N73-19227

PROGRAM ANALYSIS FOR DOCUMENTATION

G. H. Lolmaugh
Programming Methods, Inc.

Program analysis for documentation (PAD) is a technique that produces computer program documentation in three steps. It is FORTRAN oriented but could just as well be directed toward any other programming language. It currently gets little help from the computer, but this is hopefully only a temporary hiatus in its development cycle. The three steps to the program analysis include describing the variables, describing the structure, and writing the program specifications. This is clearly the opposite order of the normally accepted way of doing things, but is consistent with the way much programming is done.

The questions of why or how programs get written with or without beforehand understanding of the problem statement or why one should bother documenting the intricacies of something that works will not be discussed here. Only the premises that a program exists in compilable form and a decision has been made to document it will be of importance.

Before proceeding, it should be made clear that a program that contains few or no lucid internal comments, for which no programmer's notes are available, and whose author is unavailable, is going to be difficult, if not impossible, to document properly. But this cannot be used as an excuse for refusing to document such a program. Facts will become known as the calling and called routines are analyzed, and after two or three analysis passes through the entire system are made, much information will emerge. Articulate documents can appear when an organized system is consistently used for posting facts as they are found.

Large systems containing many routines usually have a more or less elaborate scheme of blocked common variables. It is convenient for the documenter to develop a completely separate common document where all the known facts about each common variable are posted. This makes it unnecessary to describe a common variable thoroughly and redundantly in each routine that uses it.

Programmers will undoubtedly balk at the notion of describing every variable. For the moment, it will be assumed that every variable is important, or it would not be in the program. Unimportant and abandoned variables have been known to cause trouble. Being able to decide what to leave out of a document without compromise is what makes a documenter a skilled professional. As this analysis proceeds, a method will be developed for specifying in advance the criterion of impunity.

LISTING THE VARIABLES

The variables in a FORTRAN program have several attributes that are of interest to the documenter. These attributes are well known to the compiler, and, in fact, the compiler produces, or can be directed to produce, listings in various orders of sort and with various degrees of completeness of facts. A proposal for an additional listing will be specified shortly that will save much of the labor and tedium about to be described.

Describing the terms to be used at the beginning of a technical report has been an accepted standard procedure for many years. This enables the author to use his terms as symbols in the body of his report and keep his report concise, orderly, and fast moving. With PAD, a program writeup can be organized in the same way. Describe the terms (variables) first, then write the body of the report (program specifications). This may seem backward to those who view specifications as having been written before the program is written, but it does not seem at all backward to the programmers, users, and other technicians for whom the writeup is, after all, being written.

Each of the variables in a FORTRAN program may be described as either arguments, common, or internal. The first two may be classed together with the read and write statements, as input/output (I/O). I/O, it turns out, is the single class of information of most concern to the greatest number of readers.

DESCRIBING THE STRUCTURE

The structure of a program is simple or complex in proportion to the amount and complexity of the branching being done. This includes looping, which is a special form of branching. If few branches are involved, or the program is short, the description of the program structure need not be separately documented. However, if more than 5 IF's or DO's are present in the program, an author is well advised to construct a flowchart. The automated flowchart programs available, particularly AUTOFLOW, may be used to advantage.

Hand-produced flowcharts, neatly and precisely drawn, lend considerably to the credibility of the finished document, particularly if the flowchart is accurate. The choice of symbols and shapes used is of less importance than consistency and style. Above all, a flowchart should flow.

For very complex programs, particularly those more complex than they should have been, an automatically produced flowchart is probably more economical and more accurate. However, unless some nesting or editing technique was employed in generating the flowchart, it may be difficult to follow. Flowcharts should never be typed, except by an illustrator or technical typist especially trained to do this work.

WRITING THE PROGRAM SPECIFICATIONS

Program specifications should be easy to write after a program is written, but they seldom are. The fact that they were not written beforehand usually means that a program is something less than well designed and orderly. The foregoing descriptions of variables and flowcharts, together with compiled knowledge of called and calling routines, data

formats, tabulated output, error messages, and other researched material, make it now possible to outline the problem statement the programmer wishes he had available when he commenced work. How much flesh appears in this outline will depend on how much research he has had time to do, how thorough was his work, and how many times he has reedited the writeup for each interrelated subroutine in the system.

Because this paper deals with program analysis, it will not endeavor to show how to do technical writing. At this point, the program analyst must decide who his readers are. Appendix A describes several possible readers and some of their needs. A good checklist of topics and objectives should include the following:

(1) A statement of the mathematical model, equations, and formulas. If copied from, or based on, a textbook case, copy the material here, with credit given in the references.
(2) A statement of the technique, such as sorting or merging, and a description of the sort key or collating sequence.
(3) The decision criteria and tables.
(4) A description of the broad aspects of the I/O consistent with the details already specified.
(5) A stressing of what the program does, rather than how it is done, except where the means of accomplishment is tricky and will not be immediately obvious by examining the code listing.

MANUAL AND AUTOMATED METHODS

The current manual PAD method and the preliminary specifications for a proposed compiler option are described in appendixes B and C, respectively.

APPENDIX A—TECHNICAL EDITOR'S NOTES ON EDITING CRITERIA FOR REVIEWING PROGRAM DOCUMENTATION

Review the writeup on the basis of the needs of the intended reader(s). The possible readers and their needs include:

(1) *The maintenance programmer*—adds, deletes, and changes the program on the basis of new specifications. He needs to know, in addition to what the program does and how it works, the impact of any change he may make on other programs. Any I/O variable (argument, common, read/write) may affect any calling or called routine. Changes in logical tests may change the meaning of messages.
(2) *The user programmer*—needs calling sequence details, as well as other interfaces required to transplant a routine to another system.
(3) *The reprogrammer*—will be involved in transplanting this system to a next generation computer, probably rewriting portions of the code for optimization or new specifications. Needs considerable information about the current program specifications.

(4) *The system user*—may require additional information not found in the user's manual to pinpoint unusual data trouble, machine trouble, compiler trouble, etc.
(5) *The mathematician-analyst*—needs to know whether this program does exactly what he wishes to do, avoids what he wishes to avoid, is otherwise suitable to his needs, or holds promise of being suitably modifiable.
(6) *The project director, technical writer, and others*—responsible for writing and maintaining user's manuals, maintenance manuals, and other technical reports. The programmer's workbook should be the major repository of information and should be sufficiently up to date to enable the compiling of reports on short notice.

APPENDIX B—TECHNICAL WRITER'S NOTES FOR PAD

To methodically analyze (document) a FORTRAN subroutine:

(1) Compile the code. A source listing and a cross-reference are needed.
(2) Make a Xerox copy of the source listing. The ISN's and the statements will fit the 8½-in. width of standard paper. Omit card numbers.
(3) Get red, green, and black thin line marking pens.
(4) Underscore the Xerox copy of the listing in green for common block names, calls, returns, entry points, and other program interface elements; in red for read, write, and format statements, name lists, and other hardware interface elements; and in black for IF's. Bracket the DO loops in black.
(5) Build an argument-list skeleton:
    (a) List each variable in order of its appearance in the calling sequence.
    (b) Note any word size or mode other than implicit.
    (c) Show dimension. If equivalenced, consider so noting.
    (d) Determine I/O status. Use the rules at the end of these notes.
    (e) For indicators or flags, assign a three- or four-word plain text descriptive name, then tabulate all values and their meanings. For logicals, only the "usual" condition needs describing, unless the opposite status has other than the opposite meaning.
(6) Build a common-table skeleton. For each variable,
    (a) Check the name to see that it is actually used by this program. Use the symbol table to see that an ISN greater than the first executable statement is present. Skip unused names.
    (b) Note any word size or mode other than implicit.
    (c) Show dimension. If equivalenced, consider so noting.
    (d) Determine I/O status.
    (e) Refer to the common writeup for this block; determine that the description here is consistent with that in the common writeup. If the usage here adds to the knowledge in the common writeup, update the common writeup.
    (f) Note any pertinent comments in this program listing.

(g) Check the usage of this variable in several places in the code (use the symbol table) to see that information in (6(d)) and (6(e)) makes sense and is current.
(h) Describe the purpose or usage, as pertinent to this program. Use a brief, concise, terse form. The common writeup should contain the total information about a variable, and lengthy details should be documented as program specifications.

(7) Build a read/write table:
(a) Make one entry for each possible read or write statement.
(b) For a card, mention what the card is for, how many, any preconditions (IFs), and make reference to the card layout figure.
(c) For printed messages, state the conditions for the message, state the message precisely, and mention or show any tabulations that follow. Obtain sample printouts whenever possible.
(d) For a tape read or write, use the same general rule as for a card read, specifying the record and file structure and referencing a tape layout figure.
(e) For disk, data cell, and similar data sets, explain the define file parameters.

(8) Start building an internal variable table. Add to it as analysis proceeds. Unimportant variables may be omitted. An unimportant variable is one whose purpose or usage is immediately obvious. The use of (9) as the index in an unnested DO loop is obvious.
(9) Construct a flowchart, if necessary. This may not be needed if the program contains less than four decision statements.
(10) Write the program specifications.
(11) Write the error procedures, if any. Clues to error procedures are error messages, flags in the argument list or in common, and STOP and PAUSE statements.
(12) Name the routines calling this one (if known).
(13) Name the routines called by this one. Look for FUNCTION names, note program names for alternate entry points.
(14) Add the following sections, as required:
(a) References
(b) Flowcharts
(c) Attachments including tables, card and tape layouts, and sample output, input, job control language.

I/O rules. When describing a variable in a FORTRAN routine as input and/or output, the following rules apply:

(1) Input and/or output pertains to input or output usage of a variable as seen from this routine's viewpoint.
(2) A variable is input if this routine needs it for computation, testing, or other internal purposes.
(3) If a variable first appears in this routine on the right side of an equal sign, the variable is input.
(4) If a variable first appears in an IF argument, the variable is input.
(5) A variable is output from a routine if the routine changes its value in any way.

(6) If a variable appears anywhere in this routine on the left of an equal sign, this variable is output.
(7) A variable may be both input and output.
(8) An argument is input and/or output in this routine consistent with its usage as an argument in a called subroutine.
(9) Arguments or common variables that appear in read statements are output from the routine because values are (or may be) changed. Arguments or common variables that appear in write statements are input to a routine because a value is expected of them.
(10) I/O is applicable to arguments, common, and, conceivably, registers.
(11) Every routine must have at least one input or output item, or the writeup must explain the discrepancy.

APPENDIX C—FORTRAN PROGRAM ANALYZER FOR DOCUMENTATION (PRELIMINARY SPECIFICATIONS)

The purpose of this program is to produce a checklist of items that must be contained in the program documentation and to include as many facts about these items as may be available. Provision is made for communication between internal and external documentation.

Using information available from the FORTRAN compiler, produce tabulated lists of variables and lists of other items to be covered in the documentation. Lists of variables include calling sequence arguments, common variables, variables in read/write statements, and important internal variables. Other items include text of write statements and comments concerning error procedures.

Calling Sequence Arguments

Generate a list of the argument names showing I/O context, dimension, mode (other than implicit), equivalence, etc., in the following form:

I	ARG1	(E)	I*4
O	ARG2(*n*)		L*1
I/O	IARG(*n*)	(E)	
	.		
	.		
	.		

where (*n*) is the dimension (if any), (E) denotes some equivalence, and I*4 and L*1 are modes that are not implicit.

Elements of this table are to be printed one per line (double-spaced option), suitable for later manual entry (by the programmer) of descriptive text. When operating in the internal documentation mode, the program will scan the first word of each existing comment card containing a delimiter (–) and include the text of that comment. Variables are listed in order of their appearance in the argument list.

I/O context for each variable is to be labeled I, O, or I/O according to the rules in appendix B. These rules are designed to show a user programmer which values he is expected to furnish and which values will (or may) be changed by this routine.

Common Variables

Generate a list of common variable names showing I/O context, block, dimension, mode (other than implicit), equivalence, etc., in the following form:

```
I      BLOKA    VAR1(n)
O               ARRY(i,j)                R*4
I/O    BLOKB    VAR                 (E)
I               A
O               C
```

where the meaning of the symbology is now obvious.

The same characteristics and rules as for calling sequence arguments apply here. However, only those variables appearing in an executable statement or equivalenced to a variable in an executable statement are normally listed. On option, list all the variables, showing NR as applicable, to facilitate the building or checking of the complete common directory. Note that the NR test used by the FORTRAN compiler differs because of the executability specification. A variable must appear somewhere in the program following the beginning of the first executable statement to be considered executable here. A format statement is considered nonexecutable, but the variables appearing within it are executable, nonetheless.

Block and variable names are listed in the same order as their appearance in their definition statements at the beginning of the program.

Read/Write Statements

Generate a list of read and write statements naming the data set, format or define file number, etc., in the following form:

```
(27)   W   6   END DATA HANDLER – ELAPSED TIME . . (T) . . . SEC
(12)   R   5   REFDAY(3)
               IPRT                                    L*1
                  T =
                  F =
```

These entries will mostly provide blank space for the author to write his descriptions. However, the FORTRAN compiler can recognize many elements and post them accordingly.

Messages will be printed verbatim, followed by two blank lines to be used to describe the conditions under which the message is printed. A message is any H-type statement.

Name lists, either in or out, are displayed in the following form:

```
&NAME.

    ARG1
    ARG2(n)
      .
      .
      .
```

with dimension, mode, equivalence, etc., as specified for calling sequence arguments.

Read or write statements that can be recognized as cards will have the variables tabulated in a manner conducive to generating card layouts.

BCD write statements to other than card or printer data sets will have the variables tabulated in a manner conducive to generating tape record layouts.

Direct access data-set references will have the define file statement tabulated in an appropriate manner.

Sample output is usually difficult to find for documentation purposes and often lacks the generality the author wishes because of the conditions of the run. Making a test run in which all cases and all error messages are displayed is challenging. Therefore, in the DOCEXEC mode, all output statements will be executed once, consistent with the included GO and DD statements. Variable quantities in this display will be dots, indicating the field size specified by the format statement.

Switches

All logicals and all I*1 and I*2 variables will have extra line spaces for entering logical conditions in addition to the name of the variable. Logicals will provide T/F indication lines; integers will provide an arbitrary three extra lines.

Arrays

An arbitrary three extra lines will be provided following each subscripted variable name for the purpose of describing the individual variables in an R-type or I*4 (or larger) array.

Care should be exercised here not to generate too much blank paper. For instance, each of the elements of a transformation matrix need not be described. However, in the case of an array, SPEC(20, 7), where the I's are characteristics and the J's are stations, 21 lines should be generated; 1 on which to describe SPEC, and 20 on which to describe each of the characteristics. Conversely, if the I's were stations and the J's the characteristics, eight lines should be generated.

Internal Variables

Generate a list of all internal variables showing dimension, mode, and equivalence. All variables whose names contain two to five characters and that cannot be defined as arguments, common, or read/write variables are considered internal variables. The size restriction

permits a programmer to assign variables whose usage will be intuitively obvious without their being forced into the documentation. A PAD calling argument (OLD) will defeat this test for documenting preexisting programs.

Summary

These preliminary specifications are being revised as time, inclination, and additional interest are shown. Readers wishing to participate in developing these specifications are invited to send in their contributions.

DISCUSSION

MEMBER OF THE AUDIENCE: Where do you stand at the moment on the development of this system?

LOLMAUGH: It is an idea.

MEMBER OF THE AUDIENCE: Have you done any development on it?

LOLMAUGH: The manual portion of it (apps. A through C) does exist, and I already use it. I also use it as a tutorial for the tech writers, programmers, or anyone else who helps me with documentation. I had planned to discuss that at greater length because I know the programming people in this group would have liked to hear more about it, but I was discouraged from that because it does not involve automation and this was after all a symposium on automated documentation. I do use a symbol table from compiled routines. I hope my paper was able to illustrate the volume of hand work that I do that I know the computer can help me do. I join several of the previous speakers in requesting that the compiler be put to more work. I think it can be put to more useful work, at least where people want documentation.