TWO-STAGE PROGRAMMING: INTERACTIVE OPTIMIZATION AFTER STRUCTURED PROGRAMMING

TAKESHI CHUSHO
Hitachi, Ltd.,
Systems Development Laboratory
Kawasaki, Japan

TOSHIHIRO HAYASHI
Hitachi, Ltd.,
Omika Works
Ibaraki, Japan

Two-stage programming is a solution in order to apply structured programming to a field with severe object efficiency requirements. The methodology proposed in this paper insists that the two conflicting criteria of well-structured and efficiency should be considered separately, and optimization is performed after structured programming. The purpose of two-stage programming is to make programming easier by expanding Dijkstra's principle into "one decision at a time with one criterion". An interactive system, CROPS, is designed so as to support this method. CROPS provides commands for restructuring an initial program, optimizing a well-structured program and partitioning the main memory among program components. Some optimizing commands automatically prove the equivalence of an optimized program and the pre-optimization program, and do not modify a well-structured source program. For other optimizing commands, many verification commands and additional commands are provided and equivalence can be interactively proved by using them. These functions are presented with a sample programming experiment.

1. INTRODUCTION

For the past few years, considerable effort has been made to improve software productivity, reliability and maintainability. This problem has now become even more important because of recent rapid increases in the amount of software produced. Research has been dealing with almost every part of the software development processes and, above all, the greatest attention has been focused on "structured programming".

The meaning of structured programming is not definitely defind but the common essence is to produce a well-structured program. There are well-known techniques for structured programming such as structured coding which restricts control statements within sequence, selection and iteration [1], and top-down development by stepwise refinement [2], [3]. Recently, modularization techniques such as decomposition based on information hiding [4], composite design [5], etc. and abstraction techniques such as CLU [6] have been also investigated.

On the other hand, adequate attention has not focused on the main disadvantage of structured programming, that is, degradation of object efficiency. This is because software productivity has been considered more important than object efficiency in large computer systems. In minicomputer and microcomputer systems, however, this is not necessarily true because the main memory capacity is limited and efficient memory use is very important. Therefore, appling structured programming to such fields, software must be developed with the following two conflicting criteria:

(1) to produce well-structured programs
(2) to produce efficient programs

Two-stage programming is proposed in order to solve this problem. That is, first a well-structured program is developed without considering efficiency and then it is optimized in the second stage. The most important consideration is to maintain the well-structured program in the second stage. Therefore, an interactive system, conversational restructuring, optimizing and partitioning system (CROPS), was designed to support this method. CROPS has the following three functions:

(1) restructuring an initial program
(2) optimizing a well-structured program
(3) partitioning the main memory among program components.

In particular, this optimization is the leading function of CROPS and is examined in detail. The accuracy of two-stage programming and CROPS is confirmed through experiments using a specially developed structured programming language (SPL).

2. TWO_STAGE PROGRAMMING

2.1 Problem of Structured Programming

Recently, the improvement of productivity has become important in fields with severe object efficiency requirements such as system programs and real-time programs. Therefore, the application of structured programming to such fields should result in improved productivity. In this case, however, both the well-structured and efficiency aspects which are often conflicting must be considered at the same time while developing programs. Consequently, programmers are liable to care more about efficiency than about structure and thus structured programming is neglected. For example, although Wirth considered efficiency in each stage when producing a sample program by stepwise refinement [3], most programmers are not prepared to do so. Therefore, an adequate programming methodology is necessary for the effective application of structured programming.

2.2 Optimization after Structured Programming

The simplest solution to the above-mentioned problem is to separate optimizing from structuring. This methodology named two-stage programming in this paper expands Dijkstra's principle [2] of "one decision at a time" into "one decision at a time with one criterion" and makes programming easier.

2.2.1 Structuring stage

In this stage, programmers concentrate on producing a well-structured program from the viewpoints of high reliability, maintainability and documentability.
At the end of this stage, the testing and validation of a coded program should be completed.

2.2.2 Optimizing Stage

In this stage, the program developed in the first stage is modified to satisfy the restrictions of an actual object machine such as the main memory capacity and execution time. The most important consideration at this time is to optimize without the destruction of the well-structured program. If the final structure of the source program is the same as a conventional one which is complex and difficult to understand, maintainability is not improved and maintenance costs, which are now larger than development costs, are not reduced.

The best way to preserve a well-structured program is to not modify the source program but only to alter the object program by giving commands to a compiler. Even if modification of the source program is necessary, it should not be destruction but restruction.

2.3 Interactive System for Two-Stage Programming

Two-stage programming is composed of the following four steps:
 step 1 : initial programming
 step 2 : restructuring
 step 3 : optimizing
 step 4 : partitioning
First, an initial program is produced by using structured programming. In the next step, this program is reviewed and restructured from the same viewpoints as in the first step. Then, general optimizations are applied to this program in the third step. Last, the main memory is partitioned among program components.

The interactive system, CROPS, was designed in order to apply this method to practical use. This system provides commands for the second, third and fourth steps. Restructuring commands are prepared so as to modify the source program. Optimizing commands and partitioning commands are provided for alterations in the object program.

Some optimizing commands automatically prove the equivalence of an optimized program and the pre-optimization program and the other optimizing commands do not. Consequently, the former commands preserve the source program structure but the latter commands modify it. If the source program is not modified in the latter case, the source program function may be different from the function of the optimized object program and it is very difficult to debug the source program on the basis of the execution result of the object program. The system provides verification commands and other additional commands in order to supplement such a defect of the latter commands, and then, equivalence can be interactively proved by using these commands.

3. SAMPLE PROGRAMMING

An experiment was conducted with sample programming for the purpose of confirming the effect of two-stage programming. The sample program was written in structured programming language SPL [7], [8].

3.1 Description Language SPL

The description language SPL was developed for the production of industrial real-time software and is provided with ample and powerful facilities for modularization, stepwise refinement and structured coding. Each program consists of an environment module and a process module. The environment module is composed of declarations of variables, data types, etc. and constructs a tree structure in order to exactly express these scopes. The process module is a set of procedures called functions and is positioned unuer an suitable environment module as its descendant. Consequently, an SPL program becomes a hierarchy of these modules. A sample is shown in Fig. 1.

```
E_A -----------  (A)
       |
       +----- E_B ----------  (B)
       |
       +----- E_C ----------  (C)
       |              |
       |              +----- E_C1 --------  (C1)
       |              |
       |              +----- E_C2 --------  (C2)
       |              |
       |              +----- E_C3 --------  (C3)
       |
       +----- E_D ----------  (D)
                      |
                      +----- E_D1 --------  (D1)
                      |
                      +----- E_D2 --------  (D2)
                      |
                      +----- E_D3 --------  (D3)
                      |
                      +----- E_D4 --------  (D4)
                      |
                      +----- E_D5 --------  (D5)

          E_aa :  an environment module
          (aa) :  a process module
```

Fig. 1. Module hierarchy of sample program

3.2 Sample Program

A paging process analysis program of an SPL compiler was selected as a sample program because this seemed appropriate in size and complexity. The specifications of this program are:
(1) Input data is a sequence of auxiliary memory addresses referenced by the SPL compiler.
(2) Analyses of the LRU property and static frequency distribution of input data.
(3) Comparison of five paging algorithms applied to input data.
The purpose of this analysis program is to find suitable page size and paging algorithm for the SPL compiler.

3.3 Programming Process

The sample program was developed in accordance with two-stage programming composed of four steps. This process is briefly described in this section and the commands used in the restructuring, optimizing and partitioning steps are detailed in Chapter 4.

(1) Initial Programming step
The initial program was developed by stepwise refinement. First, variables and constants referenced from the whole system were declared in environment module E_A and the top level procedure was defined in process module A, as shown in Fig. 2. Then, three functions referenced in A were refined. For example, the second function and its environment module are shown in Fig. 3. The two variables in E_C are referenced in common from three functions referenced in C. The data type PAGE-SEQUENCE in E_A was refined in E_B as follows:

```
  type PAGE_SEQUENCE = array (MAXNO);
```

Such stepwise refinements were repeated and the final module composition of this program is shown in Fig. 1.

```
environment  E_A;
  dcl;
    var  PSIZE,
         NPAGE;
    var  PSEQ:  PAGE_SEQUENCE,
         NDATA;
    const MAXADR = 16384,
          SSIZE  = 64,
          MAXPG  = 256;
    const ENDMK  = -1;
  end;
end  E_A;

process  A (E_A);
  function  PAGING    opt (main);
    for  PSIZE = SSIZE, SSIZE*8  through  PSIZE*2
      repeat
        INITIALIZE PAGING;
        ANALYZE PAGE_SEQUENCE;
        APPLY PAGING ALGORITHMS TO PAGE_SEQUENCE;
    end;
  end  PAGING;
end  A;
```

Fig. 2. Top level modules of sample program

```
environment  E_C (E_A);
  dcl;
    var  LTABLE(MAXPG);
    var  STABLE(MAXPG)  :  (FREQ, PGNO);
end  E_C;

process  C (E_C);
  function ANALYZE PAGE_SEQUENCE opt (open);
    EXAMINE LRU_PROPERTY;
    EXAMINE STATIC REFERENCE FREQUENCY;
    COMPARE LRU WITH STATIC;
  end ANALYZE;
end C;
```

Fig. 3. Sample of second level modules

(2) Restructuring step
The abovementioned initial program was reviewed and modified. The completed program has 12 environment modules and 13 process modules which include 33 functions. The object length is 5910 words and execution time is 377 seconds as shown in Table 1 (a).

(3) Optimizing step
The optimizations of the object length and execution time were performed individually. As a result, the length was reduced by 34% and the time was reduced by 47% as shown in Table 1 (b) and (c).

Table 1. Results of experiments

		object length(w)	ratio to(a)	execution time(sec)	ratio to(a)
a	well-structured program	5910	1.00	377	1.00
b	memory-optimized program	3926	0.66	378	1.003
c	time-optimized program	5927	1.003	200	0.53

(4) Partitioning step
There are various ways to partition the main memory and they depend on the requirements of each program. No particular requirements were assumed in this experiment. Instead all possible cases were examined.

3.4 Discussion of Results

This experiment confirmed the following effect of two-stage programming. The program could be produced easily and quickly in the first stage since the program structure was concentrated on. There were only two logical errors in the first stage program, and moreover, it was easy to find and debug them. In the second stage, the program efficiency was satisfactorily improved though the effect of optimizations depended on each program feature. It is interesting that the execution time of the memory-optimized program and the object length of the time-optimized program scarcely increased as shown in Table 1 because some of optimizations reduced both time and memory.

As for proof of program correctness, it is the most important but the most difficult problem in software engineering. Two-stage programming, however, reduces this difficulty by separating the verification process into two stages as follows:
(1) correctness proof of a well-structured program.
(2) equivalence proof of an optimized program and the pre-optimization program.
The object of the correctness proof is limited to a well-structured program. The equivalence proof is somewhat easier and is semi-or fully-automatically performed by CROPS.

4. BASIC FUNCTIONS OF CROPS

4.1 Restructuring Functions

This section decribes typical commands related to structured programming but except commands such as any text editor has.

(1) MOVE command of declarations
This command is used to move declarations for variables, constants, etc. to the upper environment module when it is necessary to refer to them outside their scopes. The system displays the altered range of their scopes and checks if this command harms the scope of other identifiers of the same name as moved identifiers.

(2) ADD command of functions
One solution to refering to a variable outside its scope is mentioned above. This solution, however, may degrade program reliability because of unnecessary extension of the scope. The second solution is that a new function for the reference to the variable is added to a descendant process module of the environment module which contains the variable declaration. Thus, the variable can be referenced through

this new function. This approach is similar to Parnas's information hiding [4] or Liskov's data abstraction [6]. The ADD command for the function is used in this case.

(3) MERGE or SPLIT command of process modules
An initial program should be reviewed in order to make its module composition easier to understand. That is, the module composition should be improved by increasing individual module strength and decreasing mutual module coupling, which are Myers's measures [5]. The MERGE or SPLIT command of process modules is used in this case. The system examines the necessity of modifying the scopes of variables etc. referenced by a moved module.

4.2 Optimizing Functions

Several typical optimizing functions of CROPS are presented with samples in this section.

(1) SHARE command of variable storage areas
It is recognized that some variables can share their storage areas for memory saving since a dynamic control structure of a hierarchical program is easy to understand. The SHARE command is used in this case. The system examines command correctness by analyzing the module composition and function reference relations. If a command is doubtful, the system displays the reason and demands that the user should confirm its correctness. An automatic sharing command is also provided.

(2) Commands for change of similar parts to sub
In the sample program, there are six variable initializations which are the same descriptions except for their variables. These similar parts are unified to a subroutine for memory saving by the following process:

(i) A new function is defined as follows:

```
function CLEAR(TABLE(MAXPG) ) opt(open);
  var NO;
  for NO = 1, NPAGE
    repeat   TABLE(NO) = 0;
  end;
end CLEAR;
```

(ii) The six similar parts are replaced by references to CLEAR.

(iii) Each reference is temporarily expanded into the body of CLEAR and the expanded part is compared with the pre-optimization program for equivalence proof. If both programs are the same, this optimization is correct.

(iv) The function option of CLEAR is changed from open to subroutine.

(v) The storage option of the variable NPAGE declared in E_A of Fig. 2 is changed to common.

In this process, (i) and (ii) are performed by restructuring commands, ADD and REPLACE, and (iii), (iv) and (v) by optimizing commands, EXPAND, COMPARE and OPTION.

(3) Commands for unification of the same parts
In the following refinement of the third function referenced in A of Fig. 2, four similar functions, (a), (b), (c) and (d), are referenced.

```
function APPLY PAGING ALGORITHMS TO PAGE_SEQUENCE;
  var  I;
  for  I = 1, BLOOP
    repeat    BPAGE = BSIZE(I)/PSIZE;
(a)               APPLY FINUFO;
(b)               APPLY FIFO;
(c)               APPLY FIVE;
(d)               APPLY OPTIMUM;
                  COMPARE RESULTS;
    end;
end APPLY;
```

For memory and time saving, therefore, it is possible to unify the same parts in the bodies of these functions through the following process:

(i) These function references are expanded into their bodies as follows:

```
(a)  INITIALIZE APPLY_FINUFO;
     for  NO = 1, NDATA
       repeat   GET (NO)TH PAGE_NUMBER (PGNO);
                DEMAND (PGNO) WITH FINUFO;
     end;
(b)  INITIALIZE APPLY_FIFO;
     for  NO = 1, NDATA
       repeat  GET (NO)TH PAGE_NUMBER (PGNO);
               DEMAND (PGNO) WITH FIFO;
     end;
(c)  ....
(d)  ....
```

(ii) Four initialization functions are gathered at the top of the expanded program.

(iii) Four for loops are unified.

(iv) Four GET function references are gathered at the top of the loop, and unified as follows:

```
INITIALIZE APPLY_FINUFO;
INITIALIZE APPLY_FIFO;
INITIALIZE APPLY_FIVE;
INITIALIZE APPLY_OPTIMUM;
for  NO = 1, NDATA
  repeat  GET (NO)TH PAGE_NUMBER (PGNO);
          DEMAND (PGNO) WITH FINUFO;
          DEMAND (PGNO) WITH FIFO;
          DEMAND (PGNO) WITH FIVE;
          DEMAND (PGNO) WITH OPTIMUM AT (NO)TH;
end;
```

In this process, EXPAND in (i), MOVE in (ii) and (iv), and DELETE in (iii) and (iv) are used to modify the program. On the other hand, the permutation correctness in (ii) and (iv) is automatically verified by examining the local independence of moved statements. The equivalence proof of for loop unification in (iii) is performed by LOCAL command which examines the local independence of the loop body of each pre-optimization program. The unification of GET function references in (iv) is verified by EFFECT command which examines the side effect of the function.

(4) Commands for unification of functions
It is possible to unify similar functions for memory saving. For example, three functions, DEMAND (PGNO) WITH FINUFO, FIFO and FIVE, are unified by the following process:

(i) Different parts in these function bodies are distinguished.

(ii) The unification function is defined while describing the different parts with compile time if statements (%if) as follows:

```
function DEMAND (PGNO) WITH (ALGOR) opt(open);
        :
        :
  if  T(PGNO).PTR
   then  %if  ALGOR  is  FINUFO   then  ...
                     is  FIFO     then  ;
                     is  FIVE     then  ...
        end;
```

(iii) A sequence of original function references are modified by using a compile time for statement (%for) as follows:

```
%for  ALGOR = FINUFO, FIVE
  repeat  DEMAND (PGNO) WITH (ALGOR);
 end;
```

(iv) The results of temporary compile time execution for this unification function are compared with the pre-optimization program for equivalence proof. If both programs are the same, this optimization is correct.

(v) Compile time statements are converted to ordinary statments, that is, %symbols are deleted.

In this process, COMPARE in (i) and (iv), ADD in (ii), REPLACE in (iii), EXPAND in (iv) and DELETE in (v) are used.

(5) EXTRACT command of common parts in if

It is possible to extract common parts of a then phrase and else phrase behind an if statement for memory saving. For example, the assignment statements TOP = PGNO in the following part of process module C1 are extracted:

```
  if  TOP .EQ. 0
  then   TOP = PGNO;
         BOTTOM = PGNO;
         LSTACK(PNGO) = PGNO;
  else   LSTACK(PGNO) = TOP;
         LSTACK(BOTTOM) = PGNO;
         TOP = PGNO;
  end;
```

The equivalence is automatically verified by examining the local independence of the assignment statement in the then phrase.

(6) Verification commands

The system provides auxiliary commands for verification because there are some optimizing commands for which automatic verification is very difficult. For example, it seems possible to transfer part (a) to part (b) while eliminating the if statement for memory and time saving as follows:

```
for   NO = 1, NPAGE
  repeat  LSUM = LSUM-LTABLE(NO);
          if  NO .NE. 1
 (a)        then  SSUM = SSUM-STABLE(NO-1).FREQ;
          end;
            :
            :
 (b)        SSUM = SSUM-STABLE(NO).FREQ;
end;
```

(a) is equivalent to (b) if the following three predicates are true:

(i) The variable SSUM is not referenced between (b) and (a) in the loop.

(ii) The value of variable STABLE(NO).FREQ does not change between (b) and (a).

(iii) SSUM is not referenced after the execution of this loop.

The truth of these predicates is proved by verification commands.

4.3 Partitioning Functions

The main memory partitioning commands are:

(1) A storage option command specifies a storage allocation of a variable with a option such as local, common, global or bulk.

(2) A function option command such as open, close or subroutine.

Generally, bulk and subroutine will be entered for memory saving, and others for time saving.

5. CONCLUSIONS

This paper proposed two-stage programming method which expanded Dijkstra's principle. That is, a designer concentrates on producing a well-structured program in the first stage, and then, considers efficiency in the second stage.

The interactive system, CROPS, was designed to support this method. The system provides commands for restructuring an initial program, optimizing a well-structured program and partitioning the main memory among program components. The most important consideration of this method is to maintain the well-structured program in the optimizing stage. Therefore, some optimizing commands automatically prove the equivalence of an optimized program and the pre-optimization program in order to conserve the well-structured program. On the other hand, many verification commands and additional commands are provided in order to supplement the other optimizing commands and equivalence is interactively proved by using these commands.

An experiment using this method confirmed the simplifications in programming, debugging and verfication resulting from this system. Further study is needed to increase the volume of optimizing commands which automatically prove equivalence.

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to H. Takai for providing the opportunity to conduct this study. They are also indebted to Dr. I. Nakata and K. Nogi who developed SPL with the authors, for their invaluable technical discussions.

REFERENCES

[1] E.W. Dijkstra, GO TO statement considered harmful, Communications of the ACM, vol. 11, no 3, March 1968, 147-148.

[2] E.W. Dijkstra, Notes on structured programming, Structured programming, Academic Press 1972, 1-82.

[3] N. Wirth, Program development by stepwise refinement, Communications of the ACM, vol. 14, no. 4, April 1971, 221-227.

[4] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, vol. 15, no. 12, December 1972, 1053-1058.

[5] G. J. Myers, Reliable software through composite design, Petrocelli Charter, 1975.

[6] B. Liskov and S. Zilles, Programming with abstract data types, SIGPLAN Notices, vol. 9, no. 4, April 1974, 50-59.

[7] T. Hayashi, K. Nogi, T. Chusho, et al., Structured programming language SPL for control computers, Proceedings of the 17th anual conference of IPSJ (Japan), November 1976, 1-8.

[8] T. Hayashi, K. Nogi, I, Nakata, et al., Top-down structured programming language for real-time computer systems - SPL, Hitachi review, vol. 26, no. 10, October 1977, 333-338.