N72-19219

AUTOMATIC PROGRAM ANNOTATION (AUTONOTE)

Michael D. Neely
and
Judy W. Tyson
ARIES Corp.

Computer program documentation, the "maps" of the computing industry, is a very neglected area of this industry. Countless hours are spent trying to find out what has been done in the past, and time and money are wasted duplicating past efforts.

However, the poor quality of these "maps" is only a secondary effect of this neglect of the field. The primary effect is the scarcity of tools that can be used in the documentation process. A determination of what types of tools are needed still must be done. This paper will attempt a preliminary identification of these tools.

Before those tools can be identified, though, the uses of the documentation to be produced must be determined. The most important uses are program maintenance, which includes enhancements and error detection and correction, and program development. Efforts in these areas to develop tools that produce good documentation will be more than repaid.

This paper will try to identify some areas of program documentation that should be automated and then will focus on a particular area and explore the possibilities of automation. In general, this discussion is directed at the assembly language program, but in some areas the remarks are germane to metalanguages. The emphasis will be on the tools that are needed, not on the means of providing those tools.

BASIC REQUIREMENTS

Program documentation usually consists of the program specifications, flowcharts, program listing, and operating instructions. Because the specifications are written before the program is and because means of automatically flowcharting programs are already available, the program listing and operating instructions will be the topics discussed here.

The program listing contains coding, the machine language instructions generated by the coding, and comments relating to the coding. When properly interpreted, the coding supplies the most accurate "map" of the program. Interpreting the coding is one of the areas in which automation can improve the use of the listing. The operating instructions provide the information necessary to use the program. They specify the interface between the program, the operator, and the peripheral devices. Some of the operating instructions can be produced as a byproduct of interpreting the coding.

Most of the information needed by the maintenance programmer can be produced by an analysis of the coding. This includes a set of consistent comments relating to the coding,

error identification, executive function identification, analysis of arithmetic operations, and various cross-reference tables. Much of this same information is necessary in the development of a program library but, in addition, it might be desirable to produce other types of information, more general in content. These can be produced by the use of control cards to identify the information. We will consider each of these areas to determine what effect automation can have on the documentation process.

Comments

Program listings normally include comments provided by the programmer; in addition to these comments, a means of automatically producing comments from the coding would be helpful. These comments could be on two different levels: An analysis of the coding or an analysis of the logic defined by the coding. This area will be explored in more detail in a later section of this paper.

Error Detection

The objective of the error detection process would be to identify those errors that are readily apparent from an analysis of the coding without going into a detailed analysis of the logic of the program. It would be impossible, for example, to determine from the coding if a program meets the program specifications. It would be possible, however, to spot other errors that are related to the mechanics of coding.

Errors that could be readily detected include instructions using invalid operators, invalid operands, undefined program labels, or doubly defined program labels. For bank-oriented computers it would be useful to identify areas where code spills over the end of a bank or where an area of core is overlaid. Another possibility in this area would be the flagging of instructions referencing items located in a different bank.

A different type of error would involve the use of computer registers. It would be possible to flag coding in which the contents of a register are destroyed. In this case, the register is loaded with one value, then loaded with another value before the first value has been used. In double-precision operations it would be possible to flag instructions that reference improperly aligned items.

Errors should be flagged whenever they are encountered in the coding; in addition there should be an error summary at the end of the listing.

Operator Interface

Executive functions, mainly input- and output-related operations, are an important segment of any program and therefore play an important role in the program documentation. It is necessary to specify which devices are used, the manner in which they are used, and what operator interface is required for those devices. All these can be provided automatically; in addition, AUTONOTE could associate calls to a particular device with the coding, thus producing a cross-reference table of input/output (I/O) calls by device. It might also be possible to associate buffer areas with the devices using those areas.

Another interesting possibility is the identification of all input and output operations on a particular data set. However, this would be more easily implemented with metalanguages than with assembly languages.

In the area of console communications it would be desirable to associate operator messages with the coding that produces the messages. When a response is required from the operator, all valid responses should be listed and default responses should be specified.

Arithmetic Operations

For arithmetic operations, the documentation should specify the limits imposed on an operation by the machine word size, mode of operation (single or double precision), or constants that are used. Iterative operations should be identified, and the limits on the number of iterations should be specified. These functions can be provided from an analysis of the coding. In addition, it may be possible to analyze algorithms to produce the formulas defined in the coding. In metalanguages it would be possible to specify the accuracy that would be obtained in an arithmetic process.

Cross-References

Program listings normally have only one cross-reference table, an alphabetical list of program labels with the instruction numbers in which the labels are referenced. Several other types of cross-reference tables would be useful and could be produced easily during an analysis of the coding. In addition to the basic alphabetical cross-reference of all items, there should be separate cross-reference tables for data constants, address constants, buffer areas, subroutines, I/O calls, and labeled instructions. The tables of constants should identify any duplicated items, and all tables should identify unreferenced items. In addition, there should be a separate cross-reference table of undefined items. These various tables would aid the programmer in debugging his program originally as well as in maintaining the completed program.

When origin instructions to the assembler program are provided, AUTONOTE would produce a basic core map showing the area used, the location of all origin statements, and the location of any overlaid areas of core.

These are all basic items that should be provided in the documentation to aid the maintenance programmer. However, in addition to these items, many organizations require that documentation be in a specified format. The information required for this documentation is usually, or at least should be, included in the program listing in the form of comments. This information includes program name, acronym, organization name, programmer, assembly date, equipment configuration, source language, core requirement, execution time, and program abstract. With the use of control cards, this information can be extracted from the program to produce the documentation in the specified format. However, this capability should be included only as an option, and the use of control cards should not be necessary to use the other features of AUTONOTE.

AUTOMATIC COMMENTS

The possibilities of automation in the area of program comments should be explored further. This area is particularly susceptible to automation because the current method, relying on programmer-supplied comments, has several inherent disadvantages: Not all programmers comment programs in the same detail; the comments may be meaningful only to the original programmer; and often program coding is changed, but the comments remain the same. In addition, in metalanguages the programmer may not be familiar with the assembly language that is generated and may not be able to determine from the coding what is taking place.

To automatically comment a program, first, a set of comments would be associated with the instruction set, as shown in figure 1. Instructions that cause an alteration in the sequential execution of code under certain circumstances would require more than one comment. These comments would then be used to explain the mechanics of the operations. An illustration of this technique is shown in figure 2. This process requires a limited ability to look ahead in the coding to determine what is taking place, but it does not require a detailed analysis of the logic defined by the coding. The automatic comments are provided in addition to the programmer's comments, not in place of those comments.

It would also be possible to comment a program by analyzing the logic, roughly the procedure used by flowcharting programs, but that would require a greater effort in looking forward and backward in a program and would be a duplication of the flowcharting process. Because the program listing is considered a complement to the flowchart, production of comments keyed to the coding should be an area of concentration.

Subroutines are an important element in any program, and the comments relating to the subroutines are important for an understanding of the program. Good documentation of subroutines is not only helpful to the maintenance programmer but also important in the development of a subroutine library. Well-documented subroutines can prevent needless duplication of effort in the development of new programs. Figure 3 shows an illustration of a relatively simple subroutine with the programmer's comments. Figure 4 shows the same subroutine with the addition of automated comments. These comments not only describe the processing within the subroutine but also tell which program registers are loaded prior to entry, where the subroutine is called from, which items are for internal use, which external items are used, and which registers are modified at the exit. Much of this information can also be provided in a cross-reference of subroutines at the end of the program listing.

INSTR.	AUTOMATIC COMMENT
CRA	CLEAR A
STA	STORE A IN XXXX
LDA	LOAD A FROM XXXX
ADD	ADD A TO XXXX
SUB	SUBTRACT A FROM XXXX
ALS	SHIFT A LEFT X BITS
LLL	SHIFT B TO A LEFT X BITS
CAS	COMPARE A TO XXXX GREATER EQUAL LESS THAN
SMI	BIT 1 OF A=1 NO YES
SR1	SENSE SWITCH 1 SET YES NO
ENB	ENABLE INTERRUPTS

Figure 1.—Instruction set and associated comments.

```
STAK    LDA     *BUFA                       *LOAD A FROM INPUT BUFFER
        SNZ                                 *A=0
        JMP     $+2                         *YES-GO TO THIS LOCATION+2
        STA     *BUFB                       *NO-STORE A IN OUTPUT BUFFER
        IRS     BUFA    FILL OUTPUT BUFFER  *INCREMENT FWA INPUT BUFFER
        IRS     BUFB                        *INCREMENT FWA OUTPUT BUFFER
        IRS     NOWD                        *WORD COUNT=0
        JMP     STAK                        *NO-GO TO STAK
        JMP     EXIT                        *YES-GO TO EXIT

*
BUFA    DAC     INBF    FWA INPUT BUFFER
INBF    RES     10      INPUT BUFFER
BUFB    DAC     OTBF    FWA OUTPUT BUFFER
OTBF    RES     10      OUTPUT BUFFER
NOWD    DATA    -10     WORD COUNT
```

Figure 2.—Operation comments.

This type of documentation provides adequate information for maintenance programmers and the information needed to develop a program or subroutine library.

As for the format of the documentation, AUTONOTE would begin each listing with the set of instructions and their associated comments (fig. 1). Then would come the program with the automatic comments. These comments would not replace the programmer's comments but would be in a separate column. Thus, for each instruction, there could be two sets of comments. As an option, the programmer could use control cards to suppress the listing of automatic comments in sections of the program. Errors would be marked at the point of origin, and there would also be an error summary at the end of the listing with a separate list of error codes and their meaning. After that would come the cross-reference table of all items and the individual cross-references. Figures 5 and 6 show examples of cross-references for data constants and address constants. In addition to the standard list of label, location, and reference points, this list contains the value of the item and a list of

```
*
* CHECKSUM ROUTINE
* X=FWA
* A=LENGTH
*
CCCK      DAC       0
          TCA
          STA       CLGH
          CRA
          STA       CHEX
          LDA       *0
          ADD       CHEX        COMPUTE CHECKSUM
          STA       CHEX
          IRS       0
          IRS       CLGH        DONE
          JMP       $-5         NO
          JMP       *CCCK       YES
*
CLGH      DATA      0           CHECKSUM COUNTER
CHEX      DATA      0           CHECKSUM
*
```

Figure 3.—Programmer's subroutine comments.

duplicate items. For bank-oriented computers there would be a core map. If control cards are used to request lists in a specified format, these would follow the cross-reference tables.

ADDITIONAL CAPABILITIES

There are many possibilities for expanding a system such as AUTONOTE. For example, with the use of a cathode ray tube it would be possible to "page" through a program. The sequential flow of the program could be interrupted to look at subroutines, with the flow resuming at the end of the subroutine. It would also be possible to modify programmer comments at the on-line terminal or edit those comments to produce a program or subroutine abstract. Subroutines could be selected for a library in this manner, and the library could then be queried from the terminal.

Of course, a system like this presents many challenges as well as opportunities. There are several technical problems that would have to be resolved. For example, in assembly languages it is often difficult to identify I/O devices or to define record layouts. Indexing and indirect addressing would also present problems for the analysis.

The assembly language program has been the topic of this presentation; other types of programs present different problems and possibilities. In metalanguages it is easier to define record layouts and identify I/O devices, and it might be possible to identify and define blocks of logic, but detailed comments might be redundant, as most metalanguages are at least partially self-documenting. Another possibility would be to use the intermediate output of compilers, the assembly language program, as the input to AUTONOTE.

Means of reducing the cost of software must be developed if the software industry is to continue expanding as it has in the past. AUTONOTE may represent a step in the right direction. It will provide reliable, consistent documentation of programs, something that has been lacking in the past. It would be a useful tool for the maintenance programmer, it would

```
*                                                    *LOADED PRIOR TO ENTRY:
*                                                    *A,X
*                                                    *ENTERED FROM CARD NOS.
*                                                    *247,654
* CHECKSUM ROUTINE
*   X=FWA
*   A=LENGTH
*
CCCK      DAC      0                                 *RETURN ADDR CALLING PROG
          TCA                                        *TWO'S COMPLEMENT A
          STA      CLGH                              *STORE A IN CHECKSUM COUNTER
          CRA                                        *CLEAR A
          STA      CHEX                              *STORE A IN CHECKSUM
          LDA      *0                                *LOAD A INDIRECT FROM X
          ADD      CHEX     COMPUTE CHECKSUM         *ADD A TO CHECKSUM
          STA      CHEX                              *STORE A IN CHECKSUM
          IRS      0                                 *INCREMENT X
          IRS      CLGH     DONE                     *CHECKSUM COUNTER=0
          JMP      $-5      NO                       *NO-GO TO THIS LOCATION-5
          JMP      *CCCK    YES                      *YES-GO TO RETURN ADDR CALLING PROG
*
CLGH      DATA     0        CHECKSUM COUNTER
CHEX      DATA     0        CHECKSUM
*
*                                                    *INTERNAL ITEMS:
*                                                    *CLGH
*                                                    *EXTERNAL ITEMS:
*                                                    *CHEX
*                                                    *MODIFIED AT EXIT:
*                                                    *A,X
*                                                    *A=CHEX
```

Figure 4.—Automated subroutine comments.

aid in the development of program libraries, and as a bonus it would be useful as a debugging tool during the original development of a program. This is the type of tool that is needed to begin the war on soaring software costs.

```
* DATA CONSTANTS
A1      511     000001      497     618     627
A2      512     000002      480
A3      513     000003      502
A4      514     000004      312     510
A5      515     000005      452
A6      516     000006      379     580     603
A7      517     000007      411
A8      518     000010      701     718
.
.
.
.
.
.
.
AHIH    103     177777      123     347
ALOW    102     000001      121     214     390
.
.
.
.
.
.
.
*DUPLICATES:
*AL     ALOW                                        VALUE: 000001
```

Figure 5.–Cross-references for data constants.

```
* ADDRESS CONSTANTS:
AFIL      202      AFL1      218      327      339      409
BFIL      311      BFL1      114      256      423
CFIL      119      CFL1      489      522      679
DFIL      450      DFL1      560
.
.
.
.
.
.
.
.
.
.
ZFIL      332      AFL1      510      736      759      840
.
.
.
.
.
*DUPLICATES:
* AFIL    ZFIL                                          VALUE: AFL1
```

Figure 6.–Cross-references for address constants.

DISCUSSION

MEMBER OF THE AUDIENCE: I believe I disagree with a lot of what you said. I think that translating what the machine is doing into another statement does not help at all. This is what our documentation assembly level has done for so long. What we really need is information that attempts to describe what the machine is doing in relation to a definition of specifications of that job and how it relates to that part of the specifications. In other words, what you really need to know most of the time is: What was to be achieved?

NEELY: I agree. This is only going to tell you what he did achieve. I think debugging the program originally would be the main use of it. No doubt this is not a panacea for the industry or for the documentation process.

MEMBER OF THE AUDIENCE: The purpose of a comment is to describe the function, and I think that if you can get a well-commented program you are at least halfway to good documentation of that program. If you discourage the people or provide something that gives you essentially a description of the microcode, the fact that you are loading and storing really gives you very little. It is important not to have programs that are uncommented because one comment should describe maybe five or six instructions and give a functional description. If you encourage people not to do that, you are really going in the wrong direction.

NEELY: I agree with you that we should not replace the programmer's comments by any means, but this would be used in conjunction with his comments.