

プログラムの検証例とその検討

岡本吉晴, 玉井哲雄, 福永光一
(株)三菱総合研究所

1. はじめに

プログラムの品質を保証するためには、単にテストを行うのでは不十分であり、もっと積極的に、プログラムがその仕様を論理的に正しく満足していることを示す（検証する）のでなければならぬといふ考え方には、現場のプログラマにもかなり普及しつつあるようである。しかし、その考え方には実際には“理想論”として受け取られてはいるのがほとんどである。たとえば、我々が行ったヒヤリングでも、自分の業務で検証を行なうことが可能であると考えてはいるプログラマはまことにないといふ結果が得られてはいる。

この考え方の普及と実践との間の隔たりの理由としては、次のようなものが挙げられるよう。

- 既存の検証例が、人手による実験システムによるとを問わず、簡単なものか使用されるプログラム言語に制約を加えたものに限られてはいるため、方法の有効性に対して疑問が持たれてはいること。
- 検証方法の多くが、述語論理等を使用しており、プログラマの日常使ひなれていける“ことは”とはかけ離れてはいるため、理解したり真似をしたりすることができ困難であること。また、手堅なことを証明するのに“七面倒臭い”議論が必要となり、手間の割に効果が上がらないようと思われるのこと。
- 通常の業務では、コスト面（人の手間、検証用システムの開発費、計算時間等）で、現行のテスト等による方法に付し優越性を感じられないこと。

これらの問題は、理論家（研究者）の側から見れば、いずれも大した問題ではなく、時間が経る研究が進めば解決するものであつたり、プログラマの質や教育の問題として片付け得るものであるかわしえない。しかし、現場のプログラマにとっては、いかにより道具があつても、その説明書がやせびや部厚いと、その道具は一部の物好き以外には決して使われることがないといふ事実が示すように、“検証”に関する方法なり道具なりが、“手堅”に使いこなせるか否かといふことが問題なのである。もちろんこの手堅さの意味には、検証を行なうときに“こつ”的なものが必要ななら（既存の検証例では、この“こつ”が暗黙のうちに使われてはいることが多い）、それが明らかにされていふことも含む。そして、この観点から見ると、既存の検証理論はどこか“手堅”と言えるような代物ではないことが明らかである。（もつとも、“手堅に使いこなす”といふことの中には、プログラマの質や教育に関する部分が少なからずあると思われ、それはまた一つの大きな問題として扱われる必要があることは確かである。しかし、検証方法が“実用的”であるためには、プログラマの側に過度に高級な能力を期待することは不適当であると考えられる。）

このように、その必要性は十分痛感されてしまふにわかかわらず、検証といふ手段が現場のプログラマの間で、手堅に使われるようになるまでには、まだかなりの距離があるようである。したがつて、現場のプログラマの観点から、この距離の原因は何であるのか、そしてその距離を小さくするにはどのふうな問題が解決されねばならないことを考えてみる必要がある。そのための手段として現在我々

は、次に述べるような方針にしたがって、実際のプログラムを検証し、その結果を検討するという作業を行いつつある。本稿では、その作業の経過と現在までに我々の得た（ごく当り前の！）結論について報告する。

2. 作業の方針

1で述べたように、我々はプログラム作成の現場で役立つような“検証”的条件について考えている。そのためには、既存の研究成果を集めて検証系を試作するよりも、プログラムの信頼性を保証するための実用的な検証の方法、あるいはプログラムの設計・作成段階への検証的な考え方の適用方法を確立させることが先決であると考える。そのためには、実際のプログラムを検証してみて問題点を明らかにするのが効果的であると考え、次ののような手順で作業を行っている。

- (1) 種類・規模・使用言語から見て、現場でよく見られるタイプと思われるプログラムを片っ端から検証してみる。検証のための道具立てとしては、最初は既存の検証方法をとり、その真似をしてみる。そして検証例が蓄積されるに従って、個々の検証例で得られた経験を必要になった工夫等を道具の中に追加して行く。
- (2) (1)の結果から検証方法、あるいは検証しつつプログラムを作成する方法についての手引きを導き出す。
- (3) (2)の手引きのうち、自動化可能なものとそうでないものを区別し、それぞれの問題点を整理する。

3. プログラムの検証例

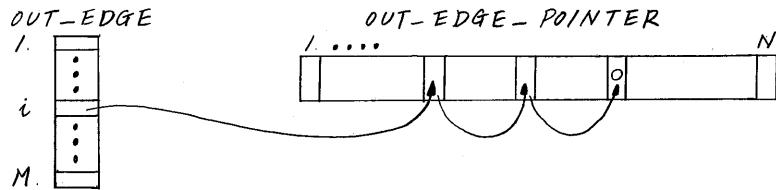
本節では、我々が現在までに行ったプログラムの検証結果について述べ、検証の段階で問題となった点について検討する。

(1) *Depth-First Spanning Tree*によるループの検出プログラムGTREEの検証
このプログラムは、プログラム・フロー・グラフを入力として、そのプログラム内のループを検出することを目的としている。プログラムは、プログラム・フロー・グラフが入力となって、入口ノードから、入力されたグラフの Depth-First Spanning Tree (DFST) を構成し、その Tree の Back Edge (Tree の子孫ノードから先祖ノードへ向かうエッジ) を検出して、それを出力するようになっている。Back Edge は、グラフの各ループ内の先頭ノードへのエッジとなっており、これは“プログラム・フロー・グラフ G から、入口ノードを root とする G の DEST の Back Edge をとり除いた G のサブグラフは、ループを持たない”という性質に基づいている。入力されるプログラム・フロー・グラフは、1つの入口ノードを持ち、入口ノードから他のすべてのノードへのパスは必ず存在すると仮定されている。PL/I で書かれており、34ステップである。

このプログラムの検証は、ステップ数も短く、アルゴリズムもすっきりした形になっていたので、Hoare の FIND の検証例に示されていよう。プログラム内の変数に対する述語論理を用ひて、厳密に検証することを試めた。この試験で次のような問題点が挙げられた。

- ① グラフを表現する入力変数の仕様が、繁雑となってしまった。プログラムでは、ノードの数 M 、エッジの数 N として、グラフは、各エッジの終点ノード番号が入れられていく配列 IN-NODE (N) と同じノードから出るエッジの集合を2つの配列 OUT-EDGE (M) と OUT-EDGE-POINTER (N) で下図の

ようにポインタで結んで表現している。



これらの変数に対する入力仕様は、10個の条件の論理積となつた。例えば。
 $\forall i \{ 1 \leq i \leq M, OUT-EDGE(i) \neq 0 \} \wedge \exists k \{ 1 \leq k \leq N, OUT-EDGE-POINTER(k) = OUT-EDGE(i) \}$
 等々。

- ② 出力仕様も、Tree であること・Back Edge であること、等の仕様が繁雑である。
 - ③ したがって証明部分も、繁雑で見にくくなることになった。
- このような短くて、割合にすくきりしたプログラムでも、そのプログラム内の変数をそのまま用いた述語論理を使用して証明することは、かなり繁雑となるので、実用規模のプログラムの検証と、このようなやり方で人手で行なうことには、限界があると考えられる。

(2) 算述演算式の処理プログラム CMATH1 の検証

このプログラムは、算述演算式が書かれている文字列が入力され、その文字列から算述演算式のシンボルとなるものを認識して、シンボルの内部コードの列を抽出し、エラーがあれば検出して、エラー・コードをセットする。ただし、算述演算式の構文解析までは行なわない。FORTRANで91ステップ(コメントを除く)のプログラムである。

このプログラムでは、

算述演算式 ::= {シンボルの列}

シンボル ::= {識別子 | 定数 | + | - | * | / | ** | < | > | }

識別子 ::= {先頭が英字、それ以降は英数字の8文字以内の文字列}

定数 ::= {FORTRANで許される整定数および実定数}

を考える。シンボルとシンボルの間は、1つ以上のブランクがあつてもよい。シンボル**の認識に関しては、文字列の中に*が2つ以上つづいて現われる場合、一般にn個つづいて現われたとすると、[n/2]個シンボル**がつづいて、その後に、nが奇数ならば1つのシンボル*があるとの認識される。

このプログラムの検証は、次のように2段階に分けて行なつた。

[I] 入力された文字列から正しくシンボルを認識して取り出し、シンボルの列を生成していることの検証

[II] 各シンボルの処理で、正しく内部コードに変換して、エラーがあれば正しく検出していることの検証

[I]では、与えられたプログラムの各部と対応させ、一般上位の抽象的な手続きに書き直し、このレベルで検証を行なつた。[II]では、各シンボルの場合分けが正しく行なわれ、各シンボルに対する処理が正しく行なわれていることを、プログラムのレベルで検証した。このようにプログラムでは、

文字列 → シンボルの内部コードの列

への変換を直接行なつてはいるが、これを

[I] 文字列 → シンボルの列

[II] シンボル → シンボルの内部コード

という具合に、段階的にとらえて検証することによって、見やすくなりやすい検証とすることができます。

(3) 等高線作図プログラムの検証

このプログラムは、格子点の高さが与えられたときに、その等高線図をプロッターに描くものである。プログラムは、Fortranで24ステップのものであり。メイン・プログラムヒ4つのサブルーチンから成る。

検証の段階で問題となつたのは、何を検証すべきかということである。この場合、等高線といふのが直観的にわかりやすいものであるために、その厳密な定義を与えるはすぐ問題が解決するかのように錯覚しがちである。ところが実際にほ、この定義はそれほど簡単ではなく、しかもプログラムの検証そのものにはこの定義が必要ないものである。等高線を描くためには何らかの曲面を想定しなければならないが、このプログラムでは、格子点を頂点とする各最小の正方形領域で、2つある対角線のうちいずれか1方にとり、2つの三角形領域に分け、各三角形領域ごとに3つの格子点を通る平面で、等高線を描くべき曲面を近似することにしておる。入力データは格子点における高さだけであり、曲面に対する記述は何もないから、この近似を検証すべきものでも何でもなく、プログラムの仕様と見なされねばならないのである。このようにすると、プログラムを検証するためには、すべての三角形領域で近似平面に対し正しく等高線が引かれていることと、曲面を分割して得られる各部分曲面で求められた等高線を合わせたものは元の曲面の等高線になつていいことが示されねばよりことになる。このプログラムの場合前者は容易に検証される。一方後者は、よく考えてみるとプログラムの検証といふ枠内で証明されるべきものではなく、むしろ“曲面の等高線”に関する問題領域の定理とみなすべきものであり、プログラムの検証においてはこのような問題に関する知識は所与の定理として扱われるべきことわかる。

この例は簡単なものではあるが、我々がプログラムを検証する前に、問題をよく見極め、検証すべきことと、所与の事実を区別することの重要性を理解するよい例となつた。

(4) 記憶域管理サブルーチン ZMNG の検証

ZMNG は一定の大きさの主記憶域しか使用できないという制限の下でマトリックス間の演算を行なうように作られた問題向き言語の、実行時の裏方を努める約200ステップの Fortran サブルーチンであり、その主な機能は以下の通りである。

- ・上位ルーチンから指定されたマトリックスが主記憶域内にあれば、その先頭アドレスを返す。
- ・指定されたマトリックスが主記憶域内になく、かつ主記憶域内にそのマトリックスを収容するのに十分な空き領域があれば、マトリックスの値を主記憶域内に読み込み、その先頭アドレスを返す。十分な空き領域がない場合は、求めマトリックス間に定められた優先順位に従い、優先順位の低いマトリックスから順に主記憶域外に追い出し、必要かつ最小限の空き領域を得た後、指定マトリックスを読み込み、その先頭アドレスを返す。（その際、主記憶域外に追い出してもいいながらマトリックスが1つだけ指定されていることがある。）

上記の機能を見れば判るように、ZMNG は、各マトリックスの状態や位置を示すテーブルヒストリックスを収容する領域を管理している。ZMNG は Fortran で書かれているので、こちらのテーブルや領域は、配列により実現されていく。

したがって ZMNG の操作は、その殆んどがこれらの配列内の値の参照および代入と、そのための添字式の計算から成る。

この ZMNG を書かれたプログラムのレベルで検証しようとすると、すぐわかるようにプログラムの仕様を表明は、これらの配列の値や添字に関する複雑な計算式の間の関係を細かく記した命題群として与えられることになる。そして、検証を行なうためには、これらの長々とした表明に対する変換操作を約200回も繰り返さなければならぬことになる。すなわち、ZMNG をこのレベルで検証しようとすると、適切な表明を見つけるのが難しく、かつ検証に要する労力が非常に大きくなることが予想される。このため、検証過程そのものに誤りが入り込む確率が高くなると考えられる。この理由としては、与えられたプログラムのレベルでの検証の場合、検証すべきものの中で ~~プログラム~~ の本來の機能と、その機能を実現するためのプログラム上の約束事が難しく、それらが混在させられてしまうことが挙げられる。したがって、ここではまず ZMNG の本來の目的を考え、それを仕様として厳密に記述し、その仕様を ZMNG が満たしているかどうかをチェックする。そして、次にその仕様をよりプログラムのレベルに近い形で記述し再びチェックを行なうといふかたちで検証を行なうことにする。

a) ZMNG の検証

指定されたマトリックスが主記憶域内にある場合の操作は極めて簡単であるので、そのマトリックスが主記憶域内になり場合についてのみ考える。

上に述べた仕様の後半を厳密に記述するためには、マトリックスの状態(主記憶域内にあるか否か)、優先順位等が定義されねばよい。これは次のようにすればよい。まず次の記号を採用する。

M: 与えられているすべてのマトリックスから成る集合

n: M の要素の数

Mem: マトリックス格納領域の大きさ(正の整数)

A: 主記憶域内にあるマトリックスの集合

B: 主記憶域外にあるマトリックスの集合

このとき、問題の意味から、 $A \cap B = \emptyset$ であり、M は A と B の直和である。

これを、 $M = A + B$

と書く。優先順位とマトリックスの大きさを記述するためには、次の二つの集合と二つの写像を考えねばよい。但し N と N' には通常の自然数の集合の意味での順序関係が入っており、N には加算が定義されているものとする。

N: 自然数の集合

$N' = \{i \mid i \in N \wedge 1 \leq i \leq n\}$

order: M から N' への全単射

size: M から N への 1 対 1 写像

このとき指定されたマトリックスを α 、追記出しへは β のマトリックスを β (もし β のようなものがない場合 $\beta = w$, $w \in M$, $M' = M \cup \{w\}$ とする) と書くことにすると、ZMNG の仕様は以下のようになら記述される。但し、仕様中 A_0 とは A の実行開始時ににおける A の値を示すものとする。(他も同様)

ZMNG の仕様

・入力表明

$$\alpha \in B \wedge \beta \in M' \wedge \alpha \neq \beta \wedge M = A + B \wedge \sum_{i \in A} \text{size}(i) \leq Mem$$

• 出力表明

$$\begin{aligned}
 & \alpha \in A \wedge \beta \in M' \wedge \alpha \neq \beta \wedge M = A + B \wedge \sum_{i \in A} \text{size}(i) \leq \text{Mem} \\
 & \wedge (\beta \in A_0 \supset \beta \in A) \\
 & \wedge \forall i \forall j (i \in A_0 \wedge j \in A_0 \wedge j \neq \beta \wedge \text{order}(i) > \text{order}(j) \wedge j \in A \supset i \in A) \\
 & \wedge (A_0 - A \neq \emptyset \supset \sum_{i \in A} \text{size}(i) + \sum_{j \in A \setminus \{i \mid \text{order}(j) = \max_{m \in A_0 - A} \text{order}(m)\}} \text{size}(j) > \text{Mem})
 \end{aligned}$$

ZMNG のプログラム中でこの仕様を実現している部分を仕様と同じレベルで書き下すようになる。

ZMNG のプログラム

begin

$$iempt := \text{Mem} - \sum_{i \in A} \text{size}(i);$$

while $iempt < \text{size}(\alpha)$ do $\cdots \cdots \cdots (*)$ — ループの入口

begin

if $A - \{\beta\} = \emptyset$ then error;

$$x := \text{out}(A, \beta);$$

$$A := A - \{x\};$$

$$B := B \cup \{x\};$$

$$iempt := iempt + \text{size}(x);$$

end;

$$A := A \cup \{\alpha\};$$

$$B := B - \{\alpha\}$$

end

たとえ $\text{out}(\alpha, \beta)$ は function でその値は、次の条件を満たすようなマトリックス x を返すものとする。

$$\begin{aligned} & i \in A \wedge \text{order}(i) = \min_{j \in A \setminus \{\beta\}} \text{order}(j) \end{aligned}$$

このプログラムが上記の入出力表明に対して正しいことを言うためには、プログラムの(*)の部分で次の表明が成立することを示せばよい。

$$\begin{aligned}
 & \underbrace{\alpha \in B \wedge \beta \in M'}_① \wedge \underbrace{M = A + B}_② \wedge \underbrace{iempt + \sum_{i \in A} \text{size}(i) = \text{Mem}}_③, \\
 & \wedge \underbrace{(\beta \in A_0 \supset \beta \in A)}_④, \quad \wedge \underbrace{\forall i (i \in A \supset i \in A_0)}_⑤, \\
 & \wedge \underbrace{\forall i \forall j (i \in A_0 \wedge j \in A_0 \wedge j \neq \beta \wedge \text{order}(i) > \text{order}(j) \wedge j \in A \supset i \in A)}_⑥, \\
 & \wedge \underbrace{(A_0 - A \neq \emptyset \supset \text{size}(\alpha) + \sum_{i \in A} \text{size}(i) + \text{size}(x) > \text{Mem} \wedge \text{order}(x) = \max_{k \in A_0 - A} \text{order}(k))}_⑦, \\
 & \wedge \underbrace{(A_0 - A \neq \emptyset \wedge A - \{\beta\} \neq \emptyset \supset \forall i (i \in A \wedge i \neq \beta \wedge j \in A_0 - A \supset \text{order}(i) > \text{order}(j))}_⑧,
 \end{aligned}$$

この表明の正しさは次のようにして言ふことができる。

i) プログラムの実行中はじめて(*)へ来たとき

- $iempt$ の計算方法より、 $iempt + \sum_{i \in A} \text{size}(i) = \text{Mem}$ が明らか。

- $A = A_0 \wedge B = B_0$ であるから

$$M = A + B, \quad \forall i (i \in A \supset i \in A_0)$$

$$\beta \in A_0 \supset \beta \in A, \quad \alpha \in B$$

は明らかに成立し、また $i \in A_0 \supset i \in A$ であるから。

$$\forall i (i \in A_0 \wedge i \in A \wedge i \neq \beta \wedge \text{order}(i) > \text{order}(\beta) \wedge i \in A \supset i \in A)$$

また $A_0 - A = \emptyset$ であるから、最後の式が成立する。

ii) ループを一周するとき

- $A := A - \{x\}, B := B + \{x\}$ は同じになって現われるから $M = A + B$ は不变、また明らかに①は不变。
- A に対する操作は $A := A - \{x\}$ のみであるから $A_0 \supset A$ より $\forall i (i \in A \supset i \in A_0)$ また $i \in A_0$ であるから $i \in A$
- (***)において、その直前の if 文の意味より $A - \{\beta\} \neq \emptyset$ だから、out の定義より $x \in A \wedge x \neq \beta$ 、よって ⑤ より $x \in A$ 。
また ⑧ より $\forall i (i \in A_0 - A \supset \text{order}(x) > \text{order}(i))$ ----- ⑨
またループが実行されているから $\text{empt} < \text{size}(x)$ である。
これと ③ より $\text{size}(x) + \sum_{i \in A - \{x\}} \text{size}(i) + \text{size}(x) > \text{Mem}$ ----- ⑩
 $\text{empt} + \sum_{i \in A - \{x\}} \text{size}(i) + \text{size}(x) = \text{Mem}$ ----- ⑪

以上より (****) において次のことが言える。

(1) $x \neq \beta$ であるから (**) で $\beta \in A$ なら (****) でも $\beta \in A$ 、よって ④ は不变

(2) ⑦ より ③ は不变

(3) $A := A - \{x\}$ の結果、 $x \in A$ ふれと $x \in A_0$ より $x \in A_0 - A$ 、さらに ⑨ ⑩ より
 $\text{order}(x) = \max_{m \in A_0 - A} \text{order}(m)$ よって ⑦ が成立する。

(4) out の定義より、 $\forall i (i \in A \wedge i \neq x \wedge i \neq \beta \supset \text{order}(i) > \text{order}(x))$
これより ⑥ ⑧ が成立する。

i) ii) の結果、(*)における ① ~ ⑧ の表明は不变に成立する。

b) 検証結果の検討

上記の検証は、ZMNG の最上位レベルでの検証である。ZMNG の仕様には、この他に主記憶域内にあるマトリックスは優先順位の順にすき間なく並べられており、主記憶域外のマトリックスは、データ・ベースと作業ファイルのいずれかに値がいれられていくこと等がある。したがって、実際の検証では、上記の作業後、マトリックスの主記憶域内アドレス等を定義し、仕様をそのレベルで書き、プログラムの対応する部分を検証するという作業を繰り返した。この検証例から得られる問題点としては次のようないちのが挙げられる。

- 問題をよく理解し、主要な概念をきちんと定式化すれば、検証に要する時間は大幅に低減する。すなわち、検証においても“抽象化”という操作が有効かつ必要であることを示している。
- しかし、それでもこの検証の場合、(a)で述べた検証と“アドレス”を定義し大段階での検証を書くと、A-4版レポート用紙で24~25枚となる。これは200ステップ程度のプログラムをある程度抽象化して検証した場合の数字である。したがって“実用的”な検証方法においては、適当な記法や証明方法の採用により、この種のドキュメントを薄くする（必要な証明の量を減らす）ための工夫が必要になると思われる。
- この検証例では、仕様を表明は述語論理風の表現で記述されているが、この方法はあまりわかりやすいためのとは言えない。たとえば、ZMNGにおいて、指定されたマトリックスを主記憶域内に入れるために必要最小限の空白領域を作るための操作が行なわれていることを言うのに、(*)における表明で ⑦ ⑧

のよう体まわりくどい表現をしなければならない。したがって問題がより複雑になる場合には、言いたいことを素直に表現できるような証明のための“こじば”を考える必要があると思われる。

(5) データ・フロー解析用およびパス列挙用プログラムの作成と検証

次の2つの例は、いずれもグラフを扱ったものである。一応、データの抽象化を考慮したプログラム設計を行ない、それに基づいて検証を試みた点に、特徴がある。

問題1 (広域的データ・フロー解析の問題)

主としてプログラムのコンパイル時の最適化を目的とした種々の解析を、一般的に定式化したものである。プログラムのフロー・グラフ（制御の流れを表わすグラフ）上の各点で、入ってくるデータに対し出ていくデータの変換規則が与えられて、こからの変換規則を満す、データの流れの定常解を求める、といつた問題である。Kildallによる定式化に従えば、入力として、フロー・グラフの構造と、データを表わす半束の構造と、グラフ上の各点での変換規則を与えて、出力として各点に対し、条件を満す半束の要素を対応づける形になる。

解法は、Kildallによるものを多少改良し、また Hecht & Ullmanによる効率化を、プログラムのごく一部の変更で導入できるようにした。

問題2 (プログラムのパスを列挙する問題)

プログラムの制御の流れを表わすフロー・グラフが与えられて、そのグラフ上でプログラムの入口点から出口点へ至るパスの集合で、プログラム上の分歧は少なくとも一度、いずれかのパスで実現される（すなはち、グラフ上の枝は、いずれかのパス上に少なくとも一度現われる）ものを求める問題である。

解法は、必ずまだ通っていない枝が少なくとも1つは含まれるように注意しながら、パスを探索していくという、直接的なものとした。

この2つのプログラムに共通していえることは、以下のようである。

- 比較的、簡単なプログラムである。いずれもPL/Iで実際にコーディングして、50ステップ前後であった（ただし入出力等を除いた、本体の部分）。
- 実際的なプログラムである。いずれも、現在我々が作成する必要があり、この検証に並行して考えたtop-downの設計に基づいて作成し、効率などのテストを行なったものである。
- 主となる対象のデータは、グラフという割合よく使われ、研究もされている構造のものである。グラフは、よく使われるが、その構造はそれほど単純ではなく、データの抽象化の考え方が段階的・複雑さをもつ。

検証は次のような形で行なった。

- 1階述語論理によるといったオーバーマルなもののせず、自然言語によった。説明易さを考えてのことであるが、厳密さは欠けていいつけたりではいる。
- 最終的なPL/Iのプログラムのレベルではなく、抽象的なデータに対する操作によって記述されたものに、直接の検証を行なった。その抽象のレベルでのプログラムが、PL/Iのプログラムに正しくあてはまるることは、ほんのり自明だと考えて、それについて証明を与えていい。

結果として、検証を行なうことにより、通常のプログラム作成とデバッフを行なったのでは、気づかなかったか、気づくのに時間がかかったと思われる、入力時の重要な仮定やアルゴリズムの誤まりを、比較的楽に発見してきた。具体的には、たとえば問題2の方で、

- フロー・グラフの条件として、入口点からすべての点へ至るパスが存在するだけではなく、出口点が必ずあって、すべての点から出口点へ至るパスが存在する必要があることは、検証によって初めて明示的になった。
- 初めに考えたアルゴリズムではプログラムの終了が証明できなかった。同時に進行していったプログラムのテストで、プログラムが終了しない例が見つかり、アルゴリズムの修正を行なって、終了の証明ができるとともに、このテスト例でも正しい結果が得られるようになった。

ここで行なったプログラム作成の方法は、Dijkstraなどのが主張するものに、(我々の理解できる範囲で)従がったものであり、それに試みた検証も、たとえば Hoare, Gries, Good などがやつておせたものと真似以上に、新しいものはおそらくないだろう。ただ、検証をしてみて過程での経験や失敗を明らかにすることに、意味があるのではないかと思う。

4.まとめ——教訓と問題点——

本節では、我々がこれまで述べたような検証を行なって得たいくつかの“教訓”や現行の(自動的な)検証方法の問題点等について簡単に述べる。まず教訓としては、次のようなものが得られた。

- 既に書かれたプログラムを検証しようとすると、殆んどの場合そのプログラムの仕様は(検証するためには)不十分である。したがって、プログラムの目的を十分吟味し、“常識”として暗黙のうちに使用されていける仮定を見出し、検証すべきこととそうでないことをはっきり区別するという作業が必要である。このような作業の結果適切な仕様を得たときは、検証作業のかなりの部分が終了していふような場合が多い。
- 書かれたプログラムを検証する場合でも抽象化は有効かつ必要である。(これはある意味でこの行為が逆立ちしていふことを物語っている。)プログラムの大まかな制御構造を見て挿入すべき表明の形式を見極めた後プログラムを分解し、分解された個々の操作に対する仕様を見つけるという具合に、検証の手続きを top-down に行なっていくという方法は有効である。
- 最初から完全な表明を見出せることはまずないのと、検証してみて失敗したら表明の手直しをするという練り返しを覚悟しておいた方がよい。このような作業のためにには、会話型の検証システムがあれば便利である。

このような教訓は、考えておれば当たり前のことがあるが、実際に検証を行なってけないとなかなかピンとこないという類のものである。またこれらは初心者のつまづきのものもあるので、“検証”を普及させるためにには適切な事例を添えた手引書のようなものがあればよいと思われる。次に、検証をしていて困難を感じた点や、もっと良い方法はないものかと思った事項を以下に挙げる。

- 検証過程で現われるいろいろな概念等を素直に表現する能力をもった表記法が必要である。(我々の場合、たとえば変数の別の時点の値に言及する際、その都度新しい記号の導入を余儀なくされた。)
- 採用された表記法に対する望ましい表明の書き方等を確立する必要がある。(プログラムの書き方を見てわからるように、同一の表記法を用いて同じことを言う場合でも、その書き方によりわかりやすさの度合が大きく変化する。)
- 検証を行なう際、日常よく使われわかりやすい推論方法を使えるようにする必要がある。(3の(5)の検証は、“----したことがある。それ故----である。”

という形式の推論が為されているため非常にわかりやすくなっているが、現在の自動的な検証システムはこの種の方法の取り扱いが不得手のようである。上記のことに関連するが、簡単なことを検証するのに要する労力が大きすぎるようと思われる。単に検証過程を自動化するだけではなく、証明の量そのものを減らさせる工夫が必要である。

これらの問題は、いざれも検証のための“ことば”と“ことばづかい”に係るものである。プログラムの作成とか検証とかいう行為が極めて人間的な行為であることを考えれば、このような“ことば”に係る問題の重要性は容易に了解されよう。したがって今後、わかりやすく表現能力に富んだ検証のための“ことば”や“ことばづかい”に対する研究を進めることが重要である。