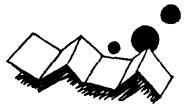
解　説



記号実行システム†

玉井哲雄††　福永光一††

1. はじめに

記号実行ということばは未だ熟しているとはいい難いが，symbolic execution の訳語としてあてたものである．ほとんど同じ意味で記号評価 (symbolic evaluation) ということばを使う人もあるが，以下では統一して記号実行という語を用いる．

プログラムの分析およびテストの1つの手段として，記号実行という技法が研究・発表され始めたのはさほど古いことではなく，1970年代半ばである．記号実行という手法が登場した背景には，主として2つの先行する技術があった．1つはプログラムの正当性の検証技術であり，もう1つはプログラムのテスト技術である．前者は60年代末から70年代に盛んに研究され，現在もなお進展をみている．既存の記号実行システムのうち，King らによる IBM 研究センターの EFFIGY と，Boyer らによるスタンフォード研究所 (SRI) の SELECT は，いずれもそれ以前に作られたプログラム検証システムを，何らかの意味で母体としている．

一方のテスト技術は，プログラミングの歴史とともにあるといってよい．従来からのテスト手法を，理論的・方法論的に見なおして，より系統的で信頼性の高いテスト技術を確立しようという気運は，やはり1960年代末から1970年代にかけて高まった．この中で，とくにプログラムの実行経路 (execution path) の分析手法，すなわち実行可能な経路の数え挙げ，各経路の実行を実現する入力データの条件の設定，等の研究がすすみ，さらにこれらの経路をテストするデータを，自動的に生成するという技法の探究にまで発展した．

これら検証技術や実行経路分析／テスト・データ生成技術において，記号実行はそれらに応用される基礎技術として発展してきたが，やがてそれ自身の直接的な利用がプログラムの動作分析に役立つことが認識され，独立したツールとしての記号実行システムがいくつか試作されるに至った．この解説は，これらの記号実行システムの紹介を中心とするが，記号実行を産み出す契機となり，またその応用分野としての意義をもつテスト・データ生成およびプログラム検証技術にも，かなりの説明をさいている．まず**2**章で記号実行とはどんな手法であるかを，例を挙げて説明する．**3**章では，この記号実行技法を直接応用した記号実行システムについて，紹介する．**4**章でテスト・データ生成系を，**5**章でプログラム検証系を，それぞれ記号実行との関連において，説明する．**6**章で，まとめを行う．

2. 記号実行とは

記号実行とは具体的にどのような手法なのか，例を用いて説明しよう．

図-1 のプログラムは，与えられた実数 P の平方根を求める FORTRAN の FUNCTION である (FOR-

```
                                        (行番号)
   FUNCTION SQROOT (P,E)                   1
     D=1.0                                 2
     X=0.0                                 3
     C=2.0*P                               4
10   IF  (D. LE. E) THEN                   5
       SQROOT=X                            6
       RETURN                              7
     ELSE                                  8
       D=D/2.0                             9
       TT=C-(2.0*X+D)                      10
       IF (TT. GE. 0.0) THEN               11
          C=2.0*TT                         12
          X=X+D                            13
       ELSE                                14
          C=2.0*C                          15
       ENDIF                               16
       GOTO 10                             17
     ENDIF                                 18
   END                                     19
```

図-1 プログラム SQROOT

† Symbolic Execution Systems by Tetsuo TAMAI and Koichi FUKUNAGA (Mitsubishi Research Institute, Inc.).

†† (株)三菱総合研究所

TRAN 77 を使用). E は精度を規定する実数値である. すなわち, この関数の返す値 $SQROOT$ は,

$$\sqrt{P}-E<SQROOT\leq\sqrt{P} \qquad (2.1)$$

を満たす. ただし, $0\leq P<1$, $0<E\leq 1$ でなければならない.

一見したところこのプログラムが正しく動くかどうか, 判然としないであろう. 従来のテストでは, P や E に具体的な数値, たとえば $P=0.5$, $E=0.1$ を与えてプログラムを実行してみる. この程度のプログラムなら, 実際に計算機によって実行してみなくても, 手で模擬実行することにより, 結果が得られるだろう. 今の場合, 多少面倒な計算の結果, 関数値として 0.6875 が得られる. これが仕様を満たすことは, (2.1) 式に代入して確かめることができる. しかし, $P=0.5$, $E=0.1$ という特定な値以外の入力に対し, このプログラムがどういう動作をするかについては, これだけでは何ともいえない.

記号実行では, P, E という入力を数値としてではなく, 記号値として与える. この記号値を, p, e としよう. すなわち, 初期値として $P=p, E=e$ を与えて, プログラムを "実行" する.

行 4 の実行が終わった時点で, 各プログラム変数は次のような記号値をもつ.

$$P=p, E=e, D=1, X=0 \qquad (2.2)$$

行 5 の IF 文によって, 実行は 2 つに分岐する. 数値による実行の際は, このどちらに分岐するかが一意に定まるが, 記号値の場合は, これが定まるとは限らない. したがって記号実行が分岐に至った時は, 場合分けをして, 各々の経路をたどる必要がある. いま, この IF 文の ELSE 段落が選ばれたとしよう. これが選ばれる条件 ($(D.\ LE.\ E)$ の否定) を記号値に直すと,

$$e<1 \qquad (2.3)$$

となる. このような条件を, 経路条件 (path condition) という.

次に行 10 の実行が終わった時点で, 各プログラム変数の記号値をみると, 次のようになる.

$$D=0.5, C=2p, TT=2p-0.5, X=0 \qquad (2.4)$$

なお, P と E は, このプログラムを通して値が変わらない (それぞれ p, e) ので, これ以降いちいち示さない. 次に行 11 で, また IF 文が登場する. 今度は THEN 段落を選ぶとしよう. この条件は, $(TT.\ GE.\ 0.0)$ すなわち,

$$2p-0.5\geq 0 \qquad (2.5)$$

である. 前の条件 (2.3) と合わせると,

$$e<1\wedge 2p-0.5\geq 0 \qquad (2.6)$$

または,

$$e<1\wedge p\geq 0.25 \qquad (2.7)$$

ここで $\wedge$ は, 論理積記号である. 次に, 行 13 が終わると, プログラムの状態は,

$$D=0.5, C=4p-1, X=0.5, TT=2p-0.5 \qquad (2.8)$$

となる. 続いて実行は, 行 17 の GOTO 文により, 再び行 5 の IF 文にもどる.

ここで, もう一度 ELSE 段落がとられるとしよう. 以下は, 前と同様に進むが, 行 11 の IF 文では, 前回と異なり ELSE 段落がとられるとしてみる. 行15 のあとで, プログラムの状態は,

$$D=0.25, C=8p-2, X=0.5, TT=4p-2.25 \qquad (2.9)$$

となる. また, このような経路がとられる条件は, 適当な簡約化をほどこすものとして,

$$e<0.5\wedge 0.25\leq p<0.5625 \qquad (2.10)$$

とかける. 再び行 10 にもどった時に, 今度は THEN 段落が実行され, ループを脱出するとしよう. この条件は, $(D.\ LE.\ E)$ すなわち, $0.25\leq e$ であるから, 経路条件は (2.10) と合わせ,

$$0.25\leq e<0.5\wedge 0.25\leq p<0.5625 \qquad (2.11)$$

この結果, プログラム終了時には, (2.9) の関係に加え, $SQROOT=0.5$ が成り立つことがいえる.

以上の記号実行で, 次のようなことが分った. このプログラム $SQROOT$ を, (2.11) 式を満たすような p, e をそれぞれ P, E に与えて実行すると, 関数値として 0.5 が得られる. その際, プログラムの実行経路は, 行番号で

1——5, 8——13, 16, 17, 5, 8——11, 14——17, 5——7

という順となる. ただし——の部分は, その間の行を連続して実行することを示す. この特定の実行経路をあらかじめ指定して, 記号実行を適用したものと仮定すれば, この経路を実現する条件が, (2.11) の形で与えられたことになる. この考え方は, 自動テスト・データ生成に直接つながる. すなわち, (2.11) を満たす数値データ, たとえば $e=0.3, p=0.5$ を, この経路のテストに対し生成してやればよい.

記号実行の結果得られる出力の値は, いまの例ではたまたま 0.5 という定数であったが, 一般には入力された記号値を含む式の形をとる. この記号出力値は,

この例のように FUNCTION から返される場合に限らず，COMMON 変数やパラメータへ出力される場合にも，同様に計算することができる．

上の例から，記号実行の特徴が，次のようにまとめられる．

（i） プログラムへの入力として，記号値を与える．ここで記号値とは，数値を代表する記号である（プログラム変数の識別子と混同しないように，注意する必要がある）．入力は，たとえば FORTRAN なら，パラメータや COMMON 変数，あるいは READ 文によってなされるから，これらに対応して記号値を与えればよい（このほかに，サブルーチン呼び出しに応じて記号値を与える必要がある場合もあるが，ここでは説明を略す）．なお，すべての入力を記号値にする必要は必ずしもなく，数値を混在させてもよい．前の例では，たとえば E には 0.1 というような数値を与え，P には p という記号値を与えるというのも，よい方法である．

（ii） 記号実行の結果は，実行する経路によって異なる．したがって記号実行を適用するには，あらかじめ経路を指定するか，分岐に際し場合分けをして，網羅的に可能な経路をたどるようにするなどの方法を，とらなければならない．

（iii） 1つの経路に対して記号実行を適用すると，2種類の情報が結果として得られる．1つは，実行終了時の各プログラム変数の記号的な値（入力された記号値を含む代数式）であり，もう1つは，与えられた経路を実現する入力条件（入力された記号値を含む論理式）である．

（iv） 記号実行の過程で生成される代数式や論理式に対し，簡単化する機能があることが望ましい．

（v） （ii）にあげた経路の選択に関連し，記号実行が IF 文のような条件判定による分岐に到達した時に，その時点で成立しているプログラムの状態および経路条件から，その IF 条件が真か偽か，あるいは両者の可能性があるかについて，決定できる推論機能があることが望ましい．

さて，このような記号実行によって，プログラムの信頼性がどの程度増したといえるであろうか．前の例で，従来のテストでは $P=0.5, E=0.1$ というような1点の入力に対して確かめられたことが，$0.25 \leq P < 0.5625, 0.25 \leq E < 0.5$ というような区間に対して確かめられたことは，1つの大きな進歩であろう．実行経路の指定をいろいろかえることにより，この入力の範囲をさらに広げることができる．しかし，この方法では結果が正しいことを確かめられはしても，なぜ正しく動くのかについて洞察を得るのが難しい．このためには，プログラムが正しいこと（仕様と一致していること）を証明して，この証明の妥当性を理解するのが，1つのよい方法といえよう．この証明に際しても，記号実行を利用することができる．

正当性証明のための記号実行

例題で，行5以下のループに注目しよう．ループのなかで，TT は使いすての変数である．そのほかの変数 X, D, C の意味をはっきりさせよう．そのために，$P=p, E=e, X=x, D=d, C=c$ として，行5から記号実行を始める．行8の ELSE，行11の THEN を選択し，再び行5にもどったとして，その経路条件は，

$$e<d \wedge c \geq 2x+d/2 \qquad (2.12)$$

であり，プログラムの状態は，

$$P=p, E=e, D=d/2, X=x+d/2, C=2(c-(2x+d/2)) \qquad (2.13)$$

となる．また，行8の ELSE，行14の ELSE を選択し，再び行5にもどったとすれば，その経路条件は，

$$e<d \wedge c<2x+d/2 \qquad (2.14)$$

であり，プログラムの状態は，

$$P=p, E=e, D=d/2, X=x, C=2c \qquad (2.15)$$

となる．これから，X と D の意味は明らかになろう．X は，目的とする平方根を近似していく変数である．Dは，ループごとに値が1/2になる，正の実数である．ループごとに，X には D が加えられるか，あるいはそのままに保たれる．すなわち，X は単調に増大し，2進の小数の精度を，1桁ずつあげる形で，目的の値を下から近似しているらしい．出力条件が，

$$\sqrt{P}-E<SQROOT \leq \sqrt{P}$$

であるが，行5では，

$$\sqrt{P}-D<X \leq \sqrt{P} \qquad (2.16)$$

が成立しているらしい．もしそうであれば，行10の IF が真の時，$D \leq E$ が成立するから，出力条件が満たされる．

問題は，C の性格である．C がどのように意味づけられるかを明らかにするには，このアルゴリズムの成立に対する洞察力がいる．次の関係を発見するには，多少の手間を要するだろう．

$$C=2(P-X^2)/D \qquad (2.17)$$

これが，行5の IF の先頭で常に成立している．このようにループに関して常に成立している関係を，不変

表明という．一たび(2.17)の関係を発見できれば，その正しさは記号実行で示すことができる．最初に行5に実行が至った時は，$C=2p, P=p, X=0, D=1$であるから正しい．次に，行5で $C=c, X=x, D=d$ とし，$c=2(p-x^2)/d$ が成立しているとする．記号実行の結果，5—8—13, 16, 17, 5 という経路では，(2.13)より，(2.17)の

左辺$=2(c-(2x+d/2))$

右辺$=2(p-(x+d/2)^2)/(d/2)$

両辺は，$c=2(p-x^2)/d$ を考慮すると簡単な計算の結果 $4(p-(x+d/2)^2)/d$ に帰着され等しいことがわかる．このとき，経路条件から，$C\geq 0$ である．行11のIF文で ELSE をとるケースでも，行5にもどったとき(2.17)が成立することは，容易に示せる．このときも，$C\geq 0$ である．このことから，(2.16)のうち，

$$X\leq\sqrt{P}$$

の不変性が示せた．同様に，$\sqrt{P}-D<X$ ないし，$P<(X+D)^2$ の不変性も同じ手法で示せる．したがって，プログラムが正しいことがわかる．

この正当性の検証における記号実行の応用にみられるように，記号実行は必ずしもプログラムの先頭から終りまで適用する必要はなく，部分的に実行した方が役にたつことがある．

以上の例で，記号実行の意味，それがテスト・データ生成やプログラム検証にどのように使われるかについて，概略がうかがえたことと思う．ただし，この例は簡単なだけでなく，記号実行にとって都合のいいものになっている．その1つの理由は，配列を含まないことである．さらに，プログラムが対象としているのが実数であり，背景の理論が，数の四則演算および大小関係に関するよく知られたものであることも，問題を簡単化しているといえよう．

3. 代表的な記号実行システム

ここでは，代表的な記号実行システムとして，EFFIGY[12]，SELECT[1]，DISSECT[8]の3つをとりあげて紹介する．これらは，ほぼ同じ時期に，互いに影響を受けながらも一応独立に研究・試作された．またこれらのシステムのねらいも，自動テスト・データ生成やプログラム検証への応用という側面は含みながらも，中心を記号実行のテスト・ツールとしての直接的な応用におくという点で，共通しているといえる．対象とするプログラムの記述言語は，いずれもたとえばFORTRAN のような数値処理向きの手続き的な言語であり，使われる技法は，2章で概略を説明したような記号実行である．以下，各々の特徴を述べよう．

EFFIGY

EFFIGY は，J. C. King を中心に IBM Thomas Watson 研究所で開発された．対象言語は PL/I のサブセット，というより，PL/I 風の記法をもったきわめて簡単なプログラム言語である．対象とするデータ型は整数型のみで，1次元配列も許される．文として，代入文，IF 文，DO 文，DO WHILE 文，GOTO 文，READ 文，WRITE 文などがあり，式として算術式と論理式がある．また，外部手続きの定義と呼び出しが可能である．

EFFIGY の特徴は，会話的な機能が重視されている点にある．たとえば，次のような機能がある．

（i） トレース機能．利用者の要求により，記号実行の過程を，各実行文についてその文番号，ソース文，計算結果などを表示することで，明らかにする．

（ii） 実行中断機能．利用者があらかじめプログラム中にもうけた中断点で記号実行を中断し，利用者に制御を渡す．利用者は，プログラムの状態をみて，必要なら変数に値を設定するなどして，実行を再開することができる．

（iii） 状態保存回復機能．複数の経路をたどろうとする時は，記号実行の途中の状態を保存したり回復したりする機能が，必要となる．

このような機能から，EFFIGYをいわゆる対話的デバッガの一種とみなすこともできる．通常のデバッガが数値のみを入力できるのに対し，EFFIGY は記号値も入力できる拡張されたデバッガであると考えればよい．実際のセッション例が，文献[11]にある．

対象プログラム言語の中に，ASSUME, PROVE, ASSERT という文があり，これを用いてプログラムの仕様や表明が記述できる．これにより，EFFIGY にはプログラムの検証機能が比較的自然に組み込まれている．式の簡約や推論機能は，King が以前に開発したプログラム検証系（ただし，検証系では記号実行の考え方は使われていない）の一部が，流用されている．

全体としては，対象言語が単純なこともあり，実験的なシステムという色彩が濃い．実験例の報告も，SELECT や DISSECT のレポートと比べて少ない．しかし，システムとしてのまとまりはよさそうで，たとえば文献7)にあるように，プログラム検証技術の教育システムなどとしても，役に立つと思われる．

SELECT

R. S. Boyer, B. Elspas, K. N. Levitt ら，SRIの人達によって開発されたSELECTは，それに先行してプログラム検証システムの開発があった点で，EFFIGYと共通点がある．ここでも式の簡約・推論機能はSRIのプログラム検証系[17]の機能を流用している．

SELECTの対象言語は，LISPのサブセットである．とはいっても，このサブセットに含まれている言語要素は，

(i) 代入 (SETQおよび配列への代入 SETA)

(ii) 算術関数 (PLUS, TIMES, DIFFERENCE, QUOTIENT, MINUS, ABS)

(iii) 制御文 (GO, FOR, WHILE, UNTIL, COND)

という，手続き的，算法的なもので，LISPらしさは失われている．SRIのプログラム検証系との関連で，入力言語がLISPの記法に従っている方が便利であることが，このような選択の事情であろう．

SELECTの1つの大きな特徴は，自動テスト・データ生成機能に重点をおいているところにある．すなわち，1つの実現可能な実行経路に対する記号実行の結果，プログラム変数の記号的な値と，その経路を実現する経路条件とが得られるが，SELECTは同時に，この経路条件を満たす入力値をも求める機能をもっている．このためには一般に，非線形不等式群の実行可能解を求めるという操作が必要であり，さらにSELECTの取り扱うデータは整数であるから，整数解でなければならないという制約が加わる．線形不等式群の場合に限って，GomoryやBendersの方法をためしたらしいが，線形という制約は強すぎたという（それよりむしろ，これらの方法の性能の方が問題だと思われるが）．そこで，最終的には共役傾斜法を採用したとのことである．整数制約をどう処理したかなどの詳細は，よくわからない．

SELECTでもEFFIGYと同様に，利用者がプログラムに表明を挿入することができる．この表明は，それを記号実行の結果で評価したり，実行経路を制約し，生成されるテスト・データの性格を規定するのに，用いられる．

SELECTについては，いくつかの小さいながらもある程度実際的なプログラムに適用した事例が，報告されている．これについては，文献1)を参照されたい．ただし文献では，システムの具体的なイメージがEFFIGY. ほど明確にされてはいない．

DISSECT

DISSECTは，W. E. Howdenとカリフォルニア大学サンディエゴの人達によって開発された．この前身となる研究開発は，Howdenがマクダネル・ダグラス社のコンサルタントだった時に，L. Stucki らとともに行っている．

EFFIGYやSELECTと比べて，DISSECTは実用性をとくに考慮しているようである．たとえば対象言語にしても，ANSI FORTRANをそのまま相手にしている．

DISSECTでは，会話型処理は意識されていない．入力は2種類のファイルによる．1つが記号実行の対象となるプログラムのファイルであり，もう1つがコマンド・ファイルである．コマンドには，入力として与える記号値や数値の指定，出力内容の指定，および記号実行を適用すべき経路の指定などがある．出力されるものには，記号実行の結果得られる各プログラム変数の記号的な値，および経路条件がある．経路条件は，記号値を含む論理式の集合で，簡約化がほどこされている．この簡約機能には，SRI検証系の簡約系が借用されており，この点に関してはSELECTと共通である．

DISSECTは，実際にいろいろなプログラムに適用され，その効果がかなり具体的に調査されているところに，特徴がある．その結果として，たとえばKernighanとPlaugerの「プログラム書法」中の"よくある誤りを含んだ例題"22に対し，記号実行と原始プログラムの静的解析技法（たとえばDAVE[13]）の組合せで，17につき誤りが発見できた，これに対し，従来のテスト手法では11しか誤りが発見できなかった，といった報告がある．

DISSECTは，その前身がプログラム検証システムではなくテスト・ツールであったことから，プログラ

表-1 3つの記号実行システムの比較

項目＼システム	SELECT	EFFIGY	DISSECT
開発者	SRI	King ら	Howden ら
発表年	1975	1975	1977
対象言語	LISP のサブセット	PL/I のサブセット	FORTRAN
主な制限	手続き呼び出しなし	データは整数型のみ	——
会話機能	具体的には不明	とくに重視	な し
その他の特徴	テスト・データ生成に重点	検証機能に特徴	テスト・ツールとしての実用性に重点

ムの検証機能はとくに組み込まれてはいない．また，テスト・データ生成機能も備えていない．逆に，記号実行をそのまま直接的にテストに利用しようという考え方が強く出ているのが，特徴になっている．

なお，DISSECT の記述言語は，UCI-LISP である．

最後に，この3つのシステムの比較を，**表-1** に示す．

4. テスト・データ生成系への応用

2 章でも述べたように，記号実行をテスト・データ生成に応用することができる．その場合，まず記号実行を行い経路条件を生成し，次にその経路条件を満たす入力データの値を求める，という手順がとられる．このような方式に基づいたテスト・データ生成系には，**3** 章で挙げた SELECT のほかに，カリフォルニア大学バークレイの Ramamoorthy らが開発した CASEGEN[14]，コロラド大学にいた Clarke が開発したシステム[3]などがある．CASEGEN と Clarke のシステムの対象言語は，ともに ANSI FORTRAN である．

本章では，CASEGEN を中心に，テスト・データ生成系への記号実行の応用方法を述べ，最後に Clarke のシステムや SELECT との比較をする．

テスト・データ生成系の構成

プログラムをテストするためには，当然のことながらテスト・データが必要である．ところが，通常のプログラムの場合，可能な入力データの組合せは無限に（あるいはそれに近いほど）あり，そのすべてに対するテストを行うことは，実際上不可能である．そこで，有効なテストを行うためのテスト・データの選択基準が必要とされる．この基準の代表的なものとして，プログラムの構造に基づく方法がある．これは，プログラム中の可能な実行経路の中で所定の条件を満たすものを列挙し，各経路を通る入力データを1組ずつ選ぶというものである．上記3システムは，いずれもこの基準を用いている．

この方式によるテスト・データ生成系の構成は，一般に **図-2** のようになる．図中の各構成要素の働きは，次のとおりである．

（i） 経路選択系

原始プログラムを読み，その中からテスト基準に合った経路を選び出す．

（ii） 制約条件生成系

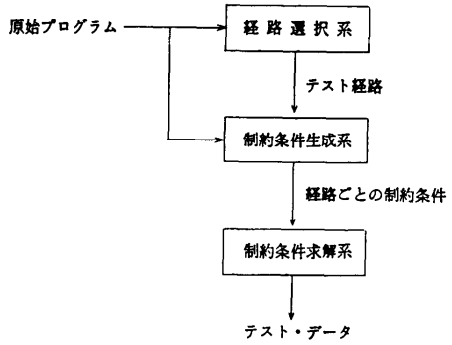


図-2 テスト・データ生成系の構成[14]

選ばれた経路ごとに，その経路が実行されるために入力データが満たすべき条件を生成する．

（iii） 制約条件求解系

生成された条件を満たす入力データの値を求める．この解が，与えられた経路を実行するためのテスト・データとなる．

CASEGEN の特徴

CASEGEN の構成も，大略図-2 に沿ったものとなっている．CASEGEN の特徴をまとめると，次のようになる．

- “すべての分岐を少なくとも1度は実行し，各ループは k 回実行する”ようなテスト・データの集合を選ぶ．（k の値は利用者が与えられるようになっている．）
- 制約条件を生成するのに記号実行を利用する．（記号実行の結果得られた経路条件が，そのまま制約条件になる．）
- 制約条件を解くのに，試行錯誤的な方法を用いる．

すべての分岐が少なくとも1度は実行されるようにテストするという基準は，特に理論的な根拠があるわけではないが，わりに広く使われている．これは，最低限この程度のテストは必要であり，しかも，これ以上に精緻なテストをしようとすると途端に手間が増大するという共通の認識があるからであろう．

ループの実行回数を固定したのは，記号実行を簡単にするためである．ループの実行回数が決まっていない場合には，ループ内の経路条件や記号値の一般形を求める必要が出てくるが，この処理は非常に難しく，今のところ実用的な技術は開発されていない．

CASEGEN の目的はテスト・データ生成だけであるから，記号実行を利用するといっても，**2** 章の例の

```
      I=5
      J=2*I-I
      IF (I. LT. 5) GO TO 40
            ⋮
40    CONTINUE
```

図-3 実行不能経路の例

ような本格的な記号実行をする必要はない．たとえば，途中結果を見やすい形式に変換する必要はない．またたとえば，図-3 の IF 文の次の GO TO 文へ行く経路は実行不能であるが，この検出（経路が実行可能か否かの判定）は，記号実行時にではなく，テスト・データ生成時（制約条件求解時）に行うものとして，GO TO 文に関する記号実行をしてしまってもかまわない．したがって，処理系の効率向上やその作成の手間の軽減のために，種々の便法を採用することができる．

CASEGEN の採った便法のうちで一番効果的なのは，配列要素の取り扱い方である．

プログラム中に配列要素に対する参照があり，その添字の値が入力データに依存する場合には，その取り扱いに特別な配慮が必要となる．たとえば，図-4 (a) のプログラム中の IF 文の判定では，I と J が等しいか否かにより，実行結果が異なる．

これについての1つの解決策は，$I=J$ と $I\neq J$ のそれぞれの場合について記号実行することである．すなわち，問題になる配列参照のところで経路が仮想的にいくつかに分かれるとして扱うもので，SELECT

〔原始プログラム〕

```
READ I, J
A(J)=2
A(I)=0
A(J)=A(J)+1
IF(A(J). EQ. 3)
```

(a)

〔記号実行後の姿〕

$I=i,\ J=j$

$A_1(j)=2$

$A_2(i)=0$

$A_2(j)=A_2(j)+1$

$IF(A_2(j).\ EQ.\ 3)$

(b)

〔テスト・データ生成時に i と j に値が与えられた後の計算〕

もし $i=i_0,\ j=j_0$ で $i_0\neq j_0$ なら

$I=i_0,\ J=j_0$

$A_1(j_0)=2$

$A_2(i_0)=0$

$A_2(j_0)=A_2(j_0+1$

$IF(A_2(j_0).\ EQ.\ 3)$

それゆえ IF の中の A の値は次のようになる

$A_2(j_0)=A_2(j_0)+1$

$=A_1(j_0)+1$

$=2+1$

$=3$

(c)

もし $i=j_0$ かつ $j=j_0$ なら

$I=j_0,\ J=j_0$

$A_1(j_0)=2$

$A_2(j_0)=0$

$A_2(j_0)=A_2(j_0)+1$

$IF(A_2(j_0).\ EQ.\ 3)$

それゆえ IF の中のAの値は次のようになる

$A_2(j_0)=A_2(j_0)+1$

$=0+1$

$=1$

(d)

図-4 配列要素の参照の取り扱い[14]

がこの方式を採っている．この方式は簡明ではあるが，取り扱う経路の数が非常に多くなり，効率が悪い．

これに対し，CASEGEN では，参照された配列要素が識別できない場合には，そのたびに新しい配列を作り，新しい配列と元からある配列との関係を記録しておくという方法をとる．そして，テスト・データ生成時に要素の識別を行う．たとえば，図-4(a) の配列 A は，同図 (b) の A_1, A_2, A_3 の相異なる3つの配列として扱われる．そして，テスト・データ生成時に，同図 (c), (d) のいずれかの方法で，A_1, A_2, A_3 の要素の値が計算される．

この方法は，テスト・データの値を決める際，配列要素より添字の値の方を先に決めることにより，記号実行中の配列要素に関する推論を不要にする，という巧妙なものである．しかし，配列要素に対する代入が起こるたびに新しい配列を作らねばならず記憶域がかさむという難点がある．

テスト・データの生成すなわち，制約条件の求解に関しては，3 章でも触れたように SELECT は共役傾斜法を用いている．Clarke のシステムでは，制約式を線型なものに限定し，線型計画法を利用している．

表-2 テスト・データ生成系の比較

比較項目＼テスト・データ生成系	CASEGEN	Clarke	SELECT
対象言語	FORTRAN	FORTRAN	LISP サブセット
記述言語	FORTRAN	FORTRAN アセンブラ Altran	LISP
ループの実行回数	指定する	指定する	指定しない（一般形を求めるための推論を行う）
記号実行中の簡約化	行わない	行わない	行う
実行不能経路の検出時期	テスト・データ生成時	テスト・データ生成時	記号実行時
添字が入力データに依存する配列要素の扱い	配列のコピーを作る	全然扱わない	経路を分割する
テスト・データ生成法	試行錯誤法	線型計画法（制約条件が線型のときにのみテスト・データを生成する）	共役傾斜法
全体評価	数式の簡約・推論機能を一切使わない手軽なツール．実用性は今一歩	能力の低い実験システム．既存の道具の組合せでこの程度のことができることを示したことに意義がある	高級な推論機能を持った研究用システム

これらとは対照的に CASEGEN では，テスト・データを乱数を用いて試行錯誤的に生成する．すなわち，変数を一列に並べて前から順に乱数を用いて値を決めていく．i 番目の変数 v_i の値を決めると，v_1, v_2, …v_i に関係する制約式がすべて満たされるか否かを調べ，満たされていれば v_{i+1} の決定に移る．満たされない場合には，v_i の値を決め直す．一定回数以上 v_i の値決定に失敗した場合はバックトラックして v_{i-1} を決め直す．

この方法は，SELECT 等の数理計画法を用いた方法に比べて，考え方が簡単であるが，やはり計算時間がかかる．実際に CASEGEN の実行時間を計測してみると，その時間の大部分は，データ生成の際のバックトラッキングに費やされているということである．

最後に，表-2 に，CASEGEN, Clarke のシステム，SELECT の比較結果をまとめておく．

テスト・データ生成系の使い方

上記の3つのシステムで生成されるテスト・データは，対象プログラムの構造を反映したものになっている．そのため，プログラムの構造そのものに関する誤りを検出できない場合がある*．しかし，このような誤りは経路条件を見ればわかる．すなわち，記号実行により生成された経路条件が，プログラムの仕様から見て必要十分なものか否かを検査すればよい．この検査に合格しない時は，経路の過不足があるか，条件判定に誤りがあることになるので，それらを修正することになる．こうした経路の分け方に関する誤りを取り除いた上で上記システムによるテスト・データ生成を行い，さらにそれらのデータによるテストを実施するというのが，妥当な手順であろう．

5. プログラム検証系への応用

記号実行をプログラム検証に利用することの利点は，検証過程がわかりやすくなるという点にある．既存の代表的な検証系[5),10),16)]は，検証作業をプログラムの実行の流れとは逆向きに進めるため，その過程が理解しにくく，プログラムに誤りがあったときに，その原因が突き止めにくいという傾向がある．これに対し，記号実行を用いた場合には，通常のプログラムの実行と全く同じ順に処理が進むので，途中で何が起っているかを把握するのが容易である．ただ，この方法は目的主導型 (goal-oriented) ではないため，検証とは直接関係ない演算も行われてしまい，結果が繁雑になるという可能性もある．

記号実行をプログラムの検証に応用する場合のおおよその筋道は，2 章で説明したとおりである．そこでは，各プログラム変数の値がそれらの入力記号値を用いて表現されていると，通常の数値入力の場合より情報量が多くプログラム変数間の関係が求めやすい，という性質が利用されていた．ただし，2 章の例では，この情報を検証に利用する過程には，人間の洞察力が介在していた．記号実行を応用した自動的な検証系を作成する場合には，この洞察力に相当する部分を，何らかの方法で機械化しなければならない．

機械化にあたって一番問題になるのは，次の2点である．

- 記号実行途中のプログラムの状態をどのように表現するか．
- プログラムのどの部分を記号実行し，その結果をもとにどのように推論をするか．

後者は，どの帰納法を適用するのかという風にいいかえてもよい．プログラムの検証では，通常，特定の入力データ値や経路に依存しないプログラムの性質を扱う．したがって，プログラムの実行ステップ数をあらかじめ有限の数で押えておくことや入力データの範囲を有限の手間で数え上げることができない場合が多い．このような場合には，単に記号実行を適用しただけでは，検証手続きは終了しない．これを解決し，有限の手間で検証が終了できるようにするために，帰納法が必要になるのである．

以下では，記号実行を利用した検証系の例として，現在ゼロックスにいる Deutsch の開発した PIVOT[4)]，エジンバラ大学にいた Topor の開発した検証系[15)]，筆者らが開発した VFPL-VS[18)] の3つをとり上げ，上記の点をどのように処理しているか，簡単に説明する．

PIVOT

PIVOT のプログラム状態は，各変数の記号値を格納した値データベースと経路条件を格納した節データベースの2つで表現される．

PIVOT の用いる帰納法は，帰納的表明法である．これは，2 章の例の (2.16) や (2.17) のような不変表明を見出し，それを利用するという方法である．

PIVOT の対象言語には，EFFIGY と同じように ASSERT 文というものがあり，これを用いて，利用

* このあたりのテスト・データ選択基準の基礎的な議論については，Goodenough[6)] を参照されたい．また，プログラムの構造に基づいたテストで発見できない誤りの具体的な例は，Howden[9)] にも見られる．

```
100  ASSERT Y=B)=0
110  X ← 0
120  LOOP
125      WHILE Y#0: BEGIN
150      X ← X + A
160      Y ← Y - 1
170      ASSERT X=A*(B-Y) AND Y>=0
190      END
200  ASSERT X=A*B
```

(a) 例題プログラム

```
PATH #1: 170-190-120-125(Y#0)-150...170
PATH #2: 170-190-120-125(Y=0)-200
PATH #3: 100...125(Y#0)-150...170
PATH #4: 100...125(Y=0)-200
```

(b) 例題プログラム部分経路

図-5 ASSERT 文から ASSERT 文へ至る部分経路[4]

者がプログラム中に入出力表明およびループの不変表明を挿入するようになっている．PIVOT は，この表明づきのプログラムを解析して，プログラム中の各 ASSERT 文から別の ASSERT 文に至り，かつその中にほかの ASSERT 文を含まない部分経路を列挙する．たとえば，**図-5** (a) のプログラムからは，同図 (b) の部分経路の集合が得られる．そして，これらの部分経路ごとに，その最初の ASSERT 文中の論理式が真であることを仮定して記号実行を行う．さらに，経路の終わりに達したときに，最後の ASSERT 文中の論理式 *out* が成立するか否か（経路条件 *pc* から導かれるか否か）を調べることにより検証を行う．(定理証明系が，$pc \supset out$ を証明できるかどうか調べる．)

この方法は実現法が簡単であるが，経路が分断されるため，プログラムの全体に渡っての見通しが得られないという難点がある．

VFPL-VS

VFPL-VS も帰納的表明法を用いるが，PIVOT のように経路を分割せずに，プログラム全体を対象として記号実行を行う．これは，VFPL-VS のプログラム状態の表現法と関わっている．

VFPL-VS では，通常の記号実行とは異なって，記号値という概念は存在せず，プログラム状態は，プログラム変数間の関係と経路条件から成る論理式で表現される．そして，各文の意味は，プログラム状態の変換規則の集合で定義される．変換規則は，Hoare の記法を用いて，次のように書かれる．

$$P\{S\}Q$$

これは，“文 S の実行直前に論理式 P が成立していれば，S の実行後，論理式 Q が成立する”ということを意味する．

記号実行は，これらの規則をプログラムの実行順に適用していくことにより遂行される．そして，**if** 文に出会った場合には，**then** 節と **else** 節をともに実行し*，それらの実行後のプログラム状態の論理和を，**if** 文の実行結果とする．

ループに関しては，あらかじめ不変表明が与えられている場合には，ループを1回記号実行し，不変表明が成立していることを確かめた上で，ループに続く次の文の実行に移る．不変表明が与えられていない場合には，それを検出しようとして，ループを何回か記号実行する．それでも検出できない場合は，利用者に問い合わせる．その後は，不変表明が与えられていた場合と同じ方法で処理を進める．

この方法は，プログラムの各点で成立する関係式が，経路に依存しない形式で求められるので，プログラムの性質が理解しやすい．

Topor の検証系

Topor の検証系の記号実行の進め方も VFPL-VS と同様である．ただし，プログラム状態の表現および文の意味の定義に操作的意味論 (operational semantics) を用いている点と，継続帰納法 (continuation induction) を採用している点が異なっている．

操作的意味論は，対象言語で書かれたプログラムを実行する抽象機械を想定し，言語の各文の意味をその機械の状態の変換規則を用いて定義するというものである．Toper の検証系の対象言語 POP-2 のような記号データを扱う言語の場合には，言語の実行システム（解釈系等）のかなりの部分をそのまま抽象機械として流用することができるので，これは非常にうまい方法である．しかし，数値データを扱う言語の場合には，この方法は適用できないので，実用的な方法とはいい難い．

継続帰納法では，プログラムの仕様を仮想プログラム**として与え，実プログラムと同じ条件でこれを記号実行し，実プログラムの実行結果と一致するか否かを調べることにより検証を行う．ただし，実プログラム中の各ループの先頭からプログラムの終わりまでの部分，および各再帰関数に対しては，それぞれ対応する仮想プログラムを用意しておかなければならない．(これは，帰納的表明法でループの不変表明を与えなければならないことに対応している)．そして，実プロ

---

* **then** 節を実行する場合には，**if** 文の条件が真であると仮定する．**else** 節の場合は，その逆．

** 実プログラムと同じ言語で書かれたプログラムではあるが，ループや再帰呼び出しは許されず，その代わりに仕様記述用の特別な関数を中に書くことができる．

表-3 検証系の比較

比較項目＼検証系	PIVOT	VEPL-VS	Topor
対象言語	ALGOL 風言語	VFPL (Pascal 風ファイル処理言語)	POP-2 サブセット
記述言語	LISP	LISP	POP-2
プログラム状態の構成要素	記号値 経路条件	プログラム変数の関係 経路条件	記号値 経路条件 制御スタック 命令ポインタ
帰納法	帰納的表明法	帰納的表明法	継続帰納法
簡約機能	あり (整数に関する知識を持つ)	あり (順序関係に関する知識を持つ)	あり (PIVOT に似ているがもっと強力)
定理証明機能	あり (それほど強力ではない)	ごく簡単 (文の変換規則の中に対象分野の知識を埋め込んでおき，大抵の推論は記号実行時に自然に行えるようにしている)	あり (それほど強力ではなく利用者との対話を重視している)

グラム中でループや再帰関数を続けて2度以上実行するときは，2度目から先に対しては仮想プログラムの方を実行する．

この帰納法は，再帰的なプログラムに対しては計算帰納法と同等の能力を有し，ループのあるプログラムでは，帰納的表明法を一般化したものになっている．

以上のように，記号実行をプログラムの検証に応用するといっても，いろいろな方法があり，それぞれ一長一短がある．しかし，それらの差は，記号実行そのものというよりは背後に控える検証に対する考え方の差に起因するものなので，ここではこれ以上触れないことにする．ただ，これらの検証系の作成者のすべてが，記号実行を採用した第一の理由として，そのわかりやすさを挙げていたことをつけ加えておく．

表-3 に，3つの検証系の比較結果をまとめておく．

6. まとめと展望

記号実行システムの研究開発は，1975年前後に相ついで行われたが，それ以降の新たな研究開発は比較的少ない（例としては，ハーバード大学の Cheatham らが開発したシステム[2] など）．

すでに発表されたシステムは，やはりあくまで実験的なものといわざるをえない．ただ，その考え方にはそれまでにない面白さがあろう．すなわち，記号実行という技法は，1つの形式的 (formal) な手法ということができ，しかも人間がプログラムの動作を理解する過程に近いものをもっているので，その形式性にかかわらず人間になじみやすい面があるといえよう．

しかしいずれにせよ，記号実行という手法，あるいは記号実行システムは，それ単独で万能な力をもつものではない．記号実行で発見しやすいプログラムの虫の種類と，そうでない虫の種類とがあろう．したがって，ほかのソフトウェア・ツールとの組合せとして考えていかなければならないだろうし，またその組合せを工夫する方向に，新たな進展があるのではないだろうか．

参考文献

1) Boyer, R. S., Elspas, B. and Levitt, K. N. : SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution, Proc. International Conference on Reliable Software, pp. 234-244 (1975).
2) Cheatham, Jr., T. E., Holloway, G. H. and Townley, J. A.: Symbolic Evaluation and the Analysis of Programs, IEEE Trans. Softw. Eng., Vol. SE-5, No. 4, pp. 402-417 (1979).
3) Clarke, L. A.: A System to Generate Test Data and Symbolically Execute Programs, IEEE Trans. Softw. Eng., Vol SE-2, No. 3, pp. 215-222 (1976).
4) Deutsch, L. P. : An Interactive Program Verifier, Ph. D. thesis, University of California Berkley (1973) ; also CSL-73-1, Xerox Palo Alto Research Center (1973).
5) Good, D. I., London, R. L. and Bledsoe, W. W.: An Interactive Program Verification System, IEEE Trans. Softw. Eng., Vol. SE-1, No. 1, pp. 59-67 (1975).
6) 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).
7) Hantler, S. L. and King, J. C.: An Introduction to Proving the Correctness of Programs, Compt. Surv., Vol. 8, No. 3, pp. 331-353 (1976).
8) Howden, W. E.: Symbolic Testing and the DISSECT Symbolic Evaluation System, IEEE Trans. Softw. Eng., Vol. SE-3, No. 4 pp. 266-278 (1977).
9) Howden, W. E.: An Evaluation of the Effectiveness of Symbolic Testing, Softw. Pract. Exper., Vol. 8, No. 4, pp. 381-397 (1978).
10) King, J. C.: A Program Verifier, Ph. D.

thesis, Carnegie-Mellon University (1969).
11) King, J. C.: A New Approach to Program Testing, Proc. International Conference on Reliable Software, pp. 228-233 (1975).
12) King, J. C.: Symbolic Execution and Program Testing, Comm. ACM, Vol. 19, No. 7, pp. 385-394 (1976).
13) Osterweil, L. J. and Fosdick, L. D.: DAVE-A Validation Error Detection and Documentation System for Fortran Programs, Softw. Pract. Exper., Vol. 6, No. 6, pp. 473-486 (1976).
14) Ramamoorthy, C. V., Ho, S. F. and Chen, W. T.: On the Automated Generation of Program Test Data, IEEE Trans. Softw. Eng., Vol. SE-2, No. 4, pp. 293-300 (1976).
15) Topor, R. W.: Interactive Program Verification Using Virtual Programs, Ph. D. thesis, University of Edinburgh (1975).
16) von Henke, F. W. and Luckham, D. C.: A Methodology for Verifying Programs, Proc. International Conference on Reliable Software, pp. 156-163 (1975).
17) Waldinger, R. J. and Levitt, K. N.: Reasoning about Programs, Artif. Intell., Vol. 5, pp. 235-316 (1974).
18) ソフトウェア産業振興協会：ソフトウェア・エンジニアリングに関する調査研究——プログラム検証系，昭和55年度報告書.

(昭和56年6月2日受付)