ソフトウェア工学 36－1
（1984. 6. 27）

プログラムの検証と再利用を考慮したプログラム作成支援システム

永田守男、福島敏夫
（慶大・理工）

1、はじめに

近年、ソフトウェア開発において、生産性と信頼性の向上が望まれている。このようなことについての研究では、ソフトウェア開発のライフサイクル（要求仕様の記述、設計、プログラミング、保守・管理といわれるもの）をまとめた上で、上記の事を考慮しなければならないが、今回は特にプログラミング段階で信頼性の高いプログラムを作成する事を支援するための方法とこれを実現したシステムに関して述べる。

これまでにも、ソフトウェア開発のライフサイクルを支援するためのGandalfプロジェクト[Ha81]、あるいはシステム設計のための方法論としてのジャクソン法[Ja75]や、プログラミング方法論としてのダイクストラの方法[Di76]などが提案されてきた。またプログラムの信頼性についてのことに限っても、プログラムの正当性の検証[Bo81]やプログラムのテスト技法[My79]などの研究が行なわれてきた。

しかしながら、これらの研究では全体の規模が大きすぎたり方法論の提案だけであったりして、実際の支援システムを製作するにあたっての問題点や使用してみての経験の蓄積などが乏しく、Lisp系統の比較的問題の整理しやすいプログラムについていくつかのシステムが報告されているにすぎない[wa82][Na80]。

そこで今回は、こういった研究をまとめた上で、主にプログラムの信頼性を上げるためにプログラミング段階で支援するシステムとして、次のような特徴を持たせることを考えた。

① 本当の意味でプログラムをトップダウンに作成する事を支援する。
② プログラムの作成と正当性の証明が並行して行なえる。
③ 既存のプログラムを検索し、それをプログラムテキスト中に挿入することで、プログラム開発において、プログラマの負担を軽減させている。

2、システム実現のアプローチ

2.1 トップダウンによるプログラム作成と正当性の検証

プログラム作成とは、設計仕様をプログラムに変換することである。ここでは、設計仕様を作成すべきプログラムの事前条件（pre-Condition）と事後条件（post-Condition）で記述するものとする。事前条件とは、プログラムが正しく実行されるために成立していなければならないプログラム変数間の関係であり、事後条件とは、実行後に成立している条件である。

プログラムは、図1に示すように作成していく。ある段階の設計仕様が与えてある時に、これに対応するプログラムとこれより下位の段階でプログラムになる部分の設計仕様に変換する。これをトップからボトムまでの各段階に対して行なうことにより、プログラムを作成していく。

また、ある段階において、その段階に対する設計仕様、その設計仕様に対するプログラム、およびそれより下位の段階でプログラムになる部分に対する設計仕

様が与えられた時に、その段階において設計仕様とプログラムが一致しているかどうかを調べる。3.2で述べる方法に従って、各段階ごとに、プログラムの正当性の検証を行なうが、このことによる利点は、次の2点である。前者はプログラマ側から見た利点であり、後者はシステム側から見た利点である。

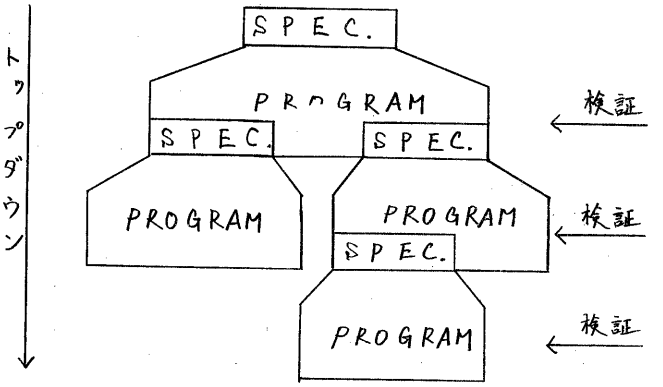

図1. プログラムの階層構造

① 正当性の証明が失敗に終わった場合、誤りのある場所が限定されているので、その原因の発見が容易である。
② プログラムの一部について正当性の検証を行なうため、検証条件の複雑化が防げる。

2.2 対象とするプログラミング言語

本システムで対象とするプログラミング言語はPascalであるが、正当性の検証を簡単にするために次のような制限を加える。

① 使用できる変数の型は、整数型だけとする。
② 使用できる文は、代入文、複合文、選択文(if文)、繰り返し文(while文)、および手続き呼び出し文とする。
③ 与えられた設計仕様は、プログラムテキストの中に記述する。

```
procedure sub(…);
  const   …;
  var     …;
  define  …;
  pre     …;
  post    …;
```

図2 宣言部

本システムにおけるPascalの宣言部は、図2に示すように、const(定数宣言)、var(変数宣言)、および一般のPascalにはないdefine、pre、postの5つからなっている。define、pre、およびpostは、次のような働きを持っている。

define：手続きの実行部の実行部で用いられるwhile文の不変条件(Loop Invariant)を関数の形式で記述する。
pre　：手続きの事前条件を記述する。
post　：手続きの事後条件を記述する。

実行部の中で予想される不変条件を関数の形式で記述することによる長所は、次の2点である。

① 繰り返しの中で値の変化する変数(繰り返しの制御変数)を関数の引数リストより、システムが簡単に認識することができる。この情報は、プログラムの正当性の検証の時と、既存のプログラムを検索する時に利用する。

② この不変条件は、事後条件を記述する時に再び現れるので、かわりに関数名を記述することにより、同じ論理式を2度記述しないで済む。

以上の事を例を挙げて説明する。

「定数aとbが与えられた時にaとbの商qと余りrを除算を使用せずに求める」という問題を用いると宣言部は、図3に示すようになる。

```
program main
  var  a,b,q,r: integer;
  define Loop(q,r): a=q*b+r & r>=0;
  pre   a>=0 & b>0;
  post  Loop(q,r) & r<b;
```

図3 宣言部の例

ここで、define文で定義された不変条件を調べることにより、繰り返しの中で変化する変数は、qとrであり、それ以外の変数は繰り返しの中では、定数であるということがわかる。

このことにより、プログラムの正当性の検証時に、繰り返しの直前で得られている必要な情報（あるプログラム変数が保持している記号的値）を失なわないで済む。

3. システムの構成およびその実現

3.1 システムの概要

システム構成は、図4に示すとおりである。図よりわかるように、プログラム作成中は、内部表現に対して編集したり、正当性の検証をしたりする。

本システムは、大きく分けて5つの副システムよりなっている。

コマンド解釈実行部は、与えられたコマンドを解釈を、他の副システムを呼び出す。

解析部は、Pascalの原始プログラムを内部表現に変換する。

生成部は、プログラムをプログラマに表示するために、内部表現を原始プログラムに変換する。

編集部は、与えられた設計仕様と同じ設計仕様を持つ既存のプログラムを検索し、プログラムテキスト中に挿入する。

検証部は、与えられたプログラムの検証を行なう。

また、原始プログラムの内部表現は、手続きの階層構造は図1に示したような構造で表わし、各手続きの宣言部および実行部は演算を前置形で表わしている。

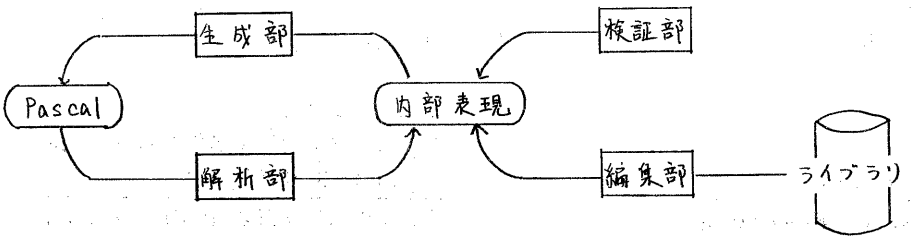

図4 システムの構成

3.2 検証条件の生成法と正当性の検証

プログラムの正当性の検証は、次に示す3段階の手順で行なわれる。

① 事前条件や事後条件などの命主張を形式的記法を用いて記述する。
② プログラム中のそれぞれの経路について、証明すべき検証条件を生成する。
③ 形式的推論規則を用いて、証明を実行する。

本システムでは、上記の3項目を次に示す方法を用いて、実現している。

① 一階の述語論理に制限を加えて用いている。
② 記号的実行の概念を用いることにより、帰納的アサーション法を使用して、検証条件を形式的に生成している[Ida82]。
③ 自然演繹システムを用いて、検証条件を証明している。

まず、検証条件を形式的に生成していく方法について説明する。そのために、いくつかの記法について述べておく。

(1)式は、状態(state) S と経路条件(path condition) PC のもとで、文の並び A の正当性を示すものである。

$$S, PC \setminus A \quad \cdots (1)$$

(2)式は、式 exp を状態 S のもとで評価する操作を示している。S を $\{a_1/x_1, \cdots, a_i/x_i, \cdots a_n/x_n\}$ (x_i という変数が記号値 a_i という値を持っていることを表わす)とすると、(2)式は exp の中に現われる x_i を a_i で置き換えればよい。

$$exp \mid S \quad \cdots (2)$$

(3)式は、状態 S を $y := M$ という置換により、新しい状態 S' を作り出す操作を示している。

$$S' = \{M/y\}S = \{M|S/y, a_1/x_1, \cdots a_n/x_n\} \quad \cdots (3)$$

ただし、$y = x_i$ の時、a_i/x_i は省く。

以上の記法を用いて、以下で5種類の文に対する推論規則を示す。

1) 代入文

$$\frac{\{exp/x\}S, PC \setminus A}{S, PC \setminus x := exp; A}$$

2) if文

$$\frac{S, PC \,\&\, B \mid S \setminus A2; A1 \qquad S, PC \,\&\, \neg B \mid S \setminus A3; A1}{S, PC \setminus \text{if } B \text{ then } A2 \text{ else } A3; A1}$$

3) confirm 文

$$\frac{PC \longrightarrow Q \mid S}{S, PC \setminus \text{confirm } Q}$$

4) while 文

$$\frac{PC \longrightarrow I \mid S \qquad S', (I \,\&\, B) \mid S' \setminus A2; \text{confirm } I \qquad S', (I \,\&\, \neg B) \mid S' \setminus A1}{S, PC \setminus \text{while } B \text{ do } A2; A1}$$

I: 不変条件
S': S の中の制御変数を初期化したもの.

5) 手続き呼び出し文

$$\frac{PC \longrightarrow P[w/x, e/y]S \quad S', PC \,\&\, Q[S'' \setminus A}{S, PC \setminus proc(w, e); A}$$

S': Sの中のwを初期化したもの
S'': {e/y, w/#x} S'
#x: 手続きが呼ばれた時のxの値を示す。

```
procedure proc(var x; const y);
  pre P;
  post Q;
  begin
    ....
  end.
```

図5 呼び出される手続き

以上の推論規則に対し、簡単に説明を加える。

1) x:=expという文が、xの値をexpに変更することを示す。

2) 許文においては、条件式Bが成立する場合と成立しない場合の2通りがあり、この2つについて検証すればよい。

3) confirm文は、プログラムが正当であるためには、QがPCに含意されなければならないことを示している。

4) while文は、次の3式を検証すればよいことを示している。1つは、現在の状態において不変条件が経路条件に含意される。2つ目は、繰り返しの本体を実行し終えた後で、再び不変条件が成立している。3つ目は、条件式が不成立の時は、繰り返しを脱出し、A1に実行が移る。

5) 手続き呼び出し文は、呼び出される手続きの事前条件と事後条件が与えられていれば、手続きの実行部はどのようなものでも検証できることを示している。

以上の推論規則を用いて生成された検証条件は、いくつかの戦略を持った自然演繹システムを用いて証明する。ここでは、ここで用いている戦略について説明する。

① 比較式の結合

前件における ある比較式の非定数項と後件における比較式の非定数項が等しい場合、後件の比較式の否定をとり前件に移し、2つの比較式を結合する。このことにより、比較式により限定される範囲をせばめることができる。

② 代入

前件の中に等式があり、その等式の非定数項が次の条件を満たす時、行なわれる。

- 1つの変数よりなる項がある。(この変数をxとする)
- その変数が他の項に含まれない。

この条件を満たす時、この等式をx=Eの形に変形し、他の比較式に現われ

るxをEに置換する。

③ 全称作用素に対する戦略

全称作用素を含む論理式が最も多く現われるのは、繰り返しの不変条件であり、本システムで対象としている式は、次の2式である。

- all j (j<i ⊃ P(j)), P(i) ⟶ all j (j<i+1 ⊃ P(j))

この式は、算術式kでjを置換することにより、次式に変換する。

(k<i ⊃ P(k)), P(i) ⟶ k<i+1 ⊃ P(k)

- all j (P(j) ⊃ Q(j)) ⟶ Q(f(j))

この式は、jをf(j)で置換することにより、次式に変換する。

P(f(j)) ⊃ Q(f(j)) ⟶ Q(f(j))

④ 推論規則を補なう場合

システムの持つ推論規則の不足により、証明できない式があった場合、次の命令により推論規則を補う。証明できない式を Γ → P(E), Δ とする時、

infer P(x) from Q(x)

というプログラマからの入力により、証明できない式を

all x (Q(x) ⊃ P(x)) → P(E), Δ

に変換する。

3.3 既存のプログラムの検索

編集部では、設計仕様の中の事後条件を鍵として、既存のプログラムを検索する。検索の対象としている事後条件は、検索のための費用が高くならないように不変条件及び脱出条件の形式で記述されているものと制限する。

検索のための手順は、次のものである。

① 不変条件と脱出条件を正規化する。ここでの正規化とは、比較式に対する正規化と論理式に対する簡単化である。

② 事後条件を記述するために使用されている変数の個数を検査し、等しいものは、LI ≡ LI' & EC ≡ EC' という式を証明する。もし証明されたなら、設計仕様が等しいものとし、プログラムテキスト中に挿入する。ここで LIとECは、それぞれ不変条件と脱出条件であり、「'」がついているものは既存のプログラムのものである。

また、LI, EC と LI', EC' は当然同じ変数で記述されているとは限らず、変数の置き換えを行なっている。この時、不変条件を記述した時に得られている情報（ある変数が繰り返しの中で変数か定数かという情報）も利用し、組み合せの数を減少させている。

4. プログラミングの例

「大きさnの配列aを並び換える」というソーティングの問題を例にプログラミングの方法を説明する。

プログラマは、まずnewというコマンドを実行し、トップレベルの手続きの宣言部を入力する（図5参照）。ここでsearchコマンドにより、既存のプログラムを検索し、もし存在すればプログラミングはここで終了する。

存在しなければ、この設計仕様に対応した実行部をeditコマンドにより、入力する（図6参照）。

この中には、minという未定義な名前があるがこのような名前はこれより下位の段階で実現される手続きとみなされる。

次にminという手続きを実現するわけだが、トップレベル以外の手続きを入力する場合、プログラマには次の2つの選択の余地がある。

① procedure宣言より入力することにより、手続きとして作成する。

② procedure宣言以外の宣言から入力することにより、上位レベルの手続き呼び出しの所に、これから入力する実行部を埋め込む。

図7は、①の方法を用いてminに対する設計仕様を入力した場合である。

ここまで入力した時に、upコマンドにより上位レベルにもどると、そのレベルにおける正当性の検証が可能になる。検証にはverifyというコマンドを用いる。

次にminの実行部を作成するわけだが、さきほどと同じようにまず検索し、もし存在しなければ自分で入力する。このことを繰り返えすことによりプログラムを作成していく。

```
program sort(input, output);
  const  n = 100;
  var  i: integer;
       a: array[1..n] of integer;
  define sortp(i): all j (1<=j & j<i : a[j]<=a[j+1]);
  pre    n > 1;
  post   sortp(i) & i=n;
```

図5　sortの宣言部

```
begin
  i := 1;
  while i <> n do
    begin
      min;
      i := i+1
    end
end.
```

図6　sortの実行部

```
procedure min
  var j: integer;
  define minp(j): all k (i<=k & k<=j : a[i] <= a[k]);
  pre  i >= 1;
  post minp(j) & j=n;
```

図7　minの宣言部

```
begin
  j := i
  while j <> n do
    begin
      exchange; j := j+1
    end
end.
```

図8　minの実行部

5 おわりに

本システムの試作を通しての検討事項をまとめる。

プログラマは、プログラムを作成していく場合、各段階において設計仕様を記述しなければならない。しかし、設計仕様を記述することは、それに対応するプログラムにおけるプログラム変数の数が多いほど困難になる。そのことにより プログラマは、変数の数を必要最少限に抑える傾向にある。

このことにより、プログラムを効率的にするための種々のテクニックが使えなくなり、結果として非効率的なプログラムが作成されてしまう。

また、本システムを使用してプログラムを作成していく過程で、設計仕様を記述することの難易度と作成されるプログラムの階層構造に次のような関係がある場合が多いということが分かった。すなわち、設計仕様はプログラムの階層構造が縦方向に深くなる場合は容易に記述できる。しかし、横方向に広がる場合は、難かしい。

本システムは、信頼性の高いプログラムを作成するということにおいて一応成功した。このシステムで扱われた問題は、設計仕様が記述できるという小さなプログラムだけであったが、より大きなものに対してもこの方法が応用できると考えられる。

本システムは、段階的にプログラムを作成するという方法論に立脚し、プログラム作成を支援したが、ここで提案した方法によって、検証を段階ごとに行なって、システムにとっては検証のための費用が従来より低くなった。また、正当性が検証されている既存のプログラムを再利用できるので、プログラム作成の労力が軽減された。

ただし、仕様の記述法については、より実用的な使いやすい記法の研究が必要である。

参考文献

[Bo 81] Boyer, R. S. and Moore, J. S. : " The Correctness Problem in Computer Science", Academic Press, (1981).

[Da 82] Dannenberg, R. B. and Ernst, G. W. : " Formal Program Verification Using Symbolic Execution " : IEEE Trans. Softw. Eng., Vol. SE-8, No. 1, (Jan. 1982).

[Di 76] Dijkstra, E. W. : " A Discipline of Programming ", Prentice-Hall, (1976).

[Ha 81] Habermann, A. N. : " System Development Environment", Technical Report, Carmegie-Mellon University, (Dec. 1981).

[Ja 75] Jackson, M. A. : " Principles of Program Design ", Academic Press, (1975).

[Na 80] Nagata, M., Akiyama, T., and Fujikake, Y. : " An Interactive Support System for Functional Recursive Programming", Proc. IFIP 80, (1980).

[Wa 82] Water, R. C. : " The Programmer's Apprentice : Knowledge Based Program Editing", IEEE Trans. Softw. Eng., Vol. SE-8, No. 1, (Jan. 1982).