Vol. 28 No. 4 情報処理学会論文誌 Apr. 1987

制御フロー解析による PASCAL プログラム計量システム†

有沢 誠†† 張 清利††

性能コンパイラは，Pascal サブセットで書かれたプログラムを，制御フロー解析を行うことによって，空間コストおよび時間コストの形で性能計量を行う道具である．プログラム中の独立な径路ごとにその部分の実行回数を表すフロー変数を，システムが機械的に割り当てる．またそれら実行回数を定めている要因を表すフローパラメータを人間があてはめることによって，プログラムの計量が行える．ある仮想機械上で必要となる空間および時間の形で計量値を計算したあと，現実の機械から仮想機械への換算の形で最終的性能を示す値を計算する．もともと Connecticut 大学で開発されたシステムを，全面的に検討しなおして，より強力で柔軟性のある新しいシステムとして，システム構成の多段階分割，対話型システムの実現，扱えるサブセットの拡大などを含むように開発した．Pascal プログラムの性能の計量だけでなく，複雑度の計量にも使用できるように作ってあり，広くソフトウェア計量支援システムとして使用することをめざしている．

1. はじめに

ソフトウェア工学の分野で取り組むべき問題のひとつは，ソフトウェアの性能や複雑さ，さらにソフトウェアの品質を定量的に計量するための，尺度や支援システムを開発することである．筆者たちの研究グループでは，いくつかのアプローチでこの問題に取り組んでいる．本論文では，言語を Pascal サブセットに限定し，制御フロー解析によってプログラムの性能や複雑度を計量するシステムの開発について報告する．

制御フロー解析にもとづいてプログラムの性能を計量するアイディアは，Connecticut 大学の性能コンパイラとよばれるシステムからとった[6]．以下このシステムを，「Connecticut 版性能コンパイラ」あるいは「Connecticut 版システム」とよぶ．本論文で報告するシステムは，Connecticut 版システムに大幅な改良を加えた外見をしており，システムそのものは独自に設計して作った[1),2)]．以下このシステムを，「山梨版性能コンパイラ」あるいは「山梨版システム」とよぶ．

Connecticut 版システムの目的は，小さな Pascal サブセットで書いたプログラムに，制御フロー解析をたすけるいくつかの追加情報を書き加えたものを入力データとして与えてやると，ある仮想的な機械上でそのプログラムが占める空間の量と，そのプログラムを実行したときに要する時間の量を計算して出力することである．前者を空間コスト，後者を時間コストとよぶ．これはプロファイラの考えかたに似てはいるが，プロファイラが具体的なデータを与えた場合にプログラムを実行して計量値を求めることに比べて，性能コンパイラでは実際にプログラムを実行せずに空間コストと時間コストが求まり，大きなプログラム群の開発の際に，部分的に完成したプログラムを簡単に計量できることが特徴である．

Connecticut 版システムがアイディアの実証のためのプロトタイプとして作られていることに比べて，山梨版システムでは，現実のプログラムの計量値を定量的に与える実用システムに拡大することを目的としている．プログラムの空間コストや時間コストだけでなく，制御フロー解析によって得られた情報から，Halstead[3] や MaCabe[5] による複雑度に相当する計量値も計算できるようにしている．

性能コンパイラによって，プログラム本体を実行せずに，特定の性格をもつデータを与えたときの実行時間の推定値を求めることができる．データの特性を変えながら実行時間（の推定値）の変化を調べることも可能である．このようなプログラム計量支援システムを，ソフトウェア開発支援環境の一部分にとりいれておくことは，きわめて意義が大きい．

本論文では，まず二つのシステムに共通な部分について山梨版システムでの実例にそって述べ，ついでに山梨版システムだけがもつ機能や性格について述べる．

2. 性能コンパイラのはたらき

性能コンパイラという名称は，プログラムの空間コ

† Pascal Program Measuring System by Control Flow Analysis by MAKOTO ARISAWA and QINGLI ZHANG (Department of Computer Science, Faculty of Engineering, Yamanashi University).

†† 山梨大学工学部計算機科学科

```
 1 (straight insertion sort program)
 2 program sinsort(input,output);
 3 perf(p)
 4     e3 = p;
 5     e4 = p;
 6     f1 = p;
 7 const
 8     max = 20;
 9 type
10     index = 0..max;
11     item = packed array[index] of integer;
12 var
13     n, m: index;
14     x: item;
15 procedure strght(var a: item; k: index);
16     perf(q)
17         e1 = q - 1;
18         e2 = (q*q + q - 2)/4;
19     var
20         i, j: index;
21         w: integer;
22     begin
23     for (e1: i := 2 to k) do
24         begin
25         w :=  a[i];
26         a[0] := w;
27         j := i - 1;
28         while (e2: w < a[j]) do
29             begin
30             a[j+1] := a[j];
31             j := j - 1
32             end;
33         a[j+1] := w
34         end
35      end;
36 begin
37 readln(n);
38 for (e3: m:=1 to n) do
39     readln(x[m]);
40 strght(f1: x,n);
41 for (e4: m:=1 to n) do
42     writeln(x[m])
43 end.
44 1
45 8
```

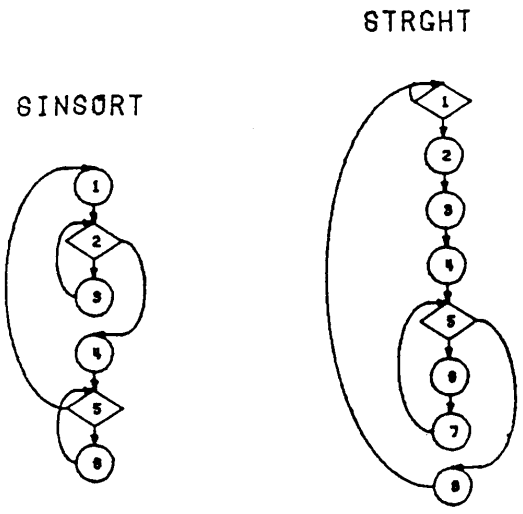
図 1 サンプルプログラム
Fig. 1 Sample program.

ストおよび時間コストを性能とみなし，またそのために入力プログラムの字句解析や構文解析などコンパイラに近い処理を行っていること，の二点から Connecticut 大学で命名したものである．入力として，**図1**のような Pascal プログラム(この例は直接挿入ソート)を与える．ここで perf 文で指定した p(3行目) や q(16行目) をフローパラメータとよび，for 文や while 文に追加した $e1$ (23 行目)，$e2$ (28 行目)，…などをフロー変数とよぶ．フロー変数はそれが書かれている部分が実行される回数を表す．たとえば $e1$ は 23 行目の for 文の実行回数を表す．プログラムの流れをグラフの形にしたとき，グラフとして独立なループごとにそれぞれのループをまわる回数を表すようなフロー変数をひとつずつ用意する．プログラムの末尾と先頭をむすぶ仮の道を作ることによって，プログラム全体もひとつのループとみなす．プログラム中の独立な径路は，これらループの組み合わせ（一次結合）で表現できる[6]．一般には個々のループの実行回数は互いに独立ではなく，プログラムに与えたデータによって定まるようなある量に依存している．フローパラメータはこの量を表しており，perf 文の直後にフロー変数とフローパラメータの関係式を書いておく．たとえば $e1=q-1$ (17 行目) という関係式は，23 行目の for 文の実行回数がフローパラメータ q の値より 1 だけ小さいことを示す．フローパラメータの値はプログラムとは別に定数の形で与える．(この例では 44〜45 行目に $p=1$, $q=8$ を与えている．)

制御フロー解析の結果は，グラフの形で図示したい．山梨版システムでは，グラフを描くための**図2**のようなデータを出力するオプションがあり，別に用意したグラフをプロッタに描くプログラムによって，**図3**のような図を得ることができる．(Connecticut 版システムにはこのオプショ

```
Performance Compiler Version 85
   1 (straight insertion sort program )
   2 program sinsort(input,output);
   ⋮
  43 end.

Cost flow graph

Module Name ----- siosort

vertex                                    control   vertex
  from fan_in fan_out line_no or dummy    to

   1      1      1      37                 2
   2      2      2      38      for        4     3
   4      1      1      40                 5
   5      2      2      41      for        1     6
   6      1      1      42                 5
   3      1      1      39                 2

Module Name ----- strght

vertex                                    control   vertex
  from fan_in fan_out line_no or dummy    to

   1      2      2      23      for        1     2
   2      1      1      25                 3
   3      1      1      26                 4
   4      1      1      27                 5
   5      2      2      28      while      8     6
   8      1      1      33                 1
   6      1      1      30                 7
   7      1      1      31                 5
```

図 2 制御グラフを書くためのデータ出力例
Fig. 2 Sample output for control flow graph drawing.



a) 主プログラム
Main program.

b) strght 手続き
Strght procedure.

図 3 制御グラフの例
Fig. 3 Sample control flow graph.

表 1 仮想機械の基本演算命令一覧表
Table 1 Instruction set for the virtual machine.

No.	名　前	省略表記	意　味
1	variable dereference	vari-dref	変数参照
2	record dereference	rec-dref	レコード型変数参照
3	dimension dereference	dim-dref	配列型変数参照
4	literal dereference	lit-dref	リテラル参照
5	integer add	int-add	整数の加算（+, −）
6	real add	real-add	実数の加算（+, −）
7	set add	set-add	和集合演算（+）
8	integer multiplication	int-mul	整数の乗算（∗, div）
9	real multiplication	real-mul	実数の乗算（∗, /）
10	set multiplication	set-mul	積集合演算（∗）
11	integer relation	int-rel	整数の関係演算（注）
12	real relation	real-rel	実数の関係演算
13	character relation	char-rel	文字の関係演算
14	enumerated relation	enu-rel	順序の関係演算
15	integer sign	int-sign	整数単項演算（+, −）
16	real sign	real-sign	実数単項演算（+, −）
17	and	and	論理演算 and
18	or	or	論理演算 or
19	not	not	論理演算 not
20	up-downto	up-downto	for 文のコントロール変数参照
21	call procedure	call-proc	手続き呼出し
22	call function	call-func	関数呼出し
23	assignment	assign	代入

注）関係演算子 〈, 〉, 〈=, 〉=, 〈〉, =

ンは用意されていない.）

性能コンパイラは制御フロー解析を終えると，プログラム中で用いている変数と命令のそれぞれの量を，種類別に求めて出力する．変数は，整数型，実数型，論理型，文字型，列挙型の5種類に分けて，何個ずつ現れたかを出力する．命令は Pascal の文や式に現れる諸機能を**表 1** に示す 23 通りの基本命令の組み合わせに分解した上で，それぞれの個数を出力する．5種類のデータ型と 23 種の基本命令群は一つの仮想機械を構成しているとみなすことができる．すなわち，性能コンパイラがここまで計算してきたことは，入力した Pascal プログラムをこの仮想機械の機械語にコンパイルした上で，空間コストや時間コストを計算していることに相当する．

空間コストについては，5種の変数と 23 種の命令がそれぞれ何回現れていたかを求める．時間コストについては，プログラム中の独立な径路（個々のフロー変数が代表しているループの組み合わせで表せる）ごとに，23 種の命令がそれぞれ何回現れたかを求める．このとき，手続き呼び出し（図1の例では 40 行目の strght）については，基本命令 21 番が1回現れており，これは呼び出し前後の処理に相当する．呼び出された手続き本体の時間や空間コストについては，別に計算しておいてあとで結合する．ただし readln や writeln などの標準手続きは，多くの Pascal コンパイラが生成する機械語プログラムではインライン展開されることを仮定して，一般の手続き呼び出しには含めず，特別にその数を計算している．以上の計算結果と記号一覧表は**図 4** のような形で出力する．

仮想機械上で，変数および基本命令の数がそれぞれの種類ごとにヴェクトルの形で求まったら，次のようにして空間コストと時間コストを計算する．空間コストについては，各変数や命令が何バイトずつ占めるか（たとえば整数変数は2バイトを占め，変数参照命令は4バイトを占めるなど）をヴェクトルの形で与えてやって，両者の内積をとれば空間コストがスカラ値で求まる．

時間コストについても，独立な径路ごとに，その径路に現れる基本命令の数をヴェクトルにしておき，その径路を通る回数（スカラ）をかけてやると基本命令ごとの実行回数が求まる．さらに各基本命令1回の実行に要する時間（たとえば変数参照命令は 2 μs など）をヴェクトルの形で与えて内積をとり，最後に各径路ごとの和をとれば，プログラム全体の時間コストがスカラ値で求まる．

この過程で，各変数や命令に必要なバイト数を表すヴェクトルや，各命令1回の実行に要する時間のヴェ

```
Performance Analysis Summary
Export List
Summary of Scope - sinsort

 Correspond Table ---  Primitive Operation


Space Cost

 local data -   (   23     0     0     0     0)
                (  int real bool char enum)
 Primitive Operation
            5         10        15        20
   ( 7 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0)
    READLN      =  2   WRITELN   =  2
    + strght     < 2: 6>

Time Cost

 Cycle  1
 Flow -  1
 Cost -
            5         10        15        20
   ( 5 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0)
    READLN      =  1
    + strght     < 2: 6>     parameter =   p

 Cycle  2  Line_no =  41  Loop_name = for
 Flow -  p
 Cost -
            5         10        15        20
   ( 2 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
    WRITELN     =  2

 Cycle  3
 Flow -  p
 Cost -
            5         10        15        20
   ( 2 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
    READLN      =  1

Symbol Table
       x              variable
       m              variable
       n              variable
       f1             flow variable
       e4             flow variable
       e3             flow variable
       p              flow parameter
       strght         procedure
```

図 4 性能コンパイラの出力例
Fig. 4 Sample output from the performance compiler.

クトルをとりかえることによって，複数の仮想機械上での空間コストや時間コストを求めることができる．仮想機械の代わりに実際の機械上での値をここで扱っている仮想機械上での値に適切に変換して，それを用いて計算することにより，実際の機械上での空間コストや時間コストが推定できる．

また各径路を通る回数は，フローパラメータの値と perf 文で与えた関係式に依存しているため，それらを少しずつ変えてやって，時間コストの変化を見ることもできる．

山梨版システムでは，図 4 の形の出力を得たあと，以上に述べた計算を 3 段階に分けて行うようにしてある．まず初めの段階（性能リンカと名づけている）では，空間コストについては，主プログラムと手続きや関数の占める変数や命令の個数の合計を計算して，ひとつのヴェクトルにまとめる．時間コストについては，独立な径路ごとに，変数の個数を別々のヴェクトルにしておく．性能リンカ実行終了までの結果が**図 5** である．

次の段階（性能インタプリタと名づけている）では，空間コストの計算は何も手を加えない．時間コストのほうは，フローパラメータの値と，perf 文で定めた関係式からフロー変数の具体的な値を求め，各命令ごとに実行回数が何回かをあてはめ，それらを主プログラム，手続きおよび関数について和をとったものを求める．この結果，各基本命令ごとに何回ずつ実行するかという形のヴェクトルが得られる．

ただしフローパラメータとフロー変数の関係式は，必ずしも理論的に厳密なものを perf 文で与えることができるとは限らないため，平均的なふるまいを表す関係式で代用することがある．図 1 のプログラムで図 18 行目の $e2=(q2+q-2)/4$ を与えたのがそのような例である．このとき，本来は整数であるべき命令の実行回数に，小数が現れることがある．性能インタプリタ実行終了までの結果が**図 6** である．11 番目の基本命令（整数の関係演算）の実行回数が 24.5 回と整数でない値になっている．

最後の段階（性能バインダと名づけている）で，仮想機械の変数や命令が占めるバイト数のヴェクトルと，各命令を 1 回実行するために必要な時間のヴェクトルを与えて，内積をとって最終的な空間コストと時間コストを計算する．性能バインダ実行終了での結果が**図 7** である．

3. 山梨版性能コンパイラの特徴

制御フロー解析によって Pascal プログラムの性能を評価するアイディアは，Connecticut 版性能コンパイラからそのまま山梨版性能コンパイラに引き継がれている．しかし，Connecticut 版はきわめて小規模な Pascal サブセットのみを扱い，たとえば，表 1 に示した基本命令のうち 8 種だけしか含めていない．山梨版システムでは，対象とする Pascal サブセットを拡げ，

```
Summary of Performance Linker

 Space Cost
 local data - (   26    0    0    0    0)
 Primitive Operation
   (21 0 8 8 4 0 0 0 0 0 0 1 0 0 0 0 0 0 0 3 1 0 6)
    READLN      =  2  WRITELN     =  2

 Time Cost
    Flow -  1
    Cost -  ( 5 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0)
    +strght     <  2:  6>   p
                                  -->
                                     q
       Flow -  1
       Cost -  ( 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
       Flow -  q - 1
       Cost -  (10 0 4 4 2 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 4)
       Flow -  (q*q + q - 2)/4
       Cost -  ( 6 0 3 2 2 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 2)

    Flow -  p
    Cost -  ( 2 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
    Flow -  p
    Cost -  ( 2 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
```

図 5 性能リンカの出力例
Fig. 5 Sample output from the performance linker.

```
Summary of Performance Interpreter

 Space Cost
 local data - (   26    0    0    0    0)
 Primitive Operation
  (21 0 8 8 4 0 0 0 0 0 1 0 0 0 0 0 0 0 0 3 1 0 6)
   READLN      =  2  WRITELN     =  2

 Time Cost
 Flow Parameter of Main Program
    p =        8.00
(213.00   0.00 96.50 82.00 49.00   0.00   0.00   0.00   0.00   0.00
  24.50   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00 26.00
   1.00   0.00 63.00)
   READLN     =    9.00  WRITELN    =   16.00
```

図 6 性能インタプリタの出力例
Fig. 6 Sample output from the performance interpreter.

入門用の教科書に現れる Pascal プログラムのかなりの割合までを受け入れるようにした．それでも，現時点の Pascal サブセットはポインタやファイルを除外するなど，まだ拡張の余地が残されている．

山梨版システムではまた，性能コンパイラから，性能リンカ，性能インタプリタ，性能バインダ，を独立させて，システム使用者が途中の出力結果を見ながら，部分的に入力を変更して再試行できる形にした．そのため，バッチ処理の Connecticut 版システムと異なり，山梨版システムは対話型にしてある．システムとの対話を状態遷移図の形で示したものが**図 8** である．

与えられた Pascal プログラムに対して，独立なループごとに一つずつフロー変数を割り当てることは，機械的に行える．図3に示したような制御フローグラフで，ある節点にはいる矢印を通った合計とその節点から出る矢印を通った回数の合計が等しい性質（フローバランスの性質）を利用すればよい．Connecticut 版システムではフロー変数の割り当てを人間が行っていたが，山梨版システムでは，これを前処理プログラムの形で実現している．このことは性能コンパイラのようなフロー解析支援システムでは本質的であり，フロー変数の自動割り当てによってずっと使いやすいシステムになった．

一方フローパラメータの選択と perf 文による関係式を定めることは，プログラムの構造を見ただけではできず，プログラムの意味論にまで立ち入る必要がある．したがって perf 文の設定は人手で行い，必要に応じて何度も設定し直す．

一般に，フロー変数もフローパラメータも与えていない普通の Pascal プログラムをまず前処理プログラムに与えると，自動的にフロー変数が $e1, e2, \cdots$ のよ

```
Summary of Performance Binder

Total Space Cost
local data
Primitive Type          Count   Bytes   Total_Bytes
integer                    26       2            52

Total                      26                    52

Primitive Operation Cost
Primitive Operation     Count   Bytes   Total_Bytes
vari_dref                  21       4            84
recd_dref                   0       4             0
dim_dref                    8       4            32
lit_dref                    8       2            16
int_add                     4       2             8
  ⋮
up_down_to                  3       2             6
call_proc                   1       2             2
call_func                   0       2             0
assign                      6       4            24
READLN                      2      20            40
WRITELN                     2      20            40

Total                      56                   254

Total Time Cost
Primitive Operation     Count   Weight  Total_Weight
vari_dref              213.00        1        213.00
recd_dref                0.00        3          0.00
dim_dref                96.50        3        289.50
lit_dref                82.00        1         82.00
int_add                 49.00        2         98.00
real_add                 0.00        4          0.00
  ⋮
up_down_to              26.00        2         52.00
call_proc                1.00        5          5.00
call_func                0.00        5          0.00
assign                  63.00        1         63.00
READLN                   9.00       10         90.00
WRITELN                 16.00       10        160.00

Total                  580.00                 1101.50
```

図 7 性能バインダの出力例
Fig. 7 Sample output from the performance binder.

うに順次番号づけして書き込まれる．同時に，フローパラメータの部分が空の perf 文が生成され，その直後に各フロー変数ごとに $e1=\cdots$ のような右辺が空の関係式が生成される．

この状態で，フローパラメータおよび関係式の設定モードにはいり，perf 文と関係式の右辺を対話形式で補う．ここまでで，図1のような形の入力データができあがる．以下，性能コンパイラ，性能リンカ，性能インタプリタ，性能バインダの順にプログラムを実行する．途中の出力結果を見た段階で打ち切りにして，フローパラメータおよび関係式の設定モードへ戻ることもできる．

Pascal プログラム本体に修正を加えたい場合は，いったん山梨版性能コンパイラシステムから抜け出し，エディタで Pascal プログラムの修正をしてから，再びシステムにはいって前処理プログラムから実行する．また Connecticut 版システムを使用する場合のように，あらかじめ図1のような形の入力データをエディタで作成しておき，前処理プログラムとフローパラメータおよび関係式設定モードをスキップして，すぐ性能コンパイラ本体のフロー解析以下を実行することも可能である．

Connecticut 版システムは，プログラムの空間コストおよび時間コストを求め，それによってプログラムの性能を計量するためのプロトタイプの実現を目的としていた．しかし，山梨版システムでは，Pascal 入門教育の水準での実用化をめざし，さらに制御フロー解析に基づくプログラム複雑度の計量にも使用することをねらっている．実際，独立な径路ごとに一つずつフロー変数を割り当てることにより，最終的に何個のフロー変数を必要としたかの個数は，McCabe の複雑度を表している．

また，性能バインダで各命令を1回実行するために要する時間をヴェクトルの形で与えて内積をとっているが，命令ごとに別の意味づけを持つ重みをヴェクトルの形で与えて内積をとれば，Halstead 尺度に似た複雑度を表すものとみなせる．本来の Halstead 尺度は静的な複雑度尺度であり，各命令の実行回数はすべて1とした場合にあてはまる．ここでは，各命令の実行回数にも依存する動的な複雑度尺度を計算できる．

もし必要なら，フローパラメータの値の設定を変えて，各命令が1回ずつ実行されるようにできるから，現在の山梨版システムでも一般性を失っていない．ただしこうして得られる尺度は，Halstead 尺度そのものではなく，プログラム中に出現する基本命令の分布によってプログラムの複雑度を計量する新しい尺度である．実際にどのような重みを与えてどう計量値を計算するかは，現在研究中である．本論文は，支援システム作成の報告にとどめ，複雑度計量にはたちいらない．

4. おわりに

山梨版性能コンパイラシステムは，山梨大学計算機科学科 MELCOM COSMO 800 Ⅲ に，Pascal 8000 を用いて作成した．システムの拡張として，より大きな Pascal サブセットを受け付けるために機能を増すことと，フローパラメータを適切に設定するための補助情報を収集する機能をもたせることが考えられる．Connecticut 大学では，主に後者に研究をしぼり，言

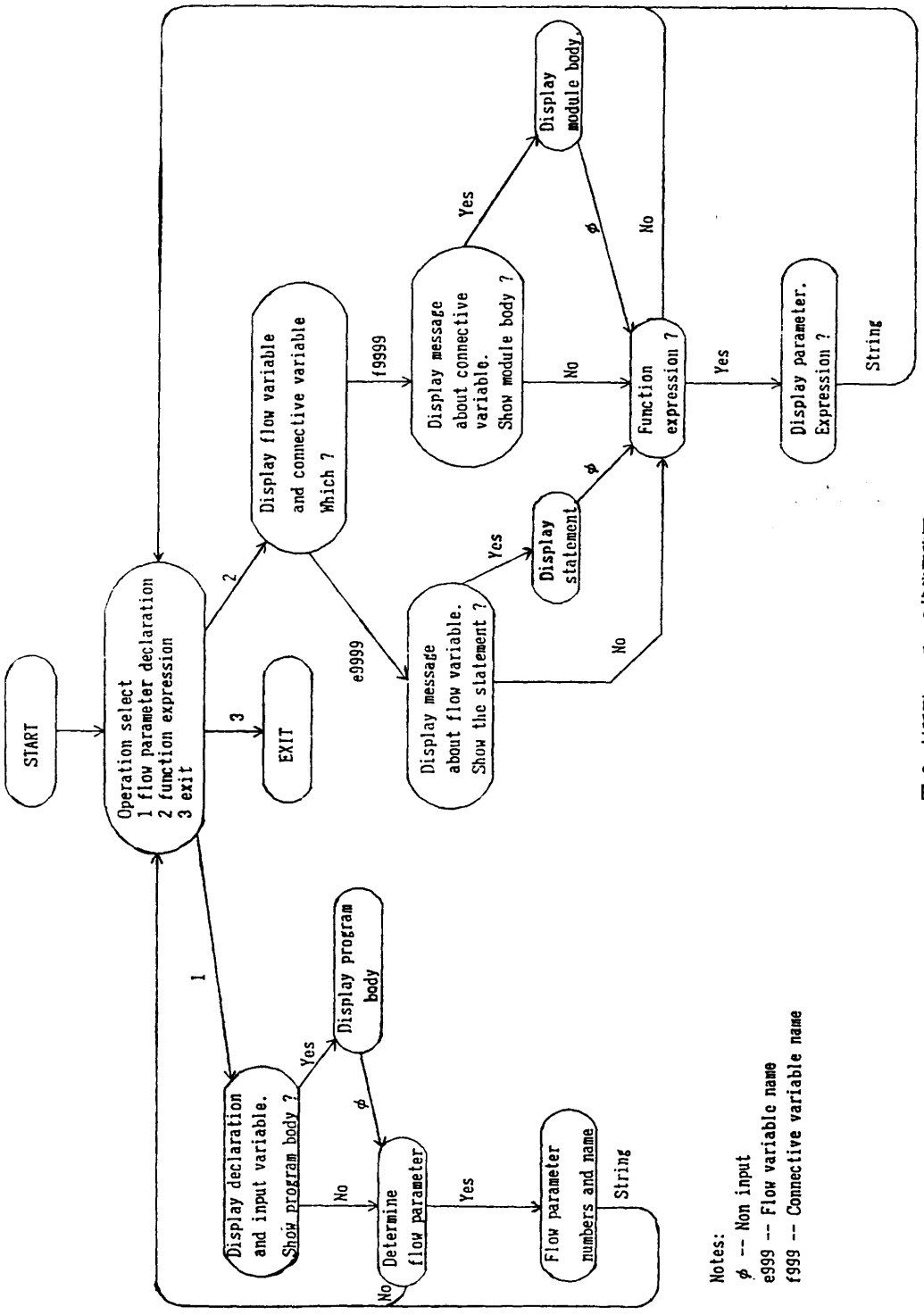


図 8 対話型システムの状態遷移図
Fig. 8 State transition diagram for system dialogue.

語は Pascal からC言語に変えたという[4]．筆者たちの今後の研究は，むしろこのような支援システムを用いて，Halstead 尺度に近いプログラム複雑度の尺度で，よりよいものを追求する方向に主力を注いでいきたい．

謝辞 本研究を支援していただいた Connecticut 大学の故 T. L. Booth 教授，および山梨大学の井内稔技官，古屋正樹君，三橋聡君はじめとする有沢研究室の諸氏に感謝いたします．また査読委員からの親切なコメントによってよりわかりやすい形にまとめることができたことを感謝します．

参 考 文 献

1) 張 清利，有沢 誠ほか：プログラム評価トゥールの開発（II），第28回情報処理学会全国大会論文集，4 B-2 (1984)．
2) 張 清利，有沢 誠：対話型ソフトウェア評価支援システム，第 31 回情報処理学会全国大会論文集，6 Q-4 (1985)．
3) Halstead, M. H.: *Elements of Software Science*, Elsevier North Holland, Amsterdam (1977)．
4) Lenk, R. M.: Measurement of Software Performance, Technical Report, Univ. of Connecticut (1981)．
5) McCabe, T. J.: A Complexity Measure, *IEEE Trans. Softw. Eng.*, Vol. 2, No. 4, pp. 308-320 (1976)．
6) Wetmore IV, T. T.: The Performance Compiler—A Tool for Software Design, Technical Report, Univ. of Connecticut (1980)．

(昭和61年9月10日受付)
(昭和62年1月14日採録)

有沢 誠（正会員）

1944年生．1967年東京大学工学部計数工学科卒業．電子技術総合研究所を経て，現在山梨大学工学部計算機科学科に勤務．工学博士．ソフトウェア工学，特にソフトウェアの評価，アルゴリズムの解析，オブジェクト指向システムなどに興味をもっている．

張 清利（学生会員）

1960年生．中国黒龍江省出身．中国政府国費留学生として来日．1984年山梨大学工学部計算機科学科卒業．1986年同大学院工学研究科修士課程修了．現在山梨県内で研修中．ソフトウェア工学の諸分野に興味をもっている．