ソフトウェア工学 18－3
（1981．5．15）

フロー解析技法を応用したプログラムテスト法

花田收悦　岡　知範　永瀬淳夫
（日本電信電話公社　横須賀電気通信研究所）

1　はじめに

プログラムの静的解析，動的解析技術の進歩やプログラミング言語の高水準化に伴い，デバッグ・テスト作業を円滑に，しかも漏れなく進めるための支援ツールが数多く製作されてきた。[1)~6)] しかしこれらのツールについては，以下のような問題がある。

(1) 作業の生産性を高める方法論を明らかにしたうえで，方法論を支援するシステムを構築すべきであるが，このようなシステムは見当らない。
(2) コンパイラ，データベース管理システム，オペレーティングシステムなどの大規模システムプログラムに使用するには不十分である。
(3) 実際に使用して作業効率向上効果などを測定したものが少ない。

本稿では，(1)及び(2)を解決するために，プログラムの内部論理に基づいて机上で矛盾を網ら的に検出するデバッグ方式とその結果得られたプログラムを計算機上で実行させて網ら的に正しさを確認する方式について述べ，ついで，このデバッグ・テスト方式を支援するシステムの機能，処理系構成及び生産性向上効果などについて報告する。

2　従来のデバッグ・テストの問題点

我々のプロジェクトにおける大規模システムプログラムのデバッグ，テスト工程の作業方法，問題点及び支援ツールの現状について述べる。

2.1　大規模システムプログラムのデバッグ・テスト

大規模システムプログラムは多数のモジュールから構成されている。開発は多人数により行われ，複数のモジュールのコーディング，コンパイル，単体デバッグが並行して実施される。単体デバッグは，仕様書，設計書及びソースプログラムをもとに机上で矛盾箇所を抽出する机上レビューと，計算機上で走行させてデバッグするマシンデバッグとからなる。単体デバッグが終了すると，モジュールを結合し，テストデータに基づいて計算機を使ってバグを抽出するマシンテストに移行する。このようなプログラムの構造的特徴と作業の特質から，デバッグ・テスト段階においては，次のような点が問題となっている。

(1) モジュール間インタフェースバグが多く，早期に検出することは難しい。
(2) 高水準言語で記述されているプログラムのデバッグは，机上レビューによる方法の方がマシンデバッグよりもバグ1件当りのデバッグ工数が少なくて済む。[7)] このため，机上レビューを充実することができれば生産性は向上するが，内部論理を網ら的にレビューする具体的な方法が少ない。
(3) 処理内容が複雑かつ膨大であるので，すべての内部論理の正しさを確認することが難しい。特に，マシンテストにおいては，テスト完了の基準が不明確なため，重複テストやテスト漏れが多く，プログラムの生産性及び信頼性向上が困難である。

このような背景から次のようなデバッグ・テスト技術の開発が望まれている。

(1) 構文，意味解析上の矛盾点だけでなく，モジュール間のインタフェースチェックなど，従来のコンパイラでは検出できなかった誤りを検出する処理系。
(2) 机上で，できるだけ網ら的に内部論理をデバッグしマシンテスト前に過半のバグを取除く技術。
(3) 冗長なテストの回避と内部論理の網ら的な実行確認を行う技術。

2.2 現状の技術及びツールの問題点

現状の技術，ツールには次のような問題点があり，実用的なものは少ない。

(1) モジュール間のアーギュメント，パラメータや共通データの属性一致の検証，変数の設定・参照関係の異常（データフローアノーマリ）の検出などを行うツールがＦＯＲＴＲＡＮなどを対象に開発されているが[8]，共通データとしてレジスタを用いたり，ポインタに共通データのアドレスを設定して使用することの多いシステムプログラムの記述言語に適用したものは見当らない。
(2) ウォークスルーやコード検査など[9]，机上レビューを重視する方法はあるが，網ら的に内部論理をレビューする方法論や支援ツールは少ない。また，レビューに必要な情報の整理もなされていない。
(3) 命令コードや分岐（エッジ）の通過率を測定して，未通過部分をテストするためのデータ作成を支援するシステムが開発されているが，すべての条件の組合せ（パス）を基準としていないので，テスト完了の基準としては不十分である。[10] 一方，コントロールフロー解析に基づいてパスを物理的に抽出すると，パス数は天文学的な値となり実用的ではない。実用的でしかも信頼性の高いテスト完了基準は見当らない。

3 網ら的デバッグ・テスト方式

前記の問題点を解決し，大規模システムプログラムのデバッグ・テストを効率的かつ網ら的に行う方法を考案した。この方法は，以下の２つの方式からなる。

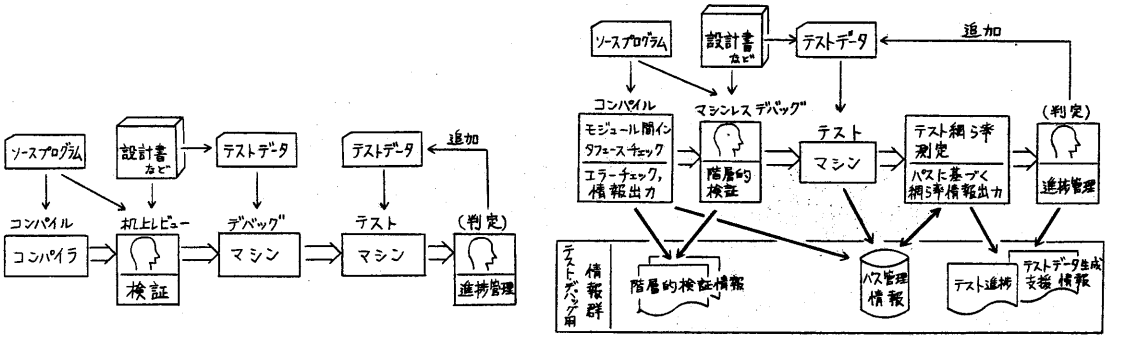
(Ⅰ) マシンレスデバッグ方式

モジュール間に渡るコントロールフロー解析及びデータフロー解析に基づき，システムプログラムに特有な構造に適用可能なモジュール間インタフェース矛盾を自動抽出する方法（インタフェースチェック）と，プログラムのパスに沿って支援機能の出力する検証用情報を用いて机上で網ら的にレビューを行う方法（階層的検証法）とからなる。

(Ⅱ) 網ら的実行確認テスト方式

仕様書や設計書をもとに，人手で作成したテストデータをプログラムに入力し走行確認を行うと同時に，支援機能により，通過したパスを記録する。さらに支援機能を用いて，未通過パスを実行するためのテスト条件（テストデータ生成支援情報）を出力し，未通過パスをねらいうちする網ら的実行確認テストを行う。

この新しい方法によるデバッグ・テスト作業の流れを，従来の作業形態と比較して図１に示す。これにより，モジュール間のインタフェース矛盾の自動抽出とモジュール内の内部論理の網ら的デバッグ及び確認が可能になる。



(a) 従来方式　　(b) 新デバッグ・テスト方式

図1　新デバッグ・テスト方式

以下に各方式について具体的に述べる。

3.1 マシンレスデバッグ方式

計算機によるインタフェースチェックとプログラム論理をテスト担当者がチェックしやすいようなデバッグ情報に基づく机上レビュー方法とからなる。

(1) インタフェースチェック

プログラムの翻訳時に，以下のモジュール間インタフェースチェックを行う。

(i) アーギュメント，パラメータ，共通データの宣言間の属性などが一致しているか否かのチェック。

(ii) 共通データのデータフローアノーマリチェック。

(iii) モジュール間のコントロールフローアノーマリチェックなど。

(2) 階層的検証法

階層的検証法は，プログラムを階層構造に基づいて分割し，トップダウンまたはボトムアップに，各分割単位毎にパスに沿った内部論理レビューを机上で行う方法である。分割単位は検証ブロックと呼び，PL/I風の言語をベースに，①外部手続き（モジュール），②内部手続き，③BEGINブロック，④ループ，からなる。階層的検証法を実施する場合，以下の階層的検証情報を使用する。

(a) 階層的ブロック関連図

ブロック間の呼出し関係及び包含関係を表示する。

(b) ブロックテキストリスト

ブロックに直接含まれる部分のテキストのリスト。このリストには，①パスを識別するための番号が文毎に，また②そのブロックに呼出されるまたは含まれる他ブロックへの入出力情報が付加される。

(c) パス一覧

ブロック内のパス一覧表。ブロックテキストリストへの対応関係を示す番号の列によって1つのパスを表現する。

(d) 機能要素リスト

パス上の代入文の列に，そのパスを実行するための条件を付加したリスト。

階層的検証法は，前記の情報に基づいて人間が以下の手順でデバッグを進める方法である（図2参照）。

step 1　階層的ブロック関連図から，上位または下位のブロックを1つ抽出する。－－－－－検証ブロックの決定

step 2　検証ブロック内のパスに沿った内部論理レビューを行う。この段階はさらに3つの詳細な手順に分かれる。

step 2.1　ブロックの機能が設計書及び上位または下位ブロックの条件を満たしているか否かを，ブロックテキストリストを用いて確認する。

step 2.2　ブロック内のパス一覧とブロックテキストリストをもとに，メインパスを中心に，入出力情報，判定，処理の妥当性を確認する。

step 2.3　重点的にレビューしたいパスについては，機能要素リストを用いて判定，処理の妥当性をチェックする。

step 3　次の検証ブロックを抽出する。すべてのブロックの検証が終うしたら階層的検証は終うする。

上記の手順においてトップダウンに検証を行う場合には，上位ブロックの検証段階で下位ブロックを1つの機能と見なしてレビューを行い，そのときに立てた機能の仮定が成立することを下位ブロックの検証段階で確認する方法で進める。

一方，ボトムアップな検証は，逆に下位ブロック内のパス上の代入文の集合によって実現される機能が正しいか否かをレビューし，上位ブロックの検証段階で下位ブロックの機能を正しく使用しているか否かを検証していくものである。

この方式は，従来の机上レビューと比較して，次の特徴をもつ。

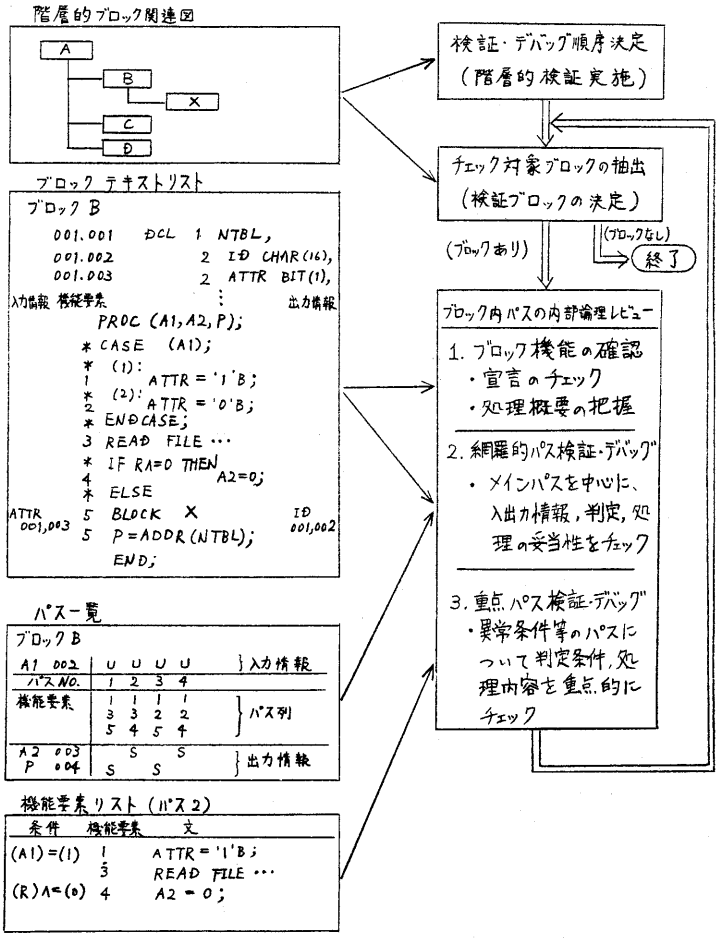


図2　マシンレスデバッグ方式の概念

(i) レビューを補助する情報が存在する。
(ii) 検証手順が明確で，内部論理の網ら的なレビューが可能である。
(iii) 大規模システムプログラムに適用する場合には，モジュール毎に並行して検証を行うことが可能である。

3.2 網ら的実行確認テスト方式

本方式は，マシンレスデバッグの完了したプログラムを計算機上で走行させ，プログラムのすべての内部論理の実行結果の正しさを確認するものである。プログラムの内部論理の実行網らを判定する基準としては，パスが最も優れているが，大規模システムの場合，パス数は膨大となり，すべてのパスをテストすることは事実上不可能である。そこで，この方式では，各検証ブロック内の全パスを少なくとも1回実行することをもってテスト完了の基準とする方法を採用した。マシンレスデバッグが，各検証ブロック内のすべてのパスについてバグのないことを保証するのに対応して，本方式は，パスの通過，未通過の記録や情報の出力などを支援機能に行わせ，実行結果の確認及び未通過パスを実行させるテストデータの作成を，人間が計算機と対話しながら，次のように進めるものである（図3参照）。

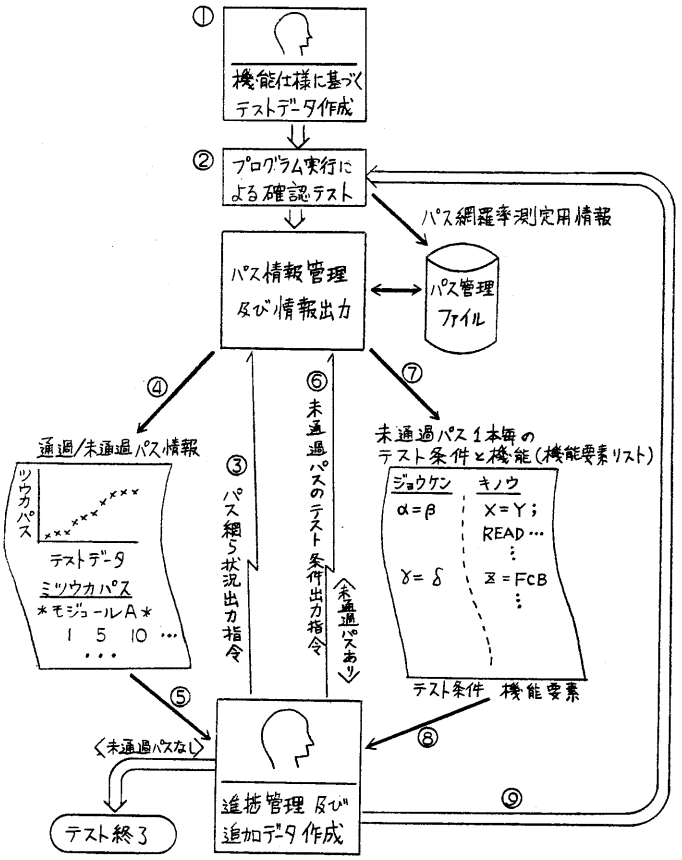


図3 網ら的実行確認テスト方式の概念

step 1 仕様書や設計書をもとに，テストデータを作成する（図3の①に対応）。
step 2 プログラムを実行し，結果を確認する（②）。
step 3 パス通過状況を問合わせる（③④⑤）。未通過パスがあれば，そのパスをテストする条件とパス上の実行文の列を出力する（階層的検証情報の機能要素リストを用いる）（⑥⑦⑧）。すべてのパスを通過したら，テスト完了とする。
step 4 出力情報に基づき，未通過パスを実行するテストデータを人間が作成し，step 2より繰返す（⑨）。

この方法は，従来のマシンテスト方式と比較して以下の特徴をもつ。

(i) 検証ブロック内のパスを基準にしているので，従来の尺度よりもテスト精度が高いうえ，検証ブロック単位にプログラムを分割するので，マシンレスデバッグ時の検証，デバッグ対象と対応づけが容易であり，しかもパス数が膨大にならず実用的である。

(ii) パスの通過，未通過などは計算機が管理するので，テスト担当者はテストの進捗を効率的かつ定量的に把握できる。

(iii) パスの概念は，階層的検証法における検証対象のパスと同一であり，しかもパスレビューを実施しているので未通過パスのテスト条件を連想しやすいうえ，機能要素リスト出力により未通過パスをねらいうちするテストデータの作成が容易である。

4 テスト支援システムPAVESの構成と機能

本章で，DIPSのシステム製造用構造化プログラミング言語SIMPLE[11]で記述されたプログラムのマシンレスデバッグと網羅的実行確認テストを支援するシステムPAVES（Program Analysis, Validation and Evaluation System）の構成と機能について述べる。SIMPLEは，PL/I風のシステム製造用言語SYSL[12]に構造化機能を導入したもので，①すべての制御構造は1入口である，②GOTO文はない，など良形な構造のプログラム以外は記述できないので，以下の点で新デバッグ・テスト方式は容易に実現できる。

(1) 検証ブロックは1入口なので，検証ブロック内のパスは一意に定まる。

(2) コントロールフローグラフ及びコールグラフが容易に求まるので，インタフェースチェックが実現できる。

(3) 通過パスの記録は，各検証ブロックに入る毎にパス情報をスタックにプッシュダウンし，出口でポップアップする方法で容易に実現できる。

PAVESは，コントロールフロー解析，データフロー解析などの静的解析結果及び，通過パスの記録などの動的解析結果を格納したプログラムデータベース（PDB）を中心として，6つの処理系から構成される総合テスト支援システムである（図4参照）。

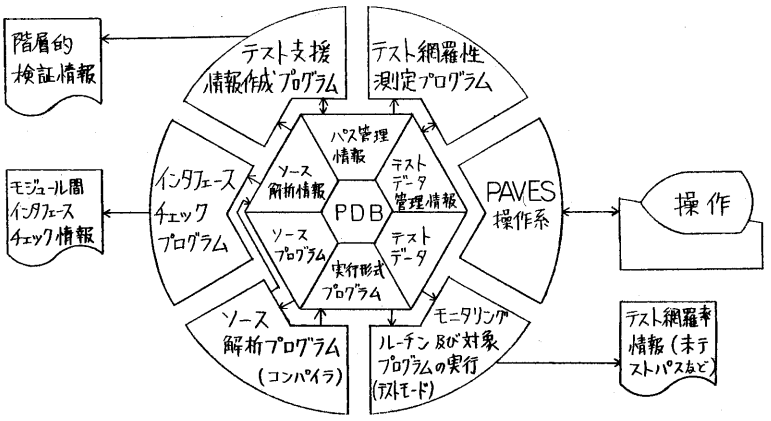


図4 PAVESの構成

4.1 プログラムデータベース

マシンレスデバッグ及び網羅的実行確認テストに必要な情報は，互いに関係をもつ。これらをデータベース化することにより，関連情報の集中管理が可能なだ

けでなく，プログラム修正時に関連情報の自動修正や再テストデータの自動抽出が可能となる。格納される情報には以下のものがある。

(1) ソースプログラム
(2) コントロールフロー情報，データフロー情報，共通データの属性情報，モジュールの特性情報，コールグラフなどを含むソース解析情報
(3) プログラム実行時の通過パスと入力したテストデータとの関係をビットベクトルで表現したパス管理情報
(4) テストデータ名とパス管理情報との対応関係を示すテストデータ管理情報
(5) テストデータ
(6) オブジェクトプログラム

4.2 処理系と機能

(1) ソース解析プログラム

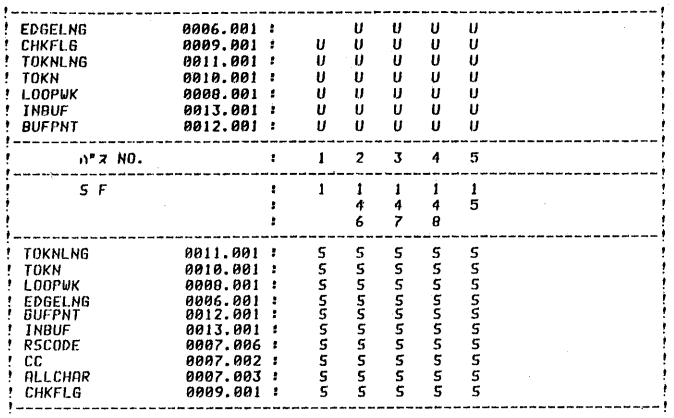
ソースプログラムの翻訳及びソース解析情報の作成を行うとともに，コントロールフロー情報から検証ブロック内のパスを抽出し，パス管理情報を作成する。さらに通過パスを記録するモニタリングルーチン呼出し命令を目的プログラム中に埋込む。

(2) 階層的検証情報作成プログラム

ソースプログラムとソース解析情報から，階層的検証情報を出力する。図5，6，7にブロックテキストリスト，パス一覧及び機能要素リストの出力例を示す。

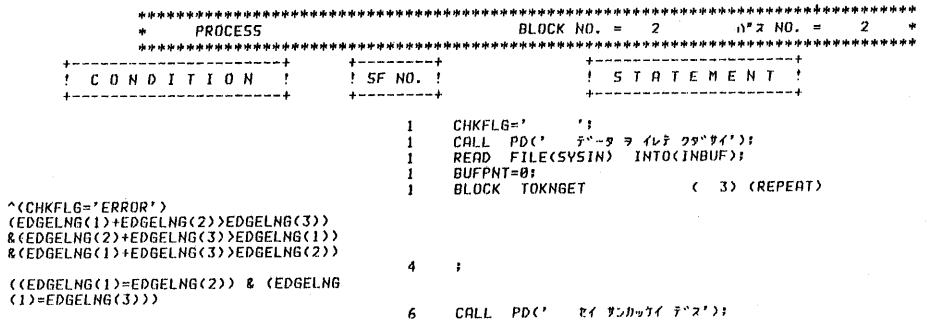
```
!-------------!---!-------------------------------------------------------------!-------------!
!USE (INT BLOCK)!5 F!                        S T A T E M E N T                  !SET (INT BLOCK)!
!-------------!---!-------------------------------------------------------------!-------------!
                    REPEAT FOREVER;
                1     CHKFLG='     ';
                1     CALL  PD('   ﾃﾞｰﾀ ｦ ｲﾚﾃ ｸﾀﾞｻｲ');                          CC        0007.002
                                                                                  ALLCHAR   0007.003
                1     READ  FILE(SYSIN)  INTO(INBUF);                             RSCODE    0007.006
                1     BUFPNT=0;
TOKN      0010.001  1   BLOCK  TOKNGET          (  3) (REPEAT)
TOKNLNG   0011.001                                                                EDGELNG   0006.001
INBUF     0013.001                                                                CHKFLG    0009.001
BUFPNT    0012.001                                                                RSCODE    0007.006
LOOPWK    0008.001                                                                CC        0007.002
                                                                                  ALLCHAR   0007.003
                                                                                  TOKNLNG   0011.001
                                                                                  TOKN      0010.001
                                                                                  BUFPNT    0012.001
                                                                                  LOOPWK    0008.001
                *     IF CHKFLG='ERROR' THEN
                *                   AGAIN;
                *     IF (EDGELNG(1)+EDGELNG(2))EDGELNG(3))
                *       &(EDGELNG(2)+EDGELNG(3))EDGELNG(1))
                *       &(EDGELNG(1)+EDGELNG(3))EDGELNG(2))      THEN
                4                                                    ;
                *     ELSE
                          DO;                          /* ﾅｶﾞｻ ﾉ ﾜ ﾁｪｯｸ              */
                5       CALL  PD('   ｴﾗｰ : ﾍﾝ ﾉ ﾅｶﾞｻ ｶﾞ ｱﾄﾏｯﾃｲﾙ ');                CC        0007.002
                                                                                  ALLCHAR   0007.003
                *       AGAIN;                                                    RSCODE    0007.006
                        END;
                *       CASE;                        /* ｻﾝｶｯｹｲ ﾉ ｾｲｼｯｻ ﾁｪｯｸ        */
                *       ((EDGELNG(1)=EDGELNG(2)) & (EDGELNG(1)=EDGELNG(3 ))):
                6         CALL  PD('   ｾｲ ｻﾝｶｯｹｲ ﾃﾞｽﾞ');                          CC        0007.002
                                                                                  ALLCHAR   0007.003
                *       ((EDGELNG(1)=EDGELNG(2)) ! (EDGELNG(1)=EDGELNG(           RSCODE    0007.006
                *       3)) ! (EDGELNG(2)=EDGELNG(3))):
                7         CALL  PD('   ｺﾄｳﾍﾝ  ｻﾝｶｯｹｲ ﾃﾞｽﾞ');                     CC        0007.002
                                                                                  ALLCHAR   0007.003
                *     ELSECASE;                                                   RSCODE    0007.006
                8         CALL  PD('   ﾌﾀｳ ﾉ ｻﾝｶｯｹｲ ﾃﾞｽﾞ');                     CC        0007.002
                                                                                  ALLCHAR   0007.003
                *       ENDCASE;                                                  RSCODE    0007.006
                      ENDREP;
```

図5 ブロックテキストリスト出力例



```
! EDGELNG          0006.001 :       U   U   U   U
! CHKFLG           0009.001 :   U   U   U   U   U
! TOKNLNG          0011.001 :   U   U   U   U   U
! TOKN             0010.001 :   U   U   U   U   U
! LOOPWK           0008.001 :   U   U   U   U   U
! INBUF            0013.001 :   U   U   U   U   U
! BUFPNT           0012.001 :   U   U   U   U   U
!-------------------------------------------------------
!      ﾊﾟｽ NO.              :   1   2   3   4   5
!-------------------------------------------------------
!      S F                  :   1   1   1   1   1
!                           :       4   4   4   5
!                           :       6   7   8
!-------------------------------------------------------
! TOKNLNG          0011.001 :   S   S   S   S   S
! TOKN             0010.001 :   S   S   S   S   S
! LOOPWK           0008.001 :   S   S   S   S   S
! EDGELNG          0006.001 :   S   S   S   S   S
! BUFPNT           0012.001 :   S   S   S   S   S
! INBUF            0013.001 :   S   S   S   S   S
! RSCODE           0007.006 :   S   S   S   S   S
! CC               0007.002 :   S   S   S   S   S
! ALLCHAR          0007.003 :   S   S   S   S   S
! CHKFLG           0009.001 :   S   S   S   S   S
```

図6　パス一覧出力例



```
*****************************************************************************
*     PROCESS                          BLOCK NO. =   2        ﾊﾟｽ NO. =    2    *
*****************************************************************************
+--------------------+     +-------+              +--------------------+
! C O N D I T I O N  !     ! SF NO. !              ! S T A T E M E N T  !
+--------------------+     +-------+              +--------------------+

                                 1    CHKFLG='     ';
                                 1    CALL  PD('   ﾃﾞｰﾀ ｦ ｲﾚﾃ ｸﾀﾞｻｲ');
                                 1    READ  FILE(SYSIN)  INTO(INBUF);
                                 1    BUFPNT=0;
                                 1    BLOCK  TOKNGET           (  3) (REPEAT)
^((CHKFLG='ERROR')
(EDGELNG(1)+EDGELNG(2))EDGELNG(3))
&(EDGELNG(2)+EDGELNG(3))EDGELNG(1))
&(EDGELNG(1)+EDGELNG(3))EDGELNG(2))
                                 4    ;
((EDGELNG(1)=EDGELNG(2)) & (EDGELNG
(1)=EDGELNG(3)))
                                 6    CALL  PD('   ｾｲ ｻﾝｶｯｹｲ ﾃﾞｽ');
```

図7　機能要素リスト出力例

(3) インタフェースチェックプログラム

ソース解析情報をもとに，以下のチェックを行う。

(i) アーギュメント，パラメータ及び共通データのフィールド個数，属性などの対応関係（レジスタで持回るデータもチェック対象）

(ii) レジスタの退避，回復の整合誤り

手続きのプロローグ，エピローグ処理のレジスタ退避，回復の対応を調べる。

(iii) アーギュメント，パラメータ及び共通データを含めたグローバルなデータフローアノーマリ解析

プログラムを有向グラフ $G=\{N, E, n_0\}$ で表現すると，各ノード $n_i \in N$ におけるデータの状態（ρ）は，定義（d），参照（r）及び未定義（u）の組合せで示される。データに対する初期設定もれ（$\rho = ur$）や冗長な代入（$\rho = dd$，du）などのデータフローアノーマリをモジュール間に渡るグローバルなデータフロー解析を行って検出する。

(iv) 制御移行のないモジュール

モジュール間の呼出し関係をコールグラフから求め，制御移行のないモジュールを検出する。

(4) モニタリングルーチンとテスト網羅性測定プログラム

モニタリングルーチンは，テスト対象プログラム実行時に起動され，通過パスを一時ファイルに記録する。テスト網ら性測定プログラムは，以下の処理を行う。

(i) 実行結果が正常であれば，テスト担当者の指令に従い，一時ファイル内の記録されたパスをもとにパス管理情報を更新する。

(ii) パス情報についての各種問合せ処理を行う。未通過パス数，パス通過率，テストデータと通過率の関係を表す進捗状況グラフなどを出力する。

5 評価と分析

考案した技法及びＰＡＶＥＳを実用のプログラム開発作業に適用し，バグ検出効果，工数削減効果について評価し実用性を確認した。

5.1 バグ検出効果

(1) インタフェースチェック機能

コンパイル完了後の３ｋＳのプログラムを対象にチェックを行ったところ，新たに検出されたエラーの数は，その後，このプログラムのテスト完了時までに検出されたバグ数の約１０％であった。

(2) 網ら的実行確認テスト

従来方式でテスト完了していた３ｋＳのプログラムのパス通過率を測定したところ，全パスの４０％が未通過であった。未通過パスをねらいうちするテストデータを作成し実行テストしたところ，全バグ数の２０％のバグが新たに検出された。また，網ら的実行確認テストを実施した１０ｋＳのプログラムについては，実用に供して１年以上経過したが，現在までに検出されたバグは１件のみである。

5.2 生産性

マシンレスデバッグの工数削減効果及び網ら的実行確認テストによる重複テスト防止効果を評価するため，次の３通りのテストをほぼ同様の処理内容のプログラムに実施し，すべてのパスの通過を確認するまでの工数を測定した。

(1) 従来形式のデバッグ，テストを行い，テスト完了となった段階で網ら率を測定し，未通過パスがなくなるまでテストする（サンプルＡ）。

(2) 机上デバッグは従来形式とし，マシンテストは網ら的実行確認を最初から行う（サンプルＢ）。

(3) マシンレスデバッグと網ら的実行確認テストを行う（サンプルＣ）。

サンプルＡ，サンプルＢは，同一担当者が実施し，サンプルＣは，別の担当者が行った。この結果，サンプルＢはサンプルＡの８５％の工数で，またサンプルＣはサンプルＡの５０％の工数でテストを終えることができた。サンプルＡとサンプルＢの担当者は同一であることを考慮すると，最初からテスト完了の基準が明確であることによって，少なくとも１５％のテスト工数削減が図られたと言える。さらにマシンレスデバッグによりバグを早期に検出するので，これ以上の工数削減が可能である。

6 むすび

大規模システムプログラムのデバッグ，テスト効率とバグ検出力の向上をねらいとしたマシンレスデバッグと網ら的実行確認テスト方式について述べ，コント

ロールフロー解析，データフロー解析などを主体とした支援システムPAVESを開発した。また，具体的なプログラム開発作業に適用することにより，バグ検出効果の向上とテスト工数の削減が図られることを確認した。このような効果が得られた理由としては，モジュール間インタフェースチェックの自動化，階層的検証情報を用いた網ら的な内部論理レビュー及び検証ブロック内のパスに基づく網ら的かつ実用的なテスト基準の採用，さらにはPAVESが，このような方法論に立脚して設計されており，しかもデバッグ，テストに必要な情報をプログラムデータベースに集約管理していることがあげられる。また，解析対象の言語が良形なプログラム以外は作成できない点も1つの理由である。

しかし，本技法は，フロー解析を中心としたプログラムの静的解析技術に基づいているので，処理抜け，条件抜けなどの設計書に対する違反や設計誤りは検出できない場合が多い。今後の課題は，静的解析技術に基づく内部論理の網ら的テストの限界を明らかにすることである。

参考文献

1) Ramamoorthy, E.V. and Ho, S.F. : Testing Large Software with Automated Software Evaluation Systems, IEEE Trans. SE., Vol. SE-1, No.1, pp46-58 (1975).
2) Stacki, L.G. : New Directions in Automated Tools for Improving Softwar Quality, Current Trends in Programming Methodology, Vol. 2, Prentice-Hall (1977).
3) Jessop, W.H. et al. : ATLAS - An Automated Software Testing System, 2nd Int. Conf. on Software Eng., pp629-635 (1976).
4) Voges, U. et al. : SADT - An Automated Testing Tools, IEEE Trans. on SE., Vol. SE-6, No. 3, pp286-290 (1980).
5) Fairley, R.E. : An Experimental Program-Testing Facility, IEEE Trans. on SE., Vol. SE-1, No. 4, pp350-357 (1975).
6) Balzer, R.M. : EXDAMS : Extendable Debugging and Monitoring System, SJCC, pp567-580 (1969).
7) Fagan, M.E. : Design and Code Inspections to Reduce Errors in Program Development, IBM SYS.J., Vol. 15, No.3, pp182-211 (1976).
8) Osterweil, L.J. and Fosdick, L.D. : DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs, Software - Practice and Experience, Vol.6, pp473-486 (1976).
9) Myers, G.J. : A Controlled Experiment in Program Testing and Code Walk-throughs/Inspections, Comm. ACM, Vol. 21, No. 9, pp760-768 (1978).
10) Myers, G.J. (松尾正信訳) : ソフトウェア・テストの技法，近代科学社 (1980).
11) 永瀬ほか：システム製造用構造化プログラミング言語SIMPLE，研究実用化報告第27巻12号 (1978).
12) 寺島ほか：DIPS-1における高能率システム製造用言語の実用化，情報処理，Vol.16, No.6 (1975).