帰納的推論による並行型プログラムの合成

寺本昌弘　　志村正道
（東京工業大学　工学部）

1．はじめに

帰納的推論によってプログラムの自動合成を行う研究には、

1)プログラムの入力リストと出力リストの複数組を仕様として、それらすべての入出力関係を満足するＬＩＳＰプログラムを合成する、

2)プログラムが実際どのような計算を行うのかを示す例を仕様として、その例の通りに実行が行われるプログラムを合成する、

というふたつのタイプがある。

タイプ2）の研究を代表するものに、Barzdin[1],Bauer[2]があるが、彼らの研究にはそれぞれ以下の理由によって、仕様としてシステムに与えるプログラムの実行例の作成が困難であるという欠点がある。

- Bauer の研究の場合、プログラムの実行例が特定の入力に対するものであるにもかかわらず、抽象的な思考によって実行例の中で多くの変数を使用しなければならない。
- Balzdin の研究の場合、プログラムの実行例がプログラムの展開形そのものの一部分であり、さらにプログラムの分岐構造をも明記しなければならない。

また、彼らの研究において合成されるプログラムはいずれも逐次型プログラムである。本来、人間の思考が完全に逐次的なものではなく、並列性も許容していることを考えると、非逐次型のプログラムの合成も逐次型のプログラムの合成と同様に試みられるべきである。

以上の２点をふまえて本稿では、並行型プログラムの実行例となるようなプロセスごとの具体的な処理手順の集合を仕様として、それから帰納的推論によって並行型プログラムを合成する方法について述べる。合成される並行型プログラムは、Hoare[3]によって提案されたCommunicating Sequential Processes（以下ＣＳＰと略記する）で記述される。

本稿で提示する方法では、仕様記述からＣＳＰのプログラムを合成する過程において、ペトリネット［4］を中間表現として用いている。ペトリネットを用いることによって、仕様記述の意味内容および合成の途中段階にあるプログラムの制御構造が明確に表現されるほか、仕様記述の中にデッドロックが潜在するか否かの判定が容易になされる。

図１にプログラム合成の流れを示す。

2．プログラムの記述

並行型プログラムの仕様記述ＣＳＰＥＣは次のふたつ組によって定義される。

CSPEC=<GNAME,ELEMS>
　GNAME:プログラムの名前
　ELEMS=<ELEM1, ‥, ELEMn>
　　ELEMi=<CNAMEi,SEQi>　(1≦ｉ≦n)
　　CNAMEi: プロセスの名前

nは仕様記述者がプログラムの実行時に発生することを意図している並行型プロセスの数であり、SEQiはｉ番目のプロセスの中で実行される処理文が実行順に並べられた有限列である。処理文には代入処理文，入力処理文，出力処理文，論理処理文の４種類があり、処理の対象は整数値，論理値，信号に限定される。

代入処理文は、単純な変数や配列に値または信号を割り当てる処理を意味する。入力処理文は、他のあるプロセスから値または信号を入力する処理を意味する。出力処理文は、他のあるプロセスへ値または信号を出力する処理を意味する。論理処理文はその直後の処理文が実行される根拠を示しており、それはふたつの対象の比較結果またはある変数の値が真である場合に限られる。なお入出力処理文の場合には、それぞれその処理文と対応する出入力処理文が

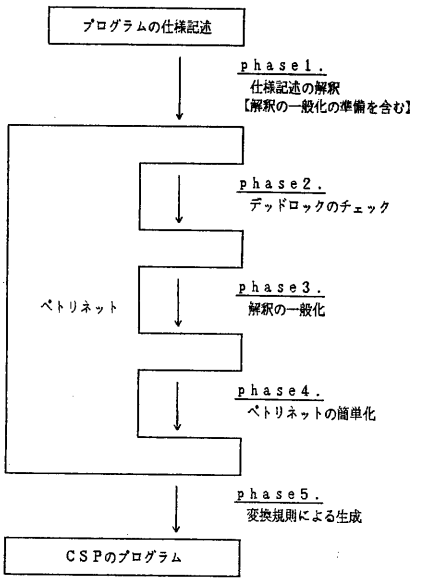


図1　プログラム合成の流れ

記述されていなければならない。ただし、ふたつの入出力処理文が対応するとは、互いに相手の処理文が属しているプロセスの名前を入出力先として明記しており、かつ一方の入力値と他方の出力値の型が適合することをいう。対応する入出力処理文の記述は、それらが同時に実行されることを暗黙的に記述しているものとする。

図２は、ふたつの正整数５４と２４の最大公約数をユークリッドの互除法を用いて求めるプログラムの仕様記述例である。GCM(0)，GCM(1)，GCM(2)はプログラムの実行時に発生する並行型プロセスである。図２の中では、たとえば次のふたつの処理文が対応する入出力処理文である。

GCM(2) KARA 6 WO UKETORU.　in GCM(0)
GCM(0) NI 6 WO OKURU.　　in GCM(2)

```
PROGRAM NO NAMAE WA 'GCM DEARU.
GCM(0) :: GCM(1) NI (54,24) WO OKURU.
          GCM(2) KARA 6 WO UKETORU..
GCM(1) :: GCM(0) KARA (54,24) WO UKETORU.
          %X NI 54-24 WO DAINYUUSURU.
          30>24 DEARU.
          %X NI 30-24 WO DAINYUUSURU.
          6<24 DEARU.
          GCM(2) NI (24,6) WO OKURU..
GCM(2) :: GCM(1) KARA (24,6) WO UKETORU.
          %X NI 24-6 WO DAINYUUSURU.
          18>6 DEARU.
          %X NI 18-6 WO DAINYUUSURU.
          12>6 DEARU.
          %X NI 12-6 WO DAINYUUSURU.
          6=6 DEARU.
          GCM(0) NI 6 WO OKURU..
OWARI.
```

図２　54と24の最大公約数を求めるプログラムの仕様記述

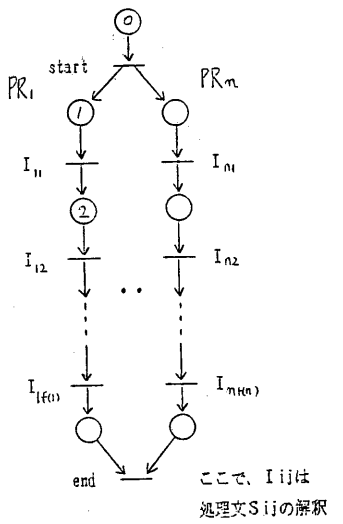


図３　仕様記述から構成されるペトリネットの一般形

３．仕様記述の解釈と一般化

３．１　ペトリネットの構成

仕様記述の中に記述されているすべての並行型プロセスをＰＲ1,‥，ＰＲn とし、各ＰＲi の中の処理文を記述順にＳ$_{i1}$，‥，Ｓ$_{if(i)}$とするとき、仕様記述に対応して図３のようにペトリネットが構成される。それぞれの処理文はその種類に応じて解釈され、次の内部表現に変換される。

1.代入処理文の内部表現
（文の名札　　ＡＳＳ
（被代入要素の構文的表現　被代入要素の意味的表現）
（代入要素の構文的表現　代入要素の意味的表現））
2.入力処理文の内部表現
（文の名札　　ＩＮ　　ソース
入力値の構文的表現　入力値の意味的表現）
3.出力処理文の内部表現
（文の名札　　ＯＵＴ　　デスティネーション
出力値の構文的表現　　出力値の意味的表現）
4.論理処理文の内部表現
（文の名札　　ＬＯＧ
論理条件の構文的表現　　論理条件の評価値）

ここで文の名札は、処理文の内部表現が意味として与えられているトランジションを同定するもので、トランジションの入力プレイスの番号と出力プレイスの番号との対である。構文的表現とは仕様記述で書かれた表現そのものであり、意味的表現とは構文的表現の中の評価可能な部分を評価した表現である。代入処理文の内部表現が作られるとき、被代入要素である変数と代入要素である整数値あるいは論理値とが、その代入処理文が含まれているプロセスの中で結びつけられる。図２の仕様記述が解釈された結果、図４に示すペトリネットの内部表現が構成される。

３．２　解釈の一般化の準備

解釈の一般化の準備では　ペトリネットの内部表現に対して次の４つの処理が行われる。

```
(("GCM(0)" ((1 2) OUT (GCM 1) (NIL 54 24) (NIL 54 24))
           ((2 3) IN (GCM 2) 6 6))
 ("GCM(1)" ((4 5) IN (GCM 0) (NIL 54 24) (NIL 54 24))
           ((5 6) ASS (%X NIL) ((- 54 24) 30))
           ((6 7) LOG (> 30 24) T)
           ((7 8) ASS (%X NIL) ((- 30 24) 6))
           ((8 9) LOG (< 6 24) T)
           ((9 10) OUT (GCM 2) (NIL 24 6) (NIL 24 6)))
 ("GCM(2)" ((11 12) IN (GCM 1) (NIL 24 6) (NIL 24 6))
           ((12 13) ASS (%X NIL) ((- 24 6) 18))
           ((13 14) LOG (> 18 6) T)
           ((14 15) ASS (%X NIL) ((- 18 6) 12))
           ((15 16) LOG (> 12 6) T)
           ((16 17) ASS (%X NIL) ((- 12 6) 6))
           ((17 18) LOG (= 6 6) T)
           ((18 19) OUT (GCM 0) 6 6)))
```

図４　仕様記述（図２）から構成されるペトリネット

1) すべての入力値に対してそれぞれ新たな変数を作り、入力処理文が含まれているプロセスの中でその変数と入力値とを結びつける。この変数のことを入力変数という。

2) 配列型のプロセスがあるときに、そのプロセスの中でプロセスの添字と変数＊ＩＮＤＥＸとを結びつける。もし、そのプロセスと添字だけが異なるプロセスをソースあるいはデスティネーションとする入出力処理文があれば、そのソースあるいはデスティネーションの添字を＊ＩＮＤＥＸを使って表す。

3) プロセスの添字でないすべての定数Ｃに対して、Ｃと結びつけられている変数Ｖ1,‥，Ｖn を探索し、Ｃをリスト（？　Ｃ　Ｖ1 ‥Ｖn）で置き換える。このリストのことを定数Ｃの本質候補リストと呼ぶ。なお、Ｖ1,‥，Ｖn の順序は、それぞれがＣと結びつけられた順序と逆の順序とする。定数の本質候補リストは、後に行われる解釈の一般化によってその中のひとつの要素が選択され、その結果仕様記述の中の定数の意味が決定する。

4) 二項演算の部分に関して、一方のオペランドが本質候補リストであり他方のオペランドが定数であるときには、本質候補リストの中の定数を取り除く。これは、プログラムの中で単なるふたつの定数に二項演算を適用することは意味がないと考えるからである。

図４のペトリネットに対して解釈の一般化の準備を行った結果を図５に示す。

３．３　デッドロックのチェック

デッドロックのチェックを行う準備としてまず、すべての入出力処理文に対し、それと対応する処理文を探索して組を作る。もし、どの入出力処理文とも対応しない入出力処理文があるときには、仕様記述が不正であったということなのでプログラムの合成は中止される。対応する入出力処理文の組が作られると次に、それぞれの組が同時に実行されるように同期がとられる。この方法を図６に示す。

上記の準備が整ったペトリネットに対して、以下の手順でデッドロックが存在するか否かを判定する。

1.start プレイス（番号が０のプレイス）にトークンをひとつ割り当てる。

2.発火可能なトランジションがある限り、その中の任意のものを発火させることを繰り返す。

3.もし、ペトリネットの中にトークンが無ければデッドロックは存在しないと判定し、そうでなければデッドロックが存在すると判定する。

このアルゴリズムが正しいことを保証するのは次の２点である。

- 先に述べた方法で構成されるペトリネットはマークドグラフである。即ち、すべてのプレイスはただひとつの入力トランジションとただひとつの出力トランジションとを持つ。従って、任意のふたつ以上のトランジションが発火の際に衝突することはないので、発火可能なトランジションの発火の仕方はひと通りである。
- ペトリネットの構成方法によってどのプレイスもトークンを発生するのは１回限りであり、しかも１個しか発生しない。従って、発火可能なトランジションの発火が繰り返されれば、最終的にペトリネットの中のトークンは必ずなくなる。

３．４　解釈の一般化

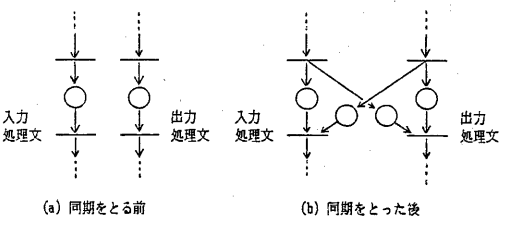
解釈の一般化では、すべての本質候補リストに対してその中のひとつの要素Ｅを選択して、そのリストをＥで置き換えることを行う。本質候補リストを含む処理文を未定処理文と呼び、未定処理文が本質候補リストを含まないようにすることをその未定処理文を確定するという。未定処理文の確定方法を述べる前に、処理文どうしのマッチングの成否を次のように定義する。

【定義】（処理文のマッチングの成否）

文の名札以外は内部表現の中の対応する部分が同じであるふたつの処理文のマッチングは成功する。また、内部表現の中の対応する部分がたとえ同じで

```
(((GCM 0)
  ((1 2) OUT (GCM (+ *INDEX 1)) (NIL 54 24))
  ((2 3) IN (GCM (+ *INDEX 2)) @VAR1))
 ((GCM 1)
  ((4 5) IN (GCM (- *INDEX 1)) (NIL @VAR2 @VAR3))
  ((5 6) ASS %X (- (? 54 @VAR2) (? 24 @VAR3)))
  ((6 7) LOG (> (? 30 %X) (? 24 @VAR3)))
  ((7 8) ASS %X (- (? 30 %X) (? 24 @VAR3)))
  ((8 9) LOG (< (? 6 %X) (? 24 @VAR3)))
  ((9 10)
   OUT
    (GCM (+ *INDEX 1))
    (NIL (? 24 @VAR3) (? 6 %X))))
 ((GCM 2)
  ((11 12) IN (GCM (- *INDEX 1)) (NIL @VAR4 @VAR5))
  ((12 13) ASS %X (- (? 24 @VAR4) (? 6 @VAR5)))
  ((13 14) LOG (> (? 18 %X) (? 6 @VAR5)))
  ((14 15) ASS %X (- (? 18 %X) (? 6 @VAR5)))
  ((15 16) LOG (> (? 12 %X) (? 6 @VAR5)))
  ((16 17) ASS %X (- (? 12 %X) (? 6 @VAR5)))
  ((17 18) LOG (= (? 6 %X @VAR5) (? 6 %X @VAR5)))
  ((18 19) OUT (GCM (- *INDEX 2)) (? 6 %X @VAR5))))
```

図５　ペトリネット（図４）に対して一般化の準備を行った結果



(a) 同期をとる前　　(b) 同期をとった後

図６　対応する入出力処理文の同期をとる方法

なくても次の6つの場合にはマッチングは成功する。

1. オペレータが可換（＋，＊，＝）、あるいはオペレータが逆向きである('<'と'>','<='と'>=')ような二項演算において、オペランドの順序を逆にしたら同じになる。
2. オペレータが'>'と'='または'<'と'='または'='と'/='である二項演算で、オペランドどうしは同じである。
3. 一方が定数Cまたは変数Vであり、他方が本質候補リストLであるとき、CあるいはVがLの中に含まれている。
4. ともに入力変数であり、他のいかなる入力変数とも対応づけられていない。
5. 一方が入力変数IVであり、他方が本質候補リストLであるとき、IVが他のいかなる本質候補リストとも対応づけられておらず、かつLの中にそれと対応づけられている入力変数が存在しない入力変数がある。
6. ともに本質候補リストであり、一方の本質候補リストの中の少くともひとつの要素が、先の3または5の条件を満足する。

```
BEGIN
  同一プロセス内のふたつの処理系列のマッチングによる精錬；          ・・・[I]
IF  未定処理文が確定しない
  THEN
    BEGIN
      相異なるプロセス内のふたつの処理系列のマッチングによる精錬；    ・・・[II]
      IF  未定処理文が確定しない
      THEN
        BEGIN
          同一プロセス内のふたつの処理文のマッチングによる精錬；      ・・・[III]
          IF  未定処理文が確定しない
          THEN
            BEGIN
              相異なるプロセス内のふたつの処理文のマッチングによる精錬；  ・・・[IV]
              IF  未定処理文が確定しない
                THEN  強制的な確定      ・・・[V]
              END
        END
    END
END
```

図7　未定処理文を確定するアルゴリズム

```
(((GCM 0)
  ((0 1) START)
  ((1 2) OUT (GCM (+ *INDEX 1)) (NIL 54 24))
  ((2 3) IN (GCM (+ *INDEX 2)) @VAR1)
  ((3 NIL) END))
 ((GCM 1)
  ((0 4) START)
  ((4 5) IN (GCM (- *INDEX 1)) (NIL @VAR2 @VAR3))
  ((5 6) ASS %X (- @VAR2 @VAR3))
  ((6 7) LOG (> %X @VAR3))
  ((7 8) ASS %X (- %X @VAR3))
  ((8 9) LOG (< %X @VAR3))
  ((9 10) OUT (GCM (+ *INDEX 1)) (NIL @VAR3 %X))
  ((10 NIL) END))
 ((GCM 2)
  ((0 11) START)
  ((11 12) IN (GCM (- *INDEX 1)) (NIL @VAR4 @VAR5))
  ((12 13) ASS %X (- @VAR4 @VAR5))
  ((13 14) LOG (> %X @VAR5))
  ((14 15) ASS %X (- %X @VAR5))
  ((15 16) LOG (> %X @VAR5))
  ((16 17) ASS %X (- %X @VAR5))
  ((17 18) LOG (= %X @VAR5))
  ((18 19) OUT (GCM (- *INDEX 2)) %X)
  ((19 NIL) END)))
```

図8　ペトリネット（図5）に対して一般化を行った結果

未定処理文の確定は、同じ形の文を作ることを基本的な方針として、処理文どうしのマッチングによって行われる。確定対象である未定処理文と、ある処理文とのマッチングが成功すると、未定処理文の中にある本質候補リストの要素から、マッチングを失敗させるものが取り除かれる。このようにして、未定処理文の中の本質候補リストの要素を絞ることを、未定処理文の精錬という。

未定処理文を確定するアルゴリズムを図7に示す。図7の中の[I]および[III]は、同じ形の処理文のみならず同じ形の処理系列を作ることを目的として行われる。また、マッチングの対象となる処理系列は次のふたつの条件を満足するものである。

- 確定したい未定処理文Sが含まれているプロセスの中にあって、入力処理文または論理処理文を先頭とし、Sを最後とする。
- 処理系列の先頭と最後以外の文は、代入処理文または出力処理文である。ただし、Sが論理処理文の場合にはSとのマッチングが成功しない論理処理文が含まれていてもよい。

図7の中の[V]は、本質候補リストの要素がひとつに絞りきられなかったときに行われる。本稿の方法では、その時点で残っているリストの要素の中で、定数と結ぶつけられたのが最も遅い変数を採択している。その理由は次の通りである。「自然言語の文章の場合、ある文中の指示代名詞が指示し得る対象が複数存在するときには，われわれは通常その指示代名詞に最も近い対象を指示すると考える。仕様記述の中の定数は、仕様記述者がある変数を指示して書いた代名詞的なものであると考えることができる。」

図5のペトリネットに対して、解釈の一般化を行った結果を図8に示す。

4．ペトリネットの簡単化

ペトリネットの簡単化は、

1)プロセスレベスの簡単化

2)プログラムレベルの簡単化

の2段階で行われる。その際、次に定義する処理文の相似性が使用される。

【定義】（ふたつの処理文の相似性）

1. タイプIで相似であるための条件
   - 同じ代入処理文である
   - 同じプロセスをデスティネーションとし、かつ対応する入力変数の違いを除いては同じ出力値を持

つような出力処理文である
- 同じプロセスをソースとし、かつ対応する入力変数の違いを除いては同じ入力値を持つような入力処理文である
- 対応する入力変数の違いを除いて同じ論理処理文である

2.タイプIIで相似であるための条件
- 添字だけが異なる配列型のプロセスをソースとし、かつ対応する入力変数の違いを除いては同じ入力値を持つような入力処理文である
- 同じプロセスをソースとし、かつ対応する入力変数の違いを除いても入力値が異なるような入力処理文である
- 論理条件の関係子だけ（たとえば、'<' と'='）が異なり、それ以外は対応する入力変数の違いを除いて同じであるような論理処理文である

3.相似でないための条件
- 上記以外の処理文である

4．1　プロセスレベルの簡単化

プロセスレベルの簡単化は、プログラムの構成要素であるそれぞれのプロセスを表すペトリネットの部分に対して独立に行われる。プロセスごとの部分ペトリネットの構造は、簡単化を行う前は直列である。プロセスレベルの簡単化では、この直列構造に対して
- 対応する処理文がいずれもタイプIで相似であるようなふたつの処理系列を検出してループ構造を生成する
- タイプIIで相似であるようなふたつの処理文を検出して分岐構造を生成する

等の操作を試みる。

プロセスレベルの簡単化の特徴は、次の5点である。

1. ループ構造の開始点および分岐構造の分岐点は、ともに入力処理文または論理処理文とし、いずれか一方に優先権を与えることはしない。
2. ループ構造の生成を分岐構造の生成よりもに優先的に行う。

```
(((GCM 0)
  ((0 1) START)
  ((1 2) OUT (GCM (+ *INDEX 1)) (NIL 54 24))
  ((2 3) IN (GCM (+ *INDEX 2)) @VAR1)
  ((3 NIL) END))
 ((GCM (*INDEX 1 2))
  ((0 4) START)
  ((4 5) IN (GCM (- *INDEX 1)) (NIL @VAR2 @VAR3))
  ((5 6) ASS %X (- @VAR2 @VAR3))
  ((6 7) LOG (> %X @VAR3))
  ((7 6) ASS %X (- %X @VAR3))
  ((6 20) LOG (< %X @VAR3))
  ((20 21) OUT (GCM (+ *INDEX 1)) (NIL @VAR3 %X))
  ((21 NIL) END)
  ((6 22) LOG (= %X @VAR3))
  ((22 23) OUT (GCM 0) %X)
  ((23 NIL) END)))
```

図9　ペトリネット（図8）に対して簡単化を行った結果

3. ループ構造は、大きさの小さなものから順に生成を試みる。
4. ループ構造の開始点ができるだけ処理系列の先頭に近くなるように、ループ構造の生成を試みる。
5. 簡単化の過程において生成されるループ構造それ自体は直列構造であり、その構造に対してプロセスレベルの簡単化が再帰的に行われる。

4．2　プログラムレベルの簡単化

プログラムレベルの簡単化は、相異なるふたつのプロセスを表す部分ペトリネットの吸収と融合によって行われる。

部分ペトリネットの吸収とは、ふたつの部分ペトリネットPN1とPN2が次のふたつの条件を満足するとき、PN2の処理内容をPN1の処理内容で代用することである。
- PN1の処理内容がPN2の処理内容を含んでいる。
- PN2のトランジションの中でend トランジションの直前に位置するトランジションがすべて、PN1においてもend トランジションの直前に位置している。

部分ペトリネットの融合とは、部分的に異なる処理内容を持つふたつの部分ペトリネットPN1とPN2から、PN1とPN2の両方の処理内容を持つ部分ペトリネットを生成することである。

図8のペトリネットに対して簡単化を行った結果を図9に示す。

5．ペトリネットからCSPプログラムへの変換

CSPのプログラムは、最終的に次の手順でペトリネットからひと通りに合成される。

1) プロセスを表すそれぞれの部分ペトリネットに対してその中の部分構造に以下の[1] から[4] の変換規則を繰り返し適用することによって、プロセスごとのCSPの部分プログラムが合成される。
2) 仕様記述の中に書かれたプログラムの名前をNAMEとし1)で合成されたプロセスごとの部分プログラムをそれぞれP1,‥ ,Pn とするとき、

NAME=[P1 ‖ P2 ‖‥‖ Pn]

なるCSPのプログラムを生成する。

【ペトリネットからCSPへの変換規則】

以下、A',Ai',B'はそれぞれ部分ペトリネットA,Ai,Bから生成されるCSPの部分プログラムである。

[1].直列文の生成規則

図10(a) の部分ペトリネットに対しては、

a ; A'

なるCSPのプログラムが生成される。

[2].代替文の生成規則

図10(b),図10(c) の部分ペトリネットに対しては、

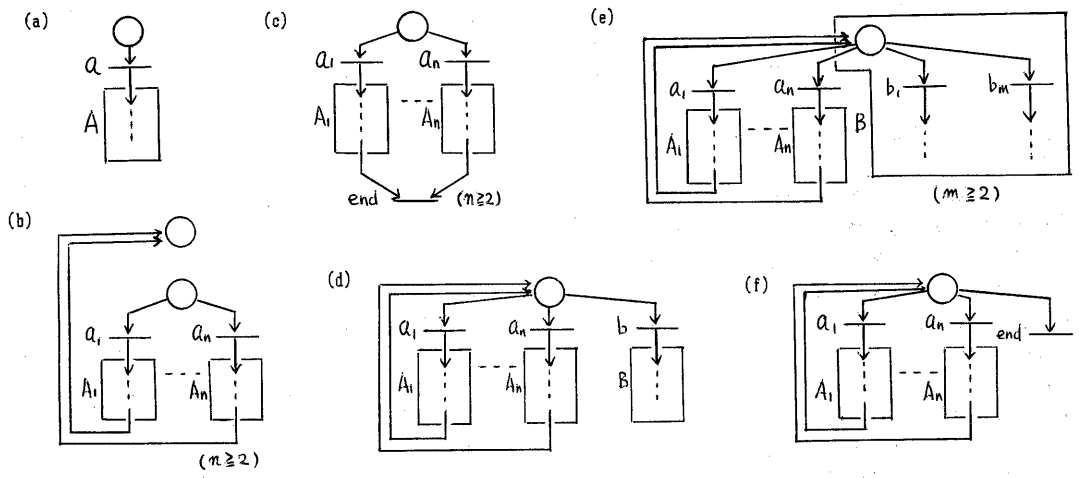


図１０　ＣＳＰへ変換されるペトリネットの構造単位

[a1 → A1'□ a2 → A2'□…□ an → An']

なるＣＳＰのプログラムが生成される。もしあるAiの中にトランジションがなければ、Ai'をskip文をする。これは次の[3],[4] においても同様である。

[3].論理式を防護とする繰り返し文の生成規則

図10(d),図10(e) の部分ペトリネットに対しては、

＊[a1 → A1'□ a2 → A2'□…□ an → An']; B'

なるＣＳＰのプログラムが生成される。ここで、aiとb とbiはいずれも論理処理文である。

[4].入力文を防護とする繰り返し文の生成規則

図10(f) の部分ペトリネットに対しては、

＊[a1 → A1'□ a2 → A2'□…□ an → An']

なるＣＳＰのプログラムが生成される。ここで、aiはいずれも入力処理文である。

図９のペトリネットから合成されるＣＳＰのプログラムを図１１に示す。

６．おわりに

プログラムの実行例となるようなプロセスごとの具体的な処理手順の集合から、ＣＳＰで記述された並行型プログラムを帰納的推論によって合成する方法について述べた。

合成に際しては、何を行うプログラムを合成するのかについての情報は全く用いていない。仕様としてシステムに与えられた具体的な例の通りに実行が行われるようなプログラムを合成することを目標とした。

これまで、本稿で述べた方法によって

- プロセスごとに再帰的な処理が行われるプログラム
- ひとつのプロセスの中の、guard(防護) の入れ子の深さが浅いプログラム

の合成が可能であることが分った。

今後はタイプの異なるプログラムの合成を試みるとともに、次の課題点を検討してゆきたいと思う。

1. 階層的な仕様記述を導入することによって、ひとつのプロセスの処理が大規模あるいは複雑であるプログラムの仕様を与えやすくすること
2. システムに複数の具体例を仕様として与え、より一般的なプログラムを合成できるようにすること
3. 変数や命令に関する知識を利用することによって、仕様記述に対して柔軟に対応することができるようにすること

```
GCM=[ GCM(0) :: GCM(1)!(54,24);
                GCM(2)?#1
    ||GCM(I:1..2) :: GCM(I-1)?(#2,#3);
                     %X:=#2-#3;
                    *[%X>#3 -> %X:=%X-#3];
                     [%X<#3 -> GCM(I+1)!(#3,%X)
                     □%X=#3 -> GCM(0)!%X]]
```

図１１　ペトリネット（図9）から合成されるＣＳＰのプログラム

参考文献

[1] Barzdin, J. M. : On inductive synthesis of programs, Lecture Notes in Computer Science, Vol.122, pp.235-254, 1979.

[2] Bauer, M. A. : Programming by examples, Artificial Intelligence,Vol.12,pp.1-21,1979.

[3] Hoare, C. A. R. : Communicating sequential processes, Comm. ACM, Vol.21, No.8, pp.666-677, 1978.

[4] Peterson, J. L. : Petri net theory and the modeling of systems, Prentice-Hall, 1981.