解　説

ソフトウェアのテスト技法†

中　所　武　司††

1. まえがき

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[1]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を対象とした方法論や技法の研究およびそれを実現するツールの開発が行われてきた．

なかでもテスト工程[2]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．もちろん，高品質ソフトウェアの開発のためには，ソフトウェアの検証を要求定義や設計の各段階でも行うことが基本であるが，プログラムの作成を人手にたよる現在の開発方式では，最終的なソフトウェアの検証はプログラムの検証によって行わざるを得ない．

このプログラムの検証は，「プログラムが仕様通りに作られている」ことを確かめるのが目的なので，仕様書とプログラムが等価であることを自動検証するような方法が理想的であるが，まだ実用には程遠く，実際には動的テスト法が用いられる．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問題がある．

（1）効果的なテストデータの効率的作成方法がない．

（2）テストの準備や結果の確認作業に手間取る．

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．すなわち，構造化プログラミングの提唱者である E. W. Dijkstra 教授[3]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に基づいてテストしたか否かに依存してしまうからである．

そこで本文では，まず2章でテスト技法を静的テスト，動的テストおよびその他の技法に分け，各々の特徴を述べた後，3，4章で動的テストの問題に対する現状での実用的技法とツールについて述べる．

2. プログラムテスト技法

プログラムの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，人手で行うものとして机上テストがある．机上テストは，ソースリストを見て，それが正しく動作することを確認する作業であり，プログラム作成者以外の人が加わって行う場合はコードレビュとかコード検査[5]と呼ばれる．この方法はテストの基本であり，動的テストよりも効果的かつ効率的であるといわれている．

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を解析して実行されない命令を検出したり，さらに変数の値の定義と参照に関するデータフローを解析してその矛盾を検出するプログラム解析ツールがある．例えば，値の定義されていない変数を参照したり，あるいは定義された値が参照されないまま，再び値が変えられたりするような，変数値の誤った定義参照関係を検出する．しかしながら，このような方法で自動的に検出され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，その動作の誤りを調べる方法で，最も一般的に行われている．この動的テストを行うためには次のような作業が必要である．

（1）テストデータの作成とその予想結果の明確化

（2）テスト実行

† Testing Techniques for Software by Takeshi CHUSHO (System Development Laboratory, Hitachi Ltd.).

†† (株)日立製作所システム開発研究所

第1項には，それに先立って行うテストケース選択，また，第2項にはテスト環境設定や結果確認などの作業が伴う．したがって，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，各々の技法とツールについて3章，4章で詳しく述べる．

2.3 その他の技法

以上に述べた他にも，幾つかのテスト技法がある．プログラム内の変数に数値などの定数を与える代わりに記号値を与えたプログラムの動作を解析する記号実行[6]は，実行経路分析やテストデータ生成に応用されるほか，プログラムの正当性の証明にも用いられる．すでにツール化[7]されたものもある．また，プログラムの入出力やループ不変式などの表明（assertion）に基づいてプログラムの正当性証明を行うものもある．しかしながら，いずれも大規模ソフトウェアのための一般的な検証技術としてはまだ実用的でないので本文では詳細にはふれない．

また，プログラムの品質評価を主目的とした技法[8]として，意図的に埋め込んだ誤りの検出率や，意図的に変形した幾つかのプログラムのうちで誤りを生じたものに対する検出率に基づくものなどがあるが，このような統計的手法についても本文の対象としない．

3. テストデータ作成技法

ほとんどのプログラムでは考えうる入力データの数が膨大になるため，そのすべてを試すことは不可能である．そこで実際にはそのサブセットを用いてテストを行わざるを得ない．したがって，限られた時間と費用の中で高い品質保証の与えられるテストデータのセットを作成する必要があり，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 理論的背景

J. B. Goodenough ら[9]は，テストデータ選択基準に対する2つの概念，すなわち，reliable と valid という概念を導入して，理想的なテストデータセットを定義した．まず，ある選択基準Cを満たすテストデータセット T_1 がすべて正しく実行されるならば，その基準Cを満たす任意のセット T_k も正しく実行されるとき，Cは reliable であるという．また，プログラムに誤りが存在するとき，基準Cを満たし，かつ，その誤りを検出するセット T が少なくとも1つ存在するとき，Cは valid であるという．そして，reliable でかつ valid な基準Cを満たすテストデータセットが正しく実行されるプログラムは正しいことを示した．しかしながら，このような基準を得るための具体的手法は見つかっていない．

表-1 同値クラス表の使用例

入力条件	有効同値クラス	無効同値クラス
配列名称の大きさ	1～6文字の名称	0文字の名称，7文字以上の名称
次元の数	1～7個のいずれか	0個のもの，8個以上のもの
	≈	≈

3.2 機能テスト

これはプログラムの機能仕様からテストケースを選ぶもので，ブラックボックステストとか仕様テストとも呼ばれる．具体的技法は少ないが，人手による一般的方法として，機能仕様に記述された入力や出力に関する外部条件を細かく数えあげ，各々について有効な入力条件や出力条件，および無効な入力条件を表に記入していく方法がある．この方法は，各条件ごとに1個のテストデータを試せば，他のデータも同じ結果になることが期待できるように外部条件を分割するので同値分割法[5]と呼ばれる．表-1には，Fortran の配列宣言文を構文解析する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，有効同値クラスのテストケースについては，1つのテストデータが多くのテストケースを含むように選び，効率良くテストできるようにする．一方，無効同値クラスのテストケースについては，個々に対応するテストデータを作成してテスト漏れが生じないようにする．また，具体的な値の選定にあたっては，その条件の限界値やそれから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．

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，テストケースを自動生成する原因結果グラフ法[5]に基づくものがある．その大まかな手順を次に示す．

（1） 機能仕様を適当な大きさに分割し，各々について原因と結果を識別する．

（2） 原因を入力，結果を出力とした組合せ論理に制約条件を付加した原因結果グラフを作成する．

（3） このグラフをある規則の下で決定テーブルに

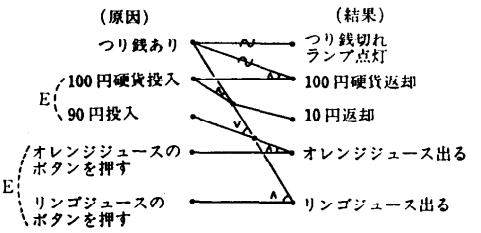


図-1 原因結果グラフの例

変換する.

(4) この決定テーブルの各列をテストケースに変換する.

本技法の実用化ツールとして, 古川らの AGENT[10] (Automated GENeration for Test cases) があるので, それに沿って説明する. 今,「90円のジュースの自動販売機」の例をとると, 原因結果グラフは**図-1**のようになる. 図中の∧, ∨, ～は各々, 論理積, 論理和, 否定を示す. 原因側のEは2つの原因の成立条件が排他的であることを示す. このような複数の原因間の制約条件としては, 他に, いずれかは必ず成立する(I), つねに1つだけ成立する(0), 特定の一方が成立すれば他方も成立する(R), 特定の一方が成立すれば他方は成立しない(M), などがあり, 不必要なテストケースの生成を防いでいる.

AGENT の入力は, このグラフを文形式で表現したものであり, そのソースリストを**図-2**に示す. 図の

```
** SOURCE LIST **
 SEQ.    SOURCE TEXT
   1  @ +S.+L.+N.+R.+C.+T.+F
   2    TITLE = '** TEST - 01 **'
   3    NODE N1 = ' ﾌﾘｾﾝ ｱﾘ '
   4         N2 = ' 100ｴﾝ ｺｳｶ ﾄｳﾆｭｳ '
   5         N3 = '  90ｴﾝ ﾄｳﾆｭｳ '
   6         N4 = ' ｵﾚﾝｼﾞ ｼﾞｭｰｽ ﾉ ﾎﾞﾀﾝｵｽ '
   7         N5 = ' ﾘﾝｺﾞ ｼﾞｭｰｽ ﾉ ﾎﾞﾀﾝｵｽ '
   8         X1 = ' ﾌﾘｾﾝ ｷﾞﾚｰ - ﾗﾝﾌﾟﾃﾝﾄｳ '
   9         X2 = ' ｺｳｶ ﾍﾝｷｬｸ '
  10         X3 = ' 10ｴﾝ ﾍﾝｷｬｸ '
  11         X4 = ' ｵﾚﾝｼﾞ ｼﾞｭｰｽ ﾃﾞﾙ '
  12         X5 = ' ﾘﾝｺﾞ ｼﾞｭｰｽ ﾃﾞﾙ '
  13  *
  14  * RELATION
  15    RELATE R1 = N1 N2 AND I1
  16           R2 = I1 N3 OR I2
  17           R3 = NOT N1 SIMP X1
  18           R4 = NOT N1 N2 AND X2
  19           R5 = I1 SIMP X3
  20           R6 = I2 N4 AND X4
  21           R7 = I2 N5 AND X5
  22  *
  23    CONST C1 = N2 N3 EXCLUSIVE
  24          C2 = N4 N5 EXCLUSIVE
  25  *
  26    END
```

図-2 AGENT のソースリストの例

前半は節点文で, グラフ内の原因に N1～N5, 結果に X1～X5の名前をつけ, その意味を記述する. 次の関係文を用いて中間節点2個と結果節点5個の論理関係を記述する. 最後に, 制約条件文を用いて2つの排他的関係を記述する.

AGENT の出力は, 図-2のソースリストの他に, 節点, 関係, 制約条件, テストケースのリスト, および**図-3**のテスト用帳票などである. この例では6個の

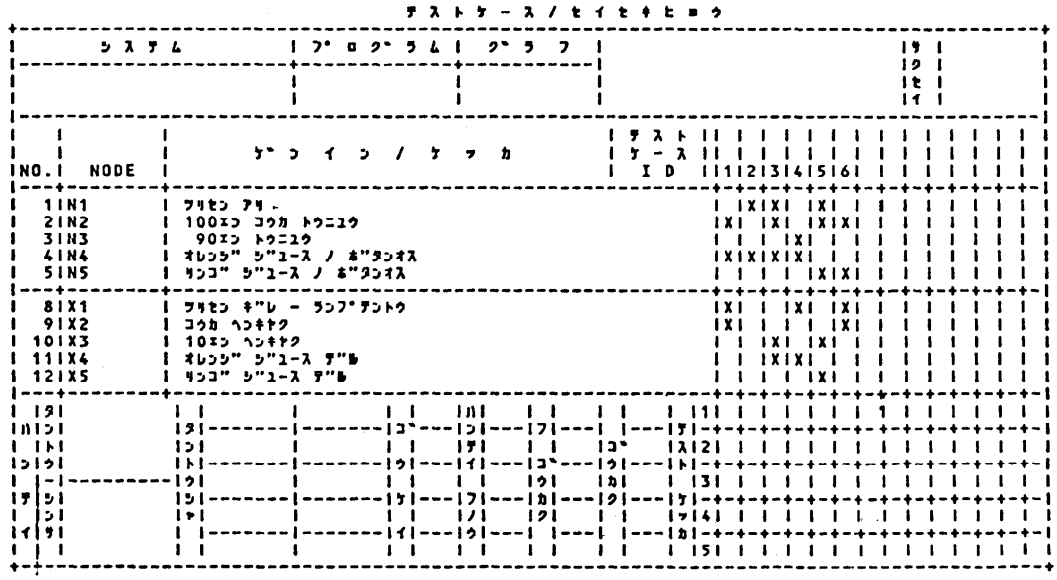


図-3 AGENT のテスト用帳票の例

テストケースが生成されている.

このような原因結果グラフ技法は，誤り発見効果の高いテストケースの選択に有効であるほか，機能仕様の不備や不明点の摘出という二次的効果のあることが確認されている．しかしながら，原因結果グラフそのものは人手で作成するため，手順の(1)で，節点が40～50程度におさまるように仕様を分割すること，および機能が組合せ論理表現向きである必要がある.

機能テストの自動化ツールとしては，原因結果グラフ法の他に状態遷移図[11]に基づくものがある．これは，機能仕様を状態遷移図で表現し，ある網ら基準を満たすように状態遷移パスの集合をテストケースセットとして選択するものである．しかしながら，プログラムの機能仕様を基本的に状態遷移図で表現できる場合でも，実際には状態遷移条件が過去の履歴の影響を受けることが多いので，純粋な形で適用可能な分野は限られる.

3.3 構造テスト

3.3.1 テスト網ら基準

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，プログラムテストなどとも呼ばれる．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ら性[12]を表す尺度が用いられる．本方式による基本的なテストデータ生成手順[13]は次のようなものである.

(1) あるテスト網ら基準を満たすように，パスの最小セットを選ぶ.

(2) 各パスを通過するためのパス条件を選ぶ.

(3) 各パス条件を満たす入力データ値を求める.

そこで，この方式を実施するためには，まずテスト網ら基準を定める必要があるが，その基本となるものはすべてのパスを網らする基準である．しかし，通常のプログラムでは繰り返し処理を含むためパスの数が膨大となり，全パス実行は不可能な場合が多い.

例えば，**図-4**に示すような，整数の列が逐次的に入力されてその最大値を求める簡単なプログラムを考えてみよう．このプログラムは0以下または101以上の値が入力されると終了するとする．そのフローチャートを**図-5**に，制御フローグラフを**図-6**に示す．制御フローグラフでは，通常基本ブロックと呼ばれる，必ず先頭から実行されて，かつ途中で分岐しない命令の集

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

図-4 プログラム例

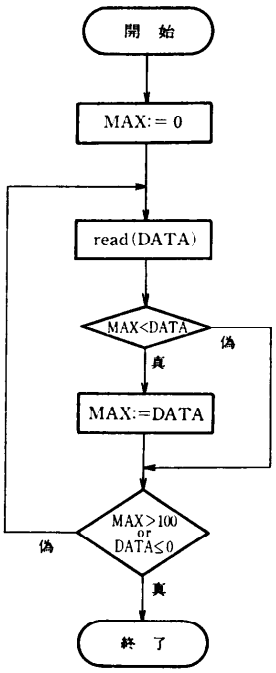


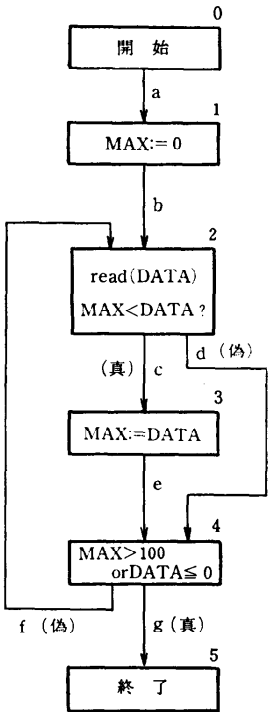
図-5 例題のフローチャート

りを1つのノードとするため，フローチャートとは異なるが，パス解析上は本質的な違いはない．さて，この図-6を見ると，部分パス cef がループになっており，その繰り返し数に上限がないので，パスは無限個あることがわかる.

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ら基準が用いられる.

(1) 全ノード網ら（全文網ら）

最も簡単な基準は，全ノードを1度以上実行するようなパスのセットを選ぶものである．図-6の例では abceg というパスを選べばよい．この基準はすべての文を1度以上実行することになるので全文網ら基準とか C_0 メジャと呼ばれることがある．なお，先のテストデータ作成手順に従えば，手順(1)で求めたパス abceg のパス条件は，変数の値の再定義されたものを



小文字の英字: アーク名
数字: ノード番号

図-6 例題の制御フローグラフ

添字で区別することにすると,

$\{MAX_1=0\} \wedge \{MAX_1<DATA_1\} \wedge \{MAX_2=DATA_1\} \wedge \{(MAX_2>100) \vee (DATA_1<0)\}$

となる. これを適当に簡約化すると

$DATA_1>100$

となり, 手順(3)でテストデータとして 101 を選べばよいことにる.

(2) 全アーク網ら (全力岐網ら)

これは全アークを1度以上実行するようなパスセットを選ぶもので, 全ノード網ら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-6の例では, 先の全ノード網らパスでは含まれていない d と f を含むように, abcefdg というパスを選べばよい. この基準はすべての分岐を1度以上実行するので全分岐網らとか C_1 メジャと呼ばれることがある.

(3) ループ繰り返し数を含む基準

繰り返し処理は通常その先頭または後尾に繰り返し処理判定条件を伴う. 全分岐網ら基準ではこの判定条件の成立回数が1回の場合だけテストすればよいが, 繰り返し数に依存した誤り検出のためには, 0回の場合や複数回 (最大数または2回) の場合もテストした方がよい. 図-6の例では, (1), (2) の2個のパスの他に, adcefdfdg を加える.

(4) 繰り返し処理を除く全パス網ら

パスの数が膨大となるのは繰り返し処理によるが, これを対象外とすればパスの数は有限となり, 全パス網ら基準が実行可能になる場合が多い. 図-6 の例では, f を対象外とすると adceg と abdg である. この基準は全分岐網ら基準と併用することになる.

(5) データフローに基づく基準

以上に述べた基準はいずれも制御フローだけに基づいていたが, S. Rapps ら[14] は, 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網ら基準を提案している. そして, 次のような基準が全分岐網らと全パス網らの間に段階的に位置することを示した.

(i) 分岐条件変数参照網ら: 2つ以上の出力アークを有するノードは分岐条件を持つが, そこで用いられている変数の参照をそのノードのすべての出力アークに対応づける. そして, そこで参照される変数値を定義しているノード(複数ありえる)からこれらのアークまでの定義参照パスを考えたとき, このような部分パスの中で, 定義ノードと参照アークの対が異なるものはすべて網らする基準を all-p-uses と呼ぶ. 図-6の例では, 上記(1)～(4)のパス例で満足されない定義参照パスとして, ノード3で定義した変数 MAX の値をアーク c(MAX<DATA) で参照するパスがある. そのため例えば abcefceg が必要である.

(ii) 全変数参照網ら: これは, 分岐条件で用いられる変数に限らず, すべての変数参照を対象とすることにより, (i)の基準を厳しくしたもので, all-uses と呼ぶ.

(iii) 全定義参照パス網ら: これは, ループを含まないすべての定義参照パスを網らするもので, 定義と参照は同じものでも途中のパスが異なればすべてテストするため, (i), (ii)よりも厳しい. all-du-paths と呼ぶ.

(6) 分岐アークの組合せによる基準

全分岐網らと全パス網らの間の格差を段階的に埋める基準として, M. R. Woodward ら[15] は n 個の逐次的に実行される分岐アークを含む部分パスを考え, n 個以下の分岐アークの組合せから成るすべての部分パスを網らする基準 TER(Test Effectiveness Ratio) を提案している.

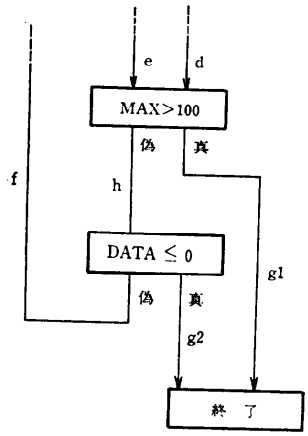


図-7　制御フローグラフの分解例（1）

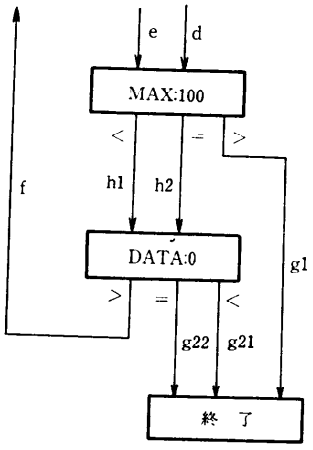


図-8　制御フローグラフの分解例（2）

以上，全パス網ら基準に対する簡易基準を6種類あげたが，このようなパス解析に基づく構造テストでは，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．そこでこの欠点を補うため，制御フローグラフを詳細化して網ら基準を厳しくする方式として，次のようなものがある．

（1）　論理式の分割：分岐条件が論理和や論理積を用いた複数条件から成る場合は，各々の真，偽の場合をテストする．図-6の例では，ノード4の分岐条件がMAX>100とDATA≤0の論理和になっているので，このノードを図-7のように分解して網らテストを行う．

（2）　比較式の分割：さらに，各条件が値の大小を比較する式の場合は，その比較演算にかかわらず，<，=，>の3種類のテストを行う．図-7の例では，分岐条件が比較式となっている2つのノードの出力アークを各々図-8のように3個にして網らテストを行う．

表-2　各テスト基準を満たすパスセットの例

テスト基準		パスセット
簡易化	全ノード（全文）網ら	①
	全アーク（全分岐）網ら	②
	繰返し条件成立回数	①（0回），②（1回），③（2回）
	繰返し以外全パス網ら	①，④
	データフロー網ら (all-p-uses)	①，②，④，⑤（ノード 1→f），⑥（ノード 3→c）
	n 個の分岐の組合せ ($n=2$)	①(c→g)，②(c→f，f→d，d→g)，③(d→f)，⑥(f→c)
詳細化	論理式分割後の全アーク網ら	①(g1)，②(g2)
	比較式分割後の全アーク網ら	①，②(h2，h1，g22)，④，⑦(g21)

（注）　パスの詳細

パス番号	パス	入力データ（例）
①	abceg	(101)
②	abcefdg	(100，0)
③	abcefdfdg	(100，1，0)
④	abdg	(0)
⑤	ebdfdg	なし
⑥	abcefceg	(1，101)
⑦	abdg (g は g21)	(−1)

本節で述べた網ら基準を図-6の制御フローグラフに適用した場合のテストケース（パス）選択とテストデータの例を表-2にまとめておく．

3.3.2　構造テストの問題点

前項で述べたパス解析に基づく構造テストには次のような問題がある．最初の2項目は運用上の問題，他は方式上の問題である．

（1）　実行不可能パスの選択

前項の初めに述べた手順（1）において，パスを機械的に選んだ場合，パス条件がつねに偽となる実行不可能なものが混じる．その原因がプログラムエラーに基づくものや，例えば入力パラメータのチェックなどの防衛的コードの存在による場合は問題ないが，その他の場合は実行可能なパスと置換える必要がある．例えば，表-2のパス⑤は図-6のノード1で定義された変数 MAX の値をアーク f(MAX≤100) で参照するための部分パス bdf を含むように選んだものである．ところがこのパス条件は，

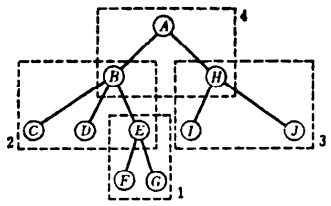


図-9 大規模ソフトウェアの構造テスト法

$\{MAX=0\} \wedge \{MAX<DATA\} \wedge$
$\{MAX \leq 100\} \wedge \{DATA>0\}$

であり，適当に簡約化すると，

$\{0<DATA\} \wedge \{DATA>0\}$

となり，この条件はつねに偽となる．そのため，パス⑤を通過するテストデータは作成不可能である．このような問題のわずらわしさを避けるため，実用化されている構造テスト支援ツールは，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を検出する機能とテスト網ら率表示だけを支援するものが多く，パスの選択は人手に任される．

（2） 大規模ソフトウェアテストの完全網ら困難性

システム全体のテストでは，網ら基準の100% 達成はコスト的に難しい．この場合，図-9[16]の例の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% を達成するようにする．実際には，システムテストのレベルでは85% とか90% の全分岐網ら基準が用いられている．

（3） 全パス網ら基準との格差

現在実用化されている構造テスト支援ツールでは全分岐網ら基準を採用しているものが多い．しかし，これでは全パス網ら基準との格差が大きいため，先に述べたような幾つかの中間基準が提案されている．

（4） 分岐条件テスト不十分性

パス網ら基準では，分岐条件が真か偽かにのみ注目するため，分岐条件自身の誤りが見逃がされやすい．そこで，前項で述べたような，論理式や比較式を分割する方式が提案されている．

（5） パス欠除検出不可

構造テストでは必要な機能，すなわち必要なパスがインプリメントされていないという誤りの検出が原理的に不可能である．このような誤りは機能テストによって検出する必要がある．

（6） 品質過大評価傾向

パス網ら基準はテストデータ作成に用いられる他，テスト十分性の尺度としても用いられ，テスト網ら率が高いほどプログラムの品質も高いと考えられる．この場合，全パス網ら基準ではテストデータ数に対するテスト率が線形特性を持つが，全分岐網ら基準などの簡易基準では1つのテストデータが複数の被網ら要素を実行するため凸曲線になる．そのため，網ら率が100% 未満の所で品質が過大に評価されるという欠点がある．そこで，筆者ら[17]はある分岐の実行に伴って必ず実行される他の分岐をテスト率測定の対象外とするような全分岐網ら基準の改良方式により，この欠点を改善している．

この方式を簡単に説明しておくと，まず有向グラフの2つのアーク p, q の関係について次の概念を導入する．

〔定義〕 p を含む任意のパスが必ず q を含むとき，q を p の相続子アークと呼ぶ．

ここで，ノード x からノード y へのアーク(x, y) が相続子となるのは，(x, y) なるアークが1つで，かつ次の4条件のいずれかを満たすものである．

(R 1) $IN(x) \neq 0 \wedge OUT(x)=1$

(R 2) $IN(y)=1 \wedge OUT(y) \neq 0$

(R 3) $OUT(x) \geq 2 \quad x \in IDOM(w)$

ただし，$\forall w \leftarrow \{w \mid (x, w) \in A \wedge w \neq y\}$

(R 4) $IN(y) \geq 2 \wedge y \in DOM(w)$

ただし，$\forall w \in \{w \mid (w, y) \in A \wedge w \neq x\}$

なお，$IN(x), OUT(x)$ はノード x の入力および出力アーク数，w は隣接ノード，$DOM(w)$ と $IDOM(w)$ は w の支配子および逆支配子ノードの集合，A はアーク集合とする．パステストの観点からは，相続子アークは被相続子アーク（上記定義の p）の網ら情報を相続するため注目する必要がない．そこで，これら4条件を相続子消去規則として適当な手順で適用するこ

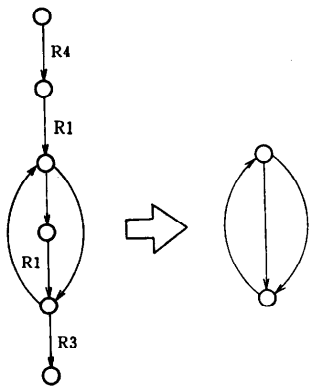


図-10 パステスト向き有向グラフ簡約法

とにより，制御フローグラフを相続子アークのないグラフに簡約化できる．このような相続子簡約グラフ上での全アーク網ら基準は，先に述べた品質過大評価傾向を減じている他，テストデータ選択に用いた場合に冗長なデータの混入防止[18]にも有効である．**図-10** は，図-6の制御フローグラフの簡約化の例である．

3.3.3 実用ツール

構造テスト支援ツールは種々開発，実用化されているが，大まかに分類すると，まず，静的なパス解析を行い，テストケースやパス条件を自動生成するものとして，Fortran を対象にした RXVP[13] や SADAT[19] などがある．しかし，大半は動的テスト時に用いるもので，テスト網ら率の測定と未実行部分の表示を行う．この種のツールは専用のものとテスト実行支援システムに組込まれたものがあり，各々の例として CIP[20]，PAVES[21] などがある．実用ツールは他にも多く，前者に属するマイコン用の CAPT[22] や構造化プログラム用の SCORE[23]，後者に属するマイコン用の HITS[24] やシステムプログラム用の HPLTD[25] などの開発例がある．

処理方式としては，プリプロセッサまたはコンパイラが被テストプログラムに計測用コードを埋め込み，その実行時に収集した網ら情報をポストプロセッサが解析する方式が一般的である．しかし，マイコン用ツールでは，先に述べた CAPT は外部ハードウェアトレーサを用いて収集した分岐命令トレースデータから求める方式，HITS は大型機によるシミュレーション実行時に情報収集する方式で，いずれもコンパイル情報を受取って解析している．

3.4 領域法

これは入力領域を同じ結果を導く部分領域に分割して，各々からテストデータを選択する方法である．この領域分割を **3.2** 節の同値分割法で行う場合は機能テスト，パス解析に基づく場合は構造テストに属するが，双方を併用する場合もあるので，節を分けてここで述べる．

E. J. Weyuker ら[26] は，**3.1**節の J. B. Goodenough らのテスト基準は全入力テストが必要となることを指摘し，それに代わる方法を提案している．すなわちある部分領域S内の1つの入力が誤りを検出するときには基準Cを満たす任意のテストデータセットも必ず誤りを検出する場合に，「C は S に対して revealing である」というプログラムに依存しない概念を導入し，これに基づいて部分領域を求める方法である．

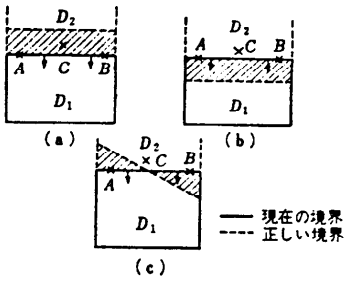


図-11 境界のずれによる3種類の誤り

別に L. J. White ら[27] は，パス解析で求めた部分領域からの効果的な入力データの選択法と個数の上下限を示している．例えば，2次元の入力領域（2入力）の場合でかつ部分領域が線形不等式で形成される場合，各境界線分ごとにその両端および境界を含まない側に少し離れた所の3点を選べば，**図-11** に示すような3種類の境界のずれをすべて検出できる．

しかしながら，このような領域法は部分領域への分割方法が難しく，適用分野は限られる．

4. テスト実行

4.1 支援機能

テストのためのプログラム実行を効率良く行うためには，テストベッドとかテストハーネスと呼ばれるテスト実行支援システムが有効である．その代表的機能としては，

（1） 単体，結合テストのための環境模擬機能

（2） 入力データと予想結果およびテスト環境などを記述するテスト手続き言語[28]

（3） 会話型デバッグ機能

（4） 構造テスト支援機能

などがある．

単体テストや結合テストによるプログラム誤りの検出，修正費用[20]はシステムテスト時に比べて1桁少ない．そのため，単体，結合テストの重要性は早くからいわれていたが，実際にはあまり徹底していない．その理由は，この種のテストでは，未作成の上位モジュールの代わりとなるドライバや，下位モジュールの代わりのスタブなどのテスト環境作成の作業がかなり発生する上に，その環境の設定誤りも多いため，テスト効率が悪いということによる．そこで，第1項のような環境模擬機能が必須となる．そして，このようなテスト環境および入力データや予想結果などはテスト手続きとしてまとめて記述しておくことにより，ライブラリ化して再利用したり，テスト作業を自動化するこ

とができる．また，このような誤りの有無を調べるテストは一括処理（バッチ処理）が効率的であるが，検出された誤りの原因究明と修正を行うデバッグ作業は会話型で行うのが効率的である．

一方，このようなテスト実行と並行して，正しく実行されたテストデータのパス網ら情報を収集，解析して，未実行部分のテストを促す構造テスト支援機能もプログラムの品質向上に不可欠である．

4.2 実用ツール

テスト実行支援システムとして実用化されているツールは多いが，大半はいわゆるシンボリックデバッガであり，**4.1**節で述べたような機能を有するものはいまだ少ない．**3.3.3**項で掲げたツールの他に，単体テストを支援し，テスト手続き記述言語を有するものとして，GE の TPL[29]，三菱の SOLDA[30]，日立の HITEST[31] などがある．

ここでは，一例として筆者らが開発した，16ビットマイコン68000用のテストシステム HITS[24),32)] (Highly Interactive Testing-and-debugging System) を簡単に紹介しておく．本システムは68000用のアセンブラおよび高級言語 Super-PL/H で記述されたプログラムの機械語オブジェクトを汎用大型計算機 HITAC Mシリーズ上でシミュレーション実行しながらテストするものである．特に設計上の特徴としては，単体テストからシステムテストまで適用可能，複数言語のシンボリックテスト支援，系統的テスト機能と効率的デバッグ機能の一元化，バッチ処理と会話型処理の両用化，などがある．主な機能としては次のようなものがある．

（1） 環境模擬機能

単体，結合テストの環境設定作業を容易化するために，**図-12**に示すように，未作成の上位，下位モジュールを模擬するドライバ，スタブ定義機能や，外部データ，入出力処理の模擬機能を設けた．

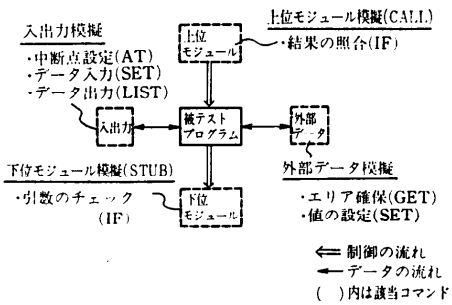


図-12 単体テストのための環境設定

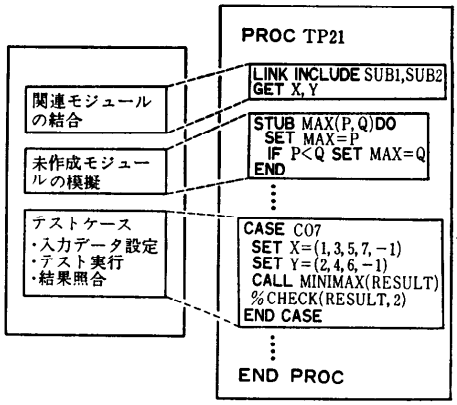


図-13 HITS のテスト手続の例

（2） テスト手続き記述言語

テストのための環境設定や入力データ，結果照合などはテスト手続きとしてまとめて記述する．その記述言語はコマンド形式とし，ライブラリ化して EXEC コマンドで実行できる他，直接に端末入力してもよい．**図-13**にテスト手続きの記述例を示す．

（3） 環境設定作業削減機能

繰り返して用いるコマンド列をマクロ化する機能，入力値だけ変化させて同じテストケースを実行する機能，複数オブジェクトのリンク機能などを設け，テスト作業を極力削減できるようにしている．例えば図-13の％CHECK は次のように定義されたマイクロコマンドである．

```
CLIST %CHECK
  IF &1=&2 LIST'<O.K.>', '&1=&2'
  IF &1<>&2 LIST '<N.G.>','&1<>&2'
END CLIST
```

（4） デバッグ機能

誤りを検出したテスト手続きは，一時的なデバッグ用コマンドを付加しながら会話型実行することができる．そのとき，文番号による中断点設定，変数名による値の設定と表示，制御トレースおよびデータトレース（変数値の変化の追跡）などができる．また，レジスタ類の値の設定，表示の他，アドレスエラー等の異常時のダンプ出力や逆トレースを実行前に指定することもできる．**図-14**は図-13のテスト手続きを用いたデバッグの例である．

（5） 構造テスト支援機能

全文網らや全分岐網らを基準としたテスト率測定，および未実行の文や分岐の表示を行う．この情報は累積することができる．

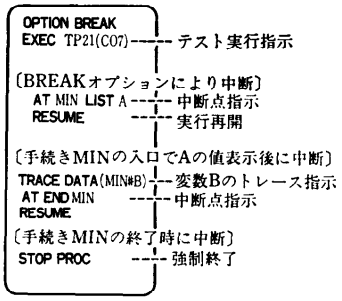


図-14 会話型デバッグの例

5. むすび

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，本文では，その方式に不可欠の効果的テストデータ作成法と効率的テスト実行のための技法およびツールについて述べた．特に詳細な説明は実用性の高いものに限定したが，それらを用いた場合のテスト手順は次のようになる．

（1） まず原因結果グラフ法が適用可能なプログラムはそれを用いてテストケースを作成する．

（2） 次に同値分割法でテストケースを選ぶ．

（3） これらのテストケースからテストデータと予想結果を作成し，テスト手続きにまとめる．

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

（5） 未実行の文や分岐が検出された場合は，それを通過するテストデータを追加作成し，テストする．

（6） テスト網ら率がある基準に達するとテストを終了する．

本文では，テスト技法として，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．

（2） 誤りの費用はその検出が遅れるほど高くなるなどの事実からも，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であると思われる．

参考文献

1) Boehm, B. W. : Software Engineering, IEEE Trans. Comput., Vol. C-25, No. 12, pp. 1226-1241 (1976).
2) Zelkowitz, M. V.: Perspectives on Software Engineering, Comput. Surv., Vol. 10, No. 2, pp. 197-216 (1978).
3) Dijkstra, E. W.: Notes on Structured Programming, in Structured Programming, Academic Press, New York, pp. 1-81 (1972).
4) 中所武司：ソフトウェアのテスト技法，電子通信学会誌，Vol. 64, No. 5, pp. 549-552 (1981).
5) Myers, G. J. (長尾，松尾訳)：ソフトウェア・テストの技法，p. 192，近代科学社，東京 (1980).
6) 玉井哲雄，福永光一：記号実行システム，情報処理，Vol. 23, No. 1, pp. 18-28 (1982).
7) 宮本 勲：プログラム・テスト支援ツール：サーベイ，情報処理，Vol. 20, No. 8, pp. 688-693 (1979).
8) Adrion, W. R. et al.: Validation, Velification, and Testing of Computer Software, Comput. Surv., Vol. 14, No. 2, pp. 159-192 (1982).
9) Goodenough, J. B. and Gerhart, S. L. : Toward a Theory of Test Data Selection, IEEE Trans. Softw. Eng., Vol. SE-1 , No. 2, pp. 156-173 (1975).
10) 古川，野木，越智：機能テストのためのテスト項目作成手法について，情報処理学会ソフトウェア工学研究会資料，16-2 (1980).
11) Chow, T. S.: Testing Software Design Modeled by Finite-State Machines, IEEE Trans. Softw. Eng., Vol. SE-4, No. 3, pp. 178-187 (1978).
12) Miller, E. F.: Program Testing: Art Meets Theory, Computer, Vol. 10, No. 7, pp. 42-51 (1977).
13) Huang, J. C.: Error Detection Trough Program Testing, in Current Trends in Programming Methodology, Vol. II, Prentice-Hall, Now Jersey, pp. 16-43 (1977).
14) Rapps, S. and Weyuker, E. J.: Data Flow Analysis Techniques for Test Data Selection, Proc. 6 th International Conference on Software Engineering, pp. 272-278 (1982).
15) Woodward, M. R. et al.: Experience with Path Analysis and Testing of Programs, IEEE Trans. Softw. Eng., Vol. SE-6, No. 3, pp. 278-286 (1980).
16) Holthouse, M. A. and Hatch, M. J.: Experience with Automated Testing Analysis, Computer, Vol. 12, No. 8, pp. 33-36 (1979).
17) 中所武司：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，Vol. 23, No. 5, pp. 545-552 (1982).
18) 中所武司，田中 厚：パステスト向き有向グラフ簡約法とそれに基づくテストケース選択法，情報処理学会第25回全国大会講演論文集，2E-5, pp. 439-440 (1982).
19) Voges, U. et al.: SADAT—an Automated

Testing Tool, IEEE Trans. Softw. Eng., Vol. SE-6, No. 3, pp. 286-290 (1980).
20) Sorkowitz, A. R.: Certification Testing: a Procedure to Improve the Quality of Software Testing, Computer, Vol. 12, No. 8, pp. 20-24 (1979).
21) 花田，岡，永瀬：フロー解析技法を応用したプログラムテスト法，情報処理学会ソフトウェア工学研究会資料，18-3 (1981).
22) 進藤，浜田，福岡：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回全国大会講演論文集，4D-8, pp. 485-486 (1982).
23) 田中 厚，中所武司：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，2E-7, pp. 443-444 (1982).
24) 中所，田中，岡本，本田，黒崎：16ビットマイコン68000用クロス型テストバッグ支援システムHITS，情報処理学会マイクロコンピュータ研究会資料，20-3 (1982).
25) 馬嶋，葉木，山野，野木：会話型テスト/デバッグ支援システム HPLTD の機能評価，情報処理学会第24回全国大会講演論文集，5N-6 (1982).
26) Weyuker, E. J. and Ostrand, T. J.: Theories of Program Testing and the Application of Revealing Subdomains, IEEE Trans. Softw. Eng., Vol. SE-6, No. 3, pp. 236-246 (1980).
27) White, L. J. and Cohen, E. I.: A Domain Strategy for Computer Program Testing, ibid., pp. 247-257 (1980).
28) 迫田行介：テストプログラム記述言語，情報処理，Vol. 22, No. 6, pp. 520-524 (1981).
29) Panzl, D. J.: Automatic Software Test Drivers, Computer, Vol. 11, No. 4, pp. 44-50 (1978).
30) 春原，大井，関本，中村：高位言語デバッギングシステム SOLDA，情報処理学会論文誌，Vol. 20, No. 5, pp. 405-411 (1979).
31) 大島，林，海永，薄井：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，Vol. 62, No. 12, pp. 47-52 (1980).
32) Chusho, T. et al.: HITS: A Symbolic Testing and Debugging System for Multilingual Microcomputer Software, Proc. NCC '83, pp. 73-80 (1983).

(昭和58年1月6日受付)