ソフトウェア工学　52－18
（１９８７・２・１４）

列概念を用いたテスト支援法について

古川善吾、牛島和夫
九州大学工学部情報工学科

本報告では、ソフトウェアのテストを体系的に行うためにプログラムのモデルを構成する方法と、構成されたモデルに基づいてテストを支援するシステムの概要とを述べる。プログラムの実行状態の遷移系列や命令の実行系列は分岐がないためにプログラムテキストと比較するとチェックや理解が容易である。このことは、プログラムのモデルを有向グラフとして構成した場合に、有向グラフの上の節点や枝の列には分岐がなくチェックや理解が容易なことに対応している。プログラムのモデルとしては、制御フローモデルがよく使用されている。しかし、制御フローモデル上でのテストは、そのテスト品質の向上に限界がある。また、計算機内の全ての記憶の値によって状態を区別する有限オートマトンは、状態数が多く実用的なモデルとはいい難い。そこで、プログラム内のいくつかの変数の値で状態を区別する「変数値状態モデル」を検討した。状態を規定する変数にプログラムの実行の順序を制御する「制御変数」を採用するのが、作成された変数値状態機械の理解の容易さや状態数の点で優れている。また、有効でない値を持つ変数に未定義値を設定することで状態数を削減できる。

変数値状態モデルに基づいてテストを支援するためには、(1)プログラムのモデルの作成、(2)明確なテスト基準に基づくテストケースの作成、(3)誤り発見能力を考慮したテストデータの作成、(4)テストの実行とテスト十分性の評価、等の機能が必要である。

A Testing Support Method with the Concept of Sequences

Zengo FURUKAWA and Kazuo USHIJIMA
Department of Computer Science and Communication Engineering, Kyushu University
Hakozaki 6-10-1, Higashiku, Fukuoka 812, Japan

We discuss a model for systematic testing of programs and an outline of a testing support system. Sequences of states and instructions of a program execution are rather understandable than the program text, because sequences don't branch. Although a control flow graph of a program is usually used for a program model, testing quality with this model is not enough. A finite automaton is another program model, in which states are defined by values of all variables in a program. A finite automaton model is not practical for software testing, because of a large number of states. We investigate a 'variable's value states model' (vvs model) of a program, in which states are defined by values of some particular variables in the program. A variable's value state machine (vvs machine) with control variables in a program is useful model from the view point of understandability and the number of states. Assignment of an undefined symbol (it is a special value) to variables which are not live can reduce the number of states of a vvs machine.

1. はじめに

計算機の応用分野の広まりに従い、信頼性の高いソフトウェアが必須である。ソフトウェアの信頼性を高める方法の1つとしてテストがある。ソフトウェアの開発では、プログラムが正しく動作することを確かめもし誤りがあるならばそれを発見するテストに多額の費用が費やされている。ソフトウェアの開発費用を削減し、ソフトウェアの信頼性を高めるために、テストの効率化とテストの品質向上が急務である。

これまで、状態遷移図と論理関係で記述した機能仕様から系統的なテストのためのテストケースを作成するシステムについて検討してきた[FUR82,FUR84]。機能仕様の入口から出口に到る状態列をテストケースとしてシステムが機械的に作成する。作成されたテストケースを実行することによって誤りを発見できることばかりでなく、テストを実行する際にテストケースを静的に分析することによって機能仕様の誤りを発見できることが信頼性向上に効果があることが分かった。特に、テストを専門に行う立場では後者の効果が大きい。プログラムを十分には理解していなくても、テストを実施するためにはテストケースの分析と理解は十分に成される必要がある。しかも、テストケースが状態の列であるので記述された機能仕様に比較し分析や理解が容易になり、誤り発見の可能性が高くなると考えることができる。

この経験に基づき、有向グラフ上の列を用いたテスト法について、Gourlay が与えたテストに関する数学的な枠組み[GOU83] に基づいて先に検討した。即ち、計算機内の記憶と命令の実行とを有限オートマトンでモデル化し、プログラムに対応して構成した有限オートマトンを有向グラフで表現する。有向グラフ上の開始節点から終了節点に到る節点あるいは枝の列を有限オートマトンのテストケースとする。有限オートマトンの状態と遷移を少なくとも1回実現するテストケースによって有限オートマトンの誤りを発見できる[FUR86]。有限オートマトンのテストケースは、プログラムの実行に対応した計算機内の記憶の値の列あるいは実行した命令の列に対応している。テストケースを実現するテストデータでプログラムを実行し、得られた値の列や実行した命令の列をテストケースと比較することによりプログラムの実行誤りを手続き的に発見できる。

条件分岐を表現できる有向グラフでのモデル化はテストの分野で多く利用され、検討されている。Huang,Howden は、プログラムの制御フローグラフに基づくテスト方法を検討している[HUA75,HOW76]。また、Chow,Jessop,渡辺、古川は、テストの実現やテストデータの作成に状態遷移図を用いている[CHO76,WAT84,JES76,FUR84]。有限オートマトンは、現実世界の自然な形式化にしばしば有用であり[HOP79]、いろいろなもののモデルとして利用されている。プログラムの有限オートマトンモデルは、制御フローグラフに比較して誤り発見能力が優れており[FUR86]、状態遷移図を用いたテストデータ作成の従来方法に比較して新たな記述を行わずに済む点で優れている。しかし、計算機の記憶の値に対応した状態を設定すると状態数が多くなり過ぎて、有限オートマトンモデルに基づく実用的なテスト支援システムを構築できない。そのために、プログラムから有限オートマトンを作成する段階で状態数を削減する方策が必要である。

本論文では、有限オートマトンに基づくテスト支援システムを構築するために実用的なモデルを作成する方策について検討し、テスト支援システムの要求機能について述べる。2章、3章で有限オートマトンに基づくテスト法について簡単にまとめた後、4章で実用モデルの構築について論じ、5章でテスト支援システムの機能要求について論じる。

2. プログラムのモデル化

2.1 プログラムの解釈

プログラムは計算機内の資源を利用して入力を出力に変換する。計算機の資源には記憶領域とCPU がある。記憶領域は保持する値で状態を区別し、命令を実行してCPU が状態を変更する。状態と変更手続きとを持つモデルとして状態機械を考えることができる。

状態機械は、状態と状態変更手続き（遷移関数）とを持つ。状態は、遷移関数を適用する前の状態と適用した後の状態として順序性を持つ。状態を節点に、遷移関数を枝に対応させることによって状態機械を有向グラフで記述できる。有向グラフ上の節点の列と枝の列とに対応して状態の列と遷移関数の列とを作成できる。状態機械では、遷移関数の適用を1ステップと考えて時間概念を導入することが出来る。最初にどの状態から開始するかを示す開始状態と、最後にどの状態に到達するかを示す終了状態がある。

プログラマは、言語の動作規則に従い入力を出力に変換するプログラムを言語の構文規則に従って記述する。計算機は、言語の構文規則と動作規則を用いてプログラムを解釈し、入力データに対応して記

憶領域内の値を命令の実行によって変更する。

ほとんどの言語には、記憶領域の値、即ち状態、に応じて次に実行する命令を選択する判定命令がある。判定命令には２つの効果がある。第１は、判定命令に至るまでの記述を共通に利用してソースプログラムの記述量を削減可能なことである。第２は、必要になるまで実行すべき命令の選択を遅らせて、命令の実行が非決定性になることを防止することである。プログラムテキストは、計算機が実行する可能性のある命令列を記述したものである。判定命令があるために入力データに応じてプログラムの異なった命令列を計算機は実行する。そのため、プログラムを検証する場合、実行する命令と同時に条件をチェックしていく必要がある。

プログラマは、プログラム実行の最終状態だけでなく途中の状態を明確にしながらプログラムを記述する。プログラマがプログラムの実行によって実現しようとする状態の列をプログラマの意図と呼ぶ。実際にプログラムを実行したときの状態がプログラマの意図に一致しない時にプログラムが誤っていると言う。プログラムの実行は必要な入力データを得ながら一連の状態変更命令を実行するものであり、命令に従って状態列を作り出す。そのために、状態列を意図と比較することにより、状態の一致を確認したり、不一致を発見することは容易である。テストでは、プログラムが正しく動作することを確認したり誤りを発見したりするためにこの考え方を利用している。

プログラムテキストは、先に述べたように、計算機が実行する可能性のある命令列を記述したものである。そのために、人間がプログラムを読んで検証するだけでは発見できない誤りが残ってしまう。従って、実際にプログラムを実行して状態列を生成し、これを用いて誤りを発見するテストが有効である。

2.2 有限オートマトンモデル

計算機内の全ての記憶の値によって状態を区別し、CPU による命令の実行を遷移関数とみなすと、計算機を有限オートマトン（状態と入力データだけで遷移関数が定まり、状態数が有限な状態機械）とみなすことができる。計算機の記憶による状態を命令で遷移させる手続きがプログラムであるので、プログラムを有限オートマトンに変換できる。

定義2.1

プログラムp の動作をモデル化した有限オートマトンをFA_pとする。

$$FA_p=(Q,I,O,T,s,F) \quad (2\text{-}1)$$

但し、Q:状態集合、I:入力アルファベット、O:出力アルファベット、 T:Q×I→Q×O:出力関数を含むように拡張した遷移関数、s:開始状態、F:終了状態集合、である。 ■

例えば、C:=A+Bというプログラムを有限オートマトンでモデル化した結果を図で表現すると図2.1 の様に成る。但し、計算機で表現できる整数の値を-32768～32767 までと仮定し、記号◇は、値が定義されていないこと示す。図2.1 は、変数A,B,C の値で状態を区別し、開始状態（３つの変数の値が未定義であるので(◇,◇,◇)）からAの値の入力でまず遷移を行い、次にBの値の入力で遷移を行い、最後にA+BをC に代入することで遷移した有限オートマトンである。

C:=A+Bの状態を(A,B,C)の3つ組で表し、AとBを順番に入力してCを代入。

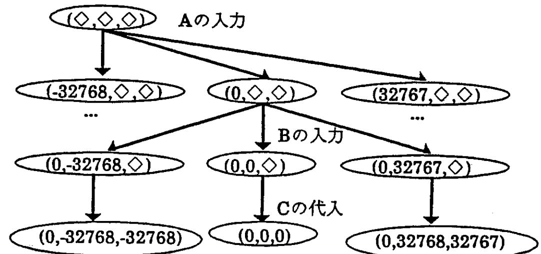
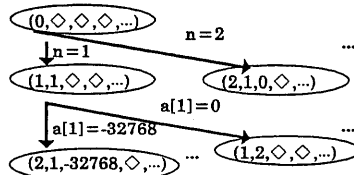


図2.1 有限オートマトンによるモデル化の例

```
1     procedure sortbin;
2        var i,j,l,r,m:integer;
3            x:integer;
4     begin
5        for i:=2 to n do begin
6            x:=a[i];l:=1;r:=i-1;
7            while l<=r do begin
8               m:=(l+r) div 2;
9               if x<a[m]    then  r:=m-1
10                           else  l:=m+1
11           end;
12           for j:=i-1 downto l do a[j+1]:=a[j];
13           a[l]:=x
14       end
15    end;
```

(a)ソースプログラム

変数pc,n,a[1..n],i,j,l,r,m,xが保持する記憶が状態を構成すると考えた有限オートマトン



(b)有限オトマトン

図2.2 2分挿入整列プログラムのモデル化

プログラムの動作を有限オートマトンでモデル化すると、状態数が多くなるため実際のプログラム言語では、状態とは異なる記憶という別の概念を導入して記述量を少なくしている。

図2.2に、2分挿入による整列プログラムのソース(a)と有限オートマトンによる表現(b)を示す。有限オートマトンは、開始状態から変数n と配列a に値の入力で遷移を行うことを示している。この例からも状態数が非常に多くなることが分かる。

2.3 制御フローモデル

プログラムの1つのモデル化としてよく用いられている制御フローグラフを考える。プログラムの制御フローの記述に有向グラフを用いる。制御フローは、プログラムの基本ブロック（連続して実行される命令列）を節点に、分岐を矢に対応させたものである。制御フローグラフCGは、次の通りである。

定義2.2

プログラムp の制御フローグラフCGは次の２つ組である。

$CG(p)=(N_c,E_c)$ (2-2)

$N_c=$ {プログラムpの基本ブロック}

$E_c=$ {(a,b):a,b∈N_c,プログラムp の中で、基本ブロックaから基本ブロックbに分岐がある} ■

但し、プログラムp を特に区別する必要のないとき単にCGと記述する。計算機内では、次に実行する命令の存在する番地をプログラムカウンタと呼ばれる記憶領域に保持している。そのプログラムカウンタの値（実際には基本ブロックの番号）で状態を区別した状態機械が制御フローモデルである。遷移関数は現在の状態と入力データ、変数領域に記憶された値によって次の状態を決定する。即ち、変数として確保された記憶領域の値を基本ブロック内の代入文や入力文で書き換え、基本ブロックの最後の文で次の節点（状態）に遷移する。

図2.3に図2.2のプログラムの制御フローグラフを示す。図2.3 でも変数nと配列aには値を入力するものとした。この制御フローグラフに基づくテストは必ずしも十分ではない[HOW78]ことが、これまでの経験からも分かっている。

3.テスト法

ソフトウェアのテスト法についてはこれまでも多くの提案が成されている。Gourlay は、テストを議論する際の枠組みとして、対象のプログラムとその

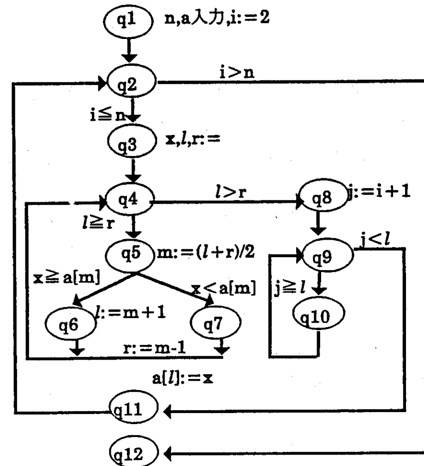


図2.3 2分挿入整列プログラムの制御フローグラフ

仕様、テストデータ、プログラムの実行結果が仕様に一致しているか否かの判断、テスト結果に基づいてプログラムが仕様を満たしているか否かの評価からなる5つ組を与えている[GOU83] 。その中で、テストデータの作成方法（「テスト法」）が最も多く検討されている。その多くは、プログラムのみに依存してテストデータを作成する仕様独立なテスト法である。仕様を利用したテスト法には、有限状態機械[FUR84,WAT84] や原因結果グラフを利用した方法が知られている[MYE79,FUR82]。

有限オートマトンに基づくテストシステムを次のように定義する。

定義3.1

有限オートマトンに基づいたテストシステムFATCは次の5つ組である。

$FATC=(P,FA_P,T,corr,ok(FA_P))$ (3-1)

但し、P:プログラム、FA_P: プログラムPの有限オートマトンモデル、T: テストデータの集合、corr:正当性の評価関数、$ok(FA_P)$:実行の正しさの判定関数、である。 ■

このテストシステムでは、$ok(FA_P)$ は、プログラムpと有限オートマトンFA_Pのそれぞれの実行列の比較として実現できる。

ところで、有限オートマトンは計算機内の全ての記憶領域の値に応じた状態を持っているために、プログラムpが実現できる全ての状態をFA_Pは持っている。故に、全ての状態を少なくとも1回は実現するテストとは、プログラムで実現できる全ての記憶の値を実現するものである。また、全ての遷移を少な

くとも1回は実現するテストとは、各状態を遷移先として持つ全ての命令を少なくとも1回は実現するものである。

全ての遷移を1回は実現するテストによって、有限オートマトンの誤りの中で遷移関数に関する全ての誤りを発見できる。全ての状態を少なくとも1回は実現するテストで状態に関する全ての誤りを発見できる。

機能テストは、プログラムが機能仕様通りに動作することを確かめるために、機能仕様からテストデータを作成してテストを実施する方法である。つまり、テストシステムで仕様をテストの判定や評価だけでなくテストデータの作成にも利用する。機能仕様のみを用いたテスト法をプログラム独立なテスト法と呼ぶ。プログラム独立なテスト法は、プログラムの利用者が望む機能をプログラムが果していることを仕様のみから確認する方法である。

有限オートマトンを、プログラムの実行モデルと考えてきた。そのために、モデルとして作成した有限オートマトンのテストデータは、プログラムのテストデータである。有限オートマトンはプログラムの実行のモデルであり、テストデータに対応した有限オートマトンの実行列とプログラムの実行列との比較で実行結果の判定を手続き的に行うことが可能になる。

4. 変数値状態モデル

プログラムをモデル化した有限オートマトンの状態数は、一般に、多くなる。例えば、図2.1 のプログラムは、値の範囲が-32768から32767 （16ビット）である3つの変数のみを使用しているが、有限オートマトンの状態数は、3*(2**16)**3=3*2**48～10**15（但し、最初の3は、プログラムカウンタの値）である。状態数が多過ぎるために、有限オートマトンモデルに基づいたテスト支援システムは、（人が介在しても、計算機そのものにとっても）実際的ではない。一方、制御フローグラフに基づいたテストは必ずしも十分ではなく[HOW78]、その支援システムは、プログラムの誤り発見には効果的であるとはいい難い。そのために、テスト支援システムの構築に際して実際に使用可能で、誤り発見に効果的なプログラムのモデルを作成する必要がある。

有限オートマトンは、計算機の全ての記憶を用いて状態を区別し、制御フローグラフはプログラムカウンタのみを用いて状態を区別する。これらの中間的なモデルとして、状態を区別するために、プログラム内の幾つかの変数を用いたモデルを考えることができる。例えば、図4.1 は、2分挿入整列プログラム（図2.2 ）を6つの変数の値で状態を区別してモデル化した例である。状態は、変数iとj,m,l,r,pc（プログラムカウウンタ）で区別されている。nの値（図では3）と配列a[1..3]の値（図では記号aとb,c） は状態とは異なった領域に記憶されている。変数の値◇は、図2.1 と同様に値が未定義であるこ

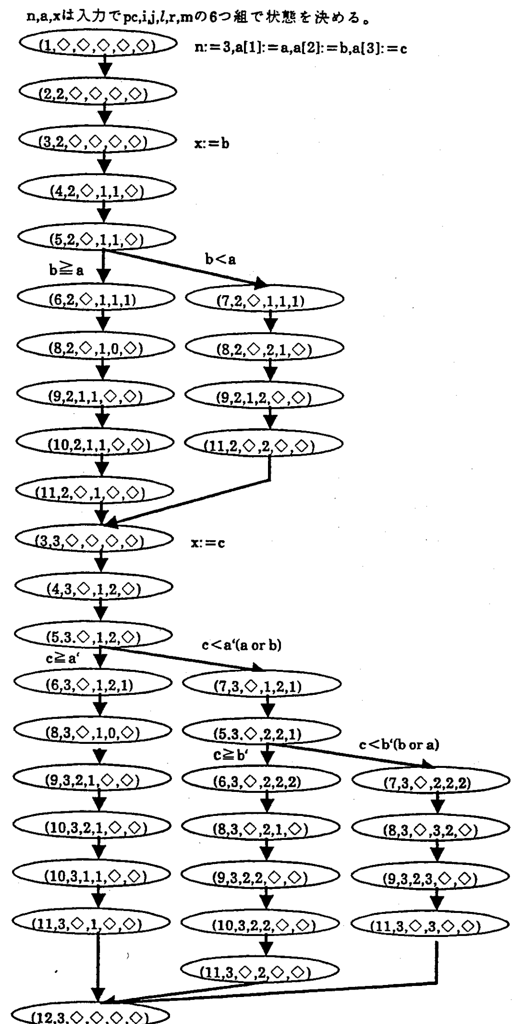


図4.1 2分挿入整列プログラムの変数値状態モデル

とを示す。図4.1 には、35の状態があり、開始状態から終了状態までに6 つの路がある。各々の路の入力条件は以下の通りである。

(1) $a<b<c$, (2) $a<c<b$, (3) $c<a<b$,
(4) $b<a<c$, (5) $b<c<a$, (6) $c<b<a$

これらの路は、制御フローグラフの状態と分岐の全てを通過する。

この様にプログラム中の変数によって状態を区別するモデルを「変数値状態モデル」と呼ぶことにする。

変数値状態モデルの状態数については以下の様なことが言える。

(1)基本ブロック番号の使用

制御フローグラフと同様に実行する命令の番号づけのために、計算機内のプログラムカウンタやソースプログラムの文番号ではなく基本ブロックの番号を使用した。基本ブロック番号は、より大きな実行命令列を単位として区別することができるために一般的には状態数を削減できる。しかし、for 文では、制御変数の初期値代入と増分代入のための命令があるために、文番号の場合より増える場合がある。また、利用者には基本ブロックの番号とソースプログラムとの対応が取れるような情報を与える必要がある。(但し、本論文では、以後もプログラムカウンタと呼ぶ。)

(2)生存変数解析による未定義値の代入

変数の値が異なってもプログラムの動作に影響を与えないことがある。プログラムの動作に影響を与える変数の範囲をプログラムのデータフロー解析の生存変数解析によって知ることができる[KIN76] 。プログラム内で変数に値が代入されており、しかもその値が参照される可能性のあるプログラムの部分を生存区間と呼ぶ。プログラムの動作に影響を与えない変数の値の違いによって状態を区別することはテストには必要のないものである。プログラムの入口で全ての変数に未定義値を代入し、変数の生存区間の終了時に変数に未定義値を代入することで無意味な状態の区別を避けることができ、状態数を削減できる。例えば、図4.1 では、変数mとl,r,j,iの生存区間の最後で未定義値を代入して状態を区別してモデルを作成している。もし未定義値を代入しないと図4.1の状態(3,3,◇,◇,◇,◇) でl,rの値が異なり状態(3,3,◇,1,0,1)と(3,3,◇,2,1,1)となる。そのため、状態(5,2,◇,1,1,◇) からの分岐が合流できなくなり状態数は49になる。

(3)変数の数と値の範囲

変数値状態モデルの状態の数は状態を決定する変数の数と値の範囲によって変化する。例えば、2分挿入整列プログラムでプログラムカウンタと変数iで状態を区別すると状態数は19になる。また、n の値を4 にして状態を6 つの変数で区別すると状態数は64になる。

(4)変数の種類

プログラム内の変数は次のように分類することができる。

(a)入力変数:プログラムの外部から値を読み込むために使用する変数。

(b)出力変数:プログラムの外部に値を出力するために使用する変数。

(c)制御変数:プログラムの実行順序を制御するために使用する変数。

(d)作業用変数:値の一時待避や計算結果の一時記憶などに使用する変数。

例えば、2 分挿入整列プログラムで、配列a は入力と出力の両方の変数として使用されている。変数n は、入力と制御変数の両方として使用されている。i,j,l,r,m は、制御変数として使用されている。状態を規定する変数としては、制御変数が適している。入力変数や出力変数は、値の範囲が広く状態数が増大するし、作業変数も値の範囲が広いことが多い。制御変数は、実行の順序を制御しているためにその変化がプログラムの特徴を表すうえに、値の範囲が狭く状態数の増大につながらないことが多い。

5. テスト支援システムの概要

図5.1は、テスト支援システムの機能構成である。テスト支援システムは、ソースプログラムを入力し、テストデータの作成とテストの実行を行う。

(1)入力

システムは、ソースプログラムを読み込む。記述言語は、Pascalのサブセットを仮定する。Pascalは、教育用の言語で、簡単であり、コンパイラの入手が容易である。

(2)出力

システムは、読み込んだソースプログラムのテストデータを作成する。テストデータは、プログラムから作成した有限オートマトンの全ての状態を通過するものである。さらに、システムは、テストを実施し発見した誤りを報告する。

(3)モデル生成系(MG)

モデル生成系(Model Generator) は、プログラムの変数値状態モデルを生成する。利用者は、状態を規定する変数を指定したり、変数の範囲についての情報を付加する。MGは、入力変数に記号値を与えて

プログラムを実行し、状態や遷移を生成する。実行途中で計算が継続できなくなった時、利用者に問い合わせる。

(4)テストケース生成系(TCG)

テストケース生成系(Test Case Generator) は、MGが作成した変数値状態モデルからテスト基準に従ってテストケースを作成する。テスト基準としては、全ての状態と遷移を少なくとも1回通過することを考える。テストケースを出力して変数値状態モデルの誤りがないかどうかを調べる。

(5)テストデータ生成系(TDG)

テストデータ生成系(Test Data Generator) は、TCG が作成したテストケースの入力変数に対する条件を満たすデータをテストデータとして作成する。このとき、誤りを発見する可能性の高い範囲の端点や桁上がりのある数などを取り出す[HOW79]。

(6)不等式の解決系(IS)

不等式の解決系(Inequality Solver) は、プログラムの記号実行や、テストデータの作成時に不等式を解く。不等式が解けないときは、利用者に問い合わせ、記号値の値を設定したり、実行を停止する。

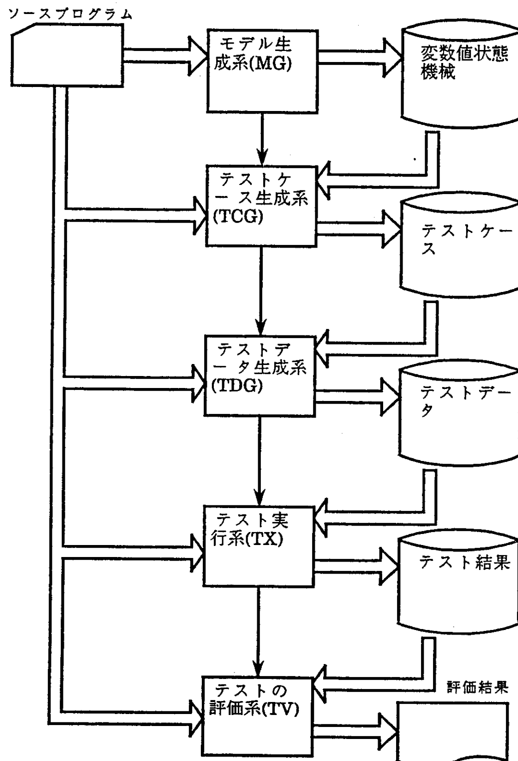


図5.1 テスト支援システムの機能概要

(7)テスト実行系(TX)

テスト実行系(Testing Executor)は、プログラムの実行に必要な資源を割り当て、プログラムを実行する。入力データは、TDG が作成したテストデータを用い、実行経過をモニタしながらMDで作成したモデルと比較し実行の異常を発見する。また、テストを評価するために必要な情報を収集する。

(8)テスト評価系(TV)

テスト評価系(Testing Evaluator) は、TXが作成した実行状況を基に、テストの十分性を評価する。

6.まとめ

本論文では、有向グラフ上の列概念に基づくテスト支援システムを構築するため、プログラムから作成した有限オートマトンの状態数を削減し実用化を計る方策を検討した。また、テスト支援システムの要求機能を明確にした。プログラムから取り出した制御フローグラフや有限オートマトンとは異なり、有向グラフ上の列は条件による分岐がないので、プログラムの実行に対応した状態や適用した遷移関数を直線的に扱うことができる。そのために、プログラマが期待した実行結果とプログラムの実行結果とが一致しているか否かを容易に判定できる。また、プログラムの実行時にモデルとして作成した有限オートマトンの状態や遷移関数、入力値に対応する事象を観測し、有限オートマトン上の列と比較することで実行結果の判定を手続き的に行うことができる。

制御フローグラフは、テスト品質を向上するためには十分な精度とはいい難い[HOW78]。一方、有限オートマトンは、状態数が多くなり実用的とは言えない。そこで、プログラム中のいくつかの変数の値で状態を区別する変数値状態モデルを検討した。変数値状態モデルは、状態を決定する変数の選び方によって状態数が変化する。また、入力変数の値の与え方にもモデルの構造は依存する。即ち、プログラムの構造に関する知識によってモデルが異なるのが難点である。基本的には、プログラムの実行順序を制御する制御変数で状態を規定することが望ましい。

変数値状態モデルでは、プログラムのデータフロー解析で変数が有効な値を持つ区間を見つけ出し、有効な値を持たない区間では変数に未定義値を設定しておくことも状態数の

削減に効果がある。

列に基づくテスト支援システムでは、モデル作成やテストデータ作成のために不等式の解法が必要になる。さらに、テストデータ作成時に誤りを発見し易いデータを組み込むために知識工学的手法が必要になろう。

[参考文献]

[ALL76] Allen, F.E. and Cocke, J. : A Program Data Flow Analysis Procedure, CACM, Vol.19, No.3, pp.137-147, 1976.
[CHO78] Chow, T.S. : Testing Software Design Modeled by Finite-State Machine, IEEE Trans. Soft. Eng., Vol.SE-4, No.3, pp.178-187, 1978.
[FUR82]Furukawa, Z. et al. : AGENT:Automatic Generation of Test Cases with Cause-Effect Graphs, Proc. of 6th ICSE Poster Session, pp. 23-24, 1982.
[FUR84]古川善吾 et al. : AGENT: 機能テストのためのテスト項目作成の一手法，情報処理学会論文誌，VOL.25, NO.5, pp.736-744, 11 1984.
[FUR85] Furukawa, Z., Nogi, K. et al. : AGENT: An Advanced Test-case Generation System for Functional Testing, AFIPS-Conf. Proc., Vol.54, pp.525-535, 1985.
[FUR86] 古川善吾，牛島和夫 ：有向グラフ上の列概念を用いたテスト法について，日本ソフトウェア科学会第３回大会論文集，pp.221-224, 1986.
[GOU83]Gourlay : A Mathematial Framework for the Investigation of Testing, IEEE Trans. on Software Engineering, VOL.SE-9, NO.6, pp.686-709, November 1983.
[HOP79]Hopcroft, J.E. et al. : Introduction to Automata Theory, Languages and Computation, Addison-Wesley Publishing Co. Inc., 1979.
[HOW76] Howden, W.E. : Reliability of the Path Analysis Testing Strategy, IEEE Trans. Soft. Eng., Vol.SE-2, pp.208-215, Sept. 1976.
[HOW78] Howden, W.E. : Theoretical and Empirical Studies of Program Testing, IEEE Trans. Soft. Eng., Vol.SE-4, No.4, pp.293-298, July 1978.
[HOW80] Howden, W.E. : Functional Program Testing, IEEE Trans. Soft. Eng., Vol.SE-6, No.2, pp.162-169, 1980.
[HUA75] Huang, J.C. : An approach to program testing, ACM Computing Surveys, 1975.
[JES76] Jessop, W.H., Kane, J.R. et al. : ATLAS-An Automated Software Testing System, Proc. of 2nd ICSE, pp.629-634, 1976.
[MYE79]Myers, G.J. : The Art of Software Testing, John Wiley & Sons Inc., 1979.
[WAT84]渡辺担 ：正則な状態遷移図の全遷移を網羅するテストデータ生成アルゴリズム，情報処理学会論文誌，VOL.25, NO.6, pp.960-969, 11 1984.
[WHI80]White, L.J. et al. : A Domain Strategy for Computer Program Testing, IEEE Trans. on Software Engineering, VOL.SE-6, NO.3, pp.247-257, May 1980.