technology, RTE-6
NASA HQS
Washington, DC 20546

Other Organizations

Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311

Dr. Robert T. Hennessy
NAS - National Research Council (COHF)
2101 Constitution Avenue, N.W.
Washington, DC 20418

Dr. Robert C. Williges
Department of Industrial Engineering and OR
Virginia Polytechnic Institute and State University
130 Whittemore Halle
Blacksburg, VA 24061

Mr. Edward M. Connelly
Performance Measurement Associates, Inc.
410 Pine Street, S.E.
Suite 300
Vienna, VA 22180

Dr. J. O. Chinnis
Decision Science Consortium
Suite 721
7700 Leesburg Pike
Falls Church, VA 22043

Dr. Richard Pew
Bolt, Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238

Psychological Documents (3 cys.)
ATTN: Dr. J. G. Darley
N-565 Elliott Hall
University of Minnesota
Minneapolis, MN 55455