プログラム解析システム——PANSY*

宇都宮公訓**　諸橋　正幸**　門倉　敏夫**

Abstract

PANSY (Program ANalysis SYstem) is a debugging tool for assembler users. It dynamically traces a given program, then generates a flow description and cross reference tables as traced results. The flow is described in a FORTRAN-like language, so PANSY could be said to be a program which consits of a dynamic tracer and a reversecompiler. There are four kinds of cross reference tables, branch, modified instruction, single data and array data cross reference tables. This paper discusses specifically the tables which store the results of trace, and reverse-compiling techniques, especially the ways how to detect DO-loops and anlyze register use.

1. まえがき

プログラムの初歩的なコーディングは文法エラーとして機械的に検出されるが，プログラムロジックの正しさは実際にデータを与え run させて調べる以外に知る方法はない．

このような test run においてバグの存在が検出されると，プログラマは，フローチャート，ソース（もしくはアセンブリ）リストそれに test run の結果を用いてデバグする．コーディングをフローチャートに変換する技術が開発[1),2),3)]され，最近では商用されているもの[4)]もあるのでそれを利用することもできるし，dump をとって利用することもできる．しかし，この種のフローチャートや dump は static な情報であり，プログラムの run の様子をそっくり反映してはいない．

一方，アセンブラ言語のユーザには，tracer[1)]という強力な debugging tool が用意されている．これは trace と dump を dynamic に行なうが，出力されるデータの量が膨大で真にデバギングに必要な情報とそうでない情報を区別しにくいという使用上の難点がある．

著者はアセンブラ言語でかかれたプログラムを対象に，

（1） 機械的に収集可能なデバギングのためのデータはすべて生成し，デバギングに便利な形に整理することを第一の目的とし，

（2） ロジックが知られていないプログラムの解読のための資料を提供することも考慮して，プログラム解析システム——PANSY (Program ANalysis SYstem) を考えた．

2. 基本設計

1節で掲げた2つの目的を達成するため，PANSY は基本的にはどのように設計されるかを述べる．

2.1 構造と動作

PANSY は，本質的には図 2.1 のように 3 つの pass からなっている．Pass 1 では，被解析プログラムを trace してその結果をいくつかのテーブルに記録する．

Pass 2 では，Pass 1 で得られたテーブルをもとにフロー解析，データ解析を行なって，被解析プログラムの run の様子をコンパイラ・レベルの言語で記述した dynamc flow description coding を出力する．このプログラムは PANSY の中枢であり，逆コンピレーションと呼んでいる．それゆえ，PANSY は，いわば，拡張された逆コンパイラ[6),7)]である．さらに，使用されたデータについては，単一データと配列データにわけて，dynamic cross reference table として出力する．branch 命令による address 参照，変更された命令もそれ専用の cross reference table として出力する．

Pass 3 では，dynamic flow description coding を

* A Program Analysis System——PANSY, by K. Utunomiya, M. Morohashi and T. Kadokura (Scientific and Engineering School of Waseda University)

** 早稲田大学理工学部電気工学科

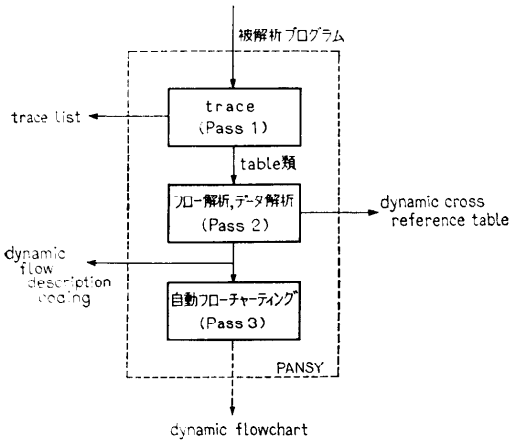

図 2.1 PASYS の構造と動作

dynamic flowchart に変換する．アセンブラ言語やコンパイラ言語でコーディングされたプログラムをフローチャートに変換することは，デバギングに実用するという観点からは，

（1） ライプリンタにかかせたフローチャートは大きく拡がり過ぎてフローを追いにくい，

（2） curve plotter にフローチャートを描かせればコストが高くなるし，curve plotter を常時使えるユーザも少ない，

（3） 被解析プログラム中で命令が変更されているとき，その記述が難しい

という難点があり，必ずしも最善の策ではない．したがって，PANSY では dynamic flow description coding を最終結果とすることを標準とする．

本論文では Pass 1 と Pass 2 だけを議論する．

2.2 Dynamic Flow Description Language

PANSY ユーザは FORTRAN 語に習熟していることが予想されるので，FORTRAN-like な Dynamic Flow Description Language (DFDL と略称する) を定義して，これで dynamic flow を記述する．DFDLは FORTRAN 語の機能をすべては必要としないが，一方，アセンブラ言語の細かな記述を表現できなければならない．

DFDL 文は文ラベルと文ボディからなっており，つぎの 15 種類がある．

（1） DIMENSION 文
（2） DO 文
（3） CONTINUE 文
（4） 算術 IF 文
（5） 論理 IF 文
（6） ASSIGN 文
（7） 割当て型 GO TO 文
（8） 計算型 GO TO 文
（9） GO TO 文
(10) 代入文
(11) EXTERN 文
(12) MODIFY 文
(13) PERFORM 文
(14) 特殊関数文
(15) アセンブラ文

（2）～(10)は FORTRAN 語におけるそれと同ど機能をもつが，（5）は overflow indicator や sense switch のテストの記述にも拡張して用いる．

(11)は外部 routine を call する命令の記述に用いる．

(12)と(13)は，それぞれ命令を別の命令で変更するとき，変更された命令を実行するときの記述に用いる．

(14)は現在のところ shift 命令に対するものしか考えていない．すなわち，shift 演算は特殊関数とみなして記述する．

(15)は概念的にどうしてもまとめられないアセンブラ命令をそのまま記述した文である．

最後に，（1）であるが，これは意味をかなり変えてしまっている．3.3 節で述べる．

data の属性に関する宣伝は DIMENSION 文によるものを除いては一切行なわず，cross reference を参照してもらうようにする．

2.3 Dynamic Cross Reference Table

dynamic cross reference は

（1） 単一データ，
（2） 配列データ，
（3） branch 命令，
（4） 変更された命令

の4つのカテゴリに分けて出力する．

単一データの cross reference table では，参照されたデータの address のあとに，そのデータを参照した命令の address が印刷される．さらに，参照されたデータは，

（ⅰ） 変数か，定数か，
（ⅱ） 命令として実行されたか，
（ⅲ） 特殊データ(たとえば program status word) か

をチェックされ，その結果も印刷される．データを参

照する命令は，その種類に応じて

（ⅰ） 論理データ，
（ⅱ） 固定小数点2進数，
（ⅲ） 固定小数点 10 進数，
（ⅳ） 浮動小数点（2進）数，
（ⅴ） 文字，
（ⅵ） それ以外

の別を調べられ印刷される．

配列データの cross reference table は，サイズ情報と参照するときに用いた index register の番号が付加えられることを除けば単一データの場合とまったく同様である．サイズ情報とは，使用された index register の最大値，最小値およびそれらの差である (3.3 節参照)．

branch 命令の cross reference table では，branch 先の address を先頭に印刷し，そのあとに，その address に branch した（または，branch することが指定されていた）命令の address を印刷する．branch 命令では次の3項目をチェックし，その結果を address のすぐあとに（ ）で括って印刷する．

（ⅰ） 実際に branch したかどうか．
（ⅱ） 3方向以上の branch 命令か．
（ⅲ） 繰返しを前提とした branch 命令（たとえば，IBM System/360 の branch on count 命令）か．

被解析プログラムの run 中に変更される命令は，変更された命令の address，変更結果，変更を施した命令の address をこの順に並べて出力する．

3. Pass 1 の詳細

Pass 1 のフローチャートを図 3.1 に示す．Pass 1 から Pass 2 に送られるテーブルは次の6つである．

（1） instruction table
（2） trace table
（3） branch table
（4） data table
（5） array table
（6） run information table

3.1 Trace Table と Instruction Table

trace table では，被解析プログラムの1命令 address あたり8ビットからなるエントリが対応している．はじめの4ビットは，それぞれ

（1） この address の命令は trace されたか，
（2） この address の命令は branch 命令か，
（3） この address の命令は変更されたことがあるか，
（4） この address は branch 命令からの entry point になっているか

を表わしており，残りの4ビットは Pass 2 で使用する．

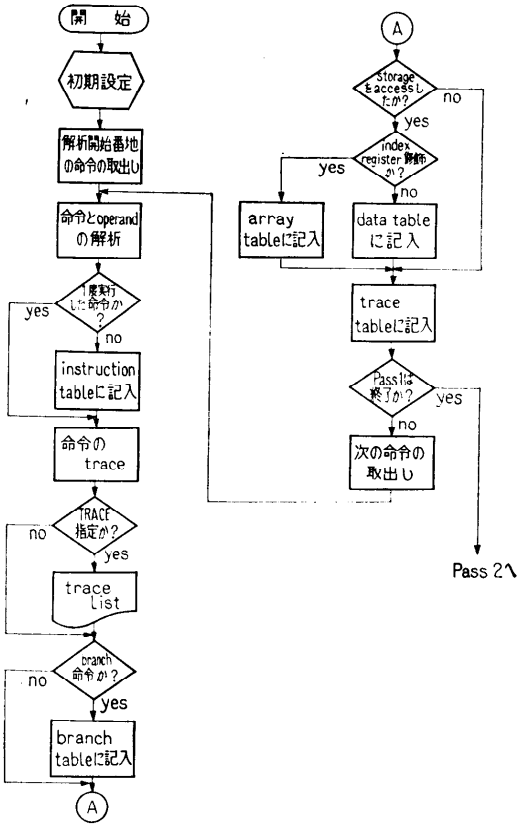

図 3.1 Pass 1 のフロー

instruction table 中の1つのエントリは図 3.2 のような構造をしている．命令の type code とは，

命 令		
code	type code	location
operand 1		
長さ（もしくは register number）	type code	pointer（もしくは immediate data）
operand 2		
長さ（もしくは register number）	type code	pointer（もしくは immediate data）

図 3.2 instruction table の各エントリの構造 (operand が2つの場合)

（1） condition code をセットするか,
（2） main storage からデータを fetch するか,
（3） main storage にデータを store するか,
（4） register の内容の変更をともなうか,
（5） register を double で使用するか,
（6） index register を使えるか

などをその命令についてチェックした結果である.

operand の type code は以下を区別した結果である.

（1） register か.
（2） immediate data か.
（3） mask bit か.
（4） (shift などの) count か.
（5） index register か.
（6） base register か.

また, 命令の一部あるいは全体が変更される場合は, instruction table 中では同じ address にいくつもの命令が存在するような形の登録が行なわれる.

3.2 Branch Table

branch 命令の trace 結果は branch table に記入される. 各エントリについて,

（1） 命令コード,
（2） その命令の location,
（3） branch 先の address,
（4） branch 先 address の1つ1つについて, 実際に branch したかどうかを示すフラグビット,
（5） 3方向以上の branch か

が記録される.

3.3 Data Table と Array Table

参照されたデータは単一データか配列データかをテストされ, 単一データは data table に, 配列データは array table に記入される.

PANSY では, index register を用いて main storage を参照するとき, その operand は配列であるとし, 逆に配列は必ず index register をきかせて参照するものとしている. しかも, 必ず1次元であるという考え方をとっている.

図 3.3 に data table の各エントリの構造を示した. 同じ data をいくつもの命令で参照するときは, pointer を用いて ARI を記入するフィールドを拡張する.

data の属性を表わしているビットの集まりは, implement するときの考え方で多少変更される.

array table address フィールドに index をきかせない non-indexed address が記入されること, index register に関する情報が追加されることを除けば data table とそっくり同じである. index register の内容に関しては最大の値と最小の値が調べられ, 必要に応じて更新される. 配列を宣言する DIMENSION 文は

DIMENSION (m_1; m_2, m_3)

の形をしている. ただし, m_2 は index register の最大値, m_3 は最小値, m_1 はそれらの差である.

symbolic name											
長さと位置						location					
ST	FE	CH	DN	BN	LG	FN	AC	SP	IN	SV	GE
初期値											
この block 中に記入されている ARI の数						ARI 1					
ARI 2						ARI 3					
ARI 4						ARI 5 (もしくは next block pointer)					

ST: この data location に store が行なわれたか.
FE: fetch されたか.
CH: 文字データか.
DN: 10 進数か.
BN: 固定小数点2進数もしくは論理データか
LG: 論理データか.
FN: 浮動小数点数か.
AC: address constant か.
SP: 特殊データ (program status word など) か.
IN: このデータの一部または全部が命令として実行されたか.
SV: register save area か.
GE: 実行の途中で generate されたデータか.
ARI: このデータを参照した命令の location.

3.3 図 data table の各エントリの構造

4. Pass 2 の詳細

Pass 2 では, 以下の作業がこの順序で行なわれる.

（1） instruction table, branch table, data table および array table を address の順に sort する.
（2） reverse branch table をつくる.
（3） register 使用解析を行なう.
（3） DO-loop を検出して DO table に登録する.
（5） 外部 routine の call を検出する.
（6） 逆コンパイルする (phase 1).
（7） 逆コンパイルする (phase 2).
（8） dynamic flow description coding と dynamic cross reference table を印刷する.

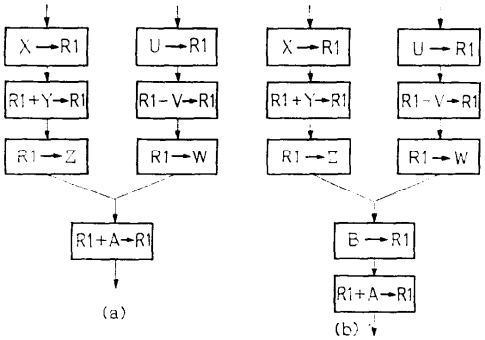

図 4.1 合流後の register 使用の例

4.1 Register 使用解析

図 4.1 のフローチャートのような2本以上の枝の joint point における逆コンパイレーションを考えてみよう．R1 を register とするとき，フローチャート(a)では合流ののち R1 にいきなり A が加えられているので，合流前の枝はそれぞれ

$Z=X+Y$　　$W=U-V$

$R1=Z$　　$R1=W$

と逆コンパイルしなければならない．一方，(b)のような場合は，R1 に A が load されており，以前の R1 の値は考慮する必要がないので，逆コンパイレーション結果は

$Z=X+Y$　　$W=U-V$

だけで十分である．

register の値が store されているにもかかわらず，(a)のようなケースを考慮して joint point の直前でいつも register に値を load するようにしたのでは，IBM System/360 のように general register が16個もある計算機では無駄な register load を沢山ふくむ質の悪い逆コンパイレーション結果が得られて，PANSY 本来の目的を遂げられなくなる．この不合理を解決するため，逆コンパイレーションに先立って register 使用解析を行なう．

まず，entry point からつぎの entry point までを1つの flow unit と考える．この際，無条件 branch で論理的には jont point にも branch point にもなっていない点は entry point とみなさない．一方，2方向以上への branch point において branch 条件が満たされない場合に実行される命令の location は Pass 1 では entry point と見なされていなかったが，register 使用解析では virtual entry point と名付け，entry point として扱う．trace table 中の各エントリの残りの4ビットのうち1ビットをこのために使用する．

${}^mU_i^j$ / ${}^mU_i^k$	0	1	2
0	0	0	2
1	0	1	2
2	0	2	2

図 4.2 Δ-opreation

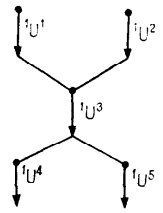

図 4.3 flow unit と register use vector の例

次に，flow unit 内での register 使用をおのおのの register について調べ，

場合(0) load しないでいきなり使用している，

場合(1) 使用していない，

場合(2) load したのち使用している

のいずれであるかを知り，それぞれの場合に対して値 0，1，2を与える．したがって，register を 16 個もつ計算機では，flow unit j に関して 16 個の成分からなる vector が対応する．これを register use vector (depth 1) と名付け ${}^1U^j$ とかく．

すべての flow unit の register use vector (RUV と略称する)が求まれば，次に，それらの間で Δ-operation を行なう．${}^mU^j$ と ${}^mU^k$ の Δ-operation ${}^mU^j \Delta {}^mU^k$ は，対応する成分 ${}^mU_i^j$，${}^mU_i^k$ $(i=1,2,\cdots)$ ごとに図4.2の表を用いて計算する．

図4.3の flow で ${}^1U^1 \Delta {}^1U^3$ は ${}^2U^1$ とかかれる．これは，flow unit 1 の RUV が 1 に続く flow unit の影響まで考慮した値であることを表わしている．すなわち，${}^2U^1$ の左肩の数字2は直列に連結された後続の flow unit の長さを表わしており depth と呼ばれる．同様にして，${}^2U^2$ が計算できる．${}^1U^3$ のように何方向かへの branch point が続いている場合は

$${}^2U^3={}^1U^3 \Delta \min\{{}^1U^4, {}^1U^5\}$$

のようにして計算する．ただし，min{ }は，{ }中のすべての vector の対応する成分のうち，最小の値をとってつくった vector を表わしている．

このようにして計算した ${}^2U^1$, ${}^2U^2$, ${}^2U^3$ でそれぞれ ${}^1U^1$, ${}^1U^2$, ${}^1U^3$ を置換える．置換えた結果の RUV 間でさらに Δ-operation を行なうというように Δ-operation を繰返す．すなわち，一般に，flow unit j に flow unit k が続くとき，

$${}^{m+1}U^j={}^mU^j \Delta {}^mU^k$$

を計算する．

被解析プログラムの各 flow unit についてこの操作を行なって depth の大きい RUV を求めることができる．しかしながら，2～3回の繰返しでかなりよい逆コンピレーション結果が得られると思われ，何十回も繰返すことは計算時間の浪費で余り意味がない．

4.3 DO-loop の検出

DO-loop の検出も逆コンピレーションの前処理の1つである．DO-loop は概念的にみて大きな block である．逆コンピレーションはある種の block 化であるので，DO-loop を可能なかぎり検出することは不可欠である．プログラムの flow 解析に関する論文[9),10),11)]が発表されているが，いずれも DO-loop の検出には直接は役立たない．

PANSY では次の条件をすべて満たす一連の命令群を DO-loop に逆コンパイルする．

（1） 2方向 branch 命令は制御変数に関するテストであり，それによって loop が解かれる．しかも，制御変数の初期セット，count up (down) が一意的に行なわれている．

（2） loop の range が一意的に定まる．

（3） loop の外部からその range 内へ branch がなされていない．

（4） loop を構成する命令の location は，すべてその range の物理的な initial address (P_i) より大きく terminal address (P_t) より小さくなければならない．

DO-loop になるフローの型をいくつか図 4.4 に示した．

DO-loop の検出は次の手順で行なう．

（1） branch table を走査して，実際に1方向もしくは2方向に branch した命令で branch 先がその命令の location よりも小さいものをピックアップする．

（2） trace table 中のその命令に対応するエントリをみて，その命令が実行され，かつ，他の命令によって変更されていないことを確かめる．

（3） その branch 命令の location を P_t，branch 先の address を P_i とし，P_i と P_t で定まる range を pysical DO range (PDRと略称する) と名付ける．

（4） PDR 中のすべての entry point の branch もとの address が PDR の外部でないことを確かめる．

（5） 1方向 branch 命令のときは PDR 中に少なくとも1つの2方向 branch 命令がなければならない．instruction table を走査し，data table を参照して，これら2方向 branch 命令の判定条件に関与する変数，すなわち，制御変数を見つける．

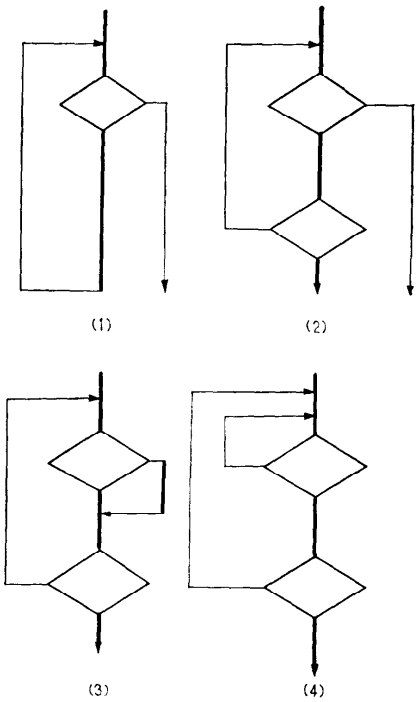

図 4.4 DO-loop になれる例

（6） 再び，instruction table と trace table を走査し，初期パラメータ，終期パラメータおよび増分パラメータを求め，loop が制御変数によって一意的に制御されていることを確かめる．

（7） P_i 以前で初期設定を行なっている命令の location を D_i とし，P_t を D_t とする．

このようにして検出した DO-loop は DO table に登録する．また，trace table 中の各エントリの残りの3ビットを利用して D_i，D_t および制御変数を取扱う命令を identify できるようにする．

4.4 逆コンピレーション

逆コンピレーションの Phase 1 では，Pass 1 から送られたテーブルを参照しながら被解析プログラムをほとんどすべて逆コンパイルし，結果は内部形式にしておく．Phase 2 では，ASSIGN 文への修正などいくらかの手直しだけが行なわれる．

2.2 節の DFDL を用いればすべての命令を逆コンパイルすることができる．条件付き branch 命令は IF 文に，無条件 branch 命令は，branch 先が固定して

いるかどうかで GO TO 文になったり，割当て型 GO TO 文や計算型 GO TO 文になる．

コントロールに関する文では ASSIGN 文がいくらか厄介である．address 部分の変更や (base) register などの値に応じて branch 先が変る branch 命令は，Phase 1 で割当て型 GO TO 文に逆コンパイルされる．しかし，それに対応する命令変更や register の値の設定は，Phase 1 では MODIFY 文や代入文になっているので，Phase 2 でこれらの文を ASSIGN 文に修正する．

したがって，代入文の逆コンピレーションがもっとも困難である．register に関する演算結果を代入文に逆コンパイルするときは，次の規則によって行なう．

規則(1) store 命令で register の値を main storage に格納するときは，その命令に出会った所で逆コンパイルする．

以下の規則で逆コンパイルするときは，左辺には固有の register 名を用いる．

規則(2) index register をきかせて main storage を参照する命令では，その直前で代入文を生成する．

規則(3) 以下のような場合は次に続く flow unit の RUV を調べ，load したのち使っている register 以外の register についとは，代入文を生成する．

(i) DO-loop に入る直前.
(ii) DO-loop から出る直前.
(iii) 外部 routine の実行に進む直前.
(iv) entry point に到達する直前.

Phase 1 実行中，register の使用に関する情報は Register Use Status Block (RUSB と略称する) に貯えられる．図 4.5 はその一例である．1つの RUSB に対応して 1 つの Register Operation Block (ROB と略称する) がつくられ，各々の register にどのような演算が施こされたかが記録される．ROB の内容は register の内容が代入文として逆コンパイルされたときクリアされる．ただし main storage への store 命令のときは，それまでの値は load された形で残る．branch point ではそれまでの RUSB, ROB が branch 先の数だけコピーされ，stack される．そして，1つの枝の1つの flow unit を処理したら，その結果の RUSB, ROB を save しておいて，別の枝の処理をする．すなわち，コントロールの流れだけから判断して，論理的に先行する可能性がある flow unit は先に逆コンパイルする．

reg 0	LN	IX	SP	SB
reg 1	LN	IX	SP	SB
reg 15	LN	IX	SP	SB

LN: ROB 中の対応する register の演算式の長さ.
IX: index register として使われたかどうかを示す.
SP: 被解析プログラムの base register として使われているかどうか示す.
SB: branch 命令で branch 先の制御に用いられたかどうか.

図 4.5 RUSB の例 (register が 16 個の場合)

4.5 命令の変更

他の命令の一部もしくは全部を変更する命令については

(1) 逆コンパイルするとき，2つ以上の文にまたがるような変更をしてはならない.

という原則をたて，一般にはこれに反しないかぎり命令変更を認めることにした．

5. あとがき

プログラマと計算機の共通語は機械語しかないので，デバギングや未知プログラムの解析のための資料を機械的に生成するに際して，プログラムの意味の解釈は機械語の世界でしか行なうことができない．これまでに提案され，使用されているデバギングツールはこの点に関する突込みが十分でないきらいがある．PANSY ではプログラムの解釈を逆コンピレーションとして行ない，さらに，プログラマのデコーディングを助けるため付帯資料としていくつかの dynamic cross reference table を与えることによってデバギングツールとしての極限をめざした．

PANSY ではプログラムの解釈は dynamic flow description coding として与えられるので，これと分岐，命令の変更に関する dynamic cross reference table を見ればコントロールの流れは一目瞭然である．プログラムスイッチや制御変数などコントロールの流れに影響するデータおよびそれ以外のデータの使用のようすは dynamic flow description coding とデータに関する dynamic cross reference table から分るようになっている．

トレーサはデバギングツールとして提供すべき資料

はすべて網羅して出力できる．トレーサと逆コンパイラを結合したものと考えられる PANSY では，トレーサ出力を逆コンパイルして整理し，無駄な情報は捨てるようにしている．したがって PANSY はデバギングツールとしてはほぼ完璧であるが，次の2点を未解決のまま残している．第1は snap shot routine を用いるように変数やレジスタの値の変化を出力しないことであり，第2は配列など構造をもつデータに対する逆コンピレーションが十分でないことである．第1の問題は容易に解決できるが，第2の問題は制御変数の解析を行なってみても限度があり，コーディングにいくつかの規約を導入しないかぎり完全に解決することはできない．一般的に言って，優れた逆コンピレーションを行なうにはコーディングの標準化が必要である．現在，PANSY の構想を IBM System/360 に適用した PANSY/360 を作成中であり，その結果とともに標準化に関する検討結果を発表するつもりである．

6. 謝　辞

本論文に関する討論に参加され，終始有意義な意見を述べて下さった小笠原道夫，折戸房雄，片山恒次の諸氏に心から感謝します．

参考文献

1) Krider, L: Flow Analysis Algorithm, JACM, Vol. 11, No. 4 (1964), pp. 429～436.
2) 前川：自動フローチャーティング，情報処理，Vol. 9, No. 3 (1968), pp. 128～136.
3) 山本，山口：自動フローチャーティング，情報処理，Vol. 11, No. 12 (1970), pp. 711～720.
4) IBM Application Program, Sytem/360 Flowchart.
5) IBM Application Program, System/360 Testran.
6) 門倉他：逆コンパイラと LOUP-TOSBAC について，昭和 38 年電気学会東京支部大会予稿集，pp. 90～91.
7) 門倉，吉川：LOUP について，第4回プログラミングシンポジウム報告集 (1963年1月)，pp. B 35～B 54.
8) Sassaman, W. A.: A Computer Program to Translate Machine Language into FORTRAN, Proc. of RJCC, 1966, pp. 235～239.
9) Karp, R. M.: A Note on the Application of Graph Theory to Digital Computer Programming, Information and Control, Vol. 3, No. 2 (1960), pp. 179～190.
10) Schurmann, A.: The Application of Graphs to the Analysis of Distribution of Loops in a Program, Information and Control, Vol. 7, No. 3 (1964), pp. 275～282.
11) Prosser, R. T.: Application of Boolean Matrices to the Analysis of Flow Diagrams, Proc. of the Eastern Joint CC, 1959, pp. 133～138.
12) 門倉，宇都宮他：プログラム解析システム，昭和46年電気学会東京支部大会予稿集，p. 78.

(昭和47年3月8日受　付)
(昭和47年5月8日再受付)