ストリームによるプログラミングのための言語とその実現方式†

久世和資††　佐々政孝†††　中田育男†††

ストリームによるプログラミングのための記述言語 Stella とその処理系の実現方式について述べる．われわれは，プログラムの記述性向上や再利用促進のためにプログラムにデータの流れであるストリームを導入することを提案している．ストリームによるプログラミングのための言語として設計，開発したのが Stella である．Stella プログラムは，一般にストリームで結合された複数モジュールのネットワークとして表現できる．プログラムを実行すると，ストリームをとおしてデータを流しながら各モジュールが並行に動作する．本論文では，Stella 処理系の単一プロセッサによる三つの実現方式について述べる．これらの方式は，擬似的に並列処理する方式と逐次プログラムに展開する方式の二つに大別できる．後者は，インライン展開と呼び，単一プロセッサ上で最大の実行時間効率を得ることができる．インライン展開としては，ソースコードから直接展開する方式とペトリネットでモデル化して展開する方式を考案し実現した．とくにモデル化により展開する方法では最小の展開コードを求めることができる．マルチプロセッサによる実行においてもプロセッサ台数よりモジュール数が多いときには，これら三つの処理方式は有効である．三つの処理系を実際に作成し，各処理系で種々の Stella プログラムの実行を行った．その結果に基づいて各処理方式の比較を最後に述べる．

1. はじめに

近年，ソフトウェアの生産性向上が，重要な課題となっている．生産性向上の一つの要因として，プログラムの記述性や読みやすさの改善があげられる．また，プログラムの部品化と再利用の促進も生産性向上に役立つ．われわれは，これらの観点から，プログラム言語にストリーム機能を導入し，ソフトウェアの生産性の向上について考察した[18),19)]．

ストリームは，一般に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),23)]，データフロー言語[1),3)]，論理型言語[22)]などにおいては，ストリームに関する研究がされている．また，Unix のパイプ，ソケット機能[21)]などもある種のストリームといえる．われわれは，汎用言語にストリームを扱う機能を導入し，より実用的なストリーム・プログラムの作成を可能にした．この言語は，Pascal をベースにして設計，実現したもので Stella[12),15)]と呼ぶ．

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．プログラムを実行すると，ストリームをとおしてデータを流しながら，各モジュールが並行に動作する．Stella の実行系は，一般には，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[5),9),10)]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[13)]．

Stella の特徴の一つは，データの流れにしたがって，アルゴリズムが素直に表現できることである．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，ストリームとみなせる[12)]．ジャクソン法[11)]における入力データ構造と出力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．アルゴリズムが素直に記述できる例としては，ライニング問題[6)]*，ハミング問題[4)]，フィボナッチ数列，素数列などがある[18)]．

さらに，複雑なシステムプログラムなども，適当な単位に処理を分割し，処理ごとにモジュールを作成し，それらをストリームで結合することによって，素直に構築できる．たとえば，ストリームとして，コンパイラにおける入力文字列，字句解析結果のトークン列，データベースシステムにおける検索データ列などが考えられる[15)]．

また，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．プログラマは，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できる．プログラムのデバッグも，流れるデータを観

† Design and Implementation of a Stream Programming Language by KAZUSHI KUSE (IBM Research, Tokyo Research Laboratory), MASATAKA SASSA and IKUO NAKATA (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 日本アイ・ビー・エム(株)東京基礎研究所
††† 筑波大学電子・情報工学系

* 入力中の文字列 'aa' を 'a' に変換，その変換結果に対して 'bb' を 'b' に変換する問題．

察できる機能を用意すれば，行いやすくなる[14]．

このような考え方は，ファイルの結合，Unix のソケットなどによっても実現できるが，われわれはストリームをプログラミング言語の標準のデータ型として導入することを主張している[12),15),18),19)]．

本論文では，Stella プログラムを実行する処理系の実現について述べる．プログラムの実行は，マルチプロセッサによる実行と単一プロセッサによる実行が考えられる（図-1）．今回は，後者の単一プロセッサ上での実行方式について述べる．しかしながら，本論文で述べる実行方式は，マルチプロセッサによる実行においても，モジュール数がプロセッサ台数以上のときには有効な手法として利用できる（この手法はストリームを言語の外付け機能として実現する方法と比べ，格段に実行効率が良い）．単一プロセッサ上での実行には，擬似並列実行とインライン展開実行[16),18),19)]の２種類がある．

擬似並列実行は，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，擬似的に並列処理する方式である．この方式は，使用上の制限が少なく適用範囲は広いが，制御の切り替えとバッファを介した通信のオーバヘッドがあり，実行効率が良くない．インライン展開実行は，複数の並行動作プロセスを一つの逐次プロセスに展開し，実行する方式で，実行効率は良い．しかし，動的にプロセスを生成するプログラムなどは展開できないなどの制限がある．

われわれは，インライン展開として，ソースプログラムから直接，一定の規則に従って展開する方式[18),19)]とペトリネット[20)]によりモデル化した上で解析し展開する方式[15),16)]の２種類を設計，実現した．

以降，最初にストリームによるプログラミングの概要と記述言語 Stella について簡単な例を用いて述べる．ついで，各処理方式の概要を述べる．さらに，実際に設計，実現した各実行系の使用上の制限，特徴，および，実行効率を比較検討し，実際のプログラムへの適用方法について考察する．

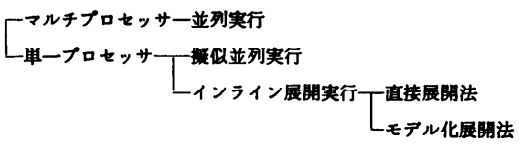


図-1 ストリーム・プログラムの実行方式の分類
Fig. 1 Classification of execution methods for stream programs.

2. ストリーム・プログラムと記述言語 Stella

2.1 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．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くるものを入力ストリーム，モジュールから出てい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．Unix のパイプ[21)]は，コマンド言語に導入されたストリーム機能であるが，入出力が各１本であるので一直線上の結合しかできない．Unix のソケットは，ストリームの考え方に類似し，ネットワーク状の結合もできるが，前述のようにわれわれはストリームを言語の標準の型として位置づけている．

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．プログラムを実行すると，ストリームをとおしてデータを流しながら，各モジュールが並行に動作する．

ストリームを用いたプログラミングには，以下のような利点がある．

（a） 逐次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．

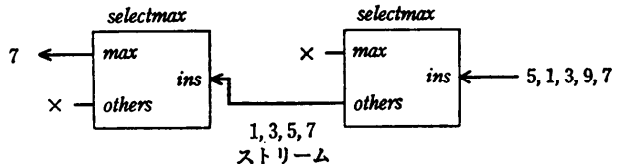
（b） データの流れに沿ったプログラミングが容易になり，ジャクソン法[11)]におけるプログラム変換などが不要になる．

（c） 単純な処理をするモジュールをストリーム結合で組み合わせて，より複雑な処理をするプログラムを階層的に作成できる．

（d） 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．

（e） 各モジュールは，独立して他のモジュールと並行に動作できるので，機能的に独立に記述や理解ができ，情報隠蔽や部品化ができる．

（f） インライン展開をすることにより，単一プロセッサ上でも高い実行効率を得ることができる．



モジュール *selectmax* は，入力ストリーム *ins* 中の最大値を出力ストリーム *max* に出力し，それ以外の値を出力ストリーム *others* に出力する．記号×は非結合ストリーム．

図-2 整数列中から2番目に大きな数を求めるストリーム・プログラム
Fig. 2 A stream program that selects the second largest number from an input number sequence.

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番目に大きな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 *selectmax* が使えるものとする．*secondl* は，二つの *selectmax* 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し，非結合ストリームと呼ぶ．

2.2 プログラミング言語 Stella

Stella の言語仕様は，Pascal の仕様を基本としており，Pascal のスーパセットとした．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，(a)ストリーム定義部，(b)モジュール宣言部，(c)結合部の三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．

(a) 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．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．

(b) 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（**図-3①**），（ ）の次に出力ストリームを宣言する（**図-3②**）．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一つ取り出すには，式の中に

next 入力ストリーム名

の形で記述する．**図-3③**は，入力ストリーム *ins* から，要素を一つ取り出し，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．終了する直前になんらかの処理をしたいときには≪ ≫内に記述し，付加する（**図-3④**）．*eos* になっても終了させない場合は，例外処理部に goto 文などを書き，制御を他へ移す．

逆に出力ストリームへ要素を一つ

```
program secondl (input, output);
  module selectmax (ins: integers)                {①入力ストリームの宣言}
                   max, others: integers;         {②出力ストリームの宣言}
    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;                            {③1要素の入力文}
    loop
      i:=next ins
        ≪next max:=lastmax≫;                      {④eos 例外処理文}
      if i>lastmax then swap (i, lastmax);
      next others:=i                              {⑤1要素の出力文}
    end
  end;
begin
  connect                                         {⑥モジュールの結合・実行文}
    selectmax (input) free, $s;                   {⑦$s はストリーム}
    selectmax ($s) output, free                   {⑧free は非結合ストリーム}
  end
end.
```

図-3 図-2のプログラムの Stella による記述例
Fig. 3 A program of Fig. 2 in Stella.

```
procedure selectmax (var ins, max, others: integers;
    var i, lastmax: integer; var proc: process);
  label 1, 2, 3, 4, 98, 99;
  procedure swap (var a, b: integer);
        .....
begin
  case proc. point of                                  {中断点への分岐}
    1: goto 1; 2: goto 2; 3: goto 3; 4: goto 4 end;
1:                                                     {lastmax :=next ins}
  with ins do
    case status of
      full, normal:                                    {バッファが空でない}
        begin
          head:=head+1;
          lastmax:=element [head];                     {1要素入力}
          if head=size then head:=0;
          if head=tail then status:=empty
                       else status:=normal
        end;
      empty:                                           {バッファが空}
          if eos then
            goto 98                                    {eos 例外処理}
          else begin
            proc. point:=1;
            goto 99
          end
    end
  while true do begin
        .....
  if i>lastmax then swap (i, lastmax);
4:                                                     {next others:=i}
  with others do
    if blocked then                                    {blocked 例外処理}
      goto 98
    else
      case status of
        empty, normal:                                 {バッファに空きあり}
          begin
            tail:=tail+1;
            element [tail]:=i;                         {1要素出力}
            if tail =size then tail:=0;
            if head=tail then status:=full
                         else status:=normal
          end;
        full:                                          {バッファに空きなし}
          begin
            proc. point:=4;
            goto 99
          end
      end
  end;
98:                                                    {モジュールの終了}
  proc. ready :=false;
  ins. blocked:=true;
  max. eos    :=true
  others. eos :=true;
99:                                                    {モジュールの中断}
end;
```

図-4 モジュール *selectmax* の擬似並列処理のための変換結果
Fig. 4 Translated "*selectmax*" code for quasi-parallel execution.

送り出すには,

next 出力ストリーム名:=式

の形で記述する (図-3⑤). 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, そのモジュール自身も終了する. 終了時の処理は≪ ≫内に記述できる.

(c) 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する (図-3⑥). connect 文内の各モジュールは, 記述順に関係なく並行に動作する.

二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-3⑦). 出力ストリームを結合しないときには, 非結合ストリームとして **free** を記述する (図-3⑧). 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, 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. この場合は処理系が自動的に関係する標準モジュールを結合する (図-9 の *reads* と *writes*).

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. *selectmax* 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[15].

3. 擬似並列処理方式

擬似並列実行は, ストリームに対してバッファを用意し, 各モジュー

ルをコルーチンとして，擬似的に並列実行する方式である．並列プログラムの実行では，スケジューラが，各モジュールの実行を管理する方式が一般的である．しかし，後述するストリーム・プログラムの特性(b)を考慮して，本処理系では，特にスケジューラは用意せず，あらかじめ固定した順序ですべてのモジュールに等しく制御が移る方式を採用した．この方式では，制御の移行を外部でスケジューラが行うのではなく，各モジュールが，独自にバッファの状態を検出して中断し，その場所を記憶し，制御を次に実行すべきモジュールへ移す．制御の移行順は，記述順とした．

モジュールが中断する条件は，空のバッファからデータを取り出そうとした場合と空き領域のないバッファへデータを送り出そうとした場合である．スケジューラを使用すると，中断の原因を記憶しておき，それが解消されたモジュールを優先的に実行することになる．これに対してわれわれの方式では，中断状態の解消，非解消にかかわらず，一定の順序で制御を移す．したがって，制御が移ったモジュールの状況が中断時と同じで何もせずに制御を他のモジュールへ移すといった冗長な制御の移行が生じる可能性がある．しかし，以下にあげる理由から本方式を採用した．

(a) スケジューラ方式における管理機構より，本方式における各モジュール内での判断機構のほうが，単純である．

(b) 冗長な制御の移行が生じる可能性が高いのは，各モジュールに初めて制御が移る場合であり，それ以後は，各ストリームにデータが蓄積されるので，その可能性は小さい．

実際には，Stella プログラムを Pascal プログラムに変換し，擬似並列実行する．変換によって，ストリームは循環バッファに，モジュールは手続きとなる．バッファのサイズは，ユーザがプログラムで指定できるが，指定のないときは標準値 100 になる．これは，いくつかの例題についてバッファのサイズを変化させて実行時間を計測した結果，100 が妥当な値と判断したからである[15]．

変換されたプログラムを実行すると，各手続きが，順々に繰り返し実行され，循環バッファから，データを取り出したり，書き込んだりする．もし，空のバッファからデータを取り出そうとしたり，空領域のないバッファへ書き込もうとすると，その時点で処理を中断し，中断点を憶えておき，その手続きを抜け，別の手続きを実行する．中断した手続きを，再び実行する際には，記憶しておいた中断点から実行を再開する．

各循環バッファ，各モジュールの局所変数および中断点は，手続きを呼び出すたびに，初期化されることなく，しかも，手続き内で，変更できなければならない．したがって，これらをグローバル領域に用意し，各手続きで変数引数として参照することにした．

動的なモジュールの生成，結合は，モジュール本体の動的生成によって実現するのではなく，モジュールの本体は一般のモジュールと同様に一つの手続きに変換し，モジュールの状態である局所変数と中断点のデータおよびストリームを動的に生成することによって行う．

図-3 に対する変換結果は，図-4 に示す．変換の詳細は，文献 15)を参照．

4. インライン展開実行

ストリームによる通信を含む並行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

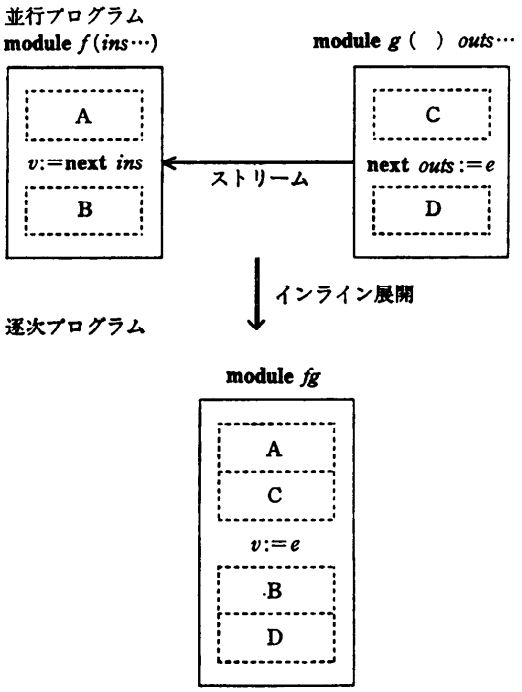


図-5 ストリーム・プログラムにおけるインライン展開概略図
Fig. 5 Overivew of stream program in-line expansion.

```
program secondl (input, output);
        label 0, 1;
        var   r: integer;                {標準入力モジュールの局所変数}
              i1, lastmax1: integer;     {selectmax (図-2 の右) の局所変数}
              i2, lastmax2: integer;     {selectmax (図-2 の左) の局所変数}
              w: integer                 {標準出力モジュールの局所変数}
        procedure swap (var a, b: integer);
              var c: integer;
        begin c:=a; a:=b; b:=c end;
begin
        if eof then goto 0;
        readln (r);
        lastmax1:=r;
        if eof then goto 0;
        raadln (r);
        i1:=r;
        if i1>lastmax1 then swap (i1, lastmax1);
        lastmax2:=i1;
1:
        if eof then begin
              w:=lastmax2
              writeln (w);
              goto 0
        end;
        readln (r);
        i1:=r;
        if i1>lastmax1 then swap (i1, lastmax1);
        i2:=i1;
        if i2>lastmax2 then swap (i2, lastmax2);
        goto 1;
0:
end.
```

図-6 図-3 のプログラム *secondl* のインライン開展結果
Fig. 6 In-line expanded code of the program "*secondl*" in Fig. 3.

る入力を，一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-5)．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-5 のような単純な変換では済まない．たとえば，*secondl* の展開結果は，図-6 に示すとおりである．

われわれは，インライン展開の手法として，ソースコードから直接展開する手法とペトリネットを用いてモデル化してから展開する手法の二つを考案し，それぞれの展開系を作成した．以下，それぞれについて述べる．なお，インライン展開するためには，プログラム上の制限がある．この制限も含めて各手法の比較を 5. に述べる．

4.1 ソースコードからの直接展開法

直接展開法は，二つのモジュールのソースコードから直接，インライン展開して一つのモジュールを得る方式である．そのアルゴリズムは，文献 7) からヒントを得て例外処理や，ある程度の最適化も考慮して作成した．この方式で展開できるモジュール数は二つなので，一般に三つ以上のモジュールが含まれるプログラムの場合には，二つずつモジュールを選んで展開し，モジュールが一つになるまで，この方式を繰り返し適用することになる．

モジュール f と g を展開して新たにモジュール h を作る例を用いて，アルゴリズムを説明する．展開の途中の状況は (S_f, S_g, S_h) の形で表現する．ここで，S_f と S_g は，これから展開する f と g の文の列，S_h は展開により生成された h の文の列をそれぞれ表す．f と g の文で，if 文以外の制御文は，あらかじめ，if 文と goto 文に変換しておく．展開の初期状態は (S_f, S_g, 〈 〉)，最終状態は (〈 〉, 〈 〉, S_h) である．ただし，〈 〉は空列を表し，(〈 〉, 〈 〉, S_h) は，S_h と等しいものと考える．アルゴリズムは以下の変換規則の集合として与えられる．なお，各変換規則は，S_f と S_g を交換した規則も含む．

(1) (**terminate**; S_f, **terminate**; S_g, S_h)
→(〈 〉, 〈 〉, S_h; **terminate**)
ただし，**terminate** は，モジュールを終了する文

(2) (s; S_f, S_g, S_h)→(S_f, S_g, S_h; s)
ただし，s は，if 文，goto 文，f と g を結ぶストリーム入出力文の 3 種類以外の文

(3) (**goto** L; S_f, S_g, S_h)→(S_L, S_g, S_h)
ただし，S_L は，ラベル L から始まる f の文の列

(4) (**if** cond **then** S_{f1} **else** S_{f2}; S_{f3}, S_g, S_h)
→(〈 〉, 〈 〉,
S_h; **if** cond **then** (S_{f1}; S_{f3}, S_g, 〈 〉)
else (S_{f2}; S_{f3}, S_g, 〈 〉))

(5) (v:=**next** *ins* ≪S_{f1}≫; S_{f2},
next *outs*:=exp≪S_{g1}≫; S_{g2}, S_h)
→(S_{f2}, S_{g2}, S_h; v:=exp)
ただし，*ins* と *outs* は f と g を結ぶストリーム

(6) (putget$_f$; S_f, putget$_g$; S_g, S_h)
→(〈 〉, 〈 〉, S_h; error)
ただし，putget$_f$ と putget$_g$ は，f と g を結ぶ異

なるストリームに対する入出力文

error は，実行時にそこに到達した場合にエラーとなることを表す

(7) (v := **next** *ins*≪S_{f1}≫ ; S_{f2}, **terminate** ; S_g, S_h)
→(S_{f1} ; **terminate** ; S_{f2}, **terminate** ; S_g, S_h)
(**next** *outs* := exp≪S_{f1}≫ ; S_{f2}, **terminate** ; S_g, S_h)
→(S_{f1} ; **terminate** ; S_{f2}, **terminate** ; S_g, S_h)

ただし，*ins* と *outs* は，f と g を結ぶストリーム

(8) (putget≪S_{f1}≫ ; S_{f2}, S_g, S_h)
→(S_{f2}, S_g, S_h ; putget≪(S_{f1} ; **terminate** ; S_{f2}, S_g, 〈 〉)≫)

ただし，putget は，f と g を結ぶストリーム以外のストリームに対する入出力文

(9) (s ; S_f, t ; S_g, S_{h1} ; u ; S_{h2})
→(〈 〉, 〈 〉, S_{h1} ; L : u ; S_{h2} ; **goto** L)

ただし，u は，以前に s と t のつき合わせで生成された文（履歴を保存しておくことにより判定する）

以上のアルゴリズムにおいて，生成コードが長くなるのを避けるため，規則(9)をもっとも高い優先順位とし，規則(4)と(8)をもっとも低い優先順位とする．実際の展開例として二つの *selectmax* モジュールの展開過程を図-7 に示す．

この方式では，三つ以上のモジュールを展開する際，規則の適用順によって展開可能な場合と展開不可能な場合が生じることがある．これは，f と g の両方に規則(8)が適用できる際の非決定性が原因となっている．この非決定性は，f と g を展開する際に，第3のモジュール h の内容を考慮すれば解決できる．しかし，h を固定することは，実用性を低下させる．したがって，非決定な部分については，すべての可能性を求めておくことが考えられるが，これでは展開を繰り返すと，その展開結果が膨大になってしまうという欠点がある．

そこで，アルゴリズムを改良して，両方のモジュールについて規則(8)が適用できる場合は，出力文を優先することにし，さらに出力文が展開不可能になった場合に，その出力結果を一時蓄えるサイズ1のバッファを設けることによって展開できるようにした．これらの改良によってアルゴリズムの非決定性のために生じる展開不可能は，ほとんど回避できる．

```
初期状態
(lastmax2:=next ins2;…, lastmax1:=next ins1;…, 〈 〉)

規則(8)を適用
(lastmax2:=next ins2;…, L1: i1:=next ins1;…, lastmax1:=next ins1)

規則(8)を適用
    ここで，next ins1 の eos 処理部は，非結合の max1 に対する出力なので除外する
(lastmax2:=next ins2;…,
 if i1>lastmax1 then swap(i1, lastmax1);…,
 lastmax1:=next ins1; i1:=next ins1)

規則(4)を適用
(〈 〉, 〈 〉,
 lastmax1:=next ins1;
 i1:=next ins1;
 if i1>lastmax1 then (lastmax2:=next ins2;…,
                      swap (i1, lastmax1);…,
                      〈 〉))

規則(2)を適用
(〈 〉, 〈 〉,
 lastmax1:=next ins1;
 i1:=next ins1;
 if i1>lastmax1 then (lastmax2:=next ins2;…,
                      next others1:=i1;…,
                      swap (i1, lastmax1)))

規則(5)を適用
(〈 〉, 〈 〉,
 lastmax1:=next ins1;
 i1:=next ins1;
 if i1>lastmax1 then (L2: i2:=next ins2≪next max2:=lastmax2≫;…,
                      goto L1;…,
                      swap(i1, lastmax1); lastmax2:=i1))

規則(3)を適用
(〈 〉, 〈 〉,
 lastmax1:=next ins1:
 i1:=next ins1;
 if i1>lastmax1 then (L2: i2:=next ins2≪next max2:=lastmax2≫;…,
                      L1: i1:=next ins1;…,
                      swap (i1, lastmax1); lastmax2:=i1))

                以下，略
```

二つの *selectmax* の変数名を区別するために，1（図-2 の右）と 2（図-2 の左）を付加する L1 と L2 は，loop 文に対応するラベル

図-7　二つの *selectmax* モジュールの直接インライン展開（一部）
Fig. 7　Direct in-line expansion of two instances of "*selectmax*".

4.2 ペトリネットによるモデル化展開法

モデル化展開法は，Stella プログラムをペトリネット[20]でモデル化し，その上で動作解析し，インライン展開する手法である[13),16)]．直接展開法が二つのモジュールに適用されるのに対し，これはモジュールの数に関係なくプログラム全体に適用される．直接展開法に比べ，モデル化の時間を要するが，最小コード長の展開結果が得られるといった利点がある．また，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[13)]も行える．これらの解析やインライン展開は，記述言語の Stella とは独立に適用できる．以下，ペトリネットによるモデル化，マーキンググラフの生成，マーキンググラフを用いたインライン展開の順で述べる．

4.2.1 ペトリネットによるモデル化

まず，ペトリネットを用いてストリーム・プログラムをモデル化する．モデル化は以下の手順で行う．

（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．一つのトランジションは，一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．then 側にも else 側にも入出力文および goto 文がない if 文は，まとめて一つのトランジションにする．分岐の一方または両方に入出力文または goto 文が入っている if 文は，排他的 OR トランジション[20]を使ってモデル化する（図-9 の 23）．排他的 OR 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一つにトークンを移す．

（2） ストリームは，そのデータを蓄える容量付きプレース[17]でモデル化し，対応するストリーム入出力文のトランジションと結ぶ．容量付きプレースとは入るトークン数を制限したプレースである．容量は任意に与えることができる．また，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もったプレースである．容量 0 プレースはトランジションで代用することができるが，容量 0 も容量 1 以上の場合と同様に扱うために導入した．

（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を用意し図-8 の要領で結ぶ．例外処理には，抑止アーク (inhibitor arc)[20]を用いる．これは，矢印の代わ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ときに発火可能となる．（たとえば，ストリームから要素を入力する際に，*eos* フラグにトークンが入っており，ストリームに対応するプレースにトークンが入っていないとき，*eos* 例外処理トランジションが発火できる）

（4） すべてのモジュールをストリームで結合する．具体的には各ストリームに対応するデータのプレースと二つのフラグに対応するプレースを介して結合する．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 *selectmax* に（1）から（3）を適用すると図-8 のペトリネットが得られる．図-8 のペトリネット二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-9）．*reads* は，入力ファイルを調べて空でないときには，値をファイルから読み込み，その値を出力ストリームに出力する．この空の判定に排他的 OR トランジションを用いる．

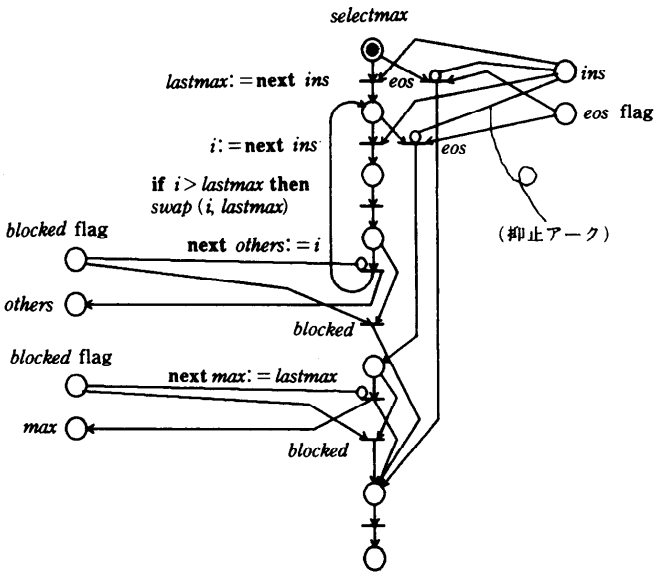
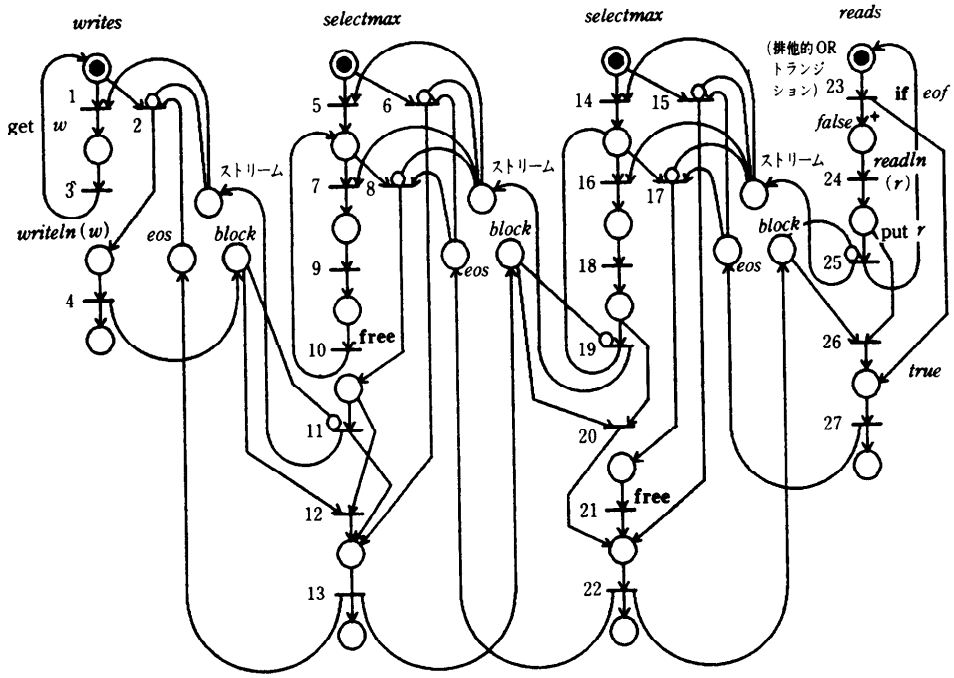


図-8 モジュール *selectmax* のペトリネットによるモデル化
Fig. 8 Modelling of "*selectmax*" using a Petri net.



各トランジションに対応するコードを以下に示す

5, 14　*lastmax*:=**next** *ins*
7, 16　*i*:=**next** *ins*
9, 18　**if** *i*>*lastmax* **then** *swap*(*i*, *lastmax*)
10, 19　**next** *others*:=*i*
11, 21　**next** *max*:=*lastmax*

ただし，二つの *selectmax* の変数を区別するために 1 (右) と 2 (左) の数字を付加する
10, 21 は非結合ストリームに対する出力文である

図-9 プログラム *secondl* のペトリネットによるモデル化
Fig. 9 Modelling of the program "*secondl*" using a Petri net.

4.2.2 マーキンググラフの生成

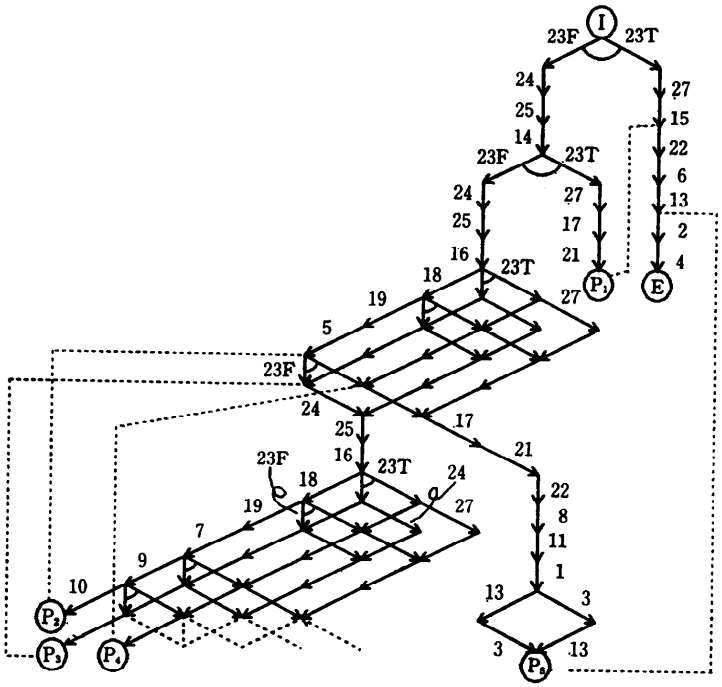
ペトリネットの解析には一般に到達可能木[20]が用いられるが，ここでは，それに類似したマーキンググラフを使用する．マーキンググラフは，ペトリネットの状態を示すノードとそれらの間の遷移を示すエッジから構成される．到達可能木との主な違いは，マーキンググラフでは，初期状態ノードから同じ距離に同一の状態を表す複数のノードが出現した場合，それらをマージして一つのノードとすることである．詳細は文献 13) を参照．

マーキンググラフは，対象のペトリネットが取りうるすべての状態遷移を表したもので，各バッファの容量の設定によって異なる．図-9 のペトリネットにおけるすべてのバッファ容量が 0 のときのマーキンググラフは次のような要領により図-10 のようになる．ただし，ノードの内容は省略する．

図-9 の 4 つのトークンが初期状態ノードⅠに相当する．ノードⅠから，唯一，発火可能な排他的 OR トランジション 23 の条件分岐に対する二つのエッジを作る．*false* 部では，トランジション 24, 25 が続けて発火できるので，それら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Ｅ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．

4.2.3 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



エッジに付加した番号は図-9におけるトランジション番号を示す
各ノードの内容は省略する
Iは初期状態ノード，Eは終了状態ノード，
Pは（点線で示される）ノードと同一の再出現ノード

図-10 図-9のペトリネットから求めたマーキンググラフ
Fig. 10 Marking graph of the Petri net in Fig. 9.

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一つの実行順序を特定したもので，実行可能性を制約するものではない．

（1） すべてのバッファプレースの容量が0

（2） 排他的ORトランジションの両方のエッジを含む

（3） エッジに対応するコードサイズの合計が最小

条件(1)は，ストリームの入出力文を一つの代入文に変換するための条件である．図-10は，すべてのバッファの容量が0のときのグラフなので，すでにこの条件を満たしている．

条件(2)は，分岐の条件部が実行時にしか評価できないので，両方の可能性をも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が**図-11**である．このグラフのエッジに対応する遷移を元の文に戻せばインライン展開したプログラム（図-6）が得られる．インライン展開のアルゴリズムの詳細は，文献16)を参照．

図-6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．ループの前にある二つのif文は，*eos*例外処理に対応するもので，データ数が0と1のときにプログラムを終了させる．

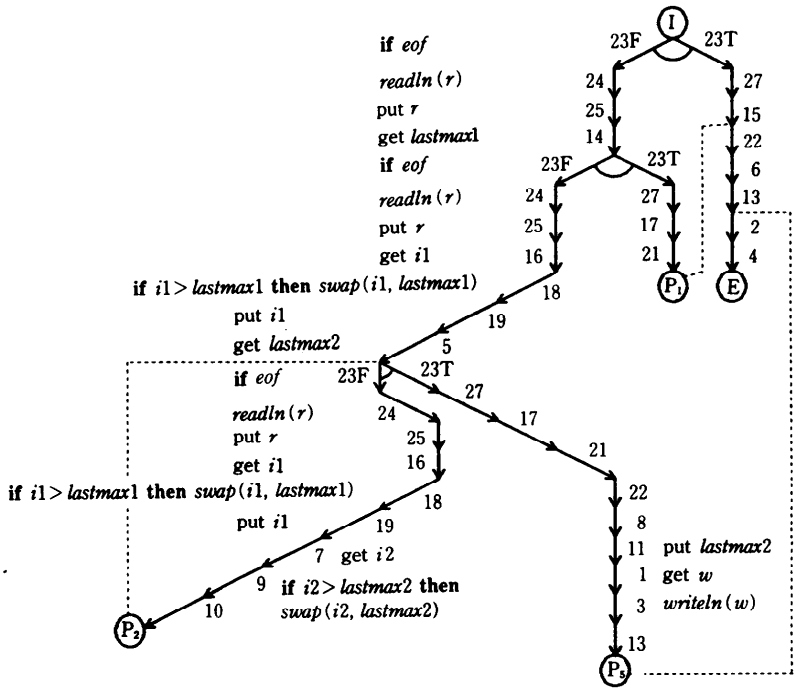
5. 各処理系の比較

それぞれの処理系を，適用範囲，変換コスト，変換コードのサイズ，変換コードの実行効率の順で比較する．変換コストと実行効率については，VAX-11/750, Unix 4.2 BSDを用いて測定した．

5.1 適用範囲

まず，各処理方式の制限について述べる．擬似並列処理方式の制限は，ほとんどないが，無限バッファを必要とする計算はできない．インライン展開方式では，ストリームにデータが蓄えられるような複数のモジュールは，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，二つのモジュール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，動的に生成されるモジュールもインライン展開できない．しかし，このような関係のモジュールを含むプログラム全体が展開不可能なわけではない．プログラム中のその他のモジュールは展開することによって効率よく実行することができる．

インライン展開の二つの手法を比べると，直接展開法は，一度に二つのモジュールしか展開できない．一方，モデル化展開法は，プログラムを部分的に展開することができない．



各遷移の左（または右）は対応するソースコード

図-11 図-10でインライン展開条件を満足する部分グラフ
Fig. 11 Part graph that satisfies the conditions of the in-line expansion in Fig. 10.

5.2 変換コスト

次に，変換にかかるコストを比較する．擬似並列処理もインライン展開も，Stella プログラムを Pascal プログラムに変換するのであるが，その変換コストは，擬似並列処理のほうが低い．擬似並列処理の場合は，入力ソースプログラムに従って，1パスで，特定の文を展開すればよい．インライン展開では，直接展開法の場合は，ソースプログラムを複数回，探索する必要があるし，モデル化展開法の場合も，ペトリネットを作成し，マーキンググラフを計算するので，変換コストは高い．インライン展開における二つの展開法を比較すると，変換に要するコストは直接展開法のほうが，マーキンググラフの計算分だけ低い．*secondl* の変換コストを表-1に示す．

表-1 各処理方式の変換時間の比較 (*secondl*)
Table 1 Comparison of translation times for described methods.

(単位：秒)

処理方式	擬似並列実行	直接展開法	モデル化展開法
変換時間	3.2	12.0	17.5

5.3 変換されたコードのサイズ

変換により生成されたプログラムのサイズを比較する．擬似並列処理では，各プログラムに共通するルーチンが追加される以外は，ストリーム入出力の各文が数行のプログラムに変換されるので，サイズは入出力文の数に依存する．擬似並列処理における変換プログラムのサイズは，以下の(式1)で与えられる．

ソースコードサイズ＋共通付加コードサイズ
＋入出力文数×1入出力文の展開サイズ （式1）

インライン展開の場合は単純な式では表せない．ここでは，ループをもった二つのモジュールを1本のストリームで結んだプログラムの展開について，サイズを評価する．ストリームの入出力を含む条件分岐がない場合は以下の(式2)で表される．

$$\frac{\text{コードサイズ}_1 \times \text{LCM}}{\text{入出力文数}_1} + \frac{\text{コードサイズ}_2 \times \text{LCM}}{\text{入出力文数}_2} \quad \text{（式2）}$$

ただし，1と2はモジュール1と2をそれぞれ表す．LCM は，入出力文数$_1$と入出力文数$_2$の最小公倍数．

たとえば，図-12 に示すインライン展開の場合，点線で示した一つの箱のサイズを1とすると，展開結果

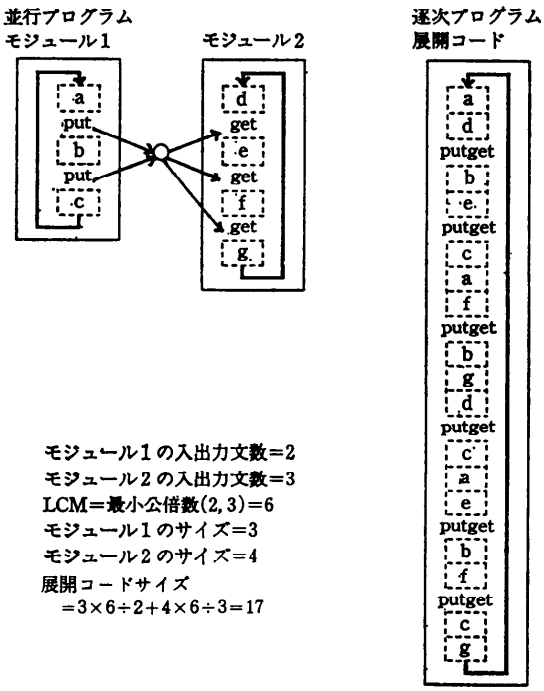


図-12 インライン展開コードのサイズの計算例
Fig. 12 Calculation of the in-line expanded code size.

のサイズは，3×6÷2+4×6÷3=17 となる．また，二つのモジュールの入出力文数が，等しい場合は，展開されたプログラムのサイズが最小で，元のサイズの和となる．

モジュールが，ストリームの入出力文を含む条件分岐をもつ場合は，(式2)が適用できない．通常，この場合は，展開サイズは，(式2)より大きくなる．

二つのインライン展開法では，モデル化展開法のほうが，展開プログラムのサイズは小さい．これは，マーキンググラフを利用して，複数の可能性のなかから最小サイズの展開を求めるためである．直接展開法では，冗長な展開コードを含む可能性がある．

展開プログラムのサイズは，そのプログラムの内容にも依存するが，一般的には，擬似並列処理，モデル化展開法，直接展開法の順で大きくなる．

5.4 実行効率

最後に，それぞれの実行効率を比較する．実行効率では，インライン展開による実行のほうが，擬似並列処理に比べて，2倍から5倍早い．これは，擬似並列処理では，モジュール間で制御を移す際の手続き呼出しの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必要なためである．前者は，バッファのサイズに依存し，バッファのサイズが大きいほど，それを扱うモジュールの中断が少なく，手続き呼出しの時間は，短くなる．バッファの操作時間は，入出力時にストリームやバッファの状態を調べ，それらの状態を管理するための時間がほとんどである．擬似並列処理による実行時間のうち，このバッファの操作時間の占める割合が大きい．

表-2 各処理方式の実行効率の比較
Table 2 Comparison of execution times for the described methods.

(単位：秒)

問題＼処理方式	擬似並列実行	インライン展開実行
secondl ($n=10^4$)	63.7	29.8
ハミング問題 ($n=1,000$)	1.1	0.6
素数列 ($n=500$)	18.6	不可
4段2進カウンタ ($n=10^4$)	23.6	7.4

ただし，nは各ストリームを流れるデータ数

二つのインライン展開方式の実行効率は等しい．前述の直接展開法による冗長な展開部分があっても，その実行時間は，モデル化展開法による最小コードの実行時間と同じである．モデル化展開コードは，同じ処理に対しては同一のコード列を再利用するが，直接展開コードは，同じ処理コードがコピーされている可能性がある．この場合，コードサイズは大きくなるが，実行時に評価される内容は変わらないので，実行時間は等しい．

擬似並列処理とインライン展開による実行時間を比較したのが表-2である．用いたプログラムは，*secondl*，ハミング問題，素数列，論理回路シミュレーション[15]である．

6. おわりに

ストリームを扱う言語 Stella と，その三つの処理系について述べた．各処理方式には，それぞれ，長所と短所があるので，状況に応じて使い分ける必要がある．各処理方式の比較を表-3にまとめた．擬似並列処理は，使用上の制限が少ないので，Stella の標準的な処理系として利用できる．しかし，実行効率ではインライン展開に劣るので，インライン展開できる部分については，展開して実行するのが望ましい．

一般的には，擬似並列処理をプログラムの開発時に

表-3 各処理系の比較
Table 3 Overall comparison of the described methods.

処理方式	擬似並列実行	直接展開法	モデル化展開法
適用範囲	◎	△	△
変換コスト	◎	○	△
変換サイズ	◎	△	○
実行効率	△	◎	◎

使用し，プログラムが完成するとインライン展開を施し，高い実行効率を得るといった使い分けが必要である．本論文で述べたストリーム・プログラムの実行方式は，Stella だけではなく一般のストリーム・プログラムの記述言語に対して適用可能である．

われわれは，処理系の他に，Stella のプログラミング支援環境も，設計，開発している．この開発支援環境は，モジュールの設計，作成，結合，デバッグなどを図式表示を利用して画面上で対話的に行えるのが特徴である[14]．支援環境を使って，新たに一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ができる．

本論文で述べた三つの処理系の状況に応じた使用と開発支援環境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．

謝辞 本研究を進めるにあたり貴重な意見をいただいた筑波大学プログラミング言語研究室，OS 研究室，日本アイ・ビー・エム東京基礎研究所の方々に感謝します．直接展開法による処理系は鳥谷憲司氏（現(株)リコーソフトウェア研究所）が作成しました．また，多数の有益なコメントをいただいた査読者の方々に感謝いたします．

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, IBM J. Res. Dev, Vol. 19, pp. 12-25 (1975).
3) 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).
4) Dijkstra, E. W.: A Discipline of Programming, Prentice-Hall (1976).
5) Dod: Ada Programming Language, ANSI/MIL-STD-1815 A (1983).
6) Grune, D.: A View of Coroutines, ACM SIGPLAN Notices, Vol. 12, No. 7, pp. 75-81 (1977).
7) 萩野：Communication Sequential Process の検証，情報処理学会ソフトウェア基礎論研究会資料，No. 2 (1982).
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) INMOS Limited: occam 2 Reference Manual, International Series in Computer Science, Prentice-Hall (1988).
11) Jackson, M. A.: Principles of Program Design, Accademic Press (1975).
12) 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).
13) Kuse, K., Sassa, M. and Nakata, I.: Modelling and Analysis of Concurrent Processes Connected by Streams, J. Inf. Process., Vol. 9, No. 3, pp. 148-158 (1986).
14) 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, pp. 1249-1253 (1986).
15) 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).
16) 久世：ストリーム・プログラムのインライン展開，情報処理学会プログラミング言語研究会資料 No. 15 (1988).
17) 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).
18) Nakata, I. and Sassa, M.: Programming with Streams, IBM Research Reports, RJ 3751 (43317) (Jan. 1983).
19) 中田，佐々：ストリームによるプログラミング，情報処理学会第25回プログラミングシンポジウム，pp. 124-135 (1984).
20) Peterson, J. L.: Petri Net Theory and the Modeling of Systems, Prentice-Hall (1981).
21) Ritchie, D. M. and Thompson, K.: The UNIX Time-Sharing System, Comm. ACM, Vol. 17, No. 7, pp. 365-375 (1974).
22) Shapiro, E. and Takeuchi, A.: Object Oriented Programming in Concurrent Prolog, New Generation Computing, Vol. 1, No. 1 (1983).
23) 田中：関数型プログラムにおけるストリーム計算，情報処理，Vol. 29, No. 8, pp. 836-844 (1988).

(平成元年8月31日受付)