データ体内時計を用いた並行プロセス同期方式†

福 岡 和 彦††

データの新しさを考慮した，新しい並行プロセス同期方式を開発した．従来の並行プロセス同期方式のなかでは，データ駆動方式が最も進んでいる．しかし，データ駆動方式は，共用データをもつ実時間システムには適用できなかった．そこで，各データに，体内時計と名づける時刻属性を与え，同一の共用データであっても，体内時計の時刻が違えば別データであるとみなす方式を考案した．この体内時計が，各共用データをフローデータの列に置き換える役目を果たすため，共用データをもつ実時間システムでも，データ駆動型で，並行プロセスの動作を制御できるようになった．データ駆動型にしたことにより，プログラムの内部に，同期制御や排他制御のための手続きを記述する必要がなくなり，また，決定性のあるシステムを容易に構築できるようになった．

1. まえがき

実時間システムでは，応答性を高めるため，多数のプロセスを並行に動作させる．しかし，データを共用する二つのプロセスを並行に動作させようとすると，両者の動作タイミングがわずかにずれるだけで，動作結果が変わってしまうおそれが生じる．つまり，システムが決定的でなくなってしまう．

決定性のないシステムには，実行順序に関する誤りが起きやすいとか，その誤りを事前に検出することがむずかしいとか，誤りに再現性がないため修正がむずかしいなどの，ソフトウェア信頼性，保守性に関する重大な欠点がある．応答性を多少犠牲にしたとしても，決定性の有るシステムを構築することが重要である．

実時間システムに決定性をもたせるためには，データを共用するプロセスの動作順序を制御し，共用データの操作順序を制御する必要がある．そのための制御機構は，すでにいくつか提案されている．

Hansen と Hoare のMonitor[1),2)] は，共用データが複数のプロセスで同時に使用されることを排除する機構である．しかし，共用データの操作順序を制御する機構をもっていないため，システムの決定性を保証する手段にはなりえない．

Campbell と Habermann の Path Expressions (順路式)[3)] は，プロセスの動作順序を定義させる機構を備えている．しかし，順路式で定義されたプロセスの動作順序が，共用データに対する正当な操作順序を反映しているかどうかは，保証の限りでない．

Hoare の CSP[4)] は，共用データが元来，任意の操作順序を受け入れてしまう性質をもっている点に着目して，共用データを使わずに，プロセス間フローデータ（プロセス間で送受信されるメッセージ）だけで実時間システムを表現させようとする方式である．この方式では，データが生成された時点で，そのデータを参照するプロセスが起動され，そのプロセスにそのデータが渡される．これはいわゆるデータ駆動[5)] であり，プロセスの動作順序とデータの操作順序が自動的に一致する．そのため，決定性のあるシステムを容易に構築することができる．

しかし，それは，フローデータだけでシステムを表現できることを前提にした話である．実際の大多数の実時間システムには，何らかの状態を保持するプロセス間共用データが存在する．

そこで，共用データの存在を許したまま，なおかつデータ駆動型でプロセスを起動する方式を考案した．この方式の最大の主張点は，共用データを時間の関数とみなす点，つまり，1個の共用データは時間軸上に並んだ無限個のフローデータと等価である，と考える点にある．これにより，プロセス間共用データに対しても単一割当規則[6)] が適用できるようになり，データ駆動が可能になり，共用データをもつ実時間システムの決定性を保証できるようになる．

この方式には，RHAPSODY (Rhythmical Applicative Software Dynamics) という名前を付けている．

2. 並行プロセスの課題

2.1 実時間システム

まず，本報告の考察対象である実時間システムについて述べる．

† Concurrent Process Synchronization Mechanism with Data Interior Clocks by KAZUHIKO FUKUOKA (Systems Development Laboratory, Hitachi, Ltd.).

†† (株)日立製作所システム開発研究所

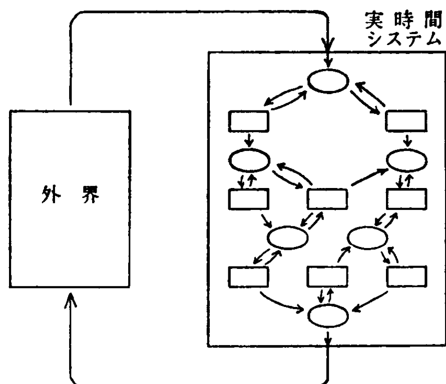

図1 実時間システムのソフトウェア構造
Fig. 1 Software structure of a real time system.

実時間システムは，複数のプロセス（プログラムを逐次的に実行する過程）と，複数のデータ（プロセスの動作結果）から構成される．各プロセスは，他プロセスによって生成された1個またはそれ以上のデータを参照して，他プロセスによって参照されるであろう1個またはそれ以上のデータを生成する．

それらのプロセスは，互いに並行に動作できるが，それぞれが独立した別個の仕事をするのではなく，データの生成・参照の関係で互いに関連をもちながら，全体としてまとまった一つの仕事をする．

実時間システムは，それだけで閉じた世界を形成するのではなく，その外界と密接な関連をもつ．外界から実時間システムへ，外界の状態を表すデータが送り込まれる．データが送り込まれるたびに，実時間システム内の各プロセスが動作し，全体としての仕事の結果が外界へ送り出される．このシーケンスが，周期的にあるいはランダムな時間間隔で，無限に繰り返される．

実時間システムの内部構造の簡単な例と，外界との関係を表したのが，**図1**である．楕円がプロセス，矩形がデータを表し，矢印がデータの生成参照関係を表す．データの生成頻度，参照頻度を無視して，たんにプロセスとデータの生成参照関係だけを，二次元構造として平面的に表現している．

2.2 プロセス動作順序の問題

実時間システムを構築する上でのむずかしい問題の一つは，互いに並行に動作する複数のプロセスが，データを共用することから生じる．

あるプロセスが生成するデータを別のプロセスが参照する場合，そのデータが生成された時点で初めてそれを参照可能にするよう，プロセス動作順序を制御しなければならない（同期制御）．また，あるプロセスが参照または更新している途中のデータを，他のプロセスが更新しないよう，プロセス動作順序を制御しなければならない（排他制御）．

同期制御や排他制御に必要な手続きを，プログラムの途中に挿入することは，容易でない．その十分性を検証することはさらにむずかしい．プログラムの内部には，同期制御や排他制御のための手続きを記述しなくてよいようにしたい．

さらに，プロセスの動作途中での，そのプロセスに対する同期制御や排他制御を不要にしたい．あるプロセスの動作途中で，他のいかなるプロセスがいかなるタイミングで動作しようとも，そのプロセスは，動作開始直前のデータから一義的に定まるデータを，必ず生成するようにしたい．つまり，実時間システムに決定性をもたせたい．

この決定性が保証されれば，タイミングによって生じる実時間システムの複雑さを解消できる．

3. データ駆動による並行プロセス同期

3.1 プロセス動作順序の制御

プロセスの動作途中での，同期制御や排他制御を不要にするために，データ駆動で，プロセスの動作を制御することにした．ここで述べるデータ駆動とは，次の三つの条件を満たすプロセス動作制御方式を意味するものとする．

[条件1] あるプロセスが参照する予定のすべてのデータが，他プロセスによって生成され，参照可能な状態になったときに初めて，そのプロセスの動作開始を許可する（そのプロセスを起動する）．

[条件2] あるプロセスの動作開始から動作終了までの間は，そのプロセスが参照するどのデータをも，他のプロセスが更新・削除・追加することを許さない．

[条件3] あるプロセスの動作が終了した時点で初めて，そのプロセスが生成したすべてのデータを，他のプロセスが参照することを許す．

条件1と条件3より，プロセスの動作途中で，参照したいデータがまだ生成されていないので待つとか，データを生成した時点で，そのデータを参照しようとして待っているプロセスの待ちを解除するなどの，同期制御を行う必要がなくなる．また，条件2と条件3より，プロセスの動作途中で，排他制御を行う必要がなくなる．

3.2 データの新しさの問題

しかし，データ駆動にすれば，ただちにシステム決定性の問題が解決するわけではない．データ駆動は本来，プロセス間共用データ（何らかの状態を保持するデータ）が存在しない世界で成り立つ方式である．ところが，ほとんどの実時間システムには，共用データが存在する．共用データをデータ駆動に適応させる問題を，解決せねばならない．

（1） 繰返しの問題

共用データの存在を認めた上で，プロセスとデータの生成参照関係を二次元平面上に表現すると，共用データを介して必ず繰返しができてしまう．あるプロセスが生成するデータを，そのプロセス自身が参照するとか，それを別のプロセスが参照し，その別プロセスが生成するデータを元のプロセスが参照する，といったデータ生成参照関係に関する閉ループができてしまう．**図2**は，その繰返しの例である．

繰返しを好意的に眺めれば，あるプロセスは，自らが過去に生成したデータ（または，そのデータを参照する別プロセスが過去に生成したデータ）を参照して，そのデータを新しい内容に書き替える，と解釈できる．しかし，穿った見かたをすれば，自らがこれから生成しようとしているデータ（または，そのデータを参照して，将来生成されるであろうデータ）を参照して，そのデータを生成する，と解釈できる．これは，明らかに矛盾している．

この矛盾をなくすためには，従来のように，共用データをそれ以上分割できない1個の対象として捉えるのではなく，ある名前をもつ1個の共用データであっても，古いデータと新しいデータを，明確に区別して表す必要がある．

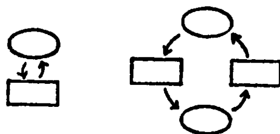

図 2 繰返しの例
Fig. 2 Examples of iteration.

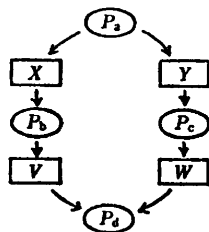

図 3 分岐合流の例
Fig. 3 An example of fork-join.

（2） 分岐合流の問題

分岐合流にも，繰返しと同様の問題がある．二次元平面上で，**図3**のように表現される分岐合流の形には，異なる2種類の意味付けが可能である．

一つは，データの新しさが一致しなければならない場合である．

P_a：データ X，Y を計算．

P_b：$V=X^2$ を計算．

P_c：$W=Y^2$ を計算．

P_d：$(V+W)^{1/2}$ を計算．

この場合，データ V と W は同一の新しさでなければならない．つまり，あるタイミングで動作したプロセス P_a が，組として生成したデータ X と Y から計算されたデータ V と W でなければならない．

もう一つは，データの新しさが一致する必要がない場合である．

P_a：外界状態 X，Y を入力．

P_b：X から，出力データ V を計算．

P_c：Y から，出力可否フラグ W を計算．

P_d：W が出力可ならば V を外界へ出力．

この場合，出力可否フラグ W は，出力データ V よりできるだけ新しいことが望ましく，決して同じ新しさである必要はない．なぜなら，P_b が出力データ V を計算中に外界状態 Y が変化し，計算開始時には出力可を示していた出力可否フラグ W が，計算終了時には出力不可を示しているかもしれないからである．

同じ分岐合流の形をしていても，データの新しさを一致させねばならない場合と，データの新しさを一致させる必要がない場合とがある．これらを区別するためには，データの新しさを明示する必要がある．

4. データ体内時計による並行プロセス同期

4.1 データの新しさの制御

実時間システムは，外界からデータを取り込み，そのデータを処理し，処理結果を外界に出力する．実時間システム内のどのデータも，外界から取り込まれたデータそのものか，またはそれが加工されたものである．そのことから，実時間システム内のどのデータも，直接的か間接的かの差はあるにせよ，外界の何らかの状態を反映しているといえる．

外界の状態は，時間の経過とともに，次々と変化していく．従来は，一つの外界状態に一つのデータを対

応させて，その外界状態の時間変化に追随して，そのデータを次々と更新していた．共用データとして扱われていたわけである．しかし，それでは，一つの外界状態に対する古いデータと新しいデータを区別できず，実時間システムの決定性を保証できない．

そこで，一つの外界状態に多数個のデータを対応させる方式を考えた．つまり，外界状態 D の時刻 t の内容を保持するデータを，$D(t)$ と表すことにし，t が異なれば別データである，と考えることにした．これは，一種の配列であり，古い時刻の状態を保持するデータから，最新の状態を保持するデータまで，順序づけられて並んでいる．

時刻 t は，外界状態の新しさを表す属性である．この時刻属性を，データの**体内時計** (interior clock) と呼ぶことにした．

データの体内時計には，そのデータを生成するときに，ある時刻を設定する．その時刻の決め方は，次のとおりである．

外界から実時間システム内に入力されるデータの体内時計には，入力を完了したあと，それを実時間システムの1データと認定した時点の時刻に設定する．入力データ以外のデータの体内時計には，そのデータを生成するために参照したすべてのデータのなかで，最新の体内時計をもつデータの体内時計と，同じ時刻を設定する．決して，そのデータを生成した時刻を設定するわけではない．これにより，どのデータの体内時計も，いずれかの入力データの体内時計と，同一の時刻を指すことになる．

いったん設定された体内時計の時刻は，それ以降，変更されることはない．また，いったん生成されたデータを，それ以降，更新することは許さない．なぜなら，更新したいならば，新しい時刻のデータを新規に生成すればよいからである．

結局，データに体内時計を与えて，古いデータと新しいデータを区別したことにより，共用データをプロセス間フローデータで置き換えることができ，見掛け上，共用データをなくすことに成功した．

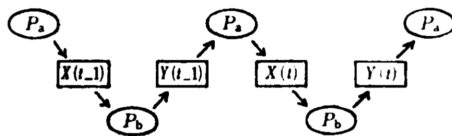

図 4 データ体内時計を利用した繰返しの表現例
Fig. 4 An example of iteration by data interior clocks.

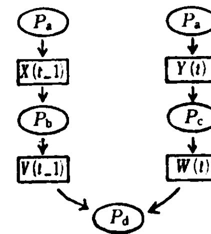

図 5 データ体内時計を利用した分岐合流の表現例
Fig. 5 An example of fork-join by data interior clocks.

体内時計を利用すれば，繰返しの問題は，たとえば**図4**のように表現できるため，繰返しの矛盾が解消される．また，分岐合流の問題で，データの新しさが一致する必要のない場合は，たとえば**図5**のように表現できるため，データの新しさが一致しなければならない場合とは，明らかに違う構造になる．

4.2 並行プロセスの三次元構造

実時間システムは，外界からデータを次々と入力する．その入力タイミングが，外界からの割込みやプロセスの処理結果で決まる場合には，入力の時間間隔が一定ではなくなる．そのため，外界データを入力し終えた時刻を，そのまま入力データの体内時計に設定しようとすると，時間間隔が一定でなくなる．

一定でない時間間隔で，入力データの体内時計の時刻を決めていったとしても，並行プロセスの同期をとる問題に支障はない．ただ，一定時間間隔で，入力データの体内時計の時刻を設定することにしておけば，仮想的な時刻（それに定数を掛ければ，実際の時刻になるような数値）で，体内時計の時刻を表すことができるし，プロセスの動作時間（処理遅れ）を考慮したプロセス間の同期を，容易に実現することができる．

そこで，一定時間が経過するたびに，その間に外界から取り込まれたデータをすべてまとめて一つの入力データとし，その体内時計に，そのときの仮想的な時刻を設定することにした．

一定時間が経過して，一つの入力データが生成されると，そのデータを参照するプロセスが起動され，そのプロセスは，そのデータと同一時刻の体内時計をもつデータを生成する．さらに，そのデータを参照するプロセスが，データ駆動の原則に従って起動され，そのデータと同一時刻の体内時計をもつデータを生成する．このようにして，同一時刻の体内時計をもつデータが，次々と生成されていく．

いま，ある同一時刻の体内時計をもつすべてのデータと，それらのデータを生成するすべてのプロセスと

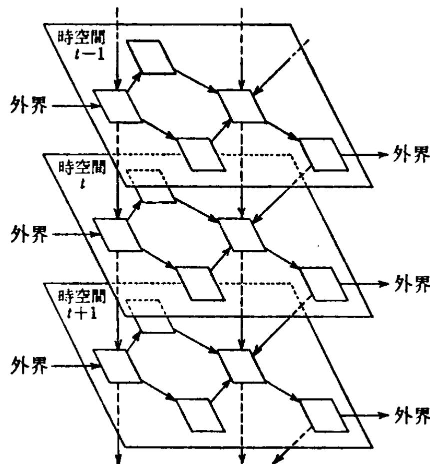

図 6 並行プロセスの三次元構造
Fig. 6 Three-dimensional structure of concurrent processes.

を，一つの二次元平面上に配置することにする．そして，プロセスとデータが配置されたその平面を，その時刻の時空間と呼ぶことにする．

時空間は，過去から未来まで，一定時間間隔ごとに存在する．それらの時空間を時刻順に並べて，プロセスとデータの生成参照関係を表現すれば，規則正しい三次元の立体構造が得られる．

図 6 は，プロセスとデータの生成参照関係を，三次元で表現した例である．矩形がプロセスを表し，矢印がプロセス間のデータの流れを表している．時空間の枠を越えて，他の時空間のプロセスとの間にデータの流れが存在する点に，特徴がある．

5. データ体内時計の実現方式

5.1 記述方式

データに体内時計を付与することにより，見掛け上共用データがなくなり，プロセス間で手渡されるフローデータだけで，実時間システムを表現できるようになった．これは，一つのデータは，ただ一つのプロセスによって生成されることを意味する．

また，これまで，一つのプロセスが複数のデータを生成することを認めた上で，議論を進めてきたが，それは避けられない条件ではない．複数のデータをまとめて一つのデータとみなすか，または，複数のデータはデータ数だけのプロセスによって生成されるとみなすことにすれば，一つのプロセスは，ただ一つのデータしか生成しない，とすることができる．

そこで，プロセスとは，そのプロセスの名前と同一の名前をもつデータを生成する関数である，と考えることができる．

プロセスは，他プロセスによって生成されたいくつかのデータを参照する．参照するデータは，名前と時間差とで識別する．時間差とは，各参照データの体内時計と，そのプロセスが生成するデータの体内時計との，時刻の差である．プロセスの記述形式（プロセス定義文）を，下に示す．

process p ($a_1_s_1, \cdots, a_r_s_n$)

ここで，p はプロセスの名前，a_i は参照データの名前，s_i は参照データの時間差，を表す．少なくとも1個の参照データが指定されなければならない．なお，$a_i_s_i$ は a_i と略記してよく，略記した場合には $s_i=0$ と仮定される．

プロセスは，プロセス定義文で定義されたデータ以外の，他プロセス生成データを参照することはできない．プロセス定義文で定義されたデータは，そのままの形で，そのプロセスに対応するプログラム内で，使用される．たとえば，a, a_1, a_2 のように，同一の名前であって体内時計の時刻だけが異なるデータが，明確に区別されて使用される．

プロセスは，すべて同一の規則に従って制御する．あるプロセスのプロセス定義文で指定された参照データが，すべて生成されたとき，そのプロセスを起動する．つまり，参照データの指定が $a_i_s_i$ であるならば，時刻 $(T-s_i)$ の体内時計をもつデータ a_i がすべて生成されたとき，時刻 T の時空間で動作するプロセスを起動する（そのプロセスの時空間属性を時刻 T に設定する）．そのプロセスは，正常に動作を終了した時点で，そのプロセスと同一の名前をもち，体内時計の時刻が T であるデータを生成する．

5.2 記述例

データに体内時計をもたせたことにより，ペトリネット[7]などのデータフロー表現で記述できる問題だけでなく，時間を考慮した問題も，容易に記述できるようになった．

（1） 状態の更新

```
process p (p_1, a : integer) : integer ;
  begin
    p := p_1 + a
  end
```

これが，最も簡単な，しかし非常に頻繁に現れる状態更新の記述例である．プログラム内部では，参照データ p_1 と生成データ p とが区別されて使用される．単一割当規則に従った記述形態になる．

(2) 移動平均

```
process p(a, a_1, a_2: real): real;
  begin
    p:=(a+a_1+a_2)/3.0
  end
```

これは，今回と前回と前々回の3回の移動平均を計算するプロセスを表現している．3回のデータに別々の名前を付けるのではなく，共通の名前で表現できるようになっている．

(3) 古いデータの利用

実時間システムのオペレータが，あるデータを画面に表示するよう，要求した場面について考察する．その要求を受けて動作するプロセスは，その要求（これも一種の入力データ）の体内時計と同じ時刻の体内時計をもつデータを，表示せねばならないとはかぎらない．その時刻のデータが生成されるまでにかなりの時間がかかるならば，すでに生成済の少し過去の時刻のデータを表示するのも，一つの方法である．そうすれば，オペレータへの応答は速くなる．

```
process p(a: boolean; b_2: integer): integer;
  begin
    if a then p:=b_2
  end
```

これが，古いデータを使用するプロセスの記述例である．この記述によれば，オペレータからの要求 a に対し，その時刻より2単位だけ過去の時刻の体内時計をもつデータ b が，画面に表示される．

(4) 生産者・消費者の問題

```
type
  buffer=record
      head: integer;
      tail: integer;
      buf: array [0..9] of integer
    end;
  output=record
      out: boolean;
      outdata: integer
    end;
process p(p_1: buffer; c_1: output;
    preq: boolean; indata: integer): buffer;
  begin
    p:=p_1;
  if c_1. out then p. tail:=p_1. tail+1;
  if preq and (p_1. head<p. tail+10)
      then begin p. head:=p_1. head+1;
          p. buf [p. head mod 10]:=indata
        end
    end
process c(p_1: buffer; creq: boolean)
    : output;
  begin
    if creq and (p_1. tail<p_1. head)
      then begin c. out:=true;
          c. outdata:=p_1. buf [p_1. tail mod
          10]
        end else c. out:=false
  end
```

これは，二つのプロセスが一つのデータを共用する場合の記述例である．プロセス p（生産者）は，登録要求 preq があったときに，バッファ p に新しいメッセージ indata を登録する．プロセス c（消費者）は，取出要求 creq があったときに，バッファ p から最も以前に登録されたメッセージを取り出す．

ここでは，バッファ p が共用データである．共用データが存在するにもかかわらず，排他制御に関する記述は不要である．それは，データ体内時計を使うことによって，バッファの古い状態 p_1 と新しい状態 p を区別することができたからである．

(5) 5人の哲学者の食事の問題[4)]

```
type
  state=(think, pickup, eat, putdown);
process phil α (phil α_1: state; fork α_1,
    fork β_1: integer): state;
      {α=(0, 1, 2, 3, 4), β:=(α+1) mod 5}
  begin
    case phil α-1 of
      think: phil α:=pickup;
      pickup: if(fork α_1=α and fork β_1=α)
            then phil α:=eat
            else phil α:=pickup;
      eat: phil α:=putdown;
      putdown: phil α:=think
    end
  end
process fork β(fork β_1: integer; phil α,
    phil β: state): integer;
      {β=(0, 1, 2, 3, 4), α:=(β-1) mod 5}
  begin
```

```
    fork β:=fork β_1;
    if(phil α=putdown or phil β=putdown)
      then fork β:=-1;
    if(fork β=-1 and phil α=pickup)
      then fork β:=α;
    if(fork β=-1 and phil β=pickup)
      then fork β:=β
  end
```

これは，有名な同期問題である．5人の哲学者 phil が，丸テーブルのまわりに座って，思考 think と食事 eat を繰り返している．2人の哲学者の中間にはそれぞれ1個のフォーク fork が置かれており，各哲学者は両側のフォークを確保して初めて，食事をすることができる．

このかなり複雑な同期問題も，データ体内時計を使って記述することができ，しかも同期に関する記述は不要である．

6. むすび

実時間システムの各データに体内時計を付与することにより，共用データの存在を許したまま，データ駆動でプロセスを起動する方式を実現した．

本方式では，各プロセスが参照する予定のすべてのデータが生成された時点で初めて，そのプロセスを起動する．また，各プロセスの動作が正常に終了した時点で初めて，そのプロセスが生成したデータを，他プロセスが参照できるようにしている．

これにより，プログラム内部に，同期制御や排他制御のための手続きを，記述する必要がなくなった．また，決定性のある実時間システムを，容易に構築できるようになった．

謝辞 最後に，本研究に際しご指導いただいた日立製作所システム開発研究所三森定道博士に，深く感謝いたします．

参考文献

1) Hansen, P. B.: *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs (1973).
2) Hoare, C. A. R.: Monitors: An Operating System Structuring Concept, *Comm. ACM*, Vol. 17, No. 10, pp. 549-557 (1974).
3) Campbell, R. H. and Habermann, A. N.: The Specification of Process Synchronization by Path Expressions, *Lecture Notes in Computer Science*, Vol. 16, pp. 89-102, Springer-Verlag, Berlin (1974).
4) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666-677 (1978).
5) Dennis, J. B.: First Version of a Data Flow Procedure Language, *Lecture Notes in Computer Science*, Vol. 19, pp. 362-376, Springer-Verlag, Berlin (1974).
6) Tesler, L. G. and Enea, H. J.: A Language Design for Concurrent Processes, *Proc. SJCC*, Vol. 32, pp. 403-408 (1968).
7) Peterson, J. L.: Petri Nets, *ACM Comput. Surv.*, Vol. 9, No. 3, pp. 223-252 (1977).

(昭和58年4月7日受付)
(昭和58年10月11日採録)