Automatic Flow-Charting

MAMORU MAEKAWA*

Abstract

For the purpose of program debugging and documentation, the automatic converting of instruction sequences into flow-charts is very useful, because such work is usually very time-consuming and cumbersome. The automatic flowcharting process is divided into four parts. The 1st step is to combine several instructions by two principles. The 2nd step is to describe flow in the computer. The list-notation is used for this description. The 3rd and 4th steps are positioning and lining, respectively.

1. *Introduction*

By way of expressing the program, the flow-chart is generally used. There are two purposes of the flow-chart. The one is for illustrating it to others and the other is for program checking. The programming process is roughly divided into two steps. The 1st step is to draw flow-charts and the 2nd step is to code according to the flow-charts.

The completed program is usually different from the first given flow-charts except a very brief program. In many cases, it is necessary to re-draw after the program has been completed. This final flow-chart must be reserved for later use. But it is usually time-consuming and cumbersome to re-draw it. Especially in the production of enormous volume of programs such as an operating system, it is a great problem. If the computer automatically does this work, it will not only save labour, but also be useful for the standardization of flow-chart.

At program debugging, it is very useful that we can easily translate instruction sequence into flow-chart. Generally, a flow-charter is different from a coder. In this case, the former passes to the latter flow-charts which indicate how to code.

But the flow-charts which are useful at debugging should indicate what the instruction sequence does. Only when these two flor-charts are compared with each other, the debugging is accomplished. For this an instruction sequence must be easily and exactly translated into flow-charts. The automatic flow-charting becomes more practical by the use of the display equipments. For example, it is very convenient for us to be able to trace on flow-charts.

In this paper, I analyze the fiow-chart by way of expressing program, from two standdoints.

The one is that of a function block, and the other is that of the flow of control. I

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 9, No. 3 (1968), pp. 129–136.

* Electronic Computer Engineering Dept., Tokyo Shibaura Electric Co., Ltd.

describe the expression of a flow-chart using the list-notation and also describe a program which authomatically combines instructions into blocks by aid of this expression and positions these blocks on a sheet of paper. Further, I describe how the program simplifies an object program automatically.

Generally, it is easier to treat a higher language than an assembler language, in automatic flow-charting. Because it is rather difficult for a computer to combine instructions into blocks automatically. In this paper I treat an assembler language, but the principles in this paper could be applied for a higher language. Hereinafter, Automatic Flew Charting Program is abbreviated by AFCP.

2. *Principles*

2. 1. *Combining instructions into blocks*

Even though, judging the logical relations of instructions, a man can easily decide how to combine these into a block, it is very difficult for a computer to do it.

Such work requires thinking. In a higher language such as FORTRAN, even if a statment corresponds with a block, the flow-chart is still useful. But, in a assembler language, such method is unfit for use.

Now, flowcharts are divided into following three levels.

i) The 1st is expressed with addresses (depend on each machine).

ii) The 2nd is expessed with not addresses but symbols (not depend on each computer).

iii) The 3rd is one in which some blocks are combined into a block.

The lst strongly corresponds with machine instructions and a block usually consists of 3–5 instructions. Fig. 1 is the example of it.

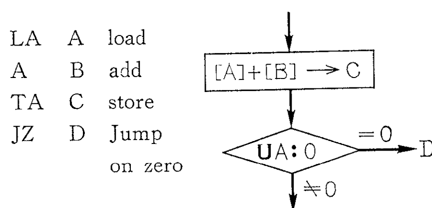

Fig. 1. Instruction sequence.

It is fit for communications between flow-charters and coders, and for dubugging with tracer. And it is also fit for program production, because it is unnecessary for coder to understand how a program works.

But, there is, of cource, a fault in this flow-chart, that is, it is difficult for others to understand it, chiefly because it is not described with symbols, but with addresses, and is too detailed. In order to deliminate this fault, the 2nd is drawn without addresses, that is, data are expressed with symbols which are easily understood by programmers. Transcript from the 1st to the 2nd is difficult, if it is automatically done by a machine unless a machnine

relies upon comments on program sheets or upon the programmers.

In the 3rd, some numbers of blocks are combined into a block in order that other persons can understand general flow of a program. This combining is done by two principles.

Principle 1. The instructions which work a function are combined into a block.

Principle 2. An instruction sequence is divided from the viewpoint of flow of control.

Maybe, there are many viewpoints of program. But two viewpoints are essential to combining. That is, the one is the flow of control, the other is a function. In the former, branch instructions take an important part. In the past, most of the published papers are analyzed according to Principle 2. Principle 1 should be considered as a matter of cource. Nevertheless, it hasn't been considered, as the establishment of judging standards is difficult. In this paper, some standards are induced from Principle 1.

A program, after all, processes some input information and produces some output information. This is a function, namely it works a function. It's natural to combine instructions to work a function into a block. The combining ranges very widely from the degree that the whole program is combined into a block, to the degree that only one instruction becomes a block.

In Principle 2, a program seems to be a flow of control. One of the most basic concepts for dividing a program into blocks is to divide into two parts at a branch instruction, and so a part of a program between two adjacent branch instructions becomes a block, and a branch instruction also becomes a block. This concept is very fit for program analysis. In program analysis, when instructions are rewritten during running, the ordinary method is no longer powerless. In this paper, such a case is not treated.

I set up several standards induced from two principles.

Standards induced from Principle 1.

1. Instructions between an instruction fetching a datum and an instruction storing a result, including both instructions, become a block.
2. A supervisor call instruction becomes a block.
3. A subroutine call instruction becomes a block.

Standards induced from Principle 2.

4. A branch instruction becomes a block.

Standards induced from Principles 1 and 2.

5. A labeled instruction designates the beginning of a new block and it is included in the new blok.
6. An unconditional jump instruction generally designates the end of a block including the instruction.

1 is the most essential standard. It is usual that an instruction sequence which works a function begins at a load-instruction then followed some calculation instructions, and ends at a store-instruction. It is necessary to examine what are load-, store-, calculation-, instructions, depending on each machine. Standards 2 and 3 are clear as a function block. TOSBAC-3400 has special instruction for these, and so the judging is easy. Standard 4 is the most

fundamental for program flow analysis.

Standards 5 and 6 are induced from Principles 1 and 2. These standards are due to man's habit that a programmer usually names the head instruction of a sequence which works a function, and the destination of a branch instruction.

They are not theoretical, but practical. Standards 5 and 6 are mutually the same, except in the case of relative addressing.

2. 2. *Flow analysis*

In 2. 1, I described the standards for combining instructions into blocks.

These generated blocks are basic elements of flow and may not be divided. I analyze the flow of a program by aid of these blocks. The chief aim is to describe flow into lists. Krider, Lee introbuced $*$-notation which is a useful expression to describe flow by lists. I slightly modify the expression so as to fit in processing by a machine, and introduced λ-operator and γ-operand in order to simplify a program automatically. These are described as follows;

Definitions

Source block, object block;

When control relinquishs from Block A to Block B, Block A is called "source block" and block B is called "object block".

$*$-operator; As a symbol to indicate that control relinquishs from A to B,

$*AB$

is used, where $*$ is an operator.

When object blocks of A are B and C,

$*^2ABC$

is used. In the same manner, when object blocks of A are B_1, - - -, Bn,

$*^nAB_1$, - - - Bn

is used. In this manner, when control relinquishes from A to other blocks, the expression, including A and $*$-operator is called $*$-notation of A.

λ-operator :

when an object block of A and an object block of B are the same block C,

λ^2ABC

is used.

γ-operand;

When an object block of A is C and an object block of B is C, too,

$*A\gamma C$, $*B\gamma C$

are used.

Ω-operand;

This indicates the logical end of a program.

loop;

In $*$-notation of A, when A is appeared again as an object block, this is called "loop", and it is called a loop of A.

minimum loop;

In the loop of A, when all the blocks between two adjacent A's are run at least twice, this is called "minimum loop".

Now, two rules and two theorems are induced from the above definitions.

Exchange rule;

$$*^2ABC = *^2ACB$$

$$\lambda^2ABC = \lambda^2BAC$$

Deletion rule;

$$*^2A*B*C\gamma D*E*C\gamma D$$
$$= *^2A*B\gamma C*E\gamma*CD$$

Theorem 1.

In $*$-notation of A, when A is appeared again as an object block, blocks between two adjacent A's, including two A's, constitute a loop, and a part of these blocks constitutes a minimum loop.

Theorem 2.

In the loop of A, when two A's are drawn each other as near as possible by aid of two above rules, it is the minimum loop of A.

3. *Dividing and combining of blocks.*

When we combine several blocks into a large block, two principles in 2. 1 can be used. But the standards 1–6 which are fit to combine into a block as a basic element can be no longer used. Now I discuss standards used for combining several blocks into a large block.

3. 1. *Standards induced from Principle 1.*

It may be impossible to combine several blocks into a larger function block by a machine automatically now.

It is necessary for the man to help a machine by comments or other aids.

3. 2. *Standards induced from Principle 2.*

I describe the standards used for combining and dividing from the standpoint of flow of control.

Standard A. Sequence

A sequence of blocks that does not include a branch instruction becomes a larger block. Fig. 2 shows such an example. In $*$-notation, it is expressed as following.

$$*A*B*C*----*P$$

In this expression, only $*$-operators of the first degree appear.

Standard B. Join

When all the flows which branch from one source block join again into a block, all the blocks are combined into a block. Fig. 3 shows such an example. In $*$-notation, it has such a form as following

$$*^2A*B*C\gamma DE*D.$$

Standard C. One after another branch.

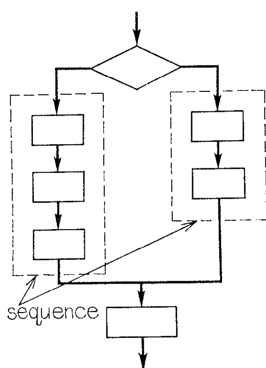

Fig. 2. Sequence.

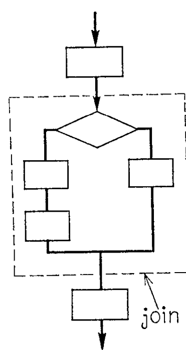

Fig. 3. Join.

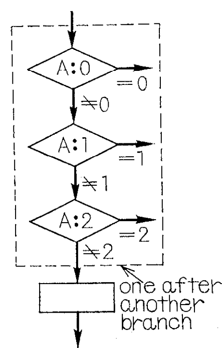

Fig. 4. Ohe after another branch.

When flows branches one after another by the value of one variable, these branches are combined into a block which has many branches. Fig. 4 shows such an example.

In $\ast$-notation, it has such a form as follows.

$$\ast^{2}AB\ast^{2}CD\ast^{2}EFC$$

Standard D. Loop

A loop becomes a block.

In $\ast$-notation, it has such a form as following.

$$\ast A\ast B\ast C\ast^{2}DAE$$

It is clear that it is possible to combine these combined blocks into a block and to divide a combined block into several blocks.

4. *Flow-Charting*

The previous description makes flow-charting procedures clear.

i) To combine instructions into a block by aid of Standard 1–6.
This is the most important and heavy burden.

ii) The relations of these blocks which are created in this manner are expressed in $\ast$-notation (list-notation in a machine). By aid of exchange rule and deletion rule, we modify it fit for flow-charting.

iii) To combine blocks

According to the degree of flow-chart, blocks are combined into a block. This work needs communications between man and machine.

iv) To position blocks on sheets of paper.

v) To line from block to block.

There are many methods for iv). In AFCP, I adopt two-dimensional positioning by the use of $\ast$-notation. When the area of a sheet of paper and an unit area of a block are $A\times B$ (length$\times$width) and $a\times b$ (length$\times$width) respectively, the number of blocks which are in accordance with following two conditions are positioned. Blocks are positioned from left upper to right lower part.

Condition 1.

The total number of blocks is under the following value.

[A/a]×[B/b]

Condition 2.

The total number of branches is under the following value.

[B/b]

This method is the easiest. It is very difficult, but very interesting, to position blocks as skillful as the man does.

5. *Automatic Flow-charting Program*

AFCP consists of six phases.

1) To combine instructions into blocks.

This is done by aid of the principles and the standards of 2. A program expressed by instructions is translated into one expressed by blocks which are function blocks.

In assembly languages, there are many difficult problems, as follows.

- relative addressing (especially jump instruction)
- expression of shift instructions
- expression of assembler pseudo instructions
- subrontine call

In assembly languages, instructions often modify other instructions. In such a case, the processing in this phase is difficult. The processing in this phase is such as a combination of an assembler and a tracer. But it is different from tracer at the point that it must trace all flows which may happen. The input of this phase is source cards of a program, and the output is $*$-notation.

At the same time, the table of each instruction corresponding to each block is generated and stored on a secondary memory as a file. A sequence number is used to distinguish each block. Furthermore, the table of blocks with γ-operand is stored in a secondary memory as another file. This is a preparation in order to simplify the program automatically.

2) The correlations of blocks with γ-operand are inspected and if there are same two blocks, the high-numbered block is deliminated. This is one of simplification.

If E is an object block of C and D, it is expressed by following $*$-notation.

$*A*C\gamma E \qquad *B*D*\gamma E$

Blocks C and D with γ-operand are recorded in every object block, and then C and D are inspected if they are the same. If they are the same, $*$-notation is modified as following.

$*A*C*\gamma E \qquad *B\gamma C$

Such a thing is done over all γ-operands.

3) Minimum loops are found in order by aid of exchange rule and deletion rule.

4) According to requests of programmers, the combining and dividing are done. The most detailed flow-chart is that of Phase i). More generalized flow-charts are generated by aid of the principles and the standards of 3.

5) Blocks are positioned by aids of two-dimensional positioning.
6) Blocks are lined to one another.

6. *Conclusion*

The levels of automatic flow-charting are full of variety. It is disposed to become much more difficult in proportion as it becomes higher degree.

Especially, it is a great problem to combine blocks as a function block and to position blocks on a sheet of paper as skillful as the man does. Furthermore, it is a problem to reduce needs of a great amount of memory and high-speed processing.

References

[1] Haibt, Lois M., A Program to Draw Multilevel Flow Charts, *Proceedings of the Western Joint Computer Conference* (1959), 131–137

[2] Karp, Richard M., A Note on the Application of Graph Theory to Digital Computer Programming, *Information and Control*, *3*, 2 (1960), 179–190

[3] Krider, Lee., Flow Analysis Algorithm, *Journal of the Association for Computing Machinary*, *11*, 4 (1964), 429–436

[4] Prosser, Resse T., Application of Boolean Matrices to the Analysis of Flow Diagrams, *Proceedings of the Eastern Joint Computer Conference*, (1959), 133–138

[5] Schurmann, A., The Application of Graphs to the Analysis of Distribution of Loops in a Program, *Information and Control*, *7*, 3 (1964), 275–282

[6] Voorhees, Edward A., Algebraic Formulation of Flow Diagrams, *Communications of the Association for Computing Machinary*, *1*, 6 (1958). 4–8

[7] W. B. Stelwagon: Principles and Procedures for the Automatic Flow Charting Program FLOW 2, Research Department.

Appendix 1. An example of combining.

A program showed in Fig. 5 is translated into Fig. 6 which is a output from a machine. Standards are showed.

Block number	Principle	Standard
1	1	1, 5
2	1	2, 5
3	1	1, 3
4	1	3
5	2	4
6	2	4
7	1	1, 3
8	1	3
9	1	1, 5
10	1	1, 3, 5
11	1	3
12	1, 2	1, 6
13	1	2, 5

This program is expressed in ∗-notation as follows,

```
   CA                  clear
   TA     A            store
E  ESM                 enter supervisor mode
   ADR    .GET.        address constant
   LAF    B            load
   MF     C            multiply
   AF     D            add
   JTL    SUBA         jump and store link address
   JZF    L            jump on zero
   JMF    F            jump on minus
   LAF    G
   DF     J            divide
   JTL    SUBB
   JAF    K            store
   J      E            jump
F  LAF    I
   DF     J
   JTL    SUBB
   TAF    K
   J      E
L  ESM
   ADR    .RUT.
   HP                  halt and proceed
```

Fig. 5.

```
X START X
    I 1
I A=0 I
    I
X E X
    I 2
II .GET. II
    I 3
I A-REG= (B)X(C)+(A) I
    I 4
II SUBA II
    I 5
* A:0 * =0 ---------------------------- X L X
    I /=0                                  I 13
    I 6                              II .PUT. II
* A:0 * LESS THAN 0 ------ X F X           I 14
    I GREATER THAN 0           I      X HALT X
    I 7                        I 10
I A-REG= (G)/(J) I     I A-REG= (I)/(J) I
    I 8                        I 11
II SUBB II             II SUBB II
    I 9                        I 12
I K= A-REG I           I K= A-REG I
    I                          I
X E X                  X E X
```

Fig. 6.

$$*1*2*3*4*^{2}5*^{2}6*7*8*9\gamma 2*10*11*12\gamma*2*13\Omega 14$$

In this flow-chart,

a variable is expressed by the address in which it is contained.

a subroutine is expressed as ‖　　‖ in which the name of the subroutine is written.

In above $*$-notation, a machine inspects blocks with γ-operand, and it finds that the block 12 may be exchanged for the block 9 by aid of the exchange rule and in the same manner, the block 11 may be exchanged for the block 8.

In such a way, the automatic simplification is done. Furthermore, these blocks may be combined. For example, the blocks 3 and 4 are combined by the standard 'sequence'.
The blocks 5 and 6 are combined by the standard, one after another branch.

Appendix 2. AFCP.

(Automatic Flow-Charting Program)

I describe only phase 4 which is the heaviest burden. The tasks to be done in this phase is divided into four parts.

1) As AFCP reads source cards it combines statements into blocks by aid of the standards 1–6. A block is recorded on a secondary memory as a record. As these records are frequently accessed, they should be recorded on random access files.

2) The name table is generated, that is, all the names that appeared in source statements are recorded in the main memory. Each name corresponds to a block number in which it is included.

3) AFCP decides a location of a block in the main memory.

4) After the preparations of 1–3 have been completed, AFCP expresses the flow of the program by aid of the list-notation. The expression in the main memory is as flollows.

n	Block number
	L_1
	L_2
	L_i

(n — brace spanning the rows below the header)

L_i: Pointer of the next block.
n: The number of degrees of $*$-operator and at the same time, the number of words.

The list-notation is superior to other expressions at the point that it can save the area of the main mamory.

For example, let us consider to express a program that has 10, 000 statements. As three statements are usually combined into a block, about 3. 300 blocks are needed. In the expression in the main memory, as a block usually needs three words, they, after all, take about 10, 000 words. This amount is able to be expressed in an ordinary machine. But if they are expressed in Boolean Matrices, they need about 3300×3300 information units, that is about 10, 000, 000. It is impossible to express them in an ordinary computer.