Vol. 23 No. 3 情報処理学会論文誌 May 1982

実行効率のよい順路式の実現方法†

土居範久†† 久良知健†††

順路式 (path expression) は，共用対象に対して可能な実行の履歴 (history) を指定することにより，並行型プロセス間の同期をとろうとするものである．

順路式の記述能力，並行型プログラムの検証等に及ぼす効果に関する研究もまだ十分行われていないが，並行型プロセスの同期を静的に指定しようとする試みは高く評価されている．しかし，その実現方法に関しては，実行効率のよい実現方法の研究は，あまり行われていない．

本論文では，実行効率と記述能力を考慮した順路式およびその実現方法を提案し，提案した実現方法の妥当性を検討するために作成した並行型プログラミングシステムについて述べる．実現方法としては，計数型セマフォに対する *P-V* 命令および *await-cause* 命令を用いる．単純順路式を機能面でかなり拡張しても，実行効率のよい実現方法があることが確かめられた．この方法は，一般順路式を実現する際に，最適化の方法の一つとして用いることも可能である．

1. はじめに

順路式 (path expression) は R. H. Campbell と A. N. Habermann によって提案されたもので，**共用対象** (shared object) に対して可能な実行の**履歴** (history) を指定することにより，**並行型プロセス** (concurrent process) 間の同期をとろうとするものである[1)~3)]．これは，同期命令をプログラムの中に埋める代りに，対象の定義の一部として，共用対象への操作の実行可能な順序を指定することによって行う．順路式に指定した操作の実行は，操作の実行履歴が，その操作の実行を許すまで遅らされる．

順路式は，共用対象への操作を順次にする必要性はその共用対象の使用者の性質ではなく，その対象自身の性質であるという考え方に基づいている．したがって，順路式は，対象の使用者であるプログラムの中ではなく，対象の定義の中に置くことになる．すなわち**型定義** (type definition) のデータ部分に書く．そこで，順路式は，その対象を表現しているデータ記録の一部になる．

順路式の定義は，その型として宣言されたすべてのデータ対象に対して一様である．そして，その型の対象は，おのおのその順路の**実体** (instance) をもつことになる．この実体は，その対象が新しく宣言されるたびに作り出される．したがって，順路式は，その型の個々の対象に対する操作の列を制御するものであって，その型の対象の全集合に対する操作の列を制御するものではない．

現在までに考案されている順路式をバッカス・ナウア記法風で定義すると，およそ次のようになる†．

〈条件型要素〉::="[" 〈条件〉：〈式〉{，〈条件〉：〈式〉}* {，〈式〉} "]"

〈数値型要素式〉::="(" 〈式〉{"−" 〈式〉}+ ")" "↑" {〈正整数〉}

〈順路要素〉::=〈操作名〉|〈条件型要素〉|〈数値型要素〉|"{" 〈式〉 "}"|"(" 〈式〉 ")"

〈繰返し要素〉::=〈順路要素〉{"＊"}

† $\{x\}$ は x がなくてもよいことを，$\{x\}^*$ は x を0個以上繰り返し続けることを，$\{x\}^+$ は x を一つ以上繰り返し続けることを，$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}$ は x か y か z のどれか一つであることを，それぞれ意味する．

† An Effective Implementation of Path Expressions by NORIHISA DOI (Institute of Information Science, Keio University) and KEN KURACHI (Bank of Tokyo).

†† 慶応義塾大学情報科学研究所
††† 東京銀行

〈因子〉::=〈繰返し要素〉{“//”〈繰返し要素〉}*

〈項〉::=〈因子〉{“;”〈因子〉}*

〈式〉::=〈項〉{{“+”|“>”|“<”}〈項〉}*

〈順路式〉::=**path**〈式〉{“&”〈式〉}* **end**

〈多重順路〉::=〈順路式〉{〈順路式〉}+

綴り記号 **path** と **end** でくくって，操作の可能な列を指定したものを**順路** (path) という．一つの順路に指定された操作は一度に一つしか実行できない．すなわち一つの順路に指定された操作は，自動的に，その順路に特定の**際どい部分** (critical section) に置かれる．

path, end による括弧対は陰にクリーネの星印 (Kleene star) を意味し，順路の終りに達したら，再び最初からその順路に入ることができる．

演算子；は**逐次演算子** (sequential operator) であり，たとえば，

path p ; q **end**

は，$pqpqpq\cdots$の順でしか操作 p, q は実行され得ないことを意味する．；は省略することもできる．

演算子＋は**排他的選択** (exclusive selection) を表わし，たとえば，

path f ; $(g+h)$; k **end**

は，fの実行とkの実行の間で，gかhを1回だけ実行しなければならないことを意味する．演算子＞および＜は，排他的選択に対し優先度を指示するためのもので，たとえば，

path f ; $(g>h)$; k **end**

は，fの実行後gとhがともに実行するよう要求されたときには，gを優先することを意味する．

演算子＊はクリーネの星印で，次に続く操作に移る前に＊の直前の操作を0回以上実行できることを意味する後置演算子である．たとえば，

path p ; $(q$; $r)$＊ ; s **end**

は，pの実行とsの実行の間で，q ; r の列を0回以上実行できることを意味する．

数値型要素 (numerical element) は，共用対象への実行順序はさして問題ないが，操作間の実行回数を制限するためのもので，たとえば，

path $(p-q)\uparrow n$ **end**

は，

$$n \geq \#(p)-\#(q) \geq 0$$

を意味する．ここで，$\#(p)$ および $\#(q)$ はそれぞれpおよびqの実行回数を表わす．これは，$(p+q)$という排他選択に，実行回数に関する制限を与えたものである．〈正整数〉の省略時解釈の値は∞である．

条件型要素 (conditional element) は，条件としての論理式が真である最左端の要素が選ばれることを意味する．最後の条件の付かない要素があるときには，その前の条件がどれも偽のときに選ばれる．

{ } は**並行型実行** (concurrent execution) を意味する．たとえば，

path $\{r\}+w$ **end**

は，いくつもの並行したrか，wかを許すことを意味する．他のプロセスがrを実行している限りrを開始できるが，すべてのrの実行が終わってからでないとwを開始することはできない．

//も並行型実行を意味する．たとえば，

path s ; $(p\,//\,q)$; t **end**

は，sの実行とtの実行の間で，pとqを両方とも実行しなければならないことを意味する．ただし，実行順序は規定しない．pとqを並行して実行してもよい．

順路をいくつも定義したとき**多重順路** (multiple path) という．順路が要素を共用していなければ，それらの順路は独立である．多重順路で共用していない操作を並行して実行可能なとき**並列順路** (parallel path) という．たとえば，

path p ; r **end**

path q ; r **end**

は，pとpの間はrで，qとqの間もrで，rとrの間はpとqとで分離することを指定しているが，pとqの実行順序は規定しない．pとqは並行して実行することも可能である．

＆は二つ以上の順路をつなげて一つの順路にする働きをもつ．＆でつなげた順路を**結合順路** (connected path) という．したがって，

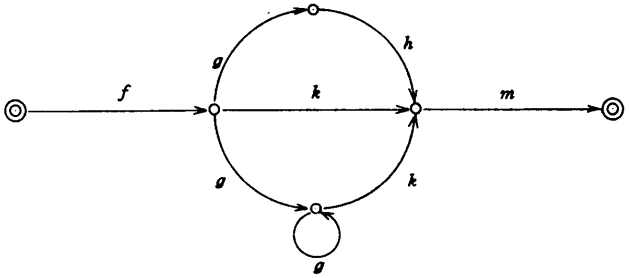
path p ; r & q ; r **end**

としたときには，pとqの実行順序は規定しないが，並行して実行することは許されない．

以上に定義したものを**一般順路式** (general path expression) という[1),3)]. これに対し，条件型要素も数値型要素も用いず，さらに演算子を＊，；および＋に限定したものを**正則順路式** (regular path expression) という[4)]. 正則順路式は有限状態機械を表わすグラフモデルに変換できる. グラフのアークが操作を表わし，節点が状態を表わす. たとえば，

path f ; (g ; h+g＊ ; k); m **end**

は次のように表現できる.



このとき，どの二つのアークも同じ名前をもたず，しかも操作の開始状態と終了状態が一意であるものを**単純順路式** (simple path expression) という[3)]. 後者を保証するためには，繰返し要素は逐次演算子にはさまれていて，しかもその前後が繰返し要素であってはならないという制限がつく. 正則順路式および単純順路式では，並行処理および操作の実行回数を規定することができないだけでなく，多少複雑な同期関係を記述しようとすると，名義的な手続きを用意する必要が生じむずかしくなる.

一般に，順路式は，状態変数，ロック機構および順路式中の各操作に付属する**プロローグ** (prologue) と**エピローグ** (epilogue) とで実現できる[3)]. プロローグの機能は，対象をロックし，状態を調べ，その操作が実行できるか否かを決定することである. 実行可能ならそのまま先に進み，実行不能ならその操作を実行しようとしたプロセスを眠らし，対象のロックをはずす. エピローグの機能は，状態を変更し，眠っているプロセスのうち実行可能なものがあれば，それらのうちの一つを選び実行させる. 実行可能なものがなければ，対象のロックをはずす.

単純順路式は，プロローグとしてP命令を，エピローグとしてV命令を用いることにより，P-V命令によって実現できる[3)]. R. H. Campbell と A. N. Habermann は；，＋，{ } が P-V 命令で実現できることを示した[2)]. 正則順路式は，P-V 命令では実現できず，状態変数として使用状況 (mode)，状態 (state) および待ち行列を用意し，各状態に対し後者ベクトル (successor vector)，各操作に対し許可ベクトル (permission vector) を設ければ実現できることを A. N. Habermann が示した[4)]. S. Andler は数値型要素および条件型要素を**述語** (predicate) としてまとめた**述語順路式** (predicate path expression) を提案し，正則順路式の実現方法に基づいて実現できることを示した (ただし，並行型要素 { } (述語順路式では後置演算子 ♯) は一つの操作にしか適用できない等の制約がいくつかある)[5)]. なお，述語順路式に対して，従来の順路式を**純順路式** (pure path expression) と呼ぶ[5)]. また，R. H. Campbell は，逐次演算子，排他選択，並行型実行と，順路およびその一部分を並行して実行できる回数を指定できる順路式を提案し**オープン順路式** (open path expression) と呼び，操作だけでなくプロセスも指定できるようにしている[6)]. オープン順路式は P-V 命令で実現されている.

本論文では，純順路式としてこれまでに提案されている演算子をすべて含めても，それらの使用を一部制限するだけで，P-V 命令および *await-cause* 命令[8)]を用いて実行効率のかなりよい実現方法があることを示す. また，提案した実現方法の妥当性を確認するために作成した並行型プログラミングシステムについて述べる. これは，順路式の実現方法および順路式の効果を調べるために作成したもので，Pascal の上に実現されており，言語，Pascal へ変換するための前処理系および並行型プロセスを擬似的に実現するための核から成っている.

2. 拡張単純順路式

P-V 命令および *await-cause* 命令を用いることで実行効率のよい実現が可能な純順路式を**拡張単純順路式** (extended simple path expression) と呼ぶ[7)]. 拡張単純順路式は，単純順路式を拡張したもので，一般順路式の部分集合であり，一般順路式に次の制約を加えたものである.

(1) 操作名は，一つの型定義中の順路式に一度しか現われてはならない.

(2) 演算子＊の用い方は，単純順路式に準ずる.

(3) { } による並行型要素の中に条件型要素を用いることはできない. また，条件型要素の中に { } による並行型要素を用いるときには，{ } 内の操作によって条件の被演算子の値を変更してはならない.

(4) // による並行型要素の被演算子の一つが条件型要素であるときには，他の被演算子の中の操作で，条

表 1　単純順路式，一般順路式と拡張単純順路式の比較
Table 1　Comparison of path expressions.

	記述のしやすさ	使用できる演算子	一つの順路式に記述できる同一操作名の数	制限	あいまいさ
単純順路式	難	+；*	1	*の使用箇所	なし
拡張単純順路式	↓	すべて	1	*および並列型要素と条件型要素の組合せ	なし
一般順路式	易	すべて	任意	な し	あり

件の被演算子の値を変更してはならない．また，条件型要素内に // による並行型要素を用いるときには，// の被演算子の操作で，条件の被演算子の値を変更してはならない．

これらの効果は次の通りである．

（1）　一般順路式のもつあいまいさがない．

（2）　単純順路式に比べ，すべての演算子および要素が使えるため飛躍的に記述能力が高まる．

（3）　*P-V* 命令および *await-cause* 命令で実現することで効率よく実現できる．

単純順路式では，ある操作の終了状態は，実行した操作によって一意に定まるのに対し，拡張単純順路式では，実行した“要素”の終了状態が，実行した“要素”によって一意に定まる．ここで，“要素”とは，条件型要素，数値型要素，{ } による並行型要素，演算子>，<，// によって結合された要素，および演算子>，<，// の直接の被演算子ではない操作名である．たとえば，順路式

path $(p \,//\, q)$; r **end**

では，$p \,//\, q$ および r が，それぞれ“要素”である．ここで，p の終了状態は q の実行の有無に依存するが，$p \,//\, q$ の終了状態が r の開始状態になる．一般順路式では，ある“要素”を選んでも，その終了状態は必ずしも一意に定まらない．たとえば，次のような多重順路では，$(p \,//\, q)$ を終了しても，s の終了の有無によって状態が異なる．

path $(p \,//\, q)$; r **end**

path s; r **end**

単純順路式，一般順路式と拡張単純順路式を比較すると**表 1** のようになる．

3. 拡張単純順路式の実現方法

プロローグおよびエピローグとしては，計数型セマフォに対する *P-V* 命令，*await-cause* 命令およびいくつかの単純な関数を用いる[7]．これらのプロローグおよびエピローグは，構文解析時に，拡張単純順路式の“要素”，すなわち部分式およびおのおのの操作，に対して決定する．以下では，部分式に割り当てたプロローグおよびエピローグを] および [で表わす．また，$S, S_1, S_2, \cdots$ でセマフォを，$C, C_1, C_2, \cdots$ でカウンタを表わす．これらのセマフォやカウンタは，型定義の**実体**が作り出されるときに，実際に，割り当てる．セマフォおよびカウンタの初期値は**括弧** [ ] 内に示す．さらに，英字の小文字で部分式または操作名を表わす．

拡張単純順路式を変換するための規則は以下の通りである．

（1）　**path** f **end** ⇨ $P(S)$; f; $V(S)$ $[S=1]$

（2）　$]f$; $g[$ ⇨ $]f$; $V(S)$, $P(S)$; $g[$ $[S=0]$

（3）　$]f+g[$ ⇨ $]f[$, $]g[$

（4）　$]f$; $g*$; $h[$ ⇨ $]f$; $V(S)$, $P(S)$; g; $V(S)$, $P(S)$; $h[$　　$[S=0]$

（5）　$](f-g)\uparrow n[$ ⇨ $P(S1)]f[V(S2)$, $P(S2)]g[V(S1)$　　$[S1=n,\ S2=0]$

数値型要素の被演算子は，部分式であってもよい．

（6）　$]\{f\}[$

⇨ $P(S)$; **if** $FIRST^{(\,)}(C)$ **then**] **fi**; $V(S)$; f $V(S)$; **if** $LAST^{(\,)}(C)$ **then** [**fi**; $P(S)$;

$[S=1,\ C=0]$

ここで，論理関数 $\mathrm{FIRST}^{(\,)}$ および $\mathrm{LAST}^{(\,)}$ の定義は次の通りである．

function $FIRST^{(\,)}$ (**var** C : *counter*) : *Boolean*;
　begin $FIRST^{(\,)} \leftarrow (C=0)$; $C \leftarrow C+1$ **end**
function $\mathrm{LAST}^{(\,)}$ (**var** C : *counter*) : *Boolean*;
　begin $C \leftarrow C-1$; $LAST^{(\,)} \leftarrow (C=0)$ **end**

S に対する *P-V* 命令で，カウンタの更新の排他性を保証する．最初に f の実行を要求したプロセスだけが] を実行し，以後に続く要求では] を実行しない．同様に，最後のプロセスだけが [を実行する．

（7）　$]f \,//\, g[$

⇨ **if** $FIRST''(C_f, C_g, S)$ **the**] $V(S_g)$ **else** $P(S_f)$ **fi**　f　**if** $LAST''(C_f, C_g, S)$ **then** [**fi**

if $FIRST''(C_g, C_f, S)$ **then**] $V(S_f)$ **else** $P(S_g)$ **fi**　g　**if** $LAST''(C_g, C_f, S)$ **then** [**fi**

$[S_f=S_g=0,\ S=1]$

ここで，C_f および C_g の構造は次の通りである．

type cc=**record** $ACTIVE, END$: *counter* **end**

各フィールドの初期値は0である．論理関数 $FIRST''$ および $LAST''$ の定義は次の通りである．

```
function FIRST'' (var C1, C2 : cc ;
                  var S : semaphore) : Boolean ;
  begin
    P(S) ;
    C1.ACTIVE←C1.ACTIVE+1 ;
    FIRST''←(C1.ACTIVE>C2.ACTIVE) ;
    V(S)
  end
function LAST'' (var C1, C2 : cc ;
                 var S : semaphore) : Boolean ;
  begin
    P(S) ;
    LAST''←(C1.END<C2.END) ;
    C1.END←C1.END+1 ;
    if LAST'' then
      C1.ACTIVE←C1.ACTIVE−1 ;
      C1.END←C1.END−1 ;
      C2.ACTIVE←C2.ACTIVE−1 ;
      C2.END←C2.END−1
    fi
    V(S)
  end
```

セマフォ S で，カウンタの引用の排他性を保証する．S_f および S_g は，それぞれ f および g と結びついたセマフォである．f または g の最初に要求された方で ‖ を実行し，他方を実行可能にする．要求が最初でなければ，私用型セマフォに対する P 命令を実行し，他方の V 命令が実行されるまで待つ（これは，最初の要求が ‖ を実行中に，他方への要求があった場合にそなえた処置である）．最初の要求か否かを調べるのに論理関数 $FIRST''$ を用いる．‖ は，f と g のうち最後に実行された方で実行する．カウンタの値は相対的な意味しか持たない．双方の実行後には，その履歴を忘れることが可能なことから，双方のカウンタを1ずつ減らす．カウンタを上述のような構造にすることで，// を入れ子にしたり，{ } の中に // を用いることが可能である．$f\,//\,g\,//\,h$ など，3項以上に適用された場合にも容易に拡張できる．

(8) $\|f>g\| \Rightarrow \| f \|,\ P(S) \| g \| V(S)\ [S=1]$

これは，二つの操作がほぼ同時に要求されたときに，それらの操作の優先順位を P-V 命令で解決するための標準的な方法である．ほぼ同時の要求のうち優先順位の高い方の要求が，双方に共通のプロローグ ‖ で封鎖されない場合以外は，到着順に処理するという解釈のもとで＞を実現している．きわめて頻繁に要求される相互排除操作に優先順位をつける必要があるような場合がもしあれば，そのときに限り問題となる場合がある．

(9) $P(sem)\ [b_1 : f,\ b_2 : g] \|$

```
⇨ P(sem) ; while not b1 do
    V(sem) ; await (E1) od ; f ‖,
    P(sem) ; while b1 and (not b2) do
    V(sem) ; await (E2) od ; g ‖
```

そして，b_1 および b_2 の値を変更する操作のエピローグの最初の部分に，

if b_1 **then** *cause* (E_1)
else if b_2 **then** *cause* (E_2) **fi fi**

を挿入する．

ここで，E_1 および E_2 は，セマフォ *sem* に結び付けた事象列である．事象列を一つにすることは可能である．同一の順路に演算子＋をはさんで条件型要素がある場合には，それぞれに別個の事象列を用意すると飢餓状態が生じ得るので注意する必要がある．ここでは，b_1 および b_2 の条件は，それぞれ部分式 f および g の全体にかかるものとしている．この方法は，条件が三つ以上の場合にも可能である．

条件型要素のプロローグが単一の P 命令でない場合，すなわち，

$\|[b_1 : f,\ b_2 : g] \|$

であって，‖ が単一の P 命令でない場合には，条件の評価の結果がすべて偽であったとき，環境を元に戻す必要があるので，我々の方法では実現は不可能である（**4.** 参照）．

(10) 演算子 & は，同一の操作名が同一の順路式には2回以上現われないという条件から，演算子＋と同一に扱うことができる．

(11) 多重順路も同様の理由から，別個に処理できる．

4. 実現上の問題点

順路式に関する記述能力上および実現上での問題点の多くは，動的なものを静的に捉えようとすることに起因するものと思われる．この章では，特に，条件型

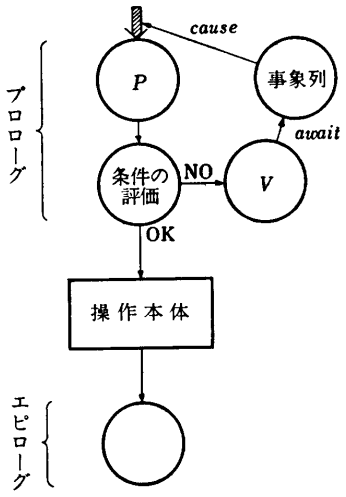


図 1 条件型要素の実現方法
Fig. 1 Implementation of conditional elements.

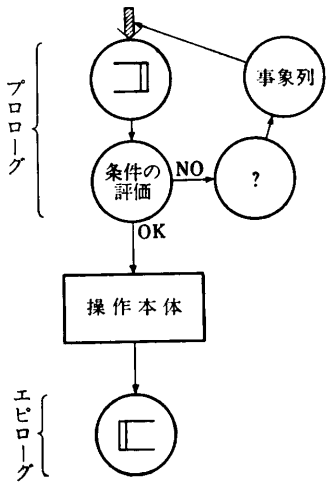


図 2 制約を付けない条件型要素の実現方法
Fig. 2 How to implement unrestricted conditional elements?

要素および { } による並行型要素に対する我々の解釈の仕方および条件型要素を置ける位置に制約をつけた理由について述べる.

4.1 条件型要素の実現上の問題点

まず,

path [i=0 : p; q, r] **end**

という順路式で, i=0 という条件は (p; q) にかかるという解釈をとっている.

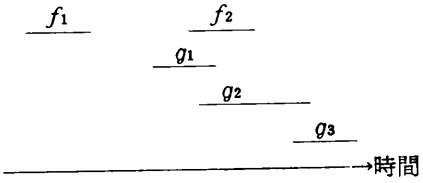
次に, 条件型要素は, そのプロローグが単一のP命令のときしか,我々の方法では実現できない. これは, 使用上, 制約を与えることになる. この理由は, 実現方法のところで簡単に述べたが, 詳しくは次の通りである. 我々の実現方法を図示すると**図1**のようになる. 条件を評価した結果がすべて偽の場合には, V命令を実行することにより環境を元に戻し, 再びプロローグを始めから検査することが可能である. プロローグが単一のP命令でない場合にも, これが保証されなければならない(**図2**参照). そのためには, あらゆるプロローグ操作の逆操作を用意すればよい. しかし, それは, 一般順路式の実現方法と比べたとき, 手間をかけたところで実行効率が向上するとは思えないので魅力がない. さらに, 一般には, 複合化した操作の結果を元に戻すことは困難であろう.

4.2 { } による並行型要素の実現上の問題点

{ } による並行型要素には, あいまいさが含まれる[2]. たとえば, 次のような順路式を考える.

path {f ; {g}} ; h **end**

次のようなタイミングでfおよびgが実行されたものとしよう.



g_1 および g_2 は明らかに f_1 に続く {g} であるが, g_3 は f_1 に続くものなのか, f_2 に続くものなのかが明確でない. 我々は, g_3 は f_1 に続くものと解釈し, あいまいさを取り除いている.

5. 実験用システム

拡張単純順路式の実現方法および記述能力の研究および確認のために, Pascal をもとに, 実験用システムを作成した[7]. これは, 順路式をもつ並行型プログラミングシステムであり, 並行型プロセスを擬似的に実現する核を含んでいる.

5.1 言語仕様

Pascal を基礎として, 次のような機能を導入している.

(1) ブロック構造

(2) **parbegin-parend**

parbegin と **parend** の間にプロセス名を記述することによって, プロセスを並行に走らせることが可能である. 親プロセスは, すべての子プロセスが終了するまで待つ.

(3) **decl** ブロック

decl ブロックによって, 型定義の実体を宣言する(**図4参照**). 型の外に**輸出** (export) する操作は,

procedure および **function** で定義し，輸出しない操作は，それらの前に **hidden** を付ける．**decl** ブロックは入れ子にすることができる．

（4） プロセスの宣言

process 〈プロセス名〉; に続けて，ブロックによりプロセス本体を記述する．プロセスは再帰的に生成できる．

5.2 プリプロセッサ

プリプロセッサは約 2,500 文から成る Pascal プログラムで，上述の言語を用いて書いたプログラムを Pascal プログラムに変換するとともに，擬似的な並行型プロセスを実現するための核の主要部分を，変換したプログラムに併合するためのものである．主な機能は次の通りである．

（1） 保護すべき名前を一意的な名前に変換する．

（2） **decl** ブロックを複数の手続きと関数に変換する．

（3） **process** を手続きに変換し，実行文の最後に terminateprocess[8] を挿入する．

（4） **parbegin** … **parend** は，initiateprocess[8]，delayprocess[8] に変換する．

（5） 順路式に従って，プロローグおよびエピローグを挿入する．

処理の過程の概略を**図 3** に示す．

5.3 核

Pascal 内で，擬似的に並行型プロセスを実現するために，**核** (nucleus) を導入した．核の設計および構造は SOS[9] に基づいているが，多少の機能の追加，削除を行っている．核の大半は，**decl** ブロックで記述しプリプロセッサで Pascal に変換している．ごくわずかのものはアセンブリ言語で記述し，最終的に連係編集プログラムで Pascal コンパイラからの出力と連係をとらせている．核の提供する機能は，*wakeme* を除いて，使用者プログラムで使用することはできない．これらの機能は，プリプロセッサの作り出す目的プログラムで引用されることになる．*wakeme* は，ミリ秒単位でプロセッサを放棄するための命令である．

Pascal 内で擬似並行処理を行うために，核では次のようなことを行う．

（1） 実行待ち状態のプロセスがないとき，内部の時間を論理的に進めて，*wakeme* により眠っているプロセスを起す．

（2） オペレーティングシステムの提供する計時割込みは使用せず，擬似的な計時割込みを用いている．

（3） 各プロセスに固有の領域を確保するために，実行時に Pascal のヒープスタック領域を変更し，各プロセスに別個のヒープスタック領域を割り当てる．(アセンブリ言語で作成した核の部分で行う．レジスタ等の環境の変更も同時に行う．)

核の概要を**表 2** に示す．

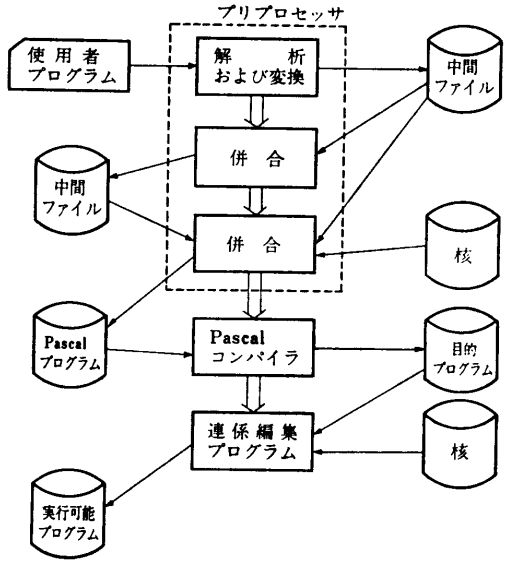


図 3 処理の過程の概略
Fig. 3 System flow of experimental system.

表 2 核の概要
Table 2 Outline of system nucleus.

レベル	抽象化	文の数	アセンブリ言語の行数
0	プロセス	2	100
1	プロセスの基礎システム	50	0
2	擬似仮想プロセッサ	123	0
3	協同型システム	94	0
		計 269	計 100

5.4 Pascal 処理系

Pascal 処理系は，慶應義塾大学情報科学研究所で作成した分割コンパイルが可能な UNIVAC 1100 用 Pascal を使用した．この Pascal 処理系に，次のような機能を追加している．

（1） ミリ秒単位で時刻を戻す関数 *clock*

（2） 引数として与えた手続きの実行時の開始番地を戻す関数 *loc*

（3） アセンブリ言語との連係を可能にするための external 文の追加

5.5 実 例

C. A. R. Hoare の解法[12]を用いた五人の哲学者の食

```
parprogram< 6 > ;        { five dining philosophers }
                         { they are thinking in their own rooms }
                         { and enter the dining room to eat.    }
  type pno = 0 .. 4 ;

  decl diningroom ;
    path [ enterroom - leaveroom ] @4 end ;
      procedure enterroom ; begin end ;
      procedure leaveroom ; begin end
  enddecl ;

  decl fork0 ;
    path pickup0 ; putdown0 end ;
      procedure pickup0  ; begin end ;
      procedure putdown0 ; begin end
  enddecl ;

  decl output ;
    path print end ;
      procedure print( i:pno ; b:boolean ) ;
      begin
        write( 'time=', clock:6, ' ':i*8+2,
               '< ', i:1, ' >') ;
        if b then
          writeln( ' start eating. ' )
        else
          writeln( ' end eating. ')
      end
  enddecl ;

  procedure eat ;
  begin wakeme( trunc( random( 1 )*100.0 ) ) end ;

  procedure think ;
  begin wakeme( trunc( random( 1 )*100.0 ) ) end ;

  process philosopher0 ;
  begin
    while true do
    begin
      think ;
      enterroom ;
        pickup0 ; pickup1 ;
          print( 0, true ) ;
          eat ;
          print( 0, false ) ;
        putdown0 ; putdown1 ;
      leaveroom
    end
  end ;

  begin
    parbegin
      philosopher0 ;
      philosopher1 ;
      philosopher2 ;
      philosopher3 ;
      philosopher4
    parend
  end.
```

(a) 五人の哲学者の食事問題

(a) Five dining philosophers' problem.

```
TIME=  5902  < 0 > START EATING.
TIME=  5916                          < 3 > START EATING.
TIME=  5928                          < 3 > END EATING.
TIME=  5943                  < 2 > START EATING.
TIME=  5959  < 0 > END EATING.                        < 4 > START EATING.
TIME=  5976                  < 2 > END EATING.
TIME=  5996          < 1 > START EATING.                        < 4 > END EATING.
TIME=  6006                                        < 3 > START EATING.
TIME=  6021          < 1 > END EATING.
TIME=  6039  < 0 > START EATING.
TIME=  6055  < 0 > END EATING.
```

(b) 実行結果

(b) Results of execution.

図 4

Fig. 4

事問題に対するプログラム例および実行結果を図4に示す.(*fork* 1～*fork* 4 に対する **decl** および *philosopher* 1～*philosopher* 4 に対する **process** の宣言は省略してある. また, 順路式中の@は, 数値型要素の↑である).

6. むすび

本論文では, 実行効率と記述能力を考慮した順路式およびその実現方法を提案し, 提案した実現方法の妥当性を検討するために作成した並行型プログラミングシステムについて述べた. 機能面で単純順路式をかなり拡張しても, 実行効率のよい実現方法があることが確かめられた. この方法は, 一般順路式を実現する際に, 最適化の方法の一つとして用いることも可能である. 代表的な同期問題は, 拡張単純順路式で記述できる.

順路式は, いくつかの制約があるにしても, かなりの面で十分使用可能と思われる. さらに, 並行型プログラムの仕様記述言語へ適用すると, きわめて都合がよいことも確かめられている[10),11)]. そのためにも, 一般順路式の効率のよい実現方法および並行型プログラムの検証等に対する順路式の効果が, 今後の研究の課題であろう.

謝辞 本稿をまとめるに際し, 査読者の的確なご指示に感謝する.

参考文献

1) 土居範久: 順路式, 情報処理, Vol. 19, No. 8, pp. 779-787 (1978).
2) Campbell, R. H. and Habermann, A. N.: The Specification of Process Synchronization by Path Expression, Lecture Notes in Computer Science, Vol. 16, Springer Verlag (1974).
3) Habermann, A. N.: Path Expression, Department of Computer Science, Carnegie-Mellon University (1975).
4) Habermann, A. N.: Imple-

mentation of Regular Path Expressions, Department of Computer Science, Carnegie-Mellon University (1979).
5) Andler, S.: Predicate Path Expressions: A High-level Synchronization Mechanizm, Department of Computer Science, Carnegie-Mellon University (1979).
6) Campbell, R. H.: A Path Pascal Language, Department of Computer Science, University of Illinois (1978).
7) 久良知健：順路式とその実現について，慶応義塾大学大学院工学研究科修士論文 (1978).
8) Brinch Hansen, P.: Operating System Principles, Prentice-Hall (1973) [田中穂積他訳：オペレーティング・システムの原理，近代科学社 (1976)].
9) 土居範久：ALGOL あるいは FORTRAN 内でコオペレーティングプロセスを実現するためのオペレーティングシステム SOS，情報処理，Vol. 16, No. 1, pp. 7-14 (1975).
10) Hirose, K., Saito, N., Doi. N. et al.: Process-Data Representation, Proc. of 3rd UJCC, pp. 225-230 (1978).
11) Hirose, K., Saito, N., Doi, N. et al.: Specification Technique for Parallel Processing: Process-Data Representation, Proc. of NCC '81, pp. 407-413 (1981).
12) Hoare, C. A. R.: Communicating Sequencial Processes, Comm. ACM, Vol. 21, No. 8, pp. 666-677 (1978).

(昭和 55 年 9 月 12 日受付)
(昭和 56 年 12 月 17 日採録)