5 テスト

5.1 概　　要

5.1.1 プログラム検証の方法

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．その方法としては，図5.1に示すように，

① 仕様書からプログラムを自動生成

② 仕様書とプログラムの等価性証明

③ テストデータによる動的テスト

などが考えられる．

最初の自動プログラミング方式はプログラム開発方式の理想形態であり，常に正しいプログラムが生成されるので，生成されたプログラムを改めて検証する必要はない．この方式を実現するためには，仕様を形式的に記述するための仕様記述言語とその言語で記述された仕様から計算機で実行可能なプログラムへの変換規則あるいは仕様を満たす既開発のプログラム部品の検索技術などが必要である．本方式はまだ研究段階にあり，実用的規模のプログラムの自動生成実現に至っていない．

次に仕様書とプログラムの等価性を自動証明する方式はプログラム検証方式

然に防止するための種々のコーディング規則を守っているか否かをチェックする．

② プログラム複雑度の計算：プログラムの構造の複雑さを測定することにより，きれいな構造のプログラム作りを奨励し，不良作り込みのポテンシャルを減少させる．複雑度の尺度としては，分岐数や構造化コーディング違反件数などのように制御構造をベースにしたものや変数の定義参照関係や制御変数の使用などを含めて分析するもの，モジュール間の関係やモジュールの外部インタフェースに着目するものなど，種々の方式が提案されている．

③ データフロー解析：各々の変数への値の代入がどこで行なわれ，その値がどこで参照されているかという分析を行なうことにより，代入された値がどこでも参照されていないとか，値が代入されていない変数が参照されている，などの不自然な処理を検出する．

④ 手続き呼び出しグラフ：ある手続きとその本体の中で呼び出している手続きとの間の呼び出し関係をプログラム全体の中のすべての手続きについて分析し，一つのグラフにまとめる．このとき，できれば実引数リストと仮引数リストの対応関係も表示する．

⑤ 制御フローグラフ：モジュール単位に制御の流れを分析し，4.3.4で述べたような図式で表示する．流れ図を出力するオートフローや PAD を出力する Auto-PAD などがある．

⑥ プリティプリンタ：コンパイラが出力するソースプログラムリストは行番号，文番号，ブロック構造やそのネストの深さレベルなどの情報を付加するものが多いが，テキスト自身はユーザが入力したときの形で表示する．それに対し，プリティプリンタは，プログラムの制御構造をみやすくするためにブロックのネストが深くなるにつれて対応するテキストの先頭カラムを右へ桁下げして表示する．この操作は自動インデンテーションと呼ばれる．

の理想形態であるが，先の自動プログラミング方式と同様の形式的仕様記述言語のほかにプログラムの意味理解が必要となり，実現は難しい．

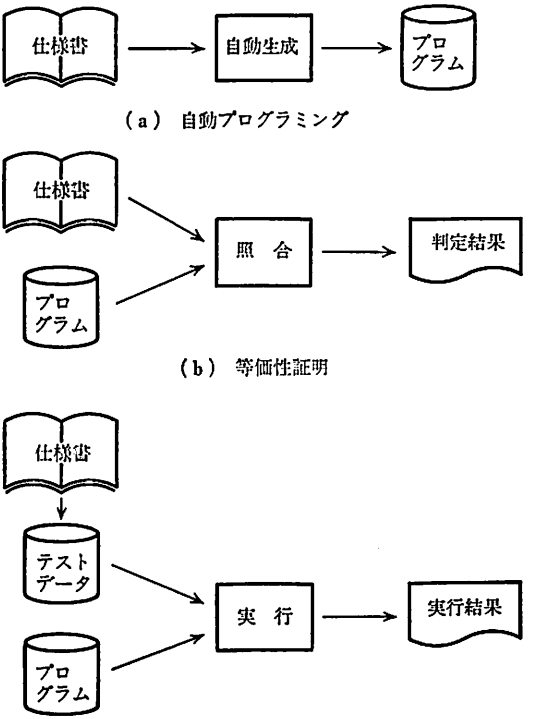


図 5.1　プログラムの検証方法

実際には第3の方法として，テストデータを用いてプログラムを実行し，その結果を確認するという動的テスト法が用いられている．すなわち，多くのテストデータを作成し，プログラムを何度も実行し，各々の実行結果が正しいか否かを確認する方法である．

本章では現在最も一般的に行なわれているこの動的テスト法を中心に述べる．

5.1.2 テスト工程

多くの労力を要する動的テスト法を実施するテスト工程は，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．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なうことが基本であるが，プログラムの作成を人手に頼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なわざるをえない．

プログラム開発工程の後半に位置するテスト工程は以下のような小工程からなる．

① モジュールテスト
② プログラムテスト
③ システムテスト
④ 検査
⑤ 受け入れテスト

このうち，①～③は開発者によるテスト，④はメーカ側の開発部門とは独立した検査部門によるテスト，⑤は顧客側のテストである．

まず，モジュールテストは，単体テストとも呼ばれ，コンパイル単位程度のプログラムモジュールがそのモジュール仕様書どおりに作られているか否かを検査するのが目的である．このテストで誤りが検出された場合は，その原因を調べる範囲が狭いので，誤り原因の除去が比較的容易に行なえるが，モジュールテストが不十分のまま次工程へ進むと，誤りの検出および除去に必要以上に手間どることになる．そのため，このモジュールテストはプログラムの生産性および信頼性向上に重要な作業であるが，このテストを実施するためにはテスト環境の作成という余分の作業が必要になることから，実際には十分に行なわれないことも多い．しかし，最近は，あとで5.3節で述べるようなモジュールテスト支援ツールが普及してきて，改善されている．

次にプログラムテストは，いくつかの関連するモジュールを結合して実行

し，互いのインタフェースが一致していることを確かめるのが目的である．したがって，モジュールテストではモジュールの内部仕様の検査が主であるが，プログラムテストではモジュールの外部仕様（外部インタフェース）の検査が主になる．

システムテストは，プログラム全体が最終的にシステムの機能仕様および性能仕様を満たし，所期の目的を達成していることを確かめるのが目的である．ここまではプログラムの内部構造をよく知っている開発者自身がテストを行なうため，細部にわたってきめ細かいテストが行なわれる反面，テストの内容が偏ったり，見落しが発生しやすい．

そこで，その欠点を補うために，プログラムの出荷前に，その開発にたずさわってこなかった検査部門の第3者が改めて行なうシステムテストが検査である．ユーザに提供する前の最終テストにあたるため，網羅的なテストや種々の異常なケースのテストが徹底して実施される．

受け入れテストは，特定の顧客の注文によって開発したプログラムの納入時に顧客自身によって行なわれるテストで，顧客の要求仕様どおりに作られていることを確認するのが目的である．このような特定の顧客向けでない，不特定多数向けのプログラム製品の場合は，メーカ側の検査部門の行なう検査が受け入れテストを兼ねることになる．

5.1.3 テスト技術

動的テスト法を用いたプログラムの検証にプログラム開発費用の半分近くを費すのは，実用面で次のような問題があるためである．

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 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「誤りをみつける」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.

次に第2の問題は,この動的テストが,図5.2に示すように,多くの人手作業を必要とすることによる.その主なものを以下に示す.

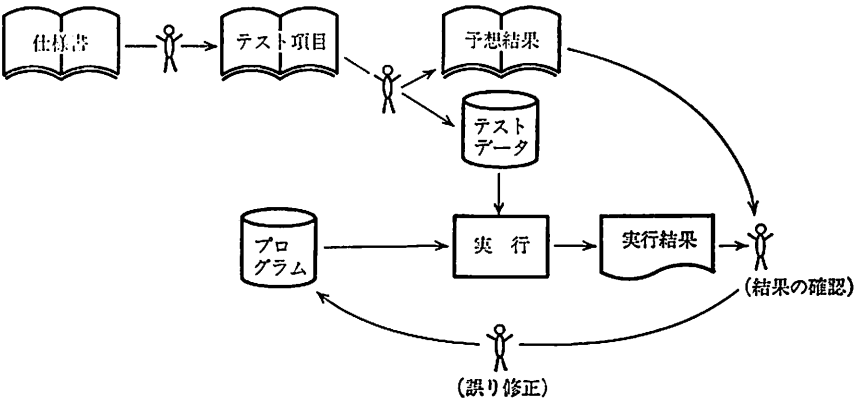


図 5.2 動的テストの実行手順

① 仕様書からテスト項目の選択
② テスト項目に対応するテストデータと予想結果の決定
③ テストデータを用いてプログラムの実行
④ 実行結果が予想結果に一致することの確認
⑤ 誤り検出時の原因究明とプログラムの修正

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以下で詳しく述べる.

5.2 テストデータ作成

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

ストを行なわざるをえない．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．

5.2.1 理論的背景

J. B. Goodenough らは，テストデータ選択基準に対する二つの概念，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 が少なくとも一つ存在するとき，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みつかっていないため，実際には経験的手法を用いてテストデータの作成が行なわれている．

5.2.2 機能テスト

これはプログラムの機能仕様からテスト項目（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．

(1) 同値分割法

機能テストの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と呼ばれる．表 5.1 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項目選択に同値分割法を適用

表 5.1 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
≈	≈	≈

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項目については，一つのテストデータが多くのテスト項目を含むように選び，効率よ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項目については，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．

たとえば，表5.1の場合，有効同値クラスのテスト項目としては，配列名称が1文字のものと6文字のものおよび次元の数が1個のものと7個のものという4種類が選択される．そして，実際のテストデータとしては，

① X(100)

② ABCDEF (10, 20, 30, 40, 50, 60, 70)

の2個のテストデータを作成すれば，上の4種類のテスト項目を満たすことになる．

次に無効同値クラスのテスト項目としては，配列名称が0文字のものと7文字のものおよび次元の数が0個のものと8個のものという4種類が選択される．無効な入力のテストは個別に行なう必要があるので，実際のテストデータとしては，たとえば，

① (100)

② ABCDEFG (10, 20)

③ X ( )

④ P(10, 20, 30, 40, 50, 60, 70, 80)

の4個のテストデータを作成する.

(2) 原因結果グラフ法

機能テストの自動化ツールとしては，機能仕様を組み合わせ論理で表現し，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み合わ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.

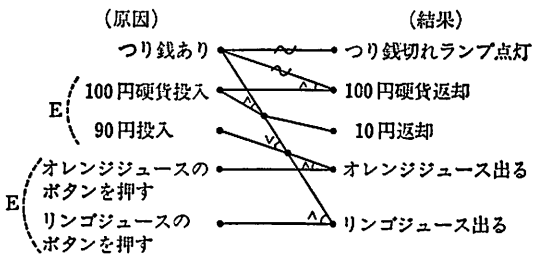


図 5.3 原因結果グラフの例

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5.3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 は二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，ほかに，いずれかは必ず成立する(I)，常に一つだけ成立する(O)，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，不必要なテスト項目の生成を防いでいる.

本技法の支援ツールの一つとして，あとの(4)項で詳しく述べる日立のAGENT (Automated GENeration for Test cases) の場合は，図5.3の原因結果グラフを入力することにより，図5.4に示すような6個のテスト項目が生成される．たとえば1番目のテスト項目は「つり銭なしの状態で100円硬貨を投入し，オレンジジュースのボタンを押した場合は，つり銭切れランプが点灯して，100円硬貨が返却される」という組み合わせが選択されている．

	原因／結果	テスト項目					
		1	2	3	4	5	6
入力	(n1) つり銭あり		×	×		×	
	(n2) 100円硬貨投入	×		×		×	×
	(n3) 90円投入				×		
	(n4) オレンジジュースのボタンを押す	×	×	×	×		
	(n5) リンゴジュースのボタンを押す					×	×
出力	(x1) つり銭切れランプ点灯	×			×		×
	(x2) 100円硬貨返却	×					×
	(x3) 10円返却			×		×	
	(x4) オレンジジュース出る			×	×		
	(x5) リンゴジュース出る					×	

図 5.4 原因結果グラフに基づくテスト項目の自動生成の例

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項目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2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，および対象とするプログラムの機能が組み合わせ論理表現向きである必要がある．

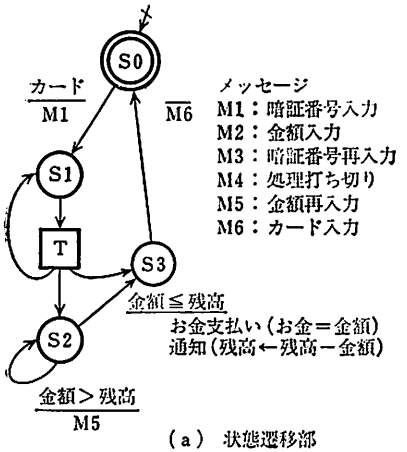
(3) 状態遷移図による方法

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ほかに状態遷移図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項目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．

(4) 機能図式による方法

先に述べた原因結果グラフおよび状態遷移図に基づく機能テスト法の双方の相補的な特徴を生かすために両者を組み合わせた機能図式に基づく方法がある．機能図式は，機能仕様の動的な部分を状態遷移で表現し，静的な部分を論理関係(決定表あるいは原因結果グラフ)で表現する．具体的には，状態遷移の各状態での条件の組み合わせを表現するのに決定表あるいは原因結果グラフを使用する．どちらを使用するかは任意に選択することができる．図5.5 に，現金自動支払機の機能仕様を機能図式で表現した例を示す．この機能図式は，動的なプログラムの機能仕様を表現することができるだけでなく，決定表により条件の組み合わせの網ら性をチェックすることもできる．また，複雑さが各状態に分散されるため，プログラム



(a) 状態遷移部

決定表T：

E	入力	暗証番号登録と一致	Y	N	N
		暗証番号3回誤り	N	Y	N
	出力	M2	*		
		M3			*
		M4		*	
		登録取り消し		*	
	遷移先	S1			*
		S2	*		
		S3		*	

(b) 論理関係部

図 5.5 機能図式による現金自動支払機の機能仕様の例

の大きさによる影響をあまり受けないという利点もある．

この機能図式から漏れや重複のないテスト項目を自動生成するツールとして著者らが開発した AGENT がある．その主な処理手順は次のとおり

である.

① 入力

テキスト形式で記述された機能図式を入力し，状態遷移部と論理関係部に分離し，同時に，機能レビュー用ドキュメントとして，ソースリストのほかにマトリクス形式の決定表リスト，状態遷移表リストなどを出力する.

② 状態遷移の構造化

状態遷移部については，より強力なテスト基準で遷移経路を網らするために，連結，選択，反復という3種類の構造だけからなる構造化状態遷移に変換する.

③ 部分テスト項目の作成

論理関係部については，各状態での条件の組み合わせを効果的に確認するのに必要かつ十分な部分テスト項目を作成する.

④ テスト項目の合成

構造化状態遷移から，選択についてはそれぞれの場合を確認し，反復については反復しない場合と反復する場合の2通りを確認するテスト経路を作成する．このテスト経路の各状態に遷移先がテスト経路の次の状態と一致する部分テスト項目を割り当てて全体のテスト項目を合成する.

本ツールの使用経験から次のような利点が確かめられている.

① 信頼性

条件の組み合わせを網らするテスト項目の作成，特に，人間にとっては考えにくい異常ケースのテスト項目の強化，機能仕様の形式化段階での不良発見，などにより信頼性が向上した.

② 生産性

従来法によって，どの程度十分なテスト項目を作成していたかによるが，同等の品質を保証するという前提で比較した場合には，テスト項目，特に，正常ケースのテスト項目の圧縮によりテスト実施作業の削減が図れた.

③ 管理性

従来法によるテスト項目の作成は，ほとんど経験者によって行なわれていたが，新人でも機能仕様を機能図式で表現することによって，品質のよいテスト項目を作成することができるようになり，新人の早期戦力化に効果があった．

5.2.3 構造テスト

(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項目を選ぶもので，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わし，そのパス解析に基づいてテスト項目を選ぶもので，そのときの選択基準としてテスト網ら性を表わ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．

たとえば，図5.6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.

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

図 5.6 例題プログラム

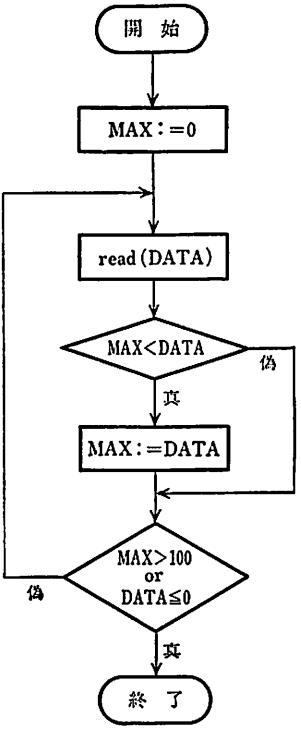


図 5.7　例題のフローチャート

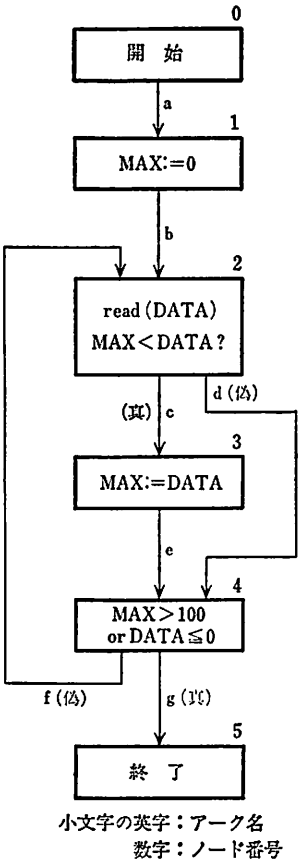


図 5.8　例題の制御フローグラフ

7 に，制御フローグラフを図 5.8 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まりを一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 5.8 をみると，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いくつかの簡易化されたテスト網ら基準が

表 5.2 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
	繰り返し条件成立回数	①（0回），②（1回），③（2回）
	繰り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個の分岐の組み合わせ ($n=2$)	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	abdfdg	なし
⑥	abcefceg	(1，101)
⑦	abdg(gはg21)	(−1)

用いられる(表5.2)．

(i) 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5.8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なる．

(ii) 全アーク網ら(全分岐網ら)

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5.8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．

(iii) 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なう．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5.8の例では，(i), (ii)の2個のパスのほかに，abcefdfdg を加える．

(iv) 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5.8の例では，fを対象外とすると abceg 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．

(v) 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，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が提案されている．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．

① 分岐条件変数参照網ら：二つ以上の出力アークを有するノードは分岐

条件をもつが，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 5.8 の例では，上記(i)～(iv)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，たとえば abcefceg が必要である．

② 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，①の基準を厳しくしたもので，all-uses と呼ぶ．

③ 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，①，②よりも厳しい．all-du-paths と呼ぶ．

(vi) 分岐アークの組み合わせによる基準

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，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み合わせからなるすべての部分パスを網らする基準 TER (Test Effectiveness Ratio)が提案されている．

$$\mathrm{TER}_{n+2} = \frac{\mathrm{Se}(n)+\mathrm{Pe}(n)}{\mathrm{S}(n)+\mathrm{P}(n)}$$

ここで，$\mathrm{S}(n)$は n 個の分岐アークの組み合わせからなる部分パスの数，$\mathrm{P}(n)$は n 個以下の分岐アークの組み合わせからなる完全なパスの数を表わし，$\mathrm{Se}(n)$ および $\mathrm{Pe}(n)$ はそれぞれの実行された数を表わす．

(vii) 分岐条件の詳細化

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なる場合は，各々の真，偽の場合をテストする．図5.8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5.9のように分解して網らテストを行なう．

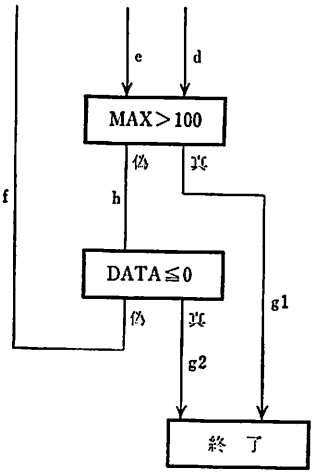


図 5.9　制御フローグラフの分解例 (1)

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種類のテストを行なう．図5.9の例では，分岐条件が比較式となっている二つのノードの出力アークを各々図5.10のように3個にして網らテストを行なう．

(vii) モジュール呼び出し

以上に述べてきた網ら基準は個々のモジュール内の制御構造を主対象としていたが，プログラムテストやシステムテストではモジュール間インタフェースのテストも重要であり，モジュール呼び出しに関連して以下の基準がある．

① 全モジュールを1回以上実行する．

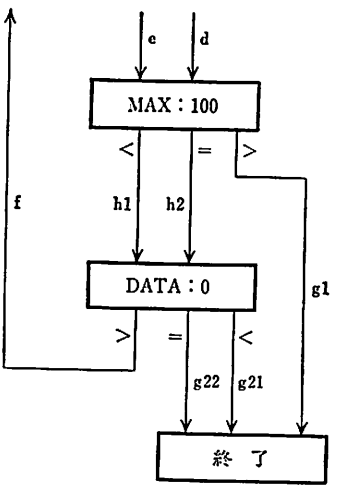


図 5.10　制御フローグラフの分解例 (2)

② 全モジュール呼び出しを1回以上実行する.

(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.

(i) 実行不可能パスの選択

前項のはじめに述べた手順①において, パスを機械的に選んだ場合, パス条件が常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の誤りに基づくものや, たと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, そのほかの場合は実行可能なパスと置き換える必要がある. たとえば, 表5.2のパス⑤は図5.8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,

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

であり, 適当に簡約化すると,

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

となり, この条件は常に偽となる. 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. このような問題の煩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.

(ii) 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5.11の例のように全体をいく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.

(iii) 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，先に述べたようないくつかの中間基準が提案されている．

(iv) 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．

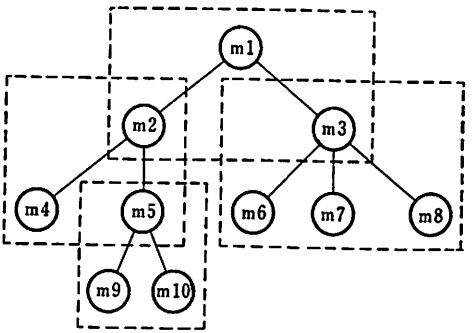


図 5.11 大規模ソフトウェアの構造テスト法

(v) 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．

(vi) 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ほか，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もつが，全分岐網ら基準などの簡易基準では一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 %未満のところで品質が過大に評価されるという欠点がある．

(3) 全分岐網らの改良方式

著者らは，「ある分岐の実行に伴なって必ず実行される他の分岐」を全分岐網ら達成に非本質的な分岐としてテスト網ら率測定の対象外とする方式を考案することにより，現在最も一般的に用いられている全分岐網ら基準の欠点を改善した．

この方式を簡単に説明しておくと，まず有向グラフの二つのアーク 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) なるアークが一つで，かつ次の4条件のいずれかを満たすものである．

(R1) $\mathrm{IN}(x) \neq 0 \wedge \mathrm{OUT}(x) = 1$

(R2) $\mathrm{IN}(y) = 1 \wedge \mathrm{OUT}(y) \neq 0$

(R3) $\mathrm{OUT}(x) \geq 2 \wedge x \in \mathrm{IDOM}(w)$

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

(R4) $\mathrm{IN}(y) \geq 2 \wedge y \in \mathrm{DOM}(w)$

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

なお，$\mathrm{IN}(x)$, $\mathrm{OUT}(x)$ はノード x の入力および出力アーク数，w は隣接ノード，$\mathrm{DOM}(w)$ と $\mathrm{IDOM}(w)$ 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．R3 の $\mathrm{IDOM}(w)$ を含む条件は，x から出発して x に戻るようなループが存在し，かつそのループがアーク (x, y) を含まないことを意味している．同様に R4の $\mathrm{DOM}(w)$ を含む条件は，y から出発して y に戻るループが存在し，かつそのループがアーク (x, y) を含まないことを意味している．

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．すなわち，被相続子アーク(p)が実行されたときは必ずその相続子アーク(q)も実行されるの

で，このような相続子アークを網ら基準の対象外としておいても，すべての被相続子アークが実行されたときはすべてのアークが実行されたことになる．

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ほか，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にも有効である．

この方式を図5.8の制御フローグラフに適用すると図5.12に示すようにアーク数が7個から3個に減少する．図中のRnはそのアークの消去に用いられた規則を示している．R1を適用したアークはもともと分岐ではないので網ら基準の対象となる分岐アークは5個から3個に減少している．

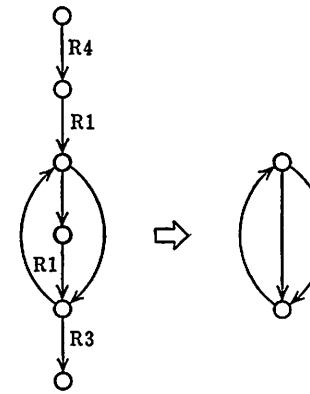


図 5.12 パステスト向き有向グラフ簡約法

(4) 支援ツール

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ない，テストケースやパス条件を自動生成するものがある．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なう．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み込まれたものがある．

著者らも専用ツールとして前項(3)で述べた全分岐網ら基準の改良方式を適用したSCORE(Source-level COverage Rate Evaluator)システムや，構造テスト機能を組み込んだテスト実行支援システムHITS (Highly Interactive Testing-and-debugging System) を開発した経験がある．

この SCORE を用いて N. Wirth が作成した PL0 パーサである 283 行の Pascal プログラムの構造テストを行なった結果を図 5.13 に示す．従来方式の C_1 基準ではこのプログラムの全分岐数 136 個を対象としているが，改良方式の C_{pr} 基準ではそのうちの 55 個が R2，R3，R4 規則により分岐網らに非本質的な分岐として取り除かれ，残り 81 個の本質的な分岐のみを対象としているために，より線形に近い特性を示し，品質の過大評価傾向を減じている．この改良方式は従来方式に比べ，着目する分岐数を約 60 %に減らせることが SCORE の適用経験から得られており，構造テストの効率化にも効果的である．

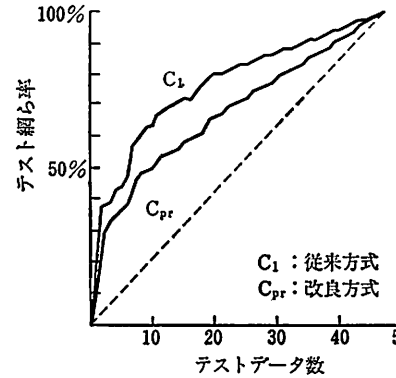


図 5.13 全分岐網ら基準の改良方式適用例

構造テストツールの処理方式としては，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．計測用コードとして実行回数をカウントする命令をソースプログラム内の分岐個所に挿入する場合の例を図 5.14 に示す．この関連処理として，カウンタ用変数の宣言およびプログラム実行終了時にカウンタの値をファイルへ書き出す命令の挿入も必要である．

```
if MAX < DATA
  then MAX := DATA
```

⇨

```
if MAX < DATA
  then begin
         COUNTER(i) : =COUNTER(i)+1;
         MAX := DATA
       end
  else COUNTER(i+1):=COUNTER(i+1)+1
```

図 5.14 テスト網ら率計測用命令の挿入例

このような計測用命令をプリプロセッサがソースプログラムに挿入する方式を図5.15に示す．プリプロセッサはソースプログラムの分岐個所と

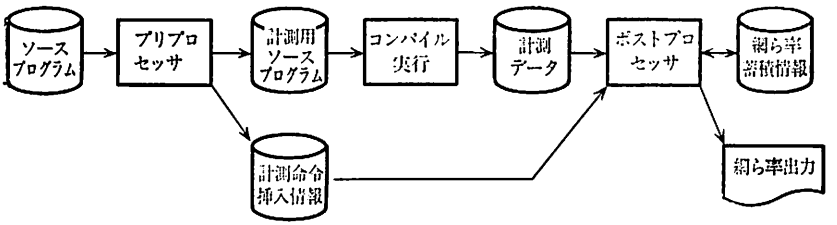


図 5.15 テスト網ら率測定方式

カウンタを対応づける計測命令挿入情報をファイルに出力しておく．ポストプロセッサは計測命令の挿入されたプログラムの実行によって得られた計測データと先の計測命令挿入情報を用いてテスト網ら率の計算および未テスト部分の検出を行なう．実際にはテスト網ら情報は複数個のテストケースの累積結果が重要なので，過去の結果の和集合を網ら率蓄積情報ファイルに保存しておく．

このような方式のほかに，マイクロコンピュータ用プログラムの場合などで，外づけの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を用いる方式や大型コンピュータによるシミュレーション実行時に測定する方式もある．

(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5.2.2(1)で述べた同値分割法で行な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．

5.2.1で述べたテスト基準は全入力テストが必要となるため，それに代わる方法として，ある部分領域 S 内の一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がある．

別に，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がある．たとえば，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ところの3点を選べば，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．

5.3 テスト実行

5.3.1 支援機能

現在の動的テスト法によるプログラムのテストには，効果的テスト技法と効率的なテスト技法の二つの課題があることはすでに5.1.3で述べた．前節5.2ではそのうちの効果的テストを目的としたテスト項目作成技法について説明した．本節ではもう一つの課題である効率的なテスト技法について述べる．

プログラムのテスト実行を効率よく行なうためには，先の図5.2のテスト手順で示したような多くの人手作業をできるだけ自動化することが重要であり，それを目的としたツールとして，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ある．その代表的機能としては，

① モジュール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
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．

まず①のモジュールテストは図5.16に示すような三つの利点がある．第

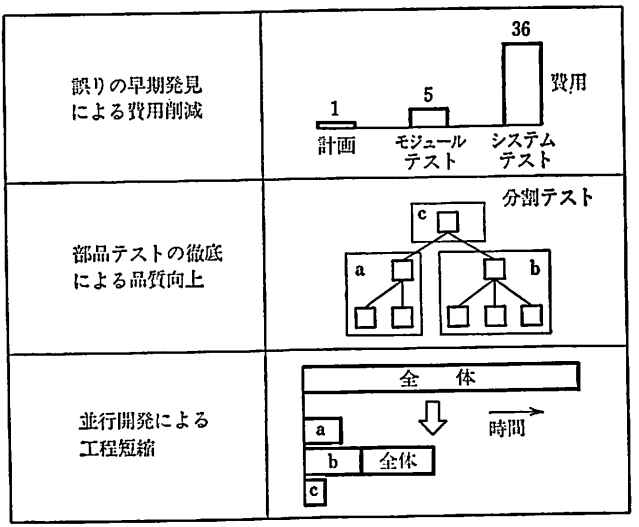


図 5.16　モジュールテストの利点

1には，プログラム誤りの検出と修正のための費用はその誤りの検出が遅れるほど増大し，モジュールテストとシステムテストではその費用が1桁違う．第2には，部品テストを徹底することにより，システムテストでは行なえないような細かいテストが可能となり，品質が向上する．第3には，多人数で大規模なプログラムを開発する場合，テスト作業を並行して実施できるため工程が短縮できる．

モジュールテストによるプログラム誤りの検出，修正費用はシステムテスト時に比べて1桁少ない．そのため，モジュール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．そのためには第②項のテスト手続き言語が必須である．

さらに，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なうデバッグ作業は試行錯誤的方法が用いられるため，第③項のような会話型デバッグ機能が必要になる．

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第④項の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．

5.3.2 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，5.3.1 で述べたような機能を有するものはまだ少ない．

ここでは，一例として，著者らが開発したテスト支援システム HITS を取り上げ，テスト支援機能を具体的に説明する．

本システムはマイクロコンピュータ用のアセンブラおよび高級言語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，モジュール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1元化，バッチ処理と会語型処理の両用化，などがある．

(1) 環境模擬機能

モジュールテストの環境設定作業を容易化するために，図5.17に示すように，未作成の上位モジュールを模擬するドライバ定義機能，未作成の

下位モジュールを模擬するスタブ定義機能のほか，入出力処理や外部データを模擬する機能がある．

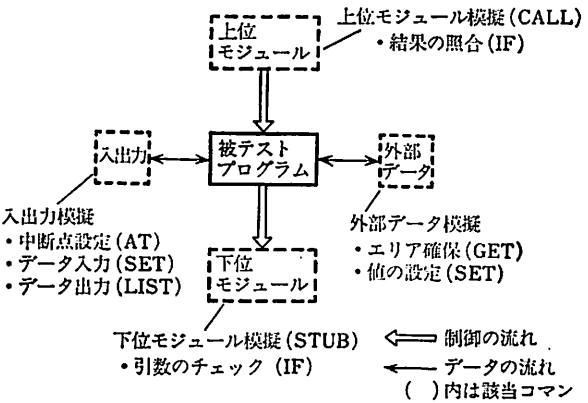


図 5.17 モジュールテストのための環境模擬機能

ドライバ定義には，被テストプログラムを呼び出す CALL コマンドや結果の照合に IF コマンドを用いる．スタブ定義では，模擬するモジュール名を STUB コマンドで指定し，変数の値の設定，照合，表示には各々 SET, IF, LIST コマンドを用いる．入出力処理も同様であるが，模擬すべき入出力の指定には中断点設定用の AT コマンドを用いる．外部データ用のメモリエリアの確保は GET コマンドを用いる．

プログラムテストでは，最上位モジュールの単体テストから始めて，段階的に下位モジュールを結合してテストするトップダウンテストとまず最下位モジュールの単体テストから始めて，上位モジュールを結合してテストしていくボトムアップテストなどのやり方がある．実際には，対象とするプログラムの性質により，これらの手法を臨機応変に組み合わせて実施することになる．

(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ほか，直接に端末入力してもよい．

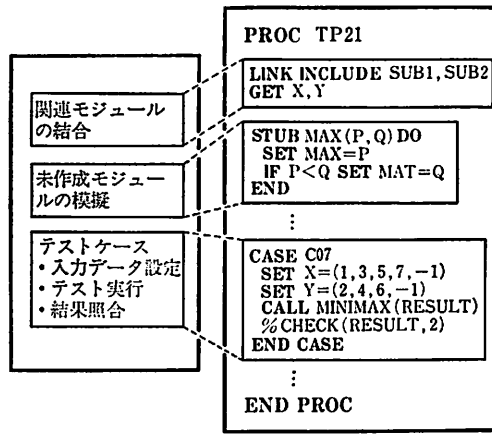


図 5.18 テスト手続きの例

テスト手続きの記述例を図 5.18 に示す．この例では，テスト手続き TP21 は，まず被テストプログラム SUB1 と SUB2 の結合と外部データ X と Y のメモリエリアの確保および未作成の下位モジュール MAX のスタブ定義を指定している．実際のテスト条件はテストケース C07 の中で指定されており，X と Y に値を設定後，手続き MINIMAX を呼び出し，その実行後に結果の照合を行なう手順になっている．% CHECK は以下のように定義されたマクロコマンドで，&1，&2 は第 1，第 2 引数をさす．

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

以上の例からも分かるように，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項目を実行する機能，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．

(3) 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．詳細は次節で述べる．

(4) 構造テスト支援機能

全文網らや全分岐網ら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なう．この情報は累積することができる．

5.3.3 テスト手順

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，前節および本節で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．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項目を作成する．

② 次に同値分割法でテスト項目を選ぶ．

③ これらのテスト項目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．

5.4 デバッグ

5.4.1 概　　要

デバッグとは，テストによって検出されたプログラムの誤りの原因を究明し，それを取り除く作業であり，誤りの原因である虫（バグ）を取るという意味で虫取りとも呼ばれる．デバッグの基本は，

① プログラムの実行の流れを調べること

② その実行の流れに沿って変数の値などの状態の変化を調べること

によって，誤ったテスト結果をもたらした原因がどこで発生したかを確認し，プログラムを修正することである．

そのためのデバッグ支援機能やツールは比較的に早くから発展してきている．その一部分についてはすでに4.3.3の特定言語向きプログラミング環境，4.4.3のデバッグ用コンパイラ，4.5節の静的プログラム解析に関連して述べた．以下では，主要なデバッグ機能について述べる．

5.4.2 デバッグ機能

(1) 状態の表示

プログラム実行のある時点でのプログラムの状態を表示する機能として，次のようなものがある．

① 変数の値の表示

② プログラム実行のために使用している主記憶メモリの16進数表示(ダンプ)

③ 中央演算装置（CPU）のプログラムカウンタやレジスタ類の値の表示

高級言語で記述したプログラムのデバッグは上記①で通常は十分であるが，特に複雑な異常現象が発生したり，アセンブリ言語プログラムのときは，②，③などもデバッグに重要な情報となる．

(2) 実行の流れの表示

プログラム内の実行文がどのような順序で実行されているかを表示する機能として，次のようなものがある．

① 指定された実行文の実行情報の表示

② 命令トレース：すべての実行文の実行情報（文番号など）を逐次的に表示

③ 分岐トレース：分岐を伴なう実行文の実行情報を表示

④ 手続きトレース：手続き呼び出しおよび戻りに関する実行情報の表示

⑤ 逆トレース：プログラム実行のある時点で，その直前の実行トレース情報をまとめて表示

⑥ プログラム実行のある時点で，メインプログラムからその手続きに至るまでの手続き呼び出しのネスト関係の表示

このようなトレース機能を多用すると出力情報が多すぎてその分析に手間取るため，最初は大まかなトレース機能を用いるとか，トレースの範囲を必要最小限の区間に限定するなどの注意が必要である．

(3) 状態の変化の監視

最初の(1)項で述べた状態の表示機能は，実際のデバッグ時には実行の流れに沿って状態が変化する様子を調べるために用いることが多く，そのための機能として次のようなものがある．

① 中断点設定：指定された実行文の実行時に一時中断し，指定項目の状態の表示

② ステップ実行：実行文を一つ実行するごとに一時中断

③ データトレース：指定されたデータの値が変化するごとにその情報の表示

(4) 一時的変更

誤りの原因を特定するためにプログラムの一部分だけを実行させたい場合に，その前準備として状態を一時的に変更する機能として，次のようなものがある．

① 変数への値の設定

② プログラムがロードされている主記憶メモリの書き換えによる実行文の変更（パッチ）

③ 中央演算装置のプログラムカウンタやレジスタ類の値の設定

5.4.3 支援形態

デバッグ作業の効率は，先に述べたデバッグ機能の有無だけでなく，その使用形態に大きく影響される．

(1) 16進数 vs. シンボリック

従来のデバッグ機能は，データやプログラムのアドレスおよびデータ値の表示や設定に16進数表示を用いるものが多かった．この場合はユーザ側で16進数と論理的な値の間の変換作業が必要となり，手間がかかる．最近は，データやプログラム内アドレスの指定にはソースプログラム内の変数名，名札名，文番号などのシンボルを用い，値の表示もそのデータ型の定数表示を用いるものが主になってきた．このような特徴を強調するときにはシンボリックデバッガと呼ぶ．

(2) バッチ処理 vs. 会話型処理

従来，プログラムの実行はその開始から終了まで一括してコンピュータに任せるバッチ処理が主であった．それに対応してデバッグも，ソースプログラム中にデバッグ用の文を挿入したり，コンパイラオプション機能としてデバッグ機能を指定して，再コンパイルとプログラム実行を行ない，その結果の出力リストを分析する方法がとられていた．しかし，この方法では1回の実行に要する手間が多いだけでなく，きめ細かい検査に不向きであった．デバッグ作業は本質的に試行錯誤的な検査を臨機応変に行なう必要があり，会話型処理が適している．最近では会話型のデバッグ支援機能を有するデバッガが一般的になっている．

(3) 他のツールと連係

デバッグ作業には，プログラム誤りの原因究明後に，ソースプログラムの修正，再コンパイル，テスト実行などの作業が伴なう場合が多い．した

がって，このような処理全体の効率化のためには，エディタ，コンパイラ，テスト支援システムなどとの連係が重要である．その1例として以下のような機能が有効である．

① デバッガの中からエディタを呼び出し，プログラムを修正する．

② エディタの中から，コンパイラやデバッガを呼び出し，編集中のプログラムのテスト実行を行なう．

③ ソースプログラムに関してコンパイラが作成したシンボル情報をデバッガが利用して高機能化する．

④ テスト支援システムのテスト手続きの実行をデバッガの制御下で行なう．

特に上記④は大規模プログラム開発時に重要である．従来，アセンブリ言語でのプログラム開発時にはデバッグ作業の苦労が多かったことなどから，デバッグという言葉の中にテスト作業を含めて用いてきた．たとえばテスト終了のことを「デバッグが終わった」などといっていた．しかし，ソフトウェア工学の発展とともに，テストこそがプログラム開発に本質的な作業でデバッグはそれに付随するものという考えが確立している．

したがって，5.3節で述べたように，テストを系統的に行なうためには，テスト仕様書を作成し，それに対応するテスト手続きを記述し，テスト支援システムで実行するのがよい．そして，誤りが検出された場合は，そのテスト手続きをデバッガを用いて会話型実行を行なうのが効率的である．このとき，デバッグ用のコマンドは一時的に用いるものなので，テスト手続きやソースプログラムの中に書き込むことはせず，端末から適宜投入する方式がよい．

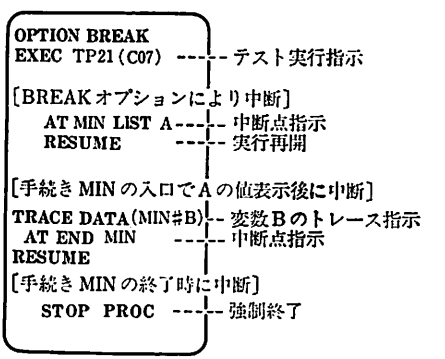


図5.19 テスト手続きを用いた会話型デバッグの例

先に 5.3.2 で紹介したテスト支援システム HITS はこのような考えで開発されたもので，誤りを検出したテスト手続きを会話型実行できる．そのとき，一時的デバッグコマンドを投入して，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，表示のほか，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 5.19 は図 5.18 のテスト手続きを用いたデバッグの例である．

5.5 動的プログラム解析

以上に述べた基本的なテスト技法のほかにも種々の技法が研究されている．

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てプログラムの動作を解析する記号実行は，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．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なうものもある．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．

実用的な動的アナライザとしてプロフィール解析ツールがある．これはプログラムを実行したときに各実行文の実行回数を測定するものである．この用途としては，

① 実行回数の分布からプログラム誤りを検出する．

② 実行回数が 0 の実行文を検出し，その文を実行するテストデータを追加する．

③ 実行回数の多い部分のプログラム処理を改良して高速化する．

などがある．

最初の ① は，5.4.1 で述べた実行の流れの表示を用いたデバッグ方法の 1 種と考えられる．次の ② は，5.2.3 で述べた構造テスト法における全文網ら

基準を用いた方法に相当する．③は4.4.4で述べたコンパイラによる最適化処理と同様のことをプログラマ自身がソースプログラム上で行なう場合に役に立つ．

各実行文の測定方式は，5.2.3で述べたテスト網ら率を測定する方式と同様に，ソースプログラムに計測用カウンタを挿入する方式が一般的である．

実用ツールとしては九州大学で開発されたFORTRAN用のFORDAP，Pascal用のPASDAPなどのほか，代入文の値の最大と最小を表示して上記①の目的を強化したものや分岐を伴なう実行文の分岐が成立した回数と不成立の回数も合わせて表示して上記②の目的を強化したものなど，あるいは各実行文の実行時間を加味して全体の実行時間の分布をより正確に測定することにより上記③の目的を主にしたものなど，種々のツールが開発されている．たとえば，UNIXのprofコマンドは，Cプログラムの関数の実行回数と実行時間を解析する．

5.6 プログラム検査

5.6.1 検査の目的

ここでいうプログラム検査とは，プログラムの出荷前にそのプログラムが製品として十分な品質に達していることを見極めることである．

この検査の前工程で行なわれるモジュールテスト，プログラムテスト，システムテストがそのプログラムの開発部門自身で実施されるのに対し，この検査は開発部門とは別組織の検査部門が実施する．したがって，検査の基本は，そのプログラムの顧客の立場に立って，要求仕様に代表される顧客の意図を満足していることを確認し，製品としての合否の判定を行なうことである．

しかし，実際には，検査部門の主な業務は，単にまったく第3者の立場で合否判定することではなく，製品プログラムを合格の基準に達した品質のものに仕上げるために開発部門に協力することである．そのため，検査部門はプログラム開発の早い段階から活動を始め，設計仕様書などの文書の検査も行ない，

設計不良による手戻りが後工程で発生しないようにしている．

プログラムのテスト技術そのものはすでに述べてきたので，本節では，プログラム検査の視点での品質管理，品質評価技法について述べる．

5.6.2 品質評価技法

(1) 品質推移・目標値管理方式

品質推移・目標値管理方式は，あらかじめ設定した摘出不良件数の目標値と実績値の比較を行ない，予実績を評価するとともに，テストの小工程ごとの摘出不良実績から品質の推移を評価する方式である．品質推移は，評価対象工程に至るまでの各工程の摘出不良件数に重みづけ係数をかけて得た数値の和によって評価値を表わし，その変化で表現する．この方式では，後工程に不良が多いほど評価値が高くなり，その逆に，前工程で不良を多く摘出し，後工程で摘出件数が下がれば，評価値が低くなるように重みづけ係数を設定している．したがって，この評価値は工程が進むにつれて下がるほどよいことになる．

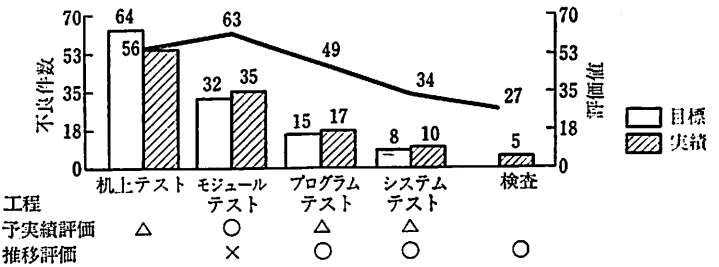


図 5.20 QC（品質管理）グラフの例（文献 59）より引用）

図 5.20 に示す QC (Quality Control) グラフは，本方式の具体例である．不良件数の目標値と実績を棒グラフ，品質推移を折線グラフで表わし，ソフトウェアの品質が一目で分かるように工夫してある．予実績評価は，目標値と実績の比較により，○，△，×の3段階評価をしている．品質推移は，折線グラフが下がれば○，同じでは△，上がると×で評価している．

(2) 不良予測技術

プログラムの中にどのくらいの不良が残っているかを推定する方法はいろいろ試みられているが，そのいくつかを紹介する．

(i) 累積不良予測

これは，時系列的な不良の累積曲線を成長曲線にあてはめて，不良の管理と予測を行なう方法である．その例を図5.21に示す．まずテスト開始前にこれから実施するテスト工程の進捗に応じて不良が摘出される度合を上限と下限の予想曲線で示す．そしてこの図の上に不良検出の累積実績値を記入していくとともにその実績に基づく不良発生予測曲線を描いていく．実績値が上下限を超えた場合や不良発生予測曲線が収束値を上回る場合は原因を調べて対策をとる必要がある．

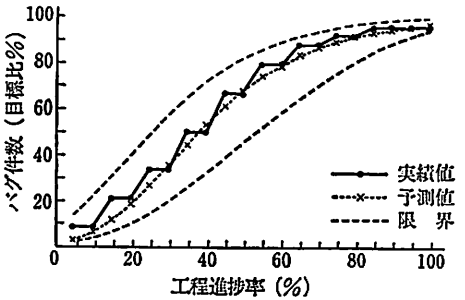


図 5.21　累積不良予測の例（文献 59）より引用）

(ii) 先取り評価手法(探針)

これは，製品の正式の検査に先立って，開発部門のテストの最終段階におけるプログラムの品質を検査部門が実際に測定，評価するものである．製品検査と同様の方式で実施し，このときの摘出不良件数から製品検査時の不良件数を統計的手法で推定することにより，品質を評価する．

この探針結果に基づいた推定残存不良件数を開発部門に提示し，その品質向上対策を逐行させる．この方式により，製品検査前の不良の先取りが行なわれる．

(iii) 統計的手法

統計的手法の一つとして，意図的に作り込んだ不良の検出率から残存不良件数を推定する方法がある．たとえば，p個の不良を意図的に作り込んだプログラムをテストしてa個の不良を検出したとする．そのa個の中に

意図的に作り込んだ不良が q 個含まれていれば，このテストでの不良検出率は q/p となる．したがって，本来の不良総数は $(a-q)\cdot p/q$，残存不良件数は $(a-q)\cdot(p-q)/q$ となる．

5.6.3 不良分析

検出された不良の分析は，残存不良件数の推定のほかにも類似の不良の除去や今後の不良防止に有用な手掛りをもたらす．そのためにはテストの初期段階から，検出した不良をあらかじめ定めたフォーマットの記録票に残すようにする．この不良記録票は次のように用いる．

① 不良の検出者は，検出日時，検出者名，検出に用いたテスト項目番号，現象などを記入する．

② 不良の対策者は，修正日時，対策者名，修正モジュール名，不良原因と修正内容などを記入する．

③ プロジェクト管理者は，未対策の不良の処理と対策完了のものの確認を行なう．

このような不良記録票を次の3種類の観点で分類して分析することも行なわれている．

① 不良の作り込みフェーズ：要求定義，設計，コーディング，テストなど．

② 不良原因：設計，インタフェース，データ定義，論理，データ処理，計算など．

③ 重要度：極めて重要，重要，普通など．

このうちの不良原因については各項目がさらに数種類の詳細項目に分かれる．この分類法に基づくいくつかの分析結果では，不良の60％以上が設計フェーズで作り込まれていること，不良原因では設計誤りが全体の20～25％を占め，最も多いことなどが明らかになっている．

6 プログラミングツールの今後の方向

6.1 山モデルの実現

本書では，変形山モデルに沿ってプログラミングツールを説明してきた．しかし，第1章で述べたように，これは現状における一つの妥協であり，将来的には，山モデルに沿ったプログラミングツールの体系を確立することが望まれる．

変形山モデルと山モデルの違いは以下のように要約される．

(1) 変形山モデルでは，プログラミング工程以降でテストや確認を行なうのに対し，山モデルでは，確認やテストを要求定義や設計と並行に進める．

(2) 変形山モデルでは，テストを合成の過程，確認を分解の過程として捕えるのに対し，山モデルでは，確認を要求定義と同じ合成の過程，テストを設計と同じ分解の過程として捕える．

山モデルでは，要求定義の後に確認を行ない，設計の後にテストを行なうというのではなく，できるだけ早い時点で不良を発見するために，要求定義と確認，設計とテストを並行に進めるのである．したがって，山モデルに沿ったプログラミングツールの体系を確立するためには，要求定義工程における確認支援ツールと設計工程におけるテスト支援ツールを実現しなければならない．

要求定義工程における確認支援ツールについては，フローネット法のところ