U.D.C. 681.32.06.001.4:519.686

ソフトウェアテスト技術とその動向

Testing Techniques for Software

中所武司* *Takeshi Chûsho*
林　利弘** *Toshihiro Hayashi*

ソフトウェアの生産性と信頼性向上のためには，その開発費用の約半分を占めるテストの効率化が重要である。ところが，従来から広範に適用されてきた動的テスト法では，効果的なテストデータ作成方法がないことや，テストの準備と結果確認作業に手間取るなどの問題があった。

これらの解決のためには，まずテストデータ作成については，プログラムの機能仕様に基づく機能テスト法と，プログラムの内部構造に対するテスト網羅率に着目した構造テスト法を組み合わせて用いるのが有効である。また，テスト作業の効率化のためには，テスト環境設定機能やテスト手続き記述言語をもつテスト実行支援システムが有用である。

本報では，これらの技法に関する最近の動向について述べる。

1 緒　言

ソフトウェアの生産性と信頼性の向上は，最近のマイクロコンピュータをはじめとする計算機応用分野の広がりに伴い，ますます重要な課題となってきている。そのため，'70年代にはソフトウェア工学という新しい研究分野が確立され，ソフトウェア開発工程のあらゆる局面の研究が行なわれてきた。

なかでもテスト工程は，ソフトウェア開発費用の約半分を占め，生産性向上に重要であるばかりでなく，品質保証に不可欠の重要工程である。このプログラムの検証は，「プログラムが仕様どおりに作られている」ことを確かめるのが目的であるが，まだ実用的な自動検証技法はなく，実際には動的テスト法[1]が用いられる。すなわち，多くのテストデータを用いてプログラムを何度も実行し，各々の結果を調べるという方法である。これは，“Exhaustive Testing”とも呼ばれ，実用面で次のような課題がある。

(1)　効果的なテストデータの作成方法
(2)　テストの準備と結果の確認作業の効率化

このうち，特に第1項は動的テストに本質的な問題である。プログラムを誤り少なく作る構造化プログラミングの提唱者であるダイクストラが，「テストは誤りの存在を示すことはできるが，誤りのないことは示せない。」と明言するように，テストは誤りを見つける目的で行なわれる。そのため，プログラムの品質は，もし誤りがあれば必ずそれを検出できるような効果的なテストデータを用いたか否かに依存してしまうからである。

そこで本報では，まず2章でテスト技法の概要を述べた後，3章でテストデータ作成技法，4章でテスト実行・評価ツールについてそれぞれ述べる。

2 プログラムテスト技法

プログラムの実用的なテスト技法は，大まかには静的テストと動的テストに分けられる。

2.1 静的テスト

これは，プログラムを実行することなく，ソースプログラムを解析して誤りを調べる方法で，人手で行なうものとしての机上テストなどが含まれる。

静的テストの自動化ツールとしては，プログラムの制御フロー解析により実行されない命令を検出したり，更に変数の値の定義と参照に関するデータフロー解析によりその矛盾を検出するものがあるが，このような方法で自動的に検出できる誤りは限られている。

2.2 動的テスト

これは，実際にプログラムを実行してその動作を調べる方法で，最も一般的に行なわれている。動的テストでは，いかに効果的なテストデータを作成するかということと，いかに効率的にテストを実行するかということが重要であり，各々の技法について以下に述べる。

3 テストデータ作成技法

一般にプログラムの入力となり得るデータの数は膨大であるが，時間的及び費用的制約のため，実際にはその一部分のデータを用いてプログラムのテストを行なわざるを得ない。そこで，できるだけ効果的なテストデータを選ぶ必要があり，その代表的技法として，機能テスト法と構造テスト法がある。

3.1 機能テスト[2]

これは，プログラムの機能仕様からテストデータを選ぶものである。その具体的技法は少ないが，人手による一般的手法として，機能仕様書に基づいて有効な入力条件や出力条件及び無効な入力条件を表に記入してゆく方法がある。この方法は，各条件ごとに1個のテストデータを試せば，他のデータも同様の結果になるように外部条件を分割するので同値分割法と呼ばれる。**表1**に，銀行の自動支払機用のプログラムのためのテストケース選択に同値分割法を適用した例の一部を示す。

次に，この同値クラス表を用いてテストデータを作成するが，そのとき，具体的な値の選定に当たっては，その条件の限界値や最小単位分だけずれたものを選ぶ。これは，プログラム内の条件判定に関する誤りの多くが限界値の判定誤りであるためである。

一方，機能テストの自動化ツールとしては，機能仕様を組合せ論理で表現して入力し，テストケースを自動生成する原

* 日立製作所システム開発研究所　** 日立製作所大みか工場

表1 同値クラス表の使用例 入力条件の各項目ごとに，正しい条件と誤った条件を表に記入し，テストデータの選択に漏れと無駄がないようにする。

入力条件	有効同値クラス	無効同値クラス
暗証番号	● 4桁の数字	● 数字以外の入力
払出金額	● 1,000円以上20万円まで	● 20万円を超えるもの ● 数字以外の入力

因結果グラフ法に基づくものがある。その一つとして，日立製作所で開発したテスト項目作成支援システムがあるが，詳細については本号の別論文[3]に譲る。

3.2 構造テスト

3.2.1 テスト網羅基準

構造テストは，プログラムの内部構造に基づいてテストケースを選ぶもので，ホワイトボックステストやプログラムテストとも呼ばれる。その主な方法は，プログラムの制御構造を制御フローグラフと呼ばれる有向グラフで表わし，そのパス解析に基づいてテストケースを選ぶもので，そのときの選択基準としてテスト網羅性を示す尺度が用いられる[4]。その基本となるものは，すべてのパスを網羅する基準である。しかし，通常のプログラムは繰返し処理を含み，パスの数が膨大となる。

例えば，**図1**の簡単なPascalプログラムを考えてみる。これは，逐次的に入力される整数値データの最大値を求めるもので，0以下又は101以上の値が入力されると終了する。このプログラムに対応する制御フローグラフを**図2**に示すが，部分パスc. e, fがループになっており，かつ繰返し数に上限がないため，パスの数は無限であることが分かる。

そこで，実際には次のような幾つかの簡易化されたテスト網羅基準が用いられる。

(1) 全ノード網羅(全文網羅)

最も簡単な基準は，制御フローグラフ上の全ノードを一度以上実行するもので，**図2**の例ではa, b, c, e, gというパスを選べばよい。これはすべての実行文を一度以上実行することになり，全文網羅基準とかC_0メジャとも呼ばれる。

(2) 全アーク網羅(全分岐網羅)

これは制御フローグラフ上の全アークを一度以上実行するもので，全文網羅基準では対象外であったif-then文の条件不成立時のテストなどが含まれる。**図2**の例では，a, b, c, e, f, d, gというパスを選べばよい。これはすべての分岐を一度以上実行することになり，全分岐網羅基準とかC_1メジャとも呼ばれる。

(3) 繰返し数を含む基準

全分岐網羅基準では，繰返し条件の成立回数が1回の場合だけテストすればよいが，繰返し回数に依存した誤りの検出のために，0回及び複数回(最大数又は2回)の場合もテストする。**図2**の例では，先の2個のパスのほかにa, b, c, e, f, d, f, d, gも選ぶ。

(4) 繰返し処理を除く全パス網羅

繰返し処理以外は全パスを実行するもので，**図2**の例ではfを対象外としてa, b, c, e, gとa, b, d, gになる。

(5) その他

以上のほかに，変数の値の定義と参照の関係を示すデータフローに着目した基準や，連続した複数の分岐の組合せをすべて網羅する基準なども提案されている。

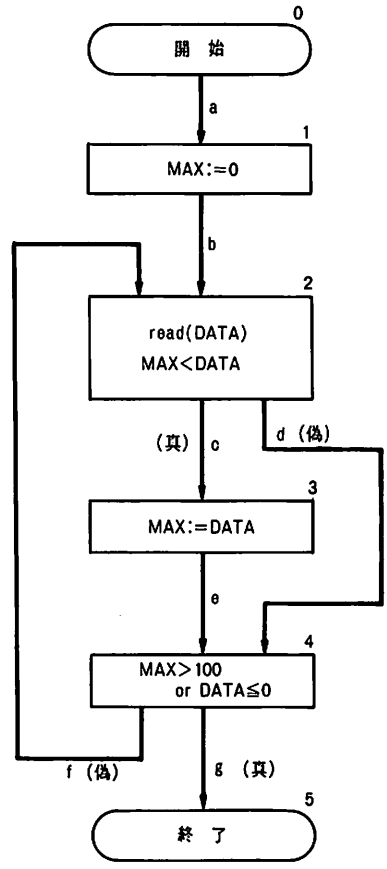
一方，このようなパス解析に基づく基準だけでは，誤りの発生しやすい分岐条件自身のテストが不十分であるため，分岐条件が論理和や論理積で結合されたものは，各々の真，偽の場合に分けてテストするとか，値の大小を比較する式ではその比較演算子に無関係に必ず＜，＝，＞の3種類のテストを行なう必要がある。

3.2.2 実用化方式

次に構造テストの実用上の問題と解決策について述べる。

```
function MAX:integer;
  var DATA:integer;
  begin
    MAX:=0;
    repeat
      read(DATA);
      if MAX<DATA
        then MAX:=DATA
    until(MAX>100) or (DATA=<0)
  end
```

図1 プログラムの例 逐次的に入力される整数値データの最大値を求めるPascalプログラムで，0以下又は101以上の値が入力されると終了する。構造テスト法の説明に用いる。



注：小文字の英字(アーク名)，数字(ノード番号)

図2 例題の制御フローグラフ 図1のプログラムで，途中からの分岐を含まず，かつ途中で分岐もしない命令の列を一つの箱(ノード)にまとめるとともに，実行の順を矢印(アーク)で表わしている。

(1) 実行不可能パスの選択

パスを機械的に選んだ場合は実行不可能なものが混じる。そこで，既存の構造テストツールではパスの選択は人手に任せ，用意したテストデータをすべて実行したときに未実行のまま残される文や分岐の検出とテスト網羅率の算出を行なうものが多い。この種のツールとして，マイクロコンピュータ用のCAPT[5]や構造化プログラム用のSCORE[6]のほか，テスト実行支援システムに組み込まれたものとして，マイクロコンピュータ用のHITS[7]などがある。

(2) 大規模ソフトウェアの完全網羅困難性

システム全体のテストでは，網羅基準の100%達成は難しいため，図3に示すように全体を幾つかのサブシステムに分割し，各々のテストで100%を達成する。

(3) パス欠如の検出不可

構造テストでは，必要な機能が欠けているという誤りの検出は原理的に不可能である。そのため，機能テストの併用が不可欠である。

(4) 品質過大評価傾向

全分岐網羅率をテスト十分性の尺度として用いた場合，テストデータ数に対するテスト率が凸曲線になるため，品質が過大評価される傾向がある。そこである分岐の実行に伴って必ず実行されるような他の分岐を，テスト率測定の対象外とするような改良方式[8]が考えられている。図4は本方式を適用するために，図2の制御フローグラフを簡約化したものである。

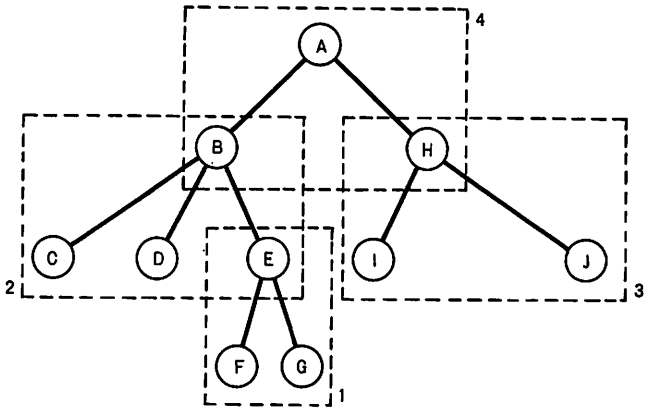


図3 大規模ソフトウェアの構造テスト法 システム全体のテスト時の網羅基準100%達成は困難のため，幾つかのサブシステムに分けてテストする。この例では4分割している。

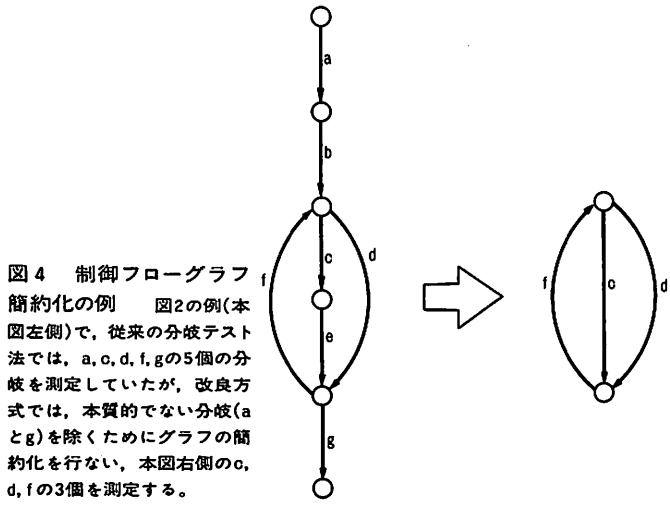


図4 制御フローグラフ簡約化の例 図2の例(本図左側)で，従来の分岐テスト法では，a, c, d, f, gの5個の分岐を測定していたが，改良方式では，本質的でない分岐(aとg)を除くためにグラフの簡約化を行ない，本図右側のc, d, fの3個を測定する。

4 テスト実行・評価支援

プログラムのテスト関連作業を効率良く行なうためには，誤り(バグ)の検出を行なうテスト作業とその誤りの原因追求・除去を行なうデバッグ作業の双方が重要である。また，品質の評価・改善のためには，テストの十分性評価と潜在バグ数の予測管理が重要である。

4.1 テスト支援

テスト作業の効率化のためには図5に示す要件を満たすテストシステムが有効であり，その例として，制御用のHITESTやマイクロコンピュータ用のHITSなどがある。このうちHITEST[9]は，プログラム単体のテストだけでなく，リアルタイムシステムに特徴的な非同期処理プログラム群の組合せテスト(マルチタスキングテスト)，更には制御対象機器を含めた実稼動環境下での総合テストまでを支援する。そして，単体，組合せ，総合テストの各段階で重複した作業が発生しないような一貫テスト方式を実現している。

また，HITSは，大形計算機を用いてマイクロコンピュータ用プログラムを効率良くテストするものであり，本号の別論文[7]で詳述する。

4.2 デバッグ支援

デバッグ作業の効率化のためには，プログラムの動きを的確に把握するための機能が重要であり，ダンプ，トレース機能が基本である。特に，マルチタスキング環境下でのシステムの動作分析を可能にしたデバッグツールとしてHITEST/DEMOがあり，本号別論文[10]で詳述する。

4.3 テスト評価支援

テスト十分性評価には前述の構造テスト機能が有効である。また，潜在バグ数予測のためには，ロジスティック曲線やゴンペルツ曲線といった統計的予測手法を用いたシステムが用いられている。

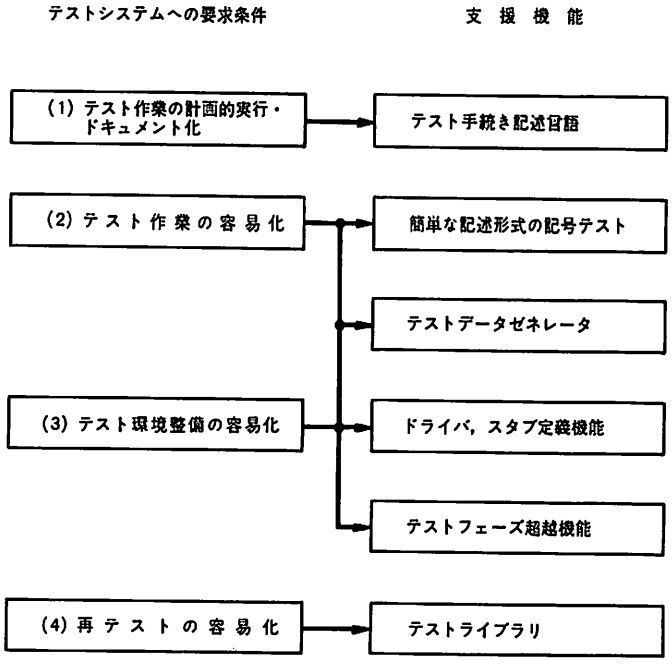


図5 テストシステムへの要求条件とその支援機能 従来のテスト方法では，その準備に始まり，テスト実行，結果の確認，更には保守時の再テストに至るまで，多くの人手作業を要していた。テストシステムでは，これらの作業の効率化，自動化が重要である。

5 結　言

現在，実用的プログラムの検証には動的テスト法が最も一般的に用いられているので，本報ではその方式に不可欠の効果的テストデータ作成技法と効率的テスト実行支援機能について述べた。実際にこれらの技法を用いたテスト手順は次のようになる。

(1) まず原因結果グラフ法の適用可能なプログラムは，そのツールを用いてテスト項目を作成する。その他のものは，同値分割法などでテスト項目を選ぶ。

(2) これらのテスト項目から限界値に着目して，テストデータと予想結果を求め，テスト手続きにまとめる。

(3) テスト実行支援システムを用いて，このテスト手続きを実行し，検出された誤りを修正する。

(4) 構造テスト機能により未実行の文や分岐が検出された場合は，それを実行するテストデータを追加する。

(5) テスト網羅率がある基準に達すると，テストを終了する。

なお，本報では，テスト技法としては計算機言語で記述されたプログラムの検証を目的とするものに限ったが，本来，テストはソフトウェア開発工程全体の各段階で行なうべきものである。特に，

(1) 誤りの多くが設計段階で作り出されること。

(2) 誤りの費用はその検出が遅れるほど高くなること。

などの事実から，今後は要求定義法，設計法，プログラミング技法も含めた，一貫したテスト思想が重要と思われる。

参考文献

1) 中所：ソフトウェアのテスト技法，情報処理，24，7，842～852(昭58-7)
2) G. J. Myers(長尾，松尾訳)：ソフトウェア・テストの技法，1980年，近代科学社
3) 野木：ソフトウェアテスト項目作成支援システム，日立評論，66，3，199～202(昭59-3)
4) E. F. Miller：Program Testing：Art Meets Theory, Computer, 10，7，42～51(July 1977)
5) 進藤，外：トレースデータに基づくマイコン用プログラムのテスト・カバレージ・アナライザ，情報処理学会第25回全国大会論文集，485～486(昭57-10)
6) 田中，外：構造化プログラム用テスト充分性評価支援ツールの開発，同上，443～444(昭57-10)
7) 黒崎，外：マイクロコンピュータ用会話型テスト支援システム"HITS"，日立評論，66，3，203～206(昭59-3)
8) 中所：パステストに本質的な分岐に着目した網ら率尺度の提案，情報処理学会論文誌，23，5，545～552(昭57-9)
9) 大島，外：制御用ソフトウェア機能一貫テストシステム"HITEST/F"，日立評論，62，12，893～898(昭55-12)
10) 大島，外：オンラインデバッグ支援システム"HITEST-DEMO"，日立評論，66，3，207～210(昭59-3)

論文抄録

プログラミング言語とその会話型支援環境

日立製作所　中所武司

情報処理　24—6，715～721（昭58-6）

プログラミング言語の目的は，人間と計算機のインタフェースをできるだけ人間側に近づけることである。そのため，言語は記述水準の高級化という形で発展してきたが，それは実用的視点から次の3段階に分けることができる。すなわち，第1段階では記述量の削減，第2段階ではプログラミング方法論の反映，第3段階では非手続き的問題記述の実現を主目的としてきた。その各々の代表的言語として，Fortran, Ada, Prologがある。

しかし，プログラムの開発はプログラムの記述をもって終了するものではなく，その後に開発費用の約半分を占める検証作業が必要である。そのため，言語の高級化によるプログラムの生産性向上には限度があり，むしろその開発支援環境が重要である。ところが，プログラムの作成と修正に用いるエディタやプログラムの実行テストに用いるデバッガなどに代表される既存のツールは，汎用的な反面，機能的には低水準である。また各ツールは個別に開発されているため，ユーザーインタフェースの不統一などの問題が生じやすい。

このような現状の問題を解決するには，

(1) 各ツールの言語適応化による高機能化

(2) 各ツールの有機的結合による統合化

が必要である。すなわち，対象プログラムの記述言語に適応した支援機能の付与とツール間の結合により，強力な統合的プログラミング環境を構築できる。

この種の特定言語向きシステムは，既に幾つか開発されている。例えば，リスト処理言語Lisp用のInterlisp(Xerox)のほか，Pascal用のMentor(INRIA)，PL/Iふうの簡易言語PL/CS用のCornell Program Synthesizer(米国コーネル大学)，システム記述言語Cの拡張言語GC用のIPE(CMU)，モジュラ言語Iota用システム(京都大学)などがある。これらはいずれも従来のテキストエディタの代わりに構文要素単位の編集機能をもつ構造エディタを備えているほか，インタプリタ又はコンパイラによる実行機能及びデバッグ機能をもっているが，まだ研究試作段階のものが多い。一方，大規模ソフトウェア開発用としては，現在，Ada用のAPSE(米国国防総省)，構造化プログラミング言語SPL用のCROPS(日立製作所)などが開発中である。

以上，プログラム開発で人間と計算機のインタフェースとなる言語とその会話型支援環境について，現状の問題点とその解決策について述べた。ここで提案した特定言語向きプログラミング環境は，まだ機能的に熟成しているとは言えないが，言語適応化による高機能化とコンパイラを含めた種種のプログラミングツールの統合化はUser-Friendlyなプログラミング環境に必須の条件であり，今後の実用化研究が注目される。その意味で，従来のようにコンパイラを単独で開発して，ユーザーに提供する時代は終わりつつあると言える。