プログラミング言語 15－2
（１９８８．２．１９）

ストリーム・プログラムのインライン展開

久世 和資
日本ＩＢＭ 東京基礎研究所

データの流れであるストリームを使ってプログラミングするための言語として設計，開発されたのがStellaである．Stellaで記述されたプログラムは，複数のモジュールと，それらを結合するストリームで構成される．プログラムの実行はストリームを介して通信しながら各モジュールが並列に動作すると考えられる．
Stellaの処理方式の１つにインライン展開がある．インライン展開はStellaで記述された並列プログラムを逐次プログラムに展開する手法である．インライン展開することにより，単一プロセッサ上で最大の実行効率を得ることができる．本報告では，ペトリネットを用いて，最小コード長の展開結果を求めることができるインライン展開の方式について述べる．

THE IN-LINE EXPANSION OF STREAM PROGRAMS

Kazushi KUSE

Tokyo Research Laboratory, IBM Japan Ltd.
5-19, Sanbancho, Chiyoda-ku, Tokyo 102

Stella is a programming language for stream programming. A Stella program is a network of concurrent processes connected by streams.

In-line expansion is a program transformation of stream-connected concurrent processes into a sequential process. It is an interesting method of implementation realizing maximum run-time efficiency. In this paper, I present a method using the marking graph of a Petri net which realizes optimal in-line expanded code. I also developed the automatic analysis and tansformation system and transformed some Stella programs using it.

1．はじめに

ソフトウェアの生産性向上のためには，プログラムの記述性と読み易さの改善，および，プログラムの部品化と再利用の促進が挙げられる．これらの観点から，ストリームを扱う機能を導入したプログラミング言語として設計されたのがStella[13)19)21)]である．

ストリームは，無限のデータ列であり，各要素の値の評価は，それが必要になるまで延期できる．これまで，関数型言語[2)8)]，データフロー言語[1)4)]，論理型言語[3)]などにおいては，ストリームに関する研究がされている．Stellaでは，汎用言語にストリームを扱う機能を導入することにより，より実用的なストリームプログラム[17)]の作成を可能した．

Stellaプログラムは，複数のモジュールとそれらを結ぶ複数のストリームで構成される．実行時には，ストリームを通してデータを流しながら，各モジュールが並列に動作すると考えられる．Stellaは，非同期方式の並行プロセス系であり，同期方式の並行プロセス系[6)9)14)]に比べて，デッドロックが起こりにくいなどの利点がある[12)]．

Stellaの特徴は，データの流れにしたがって，アルゴリズムを素直に記述できることである．この種の問題は事務処理計算に多く，特に順序ファイルなどは，ストリームと見なせる．ジャクソン法[10)]におけるデータ構造の不一致問題も，中間的なストリームを用意することによって解決できる．アルゴリズムを素直に記述できる例としては，ライニング問題[7)]，ハミング問題[5)]，フィボナッチ数列，素数列[13)]などがある．

さらに，各モジュール間の通信がストリームに限定されるため，モジュラリティが高く，プログラムの部品化，再利用に適している．部品であるモジュールを手軽につなぎ合わせてプログラムを作成することができる．プログラムのテストやデバッグも，流れるデータを観察できる機能を用意すれば，容易に行える[18)]．

Stellaの処理系には，ストリームに対してバッファを用意し，各モジュールをコルーチンにより実行する擬似並列処理系[16)]がある．この処理系は，使用上の制限が少なく適用範囲は広いが，制御の切替とバッファを介した通信のオーバーヘッドのために，実行効率が最適ではない．

ストリームを用いたプログラムの単一プロセッサ上での最適な実行効率による処理方式としてインライン展開が提案された[20)21)]．インライン展開は，複数の並列動作プロセスを１つの逐次プロセスに展開する手法である．展開することにより，制御の切り替えや通信のためのオーバーヘッドが除かれるので，プログラムを高速に実行することができる．

本稿では，ペトリネットを用いたインライン展開の方式について述べる．本方式では，まず，ストリーム・プログラムをペトリネットでモデル化する．モデル化したペトリネットから元のプログラムの全ての動作を表したマーキンググラフを生成する．インライン展開はこのマーキンググラフを使用して行う．

インライン展開コードは，一般に元のプログラムのコードよりも長くなるが，本方式では，最小のインライン展開コードを求めることができる．これは，あらかじめ，マーキンググラフによりプログラムの可能な全ての動作を解析できるためである．

また，マーキンググラフを利用してインライン展開の他に擬似並列実行時に必要なバッファ長の解析，デッドロックの検出，停止性の解析なども同時に行なえる[12)]．

この方式に基づき，ストリーム・プログラムのインライン展開と解析を自動的に行うシステム[11)]を作成した．このシステムを用いて種々のプログラムのインライン展開を行いその有効性を実証することができた．本稿では，最初にStellaによるプログラミングの概要を述べた後，インライン展開の方式とその評価を述べる．

2．ストリームによるプログラミング

ストリームとは，その要素の型が同一であるデータの連続した列である．要素値の評価は，それが実際に必要となるまで延期しておくことができる[8)]．Stellaのプログラムは，複数のモジュールと，それらを結ぶ複数のストリームで構成される．ストリームの流れる方向は，一方向であり，モジュールへ入って来るものを入力ストリーム，モジュールから出て行くものを出力ストリームと呼ぶ．Stellaでは，モジュールが扱えるストリームの数に制限はなく，一般にモジュールはストリームによりネットワーク状に結合される．

各モジュールでは，入力ストリームからデータを取り出し，そのデータを使って処理し，出力ストリームにデータを送り出す．プログラムの実行とは，ストリ

ームを通してデータを流しながら，各モジュールが並列に動作することと考えられる．

ストリームを用いてプログラミングすると以下のような利点が得られる．

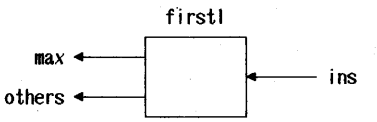
①逐次的プログラムでは表すのが困難であるコルーチン的な処理が，単純なストリームに対する通信操作によって表現できる．

②データの流れに沿ったプログラミングが容易になり，ジャクソン法[10]におけるプログラム変換などが不用になる．

③単純な処理をするモジュールをストリーム結合で組合わせて，より複雑な処理をするプログラムを作成できる．

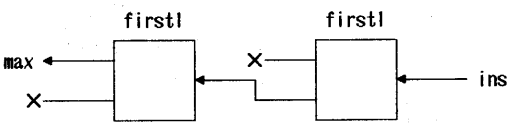
④各モジュールでは，ストリームの入力先，出力先を明示しないので，それらの結合は柔軟に行える．

⑤各モジュールは，独立して他のモジュールと並行動作すると考えることができるので，機能的に独立に記述，理解でき，情報隠蔽や部品化ができる．



整数型ストリーム ins中の最大値を maxに、
それ以外をothersに出力する

図１ モジュール firstlのストリーム図式



ins中で２番目に大きな数を maxに求める

図２ プログラム secondlのストリーム図式

ストリームによるプログラミングの簡単な例として，整数型の入力ストリームから２番目に大きい数を求めるプログラム secondlを作成する．部品として整数型の入力ストリーム ins中の最大値を maxへ，それ以外の数をothersへ出力するモジュールfirstl（図１）が使えるものとすると， secondlは，２つのfirstlを１本のストリームで結合することによって得られる（図２）．図中の記号×は，出力を他のモジュールに結合しないことを表わし，非結合ストリームと呼ぶ．このようにモジュールとストリームを，それぞれ箱と矢印で表し，モジュールの結合関係を記述した図をストリーム図式と呼ぶ．

Stellaの言語仕様は，Pascalの仕様を基本としており，Pascalの文はすべて使用できる．ここでは，前述のプログラム secondlのStellaによる記述例（図３）を用いて言語仕様の概要を述べる．Stellaプログラムは，(1)ストリーム定義部，(2)モジュール宣言部，(3)結合部の３つの部分に分類できる．以下，それぞれの部分について説明する．

（１）ストリーム定義部

ストリーム定義部は，ストリームの型を定義する部分で，

stream of 要素型

の形で任意の型が定義できる．定義した型は，各モジュールの入出力ストリームの型指定に使用する． secondlプログラムには，ストリーム定義部がないが，これは，標準ストリーム型のintegers(=stream of integer)を使用しているためである．標準ストリーム型には，整数型(integers)の他に，実数型(reals)，文字型(chars)，論理型(booleans)がある．

（２）モジュール宣言部

モジュール宣言部は，使用するモジュールを宣言する部分である．モジュールの頭部では，

module モジュール名

の後の（）内に入力ストリームを宣言し，（）の次に出力ストリームを宣言する（図３①）．入力ストリームと出力ストリームは，それぞれ，複数本使用できる．

入力ストリームから要素を１つ取り出すには，式の中に

next 入力ストリーム名

の形で記述する．図３②は，入力ストリーム insから，要素を１つ取り出し，変数 lastmaxに代入する文である．入力ストリームを生成するモジュールが終了し，要素が尽きた状態をeos(end of stream)と呼ぶ． eos状態のストリームから要素を取り出そうとすると， eos例外処理として，そのモジュール自身も終了する．したがって，文②でinsがeosなら，この文以降は実行されずにfirstlは終了する．

終了する直前に何らかの処理をしたい時には << >> 内に記述し，付加する（図3③）．文③ではinsがeosになると，その時のlastmaxを最大値としてmaxに出力し終了する．eosになっても終了させない場合は，例外処理部にgoto文等を書き，制御を他へ移す．

逆に出力ストリームへ要素を１つ送り出すには，

next　出力ストリーム名　：＝　式

の形で記述する（図3④）．出力ストリームを消費するモジュールが終了した状態をblocked(blocked stream）と呼ぶ．eos と同様に blocked状態のストリームへ要素を送り出そうとすると， blocked例外処理として，通常は，そのモジュール自身も終了する．

（3）結合部

結合部は，宣言したモジュールを，ストリームを用いて結合する部分で，connect とend の間に結合の仕方を記述したものである．２つのモジュールを結合するには，一方のモジュールの入力ストリーム部と，他方のモジュールの出力ストリーム部に，同じストリーム名を記述する．ストリーム名には，一般の引数と区別するために前に#を付ける（図3⑤）．出力ストリームを結合しない時には，非結合ストリームとしてfreeを記述する．また，ストリームを，標準入出力や外部ファイルとすることもでき，その場合は，ストリーム名の代わりに，標準入出力名またはファイル名を記述する（図3⑤）．

Stellaでは，同一のモジュールを動的に複数個生成して，ストリームで結合することもできる．firstlに，この機能を適用すると，ソーティングのプログラムが記述できる．

```
program  second1( input, output );

    module  first1( ins: integers ) max, others: integers;……①
        var  i, lastmax: integer;

        procedure  swap( var a, b: integer );
            var  c: integer;
        begin  c := a;  a := b;  b := c  end;

    begin
        lastmax := next ins;  ……②
        loop
            i := next ins
                << next max := lastmax >>;  ……③
            if  i>lastmax  then  swap( i, lastmax );
            next others := i  ……④
        end
    end;


begin
    connect
        first1( input ) free, #s;  ……⑤
        first1( #s ) output, free
    end
end.
```

図3　プログラム second1のStellaによる記述

３．インライン展開

ストリームによる通信を含む並列プログラムを逐次プログラムに展開するのがインライン展開である．これにより，単一プロセッサ上で最大の実行効率を得ることができる．

具体的には，あるモジュール中のストリームに対する出力と，別のモジュール中の同じストリームに対する入力を，１つの代入文で置き換え，それぞれの前後の処理を，その代入文の前と後ろに置く（図４）．実際には，ストリームの入出力は，ループ中や条件分岐の一方に含まれたりするので，図４のような単純な変換では済まない．たとえば， secondl（図３）の展開結果は，図５に示すとおりである．

インライン展開は単純な複数のプログラムからより複雑なプログラムを生成する手法でもあるので，プログラム合成の一種とも考えられる．

以下，インライン展開方式について，ペトリネットによるモデル化，マーキンググラフの生成，インライン展開の順に説明する．

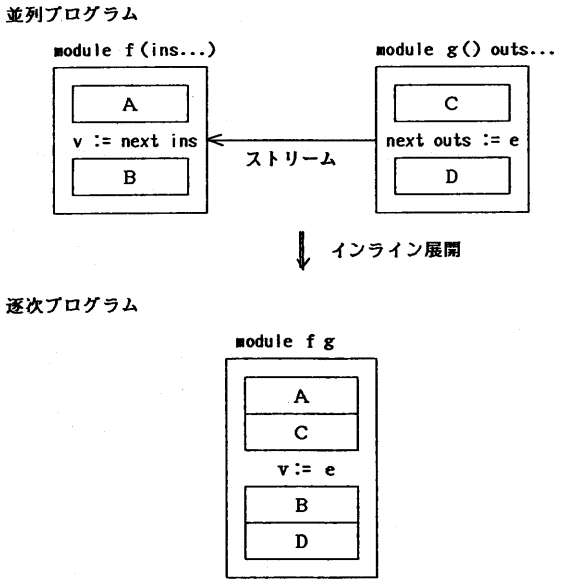


図４　インライン展開概要

３．１ペトリネットによるモデル化

まず，Stellaプログラムをペトリネット[15]を用いてモデル化し，その上で動作解析し，インライン展開する．インライン展開の他に，擬似並列実行する際に必要なバッファ長の解析，デッドロックの検出なども行なう[11][12]．解析は記述言語とは独立である．ストリームを介して通信するプロセス系であるなら，同様の手法で解析できる．モデル化は，以下の手順で行なう．

①各モジュールについて，文をトランジション，制御点をプレースに対応させてペトリネットを作る．１つのトランジションは，１つのストリーム入出力または，複数の一般の文に相当する．if文でも，そのthen側にもelse側にも入出力文，goto文がないものは，まとめて１つのトランジションに入れるが，入出力文，goto文が入っているif文は， XOR(排他的OR)トランジション[15]を使ってモデル化する（図７の23）． XORトランジションは発火すると複数の出力プレースのいずれか１つにトークンを移す．

②ストリームは，容量付きプレース[22]でモデル化し，その入出力文に対応するトランジションと結ぶ．容量

```
program secondl( input, output );
        label 0,1;
        var i1, i2, lastmax1, lastmax2: integer;

        procedure swap( var x,y: integer );
                var z: integer;
        begin z := x; x := y; y := z end;

begin
        if eof then goto 0;
        readln( lastmax1 );
        if eof then goto 0;
        readln( i1 );
        if i1 > lastmax1 then swap( i1, lastmax1 );
        lastmax2 := i1;
1:
        if eof then begin
                writeln( lastmax2 );
                goto 0
        end;
        readln( i1 );
        if i1 > lastmax1 then swap( i1, lastmax1 );
        i2 := i1;
        if i2 > lastmax2 then swap( i2, lastmax2 );
        goto 1;
0:
end.
```

図５　プログラムsecondlのインライン展開結果

付きプレースとは，中に入るトークン数を制限したプレースである．特別に容量が0のプレースも導入した．これは直後の発火で取り去られる場合に限ってトークンが入るという性質を持ったプレースである．

③例外処理に相当するトランジションと例外処理用のフラグに相当するプレース(eos flag，blocked flag)を用意し図6の要領で結ぶ．例外処理には，抑止アーク[15]を用いる．抑止アークは，矢印の代りに小さな丸のついた線で記述し，入力プレースにトークンがない時に発火可能となる．

④すべてのモジュールをストリームとフラグに相当するプレースで結合する．結合の際に非結合ストリームへの出力文に対応するトランジションは一般の文として扱う．

モジュールfirstlに①から③を適用すると図6のペトリネットが得られる．図6のペトリネット2つと標準入出力用のモジュール reads，writesのペトリネットを④により結合すれば，プログラム secondl全体のペトリネットが得られる（図7）．

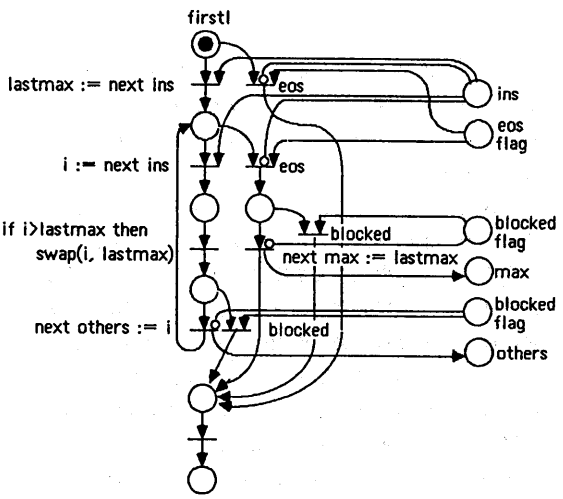


図6　モジュール firstl のペトリネットによるモデル化

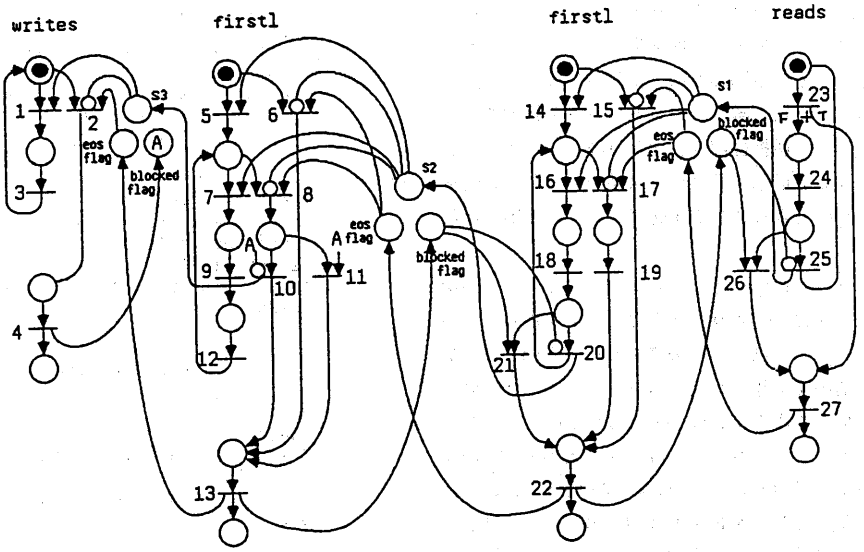


図7 プログラム secondl のペトリネットによるモデル化

３．２　マーキンググラフの作成

ペトリネットの解析には一般に到達可能木[15]が用いられるが，本方式では，それに類似したマーキンググラフを使用する．マーキンググラフは，ペトリネットの状態を示すノードとそれらの間の遷移を示すエッジから構成される．到達可能木との主な違いは，マーキンググラフでは，初期状態ノードＩから同じ距離に同一の状態を表す複数のノードが出現した場合，それらをマージして１つのノードとすることである．詳細は11)12)．

マーキンググラフは，対象のペトリネットが取り得るすべての状態遷移を表したもので，各バッファの容量によって異なる．図７のペトリネットですべてのバッファ容量が０の時のマーキンググラフは図８のようになる．ただし，ノードの内容は省略する．

図７の４つのトークンが，図８における初期状態ノードＩに相当する．ノードＩから，唯一，発火可能な排他的トランジション23の条件分岐に対する２つのエッジを作る． false部では，トランジション24，25が続けて発火できるので，それに対応するエッジを作る．トランジション25は，ストリームへの出力文に対応したもので，バッファにもトークンが入る．バッファプレースの容量は０なので，次には，このバッファからトークンを取り出すトランジション14しか発火できない．以下，同様にノードが終了状態Ｆになるか，以前に出現したノードと同一の状態Ｐになるまで，グラフの生成を続ける．ノードＰは，プログラム上でのループに対応する．

３．３　インライン展開

このマーキンググラフを用いてインライン展開する．まず，グラフから以下の条件を満足し，初期状態ノードＩを含む部分グラフを検出する．部分グラフは，各モジュールの非決定的な並列実行部分に対して１つの実行順序を特定したもので，実行の可能性を制約するものではない．

条件１　バッファプレースへトークンが入った直後にそのトークンが取られる．

条件２　条件分岐の両方を通る．

条件３　コード長が最小である．

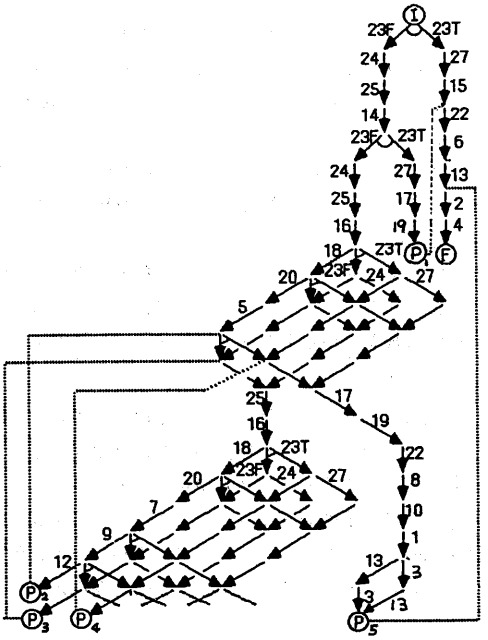


図８　図７のペトリネットから生成したマーキンググラフ

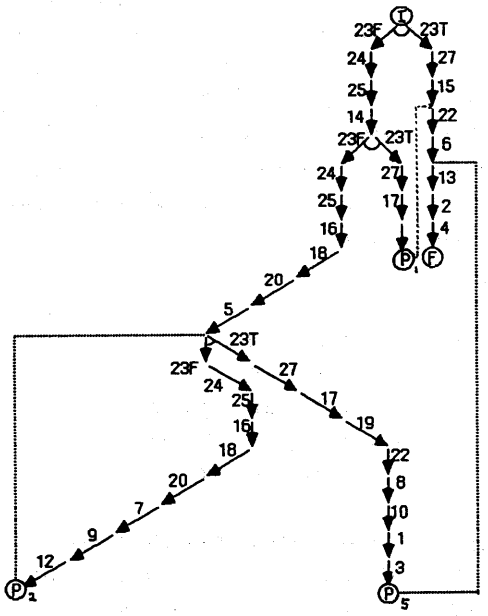


図９　インライン展開の条件を満足する部分グラフ

条件１は，ストリームの入出力文を１つの代入文に変換するための条件である．図８は，すべてのバッファの容量が０であるグラフなので，すでにこの条件を満たしている．

条件２は，分岐のための条件は実行時にしか判定できないので，両方の可能性を持つパスを検出しておくためのものである．

条件３は，最小のインライン展開結果を求めるための条件である．各トランジションに，元のソースコード長の情報を付加しておき，部分グラフ全体のコストを計算する．

以上の条件を満たす部分グラフは図９である．このグラフのトランジションを元の文に戻せばインライン展開したプログラム（図４）が得られる．

インライン展開のアルゴリズムでは，マーキンググラフ上のＩノードから，ＰまたはＦの端ノードに至るパスを結合していくことによって，条件を満足する部分グラフを求める．

インライン展開のアルゴリズムは，second1プログラムの例を使って説明する．

①Ｐノード，Ｆノードに至る各遷移列を部分グラフとする．それぞれの部分グラフに端ノードを要素とする端ノード集合を対応させる．また，各部分グラフから条件分岐列を抜き出す．この操作は，マーキンググラフの作成と同時にあらかじめ行える．

図８のマーキンググラフには，５つのＰノードと１つのＦノードがある．それぞれの端ノード集合と条件遷移列を以下に示す．

```
1 { F  } [ 23T ]
2 { P1 } [ 23F, 23T ]
3 { P2 } [ 23F, 23F, 23F ]
4 { P3 } [ 23F, 23F, 23F, 23F ]
5 { P4 } [ 23F, 23F, 23F, 23T ]
6 { P5 } [ 23F, 23F, 23T ]
```

②部分グラフうち，条件遷移列の最後の遷移以外は等しく最後の遷移が同じトランジションに対して真偽が逆であるような２つのグラフを結合し，新たな部分グラフを作る．その際，条件遷移列の最後の遷移は除き，端ノード集合はそれぞれの端ノード集合の和とする．

①で取り出した６つの部分グラフのうち，４と５から８，３と６から９の部分グラフが新たに作られる．

```
8 { P3, P4 } [ 23F, 23F, 23F ]
9 { P2, P5 } [ 23F, 23F ]
```

さらに，新たに作られた遷移列を加えて結合を続けると，以下の５つの部分グラフが作られる．結合は新たに部分グラフが作られなくなるまで続ける．

```
10 { P3, P4, P5 }          [ 23F, 23F ]
11 { P1, P2, P5 }          [ 23F ]
12 { P1, P3, P4, P5 }      [ 23F ]
13 { F,  P1, P2, P5 }      []
14 { F,  P1, P3, P4, P5 }  []
```

③条件遷移列が空の部分グラフで，端ノード集合に含まれるＰノードの戻り先がその部分グラフに含まれるものを取り出す．

ここでは，13と14の部分グラフが該当する．

④ 各グラフについてコード長を計算する．すべてのトランジションのコストが同一と仮定し，コード長を各端ノードのレベルから概算することができる．２つの部分グラフを結合する際には，それぞれのレベルを合計し，共通のレベルを差し引くことになる．

```
13 LEVEL(P2) + LEVEL(P5) - COMMON(P2,P5)
 + LEVEL(P1) - COMMON((P2,P5),P1)
 + LEVEL(F)  - COMMON((P2,P5,P1),F)
 = 20 + 21 - 12 + 8 - 5 + 8 - 1 = 39
```

```
14 LEVEL(P3) + LEVEL(P4) - COMMON(P3,P4)
 + LEVEL(P5) - COMMON((P3,P4),P5)
 + LEVEL(P1) - COMMON((P3,P4,P5),P1)
 + LEVEL(F)  - COMMON((P1,P3,P4,P5),F)
 = 21 + 21 - 20 + 21 - 12 + 8 - 5 + 8 - 1= 41
```

⑤ 最小コストの部分グラフを選ぶ．その部分グラフが最適なインライン展開である．

この例では13の部分グラフが最適である．この部分グラフの各遷移を対応するソースプログラムに変換すれば図５のインライン展開コードが得られる．

実際には，コストの計算は②の部分グラフの結合と同時に行う．空分岐遷移列が見つかれば，それよりコ

ストの高い部分グラフは，展開の対象から除くことができる．

図５の展開コードで，ラベル 1とgoto 1によるループ中で，２番目に大きな数の候補が lastmax2 に代入される．ループの前にある２つのif文は， eos例外処理に対応するもので，データ数が０と１の時にプログラムを終了させる．

４．評価

Stellaプログラムを解析しインライン展開を行うシステムをPascalで作成し，種々のストリーム・プログラムのインライン展開を行った．

実行効率は，インライン展開による実行の方が，擬似並列処理に比べて，２倍から１０倍早くなった．これは，擬似並列処理におけるモジュール間での制御の移行時間とデータをストリームへ入出力する際のバッファの操作時間が，インライン展開では除かれるためである．

擬似並列処理とインライン展開による実行時間の比較を次の表に示す．用いたプログラムは， second，ハミング問題，論理回路シミュレーション[18] である．

プログラム	インライン展開	擬似並列実行
second	２９．８	６３．７
ハミング	６．０	１１．１
論理回路	７．４	２３．６

（単位　秒）

表　実行効率の比較

インライン展開は，すべてのストリーム・プログラムに対して適用可能なわけではない．ストリームにデータが蓄えられるようなプログラムは，インライン展開できない．ストリームにデータが蓄積するのは，モジュール間の結合が特定のループ状になった場合や，２つのモジュールの間をデータの流れる速度の異なる複数のストリームで結合した場合である．また，動的にモジュールが生成されるようなプログラムも展開できない．

５．おわりに

ストリームを扱う言語Stellaとインライン展開による処理方式について述べた．本方式ではペトリネットを用いてモデル化し，最小コード長の展開結果を生成できるのが特長である．この方式に基づきインライン展開システムを作成し，種々のStellaプログラムの展開を行った．インライン展開されたコードは通信によるオーバーヘッドが取り除かれるため最大の実行効率を得ることができた．

現在，Stellaのためのプログラミング支援環境も設計，開発されている．この開発支援環境は，モジュールの結合を図式表示を利用して，画面上で設計，デバッグ等を対話的に行えるものである[18] ．支援環境を使って，新たに１つのモジュールを合成によって作成する際は，合成したモジュールをインライン展開し，実行効率の高い部品モジュールとして格納しておくことも考えられる．

インライン展開を活用することにより，ストリームを用いたプログラムのより実用的な利用が期待できる．

参考文献

1) Arvind and Brock,J.D.: Streams and Managers, Lecture Notes in Computer Science, Vol.143, pp.452-465, (1978).
2) Burge,W.H.: Stream Processing Functions,IBMJ Res.Dev, Vol.19, pp.12-25, (1975).
3) Shapiro,E.Y.: A Subset of Concurrent Prolog and Its Interpreter, Tech. Rep. TR-003,ICOT, (1983).
4) Dennis,J.B. and Weng, K.K.-S.: An Abstract Implementation for Concurrent Computation with Streams, Proc. 1979 Int. Conf. on Parallel Processing, pp.35-45, (1979).
5) Dijkstra,E.W.: A Discipline of Programming, Prentice-Hall, (1976).
6) Dod: Ada Programming Language, ANSI/MIL-STD-1815A, (1983).
7) Grune,D.: A View of Coroutines, ACM SIGPLAN Notices, Vol.12, No.7, pp.75-81, (1977).
8) Henderson,P.: Functional Programming: Application and Implementation, Prentice-Hall, (1980).
9) Hoare,C.A.R.: Communicating Sequential Processes, Comm. ACM, Vol.21, No.8, pp.666-677, (Aug. 1978).
10) Jackson,M.A.: Principles of Program Design, Accademic Press, (1975).
11) Kuse,K., Sassa,M. and Nakata,I.: Analysis and Transformation of Concurrent Processes Connected by Streams, ソフトウェア科学・工学の数理的方法研究会, 京都大学数理解析研究所講究録, No.511pp.120-143, (1984).
12) Kuse,K., Sassa,M. and Nakata,I.: Modelling and Analysis of Concurrent Processes Connected by Streams, J. Inf. Process., Vol.9, No.3 (1986).
13) Nakata,I. and Sassa,M.: Programming with Streams, IBM Research Reports RJ3751(43317), (Jan. 1983).
14) OCCAM Programming manual, INMOS Limited, (1982).
15) Peterson,J.L.: Petri Net Theory and the Modeling of Systems, Prentice-Hall, (1981).
16) 久世, 中田, 佐々: ストリームを扱う言語のトランスレータ方式による実現, 第29回情報処理学会大会, 3P-10, (1984).
17) 久世: ストリームを扱う言語Stellaによる在庫管理システムの記述,情報処理, Vol.26, No.5, pp.497-505, (1985).
18) 久世, 佐々, 中田: ストリーム・プログラミングのための図式表示を利用した開発支援環境について, 情報処理学会論文誌, Vol.27,No.12,(1986).
19) 久世: ストリームを扱う言語とその処理系の研究筑波大学工学研究科博士論文, (1986).
20) 鳥谷, 久世, 中田: ストリームを扱う言語Stellaーインライン展開方式の実現, 第30回情報処理学会大会, 1R-9, (1985).
21) 中田, 佐々: ストリームによるプログラミング, 第25回プログラミングシンポジウム, 情報処理学会, pp.124-135, (1984).
22) 松原: 容量ペトリネット, 電子通信学会論文誌, Vol.62, No.5, pp.309-316, (1979).