論　文

# オブジェクト間の協調動作を表現する並列計算モデルと言語

鵜林　尚靖[†*]　　大木　敦雄[†]　　久野　靖[†]

# Parallel Computing Model and Programming Language for Modeling Collaboration among Objects

Naoyasu UBAYASHI[†*], Atsuo OHKI[†], and Yasushi KUNO[†]

**あらまし**　現在，アクタモデルなどいくつかの並列計算モデルが提案されているが，これらは並列計算にかかわる基本的部分だけをモデル化しており，オブジェクト間の協調動作，すなわち「実世界に存在するオブジェクトが各々どのような役割を担い，どのように協調し合っているか」という観点からのモデル化能力が弱い．このため，これらのモデルで並列システムを記述した場合，システム全体の動作を大域的に理解することは容易ではない．本論文では，この問題を解決する方法として，プロデューサモデルとそれに対応するプログラミング言語 Produce/1 を提案する．プロデューサモデルとは，複数の並列オブジェクト群がプロデューサと呼ばれるオブジェクトのコーディネート下で協調動作するモデルである．アクタモデルでは，各オブジェクトが「協調動作を示すメッセージ通信路の組立て計算」と「実際のメッセージ通信とそれにかかわるオブジェクト内部の処理計算」を同じ枠組みの中で行っているが，プロデューサモデルでは，これら二つの計算を明確に分離し，それぞれをプロデューサと俳優オブジェクトに役割分担させている．そのため，オブジェクト間の協調動作はプロデューサの処理内容を見れば理解できる．

**キーワード**　並列計算モデル，オブジェクト指向，協調動作記述，メッセージ通信

## 1. まえがき

現在，並列計算モデルとして，メッセージ通信に基礎をおくアクタモデル[7]，CSP [9]，CCS [13] やタプルスペースをベースとする Linda [2] などが提案されている．これらは並列計算にかかわる基本的な機構をモデル化したものであり，計算主体（本論文ではこれをオブジェクトと呼ぶ）間の協調動作は基本機能を組み合わせて表現する必要がある．すなわち，「実世界に存在するオブジェクトが各々どのような役割を担い，どのように協調し合っているか」の記述（本論文ではこれを「オブジェクト間の協調動作記述」と呼ぶ）を組み立てる能力が弱い．そのため，これらのモデルで並列システムを記述した場合，システム全体の動作を大域的に理解することは容易ではない．

C. Hewitt により提唱されたアクタモデルを一例にこの問題を考えてみる．アクタモデルでは，アクタと呼ばれるオブジェクトの各々が独立かつ並列に計算を行う能力をもち，通信機構としては，送信相手を明示的に指定した 1 対 1 のメッセージ送信のみを許している．アクタモデルはシンプルで強力であるが，オブジェクト間の協調動作は，「各オブジェクトのメソッド中で関係するオブジェクトへメッセージ送信（処理依頼）する」という原始的な形でしか表現できない．そのため，システム全体の動作を理解するには，すべてのオブジェクトの処理内容を追わなければならない．

オブジェクト間の協調動作記述は，現在，主に，オブジェクト指向分析/設計手法[1],[10],[18] や形式的仕様記述言語[6],[8],[16],[17],[19],[20] の立場から研究されているが，前者においては形式性の欠如，後者においては記述の困難性という問題が存在する．また，これらの研究のほとんどは並列性を明示的に扱っていない．

本論文では，並列計算モデルおよび言語の両側面からオブジェクト間の協調動作記述について述べる．以下 **2.** ではオブジェクト間の協調動作を記述する能力をもった新しい計算モデルとしてプロデューサモデルを，**3.** ではこれに対応するプログラミング言語として

† 筑波大学経営・政策科学研究科経営システム科学専攻，東京都 Graduate School of Systems Management, University of Tsukuba, 3-29-1 Otsuka, Bunkyo-ku, Tokyo, 112 Japan
* 現在，東京大学総合文化研究科広域科学専攻

Produce/1 を提案する．4. で Produce/1 によるプログラミング例を紹介し，続く 5. では Produce/1 の実装について述べる．6. で Produce/1 の実装を通じて得られた知見をもとに，本提案の有効性について考察し，7. でまとめを行う．

## 2. プロデューサモデル

並列システムにおけるオブジェクト間の協調動作を演劇に例えてみる．演劇では，俳優（オブジェクト）の能力は，ある特定の演劇に依存していない．俳優はプロデューサの指示に従って一つの劇を演じているだけである．一方，プロデューサの指示のもとになる脚本（オブジェクト間の協調動作記述）も，ある特定の演劇に依存せず，演劇に出演する俳優が代わっても通用する必要がある．すなわち，演劇では，俳優の能力と脚本とは，全く独立の概念となっている．ところが，アクタモデルなど前章で挙げた計算モデルでは，俳優のみで計算モデルを構成しているため，演劇を理解するのに，どこまでが俳優の能力で，どこからが脚本なのかを区別することが難しい．

本論文で提案するプロデューサモデルでは，俳優のほかにコーディネータとしての役割を果たすプロデューサが新たに計算モデルの構成要素として加わる．図1に示すように，プロデューサモデルではプロデューサと俳優群の間を集約関係で結び，これを演劇の単位であるグループと考える．プロデューサは俳優群の協調動作に必要なメッセージ通信路（プロデューサモデルでは，これを演劇の脚本と考える）を組み立てる役割を担い，一方，俳優群はプロデューサが設定した枠組みに従ってお互いにメッセージの交換を行い，自分に与えられた役割を実行する．演劇の単位であるグループは階層的に構成することが可能であり，より大きな演劇にも対応できる．このため，プロデューサの内容のみを階層的に追えば，演劇全体の脚本，すなわち並列システムにおけるオブジェクト間の協調動作を大域的に理解できる [21],[22].

## 3. プログラミング言語 Produce/1

### 3.1 Produce/1 の言語仕様

Produce/1 は，プロデューサモデルに従うオブジェクト間協調機能をプログラミング言語として実現したものである．Produce/1 のプログラムはクラス定義の集合から構成され（図2），クラス定義はクラス宣言部とクラス実装部の二つからなる．

クラス宣言部はクラスがもつ性質を記述する部分であり，(1) グループ定義宣言部 (2) 抽象状態定義宣言部 (3) 変数宣言部 (4) 送信/受信ポートメソッド宣言部から構成される．(1) は group:で始まり，グループのメンバとなる俳優オブジェクトを列記する．クラス宣言中にこれが含まれるとプロデューサとなる．(2) は state: で始まり，抽象状態を列記する．抽象状態とはクラス

```
//宣言部
class クラス名
{   group:     $group=( 俳優オブジェクト1, 俳優オブジェクト2, .. );
    state:     $state =( 抽象状態 1, 抽象状態 2, .. );
    var:       属性定義，  // 変数宣言
    sendport:  相手クラス名 <.: ポート名 { 事前抽象状態 } ( 引数 ) { 事後抽象状態 };
    recvport:  相手クラス名 .:> ポート名 { 事前抽象状態 } ( 引数 ) { 事後抽象状態 };
}
//実装部（ポートメソッド記述）
クラス名 ::ポート名 ( 引数 )
{   メソッド内ローカル変数の宣言；
    メソッド動作の記述；   // メッセージ接続文など
                           // 構文はC++ライク
}
```

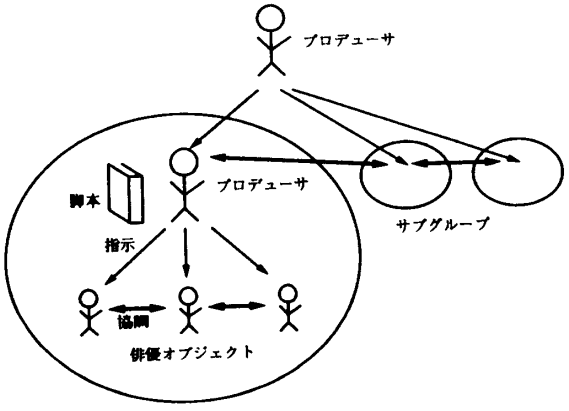



図1 プロデューサモデル
Fig. 1 Producer model.

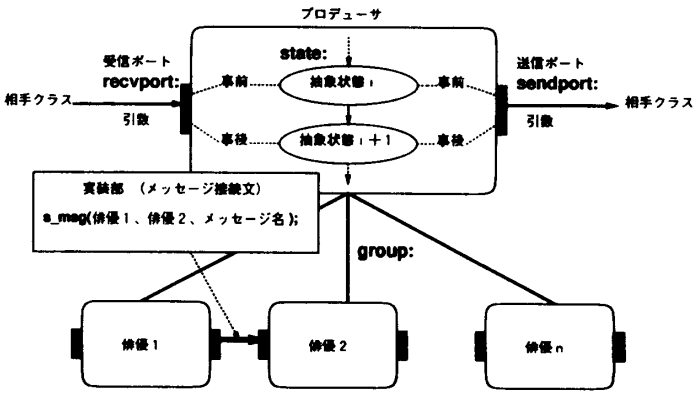



図2 Produce/1 のクラス定義
Fig. 2 Class definition of Produce/1.

のインスタンスであるオブジェクトの内部状態を抽象化したものであり，Pascal の列挙型のような形でオブジェクトの種類ごとに定義され，各メソッドにおいてオブジェクトの内部状態に対応した値に設定される．この値は$state 変数で管理される．抽象状態を記述することで，各オブジェクトは有限状態遷移機械として明示的に表現される．(3) は var:で始まり，クラスの属性を記述する．(4) は sendport:または recvport:で始まり，送信/受信ポートに対応するメソッド（以降ポートメソッドと呼ぶ）およびメソッド実行の制約表明となる事前/事後の抽象状態を宣言する．クラス名 <::ポート名は送信ポートメソッドを，クラス名::> ポート名は受信ポートメソッドを表している．Produce/1 ではメッセージはポートを経由して交換され，メッセージ送信時には送信ポートメソッドが，受信時には受信ポートメソッドが起動される．

クラス実装部には，クラス宣言部で宣言された送信/受信ポートメソッドの具体的な処理内容，すなわち，俳優オブジェクト間にメッセージ通信路を設定するためのメッセージ接続文などが C++言語に類似した構文で記述される．メッセージ接続文には，同期接続と非同期接続の 2 種類があり，前者は s_msg 文を，後者は a_msg 文を使用する．構文は両メッセージ接続文とも同じで，送信オブジェクト，受信オブジェクト（複数可），ポートを列記する．

Produce/1 プログラムはクラス定義の集まりであり，C や C++言語の main 関数に相当するものがない．そこで，Produce/1 では God クラス（ルートのプロデューサ）の life という特別な受信ポートメソッドをメインプログラムとみなしており，ここからプログラムの実行が開始される．

### 3.2 Produce/1 における協調動作表現

Produce/1 におけるオブジェクト間の協調動作はプロデューサモデルに基づき，メッセージ通信で表現される．メッセージ通信はプロデューサが傘下の俳優オブジェクト群の送信ポートと受信ポートを接続することにより行われるが，プロデューサは単にメッセージ形態に応じた通信路を開設するのみで，実際のメッセージ通信は各俳優オブジェクトが自律的に行う（図 3）．

Produce/1 では抽象状態同期 [11] の考えに基づき，有限状態遷移機械である送信オブジェクトと受信オブジェクトがそれぞれ特定の状態（抽象状態）になったときに発火が起こり，メッセージが送られる．送信

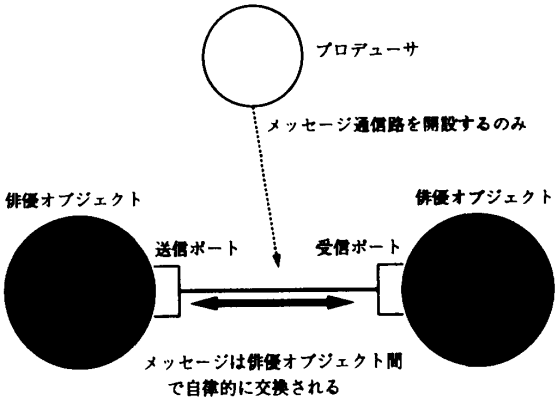



図 3　メッセージ通信機構
Fig. 3　Message passing mechanism.

のタイミングは送信側が決めるのではなく，送信側と受信側の状態の適合に基づいて決まる．すなわち，Produce/1 は拡張型状態遷移モデル [14] に従って動作する．拡張型状態遷移モデルとは本来の有限状態遷移機械がもつすべての状態の大きな集合を縮退させて簡単な抽象状態とし，その状態とクラスの属性の積でもとの有限状態遷移機械の複雑な状態を表現したモデルである．

以上をまとめると，Produce/1 によるメッセージは次の手順で送信される．

（1） プロデューサが送信側/受信側それぞれについて俳優オブジェクトとポートを指定して通信路を設定する．この通信路の集まりが俳優群の協調動作を表現する．

（2） 送信側/受信側ポートメソッドとも，実行を開始してよい抽象状態（事前）がクラス宣言部に指定されている．通信路が設定されていても，その両側のメソッドがともに実行可能になるまで送信は行われない．

（3） 両側のメソッドがともに実行可能になると，まず送信側ポートメソッドが起動されて送信すべき値を組み立て，メッセージが送られ，受信側ポートメソッドが起動されて送られてきた値に基づく処理を行う．

アクタモデルなど従来の計算モデルではメッセージトポロジー（メッセージ通信路の集まり）の組立てとメッセージの発火が分離されていなかったため，並列システム全体の動作が理解しづらくなっていた．プロデューサモデルおよびそれに対応するプログラミング言語 Produce/1 の特長は，この二つを明確に分離しているところにある．計算の進行と共に，プロデューサが先導してシステムのメッセージトポロジーの組立てを行い，俳優オブジェクトは発火条件が満たされた

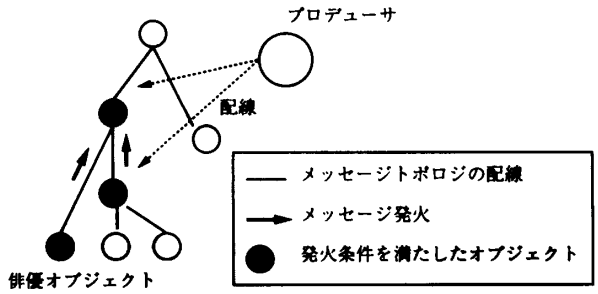



図4 通信路の設定と発火
Fig. 4 Message connection and fire.

時点で，分散かつ自律的にメッセージ通信を行う（図4）．この場合，プロデューサと俳優オブジェクトはお互い並列に動作し，システム全体の動作と各オブジェクトの分散自律性を調和する並列計算モデルになっている．アクタモデルなどがメッセージ goto 文しかない計算モデルだとすると，プロデューサモデルは構造化されたメッセージトポロジーを指向した計算モデルと考えることができる．

## 4. Produce/1 によるプログラミング例

### 4.1 哲学者の食事問題の記述

Dijkstra の哲学者の食事問題 [3] は資源競合問題として有名である．図5に示すプログラムでは，哲学者をプロデューサに，フォークを俳優オブジェクトに割り付けている．God は哲学者をまとめるプロデューサで，何人の哲学者が食事に参加するかを記述している．

食事にかかわる協調動作は，プロデューサである哲学者オブジェクトの life というメソッドの内容を追えば理解できる．s_msg（同期型メッセージ接続文）および a_msg（非同期型）はともに哲学者から二つのフォーク f1 と f2 にアトミックにメッセージ送信する．メッセージ pick が発火するのは，哲学者の抽象状態が thinking，両脇の2本のフォークの抽象状態が同時に down のときのみである．一方，メッセージ release が発火するのは，哲学者の抽象状態が eating，両脇の2本のフォークの抽象状態が up のときであるが，哲学者からどちらのフォークに先にメッセージが送信されるかは非決定的である．フォーク f1 が先に release されるかもしれないし，f2 の方が先かもしれない．

個々の哲学者の動きは上述のとおりであるが，プログラム全体ではこれらの哲学者が同時に複数動作し，競合を演じることになる．ところが，Produce/1 では，個々の哲学者の動作を抽象状態をベースに有限

```
// フォークのクラス定義（俳優）
class Fork
{  state:     $state={up, down};
   recvport:  void Fork( ) ( ) {down};          // constructor
              void ~Fork{up|down} ( ) { };      // destructor
              void pick{down} ( ) {up};
              void release{up} ( ) {down},
}
recvport void Fork::Fork( )       { $state={down}; }
recvport void Fork::~Fork( )      { $state={ };    }
recvport void Fork::pick( )       { $state={up};   }
recvport void Fork::release( )    { $state={down}, }

// 哲学者のクラス定義（プロデューサ&俳優）
class Phil
{  group:    $group=(Fork *f1, Fork *f2),
   state:    $state={start, thinking, eating, end),
   sendport: void Fork<.::pick{thinking} ( ) {eating};
             void Fork<.::release{eating} ( ) {thinking},
   recvport: void Phil( ) (Fork *fork1, Fork *fork2) {start},
             void ~Phil{end} ( ) { },
             void life{start} ( ) {end},
}
sendport void Phil::pick( )     { }
sendport void Phil::release()   { }
recvport void Phil::Phil(Fork *fork1, Fork *fork2)
{  $state={start},   f1=fork1,   f2=fork2,  }
recvport void Phil::~Phil( )    { $state={ };  }
recvport void Phil::life()  // 哲学者の協調動作
{  while(true)
   {  $state={thinking},   p1_sprint("thinking");   p1_sleep(1);
      s_msg( self, *f1 & *f2, pick ),   // 同期型メッセージ接続（' &' は同時性を示す）
                                        // self は自分自身（哲学者）を示す
      $state={eating},     p1_sprint("eating"),     p1_sleep(2);
      a_msg( self, *f1 | *f2, release ); // 非同期型メッセージ接続（ ' | ' は非決定性を示す）
   }
   $state={end},
}
// メインのクラス定義（プロデューサ）
class God
{  group.    $group=(Phil *p1, Phil *p2, Phil *p3, Phil *p4, Phil *p5),
   state      $state={start, process, end),
   var.       Fork *f1, Fork *f2, Fork *f3, Fork *f4, Fork *f5,
   sendport   void Phil::life{process} ( ) {process},
   recvport:  void God( ) ( ) {start},
              void ~God{end} ( ) { };
              void life{start} ( ) {end},
}
sendport void God .life( ) { }
recvport void God::God( )
{  $state={start};
   f1=new Fork;          f2=new Fork;          f3=new Fork,
   f4=new Fork;          f5=new Fork;
   p1=new Phil(f1, f2);  p2=new Phil(f2, f3),  p3=new Phil(f3, f4),
   p4=new Phil(f4, f5);  p5=new Phil(f5, f1),
}
recvport void God::~God( )
{  delete p1;  delete p2;  delete p3,  delete p4,  delete p5,
   delete f1;  delete f2;  delete f3,  delete f4,  delete f5;
   $state={ },
}
recvport void God .life( )
{  $state={process};   a_msg( self, *p1 | *p2 | *p3 | *p4 | *p5, life );  }
```

図5 哲学者の食事問題の記述
Fig. 5 Description of dining philosophers.

状態遷移機械として記述するだけでよく，哲学者間で発生する競合を回避するためのロジックをプログラム中に明示的に記述する必要はない．競合関係の回避は Produce/1 の処理系にまかせられており，そこで，メッセージ発火の同時性や非決定性を認識する．競合回避の具体的なメカニズムについては，5. で述べる．以下は哲学者の食事問題のプログラムを実際に動作させた結果である．哲学者たちがお互いにデッドロックに陥ることなく非決定的な順番で食事をしている様子がわかる．

```
% phil                ← プログラム起動
Phil1: thinking       ← 5人の哲学者の初期状態（瞑想中）
Phil2: thinking
Phil3: thinking
Phil4: thinking
Phil5: thinking
Phil5: eating         ← 5番目の哲学者が食事
Phil2: eating         ← 2番目の哲学者が食事
Phil5: thinking
Phil4: eating         ← 4番目の哲学者が食事
```

### 4.2 トーナメント問題の記述

Produce/1の特長の一つとして，メッセージトポロジーの記述性が挙げられる．ここでは，四つの数の中から最大値をトーナメント方式で求める問題を取り上げ，この性質を説明する（図6）．トーナメント問題のプログラムは図7のようになる．God::lifeで全体のメッセージトポロジーを定義している．トポロジーを構成する各ノードは下の二つの子ノードから値を受け取り，大きい方を親ノードに渡す．その内容はNodeクラスの送信/受信ポートメソッドsetvalに記述されている．最初にトーナメントを示す2分木構造のトポロジーと，ノードの発火条件（事前/事後抽象状態）を与えると，次々と条件を満たしたノードが並列に発火していき，最後に最大値が求まる．

アクタモデルなどをベースとする従来型の並列オブジェクト指向言語では（受信）メソッドの中で必要なメッセージ送信を行っているが，これでトーナメント問題を記述しようとすると，一つのメソッドの中でどの子ノードから値をもらったか状態管理した上で，親ノードにメッセージ送信する必要がある．すなわち，Produce/1で送信/受信ポートメソッド，メッセージ発火のための事前/事後抽象状態に区分しているものを，一つにまとめて記述しなければならない．Produce/1ではこれらを分離しているため，簡潔な記述が可能である．

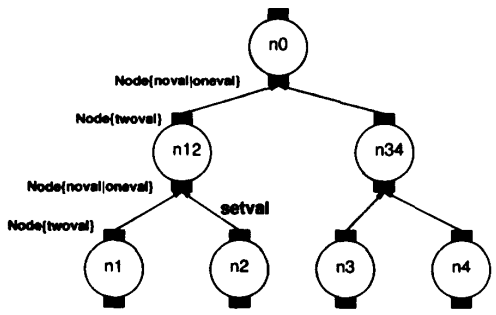



図6 トーナメント問題のメッセージトポロジー
Fig. 6 Message topology of tournament.

## 5. Produce/1の実装方式

### 5.1 Produce/1言語処理系の概要

Produce/1の処理系は，Solaris2.3上で稼動するC++言語へのトランスレータであり，lex/yaccとC++言語により記述されている[23]．Produce/1ソー

```
// ノードのクラス定義（併優）
class Node
{   state:      $state={noval, oneval, twoval};
    var:        int v1, int v2;
    sendport:   void Node<::setval{twoval}(int *p){noval};
    recvport:   void Node{ } (int v) {noval|twoval};
                void ~Node{noval} ( ) { };
                void Node::>setval{noval|oneval}(int *p){oneval|twoval};
                void prval{twoval} ( ) {twoval};
}
sendport void Node::setval(int *p)   // p   は送信パラメータ
{   if( v1 > v2 ){   *p = v1;  }
    else         {   *p = v2;  }
}
recvport void Node::Node(int v)
{   if( v == -1 ){   $state={noval};  }
    else         {   $state={twoval};   v1=v,   v2=v;  }
}
recvport void Node::~Node( )
{   $state={ };   }
recvport void Node::setval(int *p)   // p   は受信パラメータ
{   if( $state=={noval} )        {   v1= *p,   $state={oneval};  }
    else{if( $state=={oneval} ) {   v2= *p;   $state={twoval};  }}
}
recvport void Node::prval( )
{   if( v1 > v2 ) {   p1_iprint(v1),  }
    else          {   p1_iprint(v2);  }
}
// トーナメントのクラス定義（プロデューサ）
class God
{   group.      $group=(Node *n1, Node *n2, Node *n3, Node *n4,
                        Node *n12, Node *n34, Node *n0);
    state:      $state=(start, process, end);
    sendport    void Node<::setval{process} ( ) {process},
                void Node<::prval{process} ( ) {process};
    recvport    void God{ } ( ) {start};
                void ~God{end} ( ) { };
                void life{start} ( ) {end};
}
sendport void God::setval( ) { }
sendport void God::prval( ) { }
recvport void God::God( )
{   $state={start};
    n1=new Node(2),  n2=new Node(4),  n3=new Node(1);   n4=new Node(7),
    n12=new Node(-1),   n34=new Node(-1);   n0=new Node(-1);
}
recvport void God::~God( )
{   delete n1,   delete n2,   delete n3;   delete n4,
    delete n12,  delete n34;  delete n0;
    $state={ };
}
recvport void God::life( )  //  トーナメントのメッセージトポロジを定義
{   $state={process};
    a_msg( *n1, *n12, setval );     a_msg( *n2, *n12, setval );
    a_msg( *n3, *n34, setval );     a_msg( *n4, *n34, setval ),
    a_msg( *n12, *n0, setval );     a_msg( *n34, *n0, setval );
    s_msg( self,  *n0, prval  );    //   最大値を表示
    p1_terminate( );
}
```

図7 トーナメント問題の記述
Fig. 7 Description of tournament problem.

スプログラムはトランスレータにより C++言語に変換され，C++コンパイラでコンパイルされた後，Solaris2.3 マルチスレッドライブラリおよび Produce/1 ランタイムライブラリとリンクされ実行モジュールになる．Produce/1 ランタイムライブラリでは，主にメッセージ制御や送信/受信ポートメソッドへのスレッドの割付けなどに関する部分を受けもっている．実行モジュールは 4CPU の対称型共有メモリマルチプロセッサマシン SPARC Server 1000 上で動作する．

### 5.2 並列実行とスレッドの生成過程

Produce/1 では，プログラムの実行開始時に，まずスケジューラスレッドが起動され，次に Produce/1 のメインプログラムである God::life がルートスレッドとして起動される（図 8）．

God::life の中でメッセージ接続文が現れると，送信/受信オブジェクト，送信/受信ポートメソッドに関する情報がメッセージ管理テーブルに格納される．スケジューラは定期的にこのメッセージ管理テーブルを走査し，送信側/受信側双方のオブジェクトがメッセージ発火に必要な抽象状態になっているか監視する．もし，発火条件を満たすメッセージがあれば，送信/受信ポートメソッドに各々新たなスレッドを割り付け，通信の同期をとる（図 9）．受信オブジェクトがプロデューサとして振る舞う場合は，受信ポートメソッドの中で更にメッセージ接続文が現れ，同様の操作が繰り返される．このように，スケジューラのほかには一つしかなかったスレッドがねずみ算式に増えて，各々が並列に実行される．

通常の計算モデルでは条件付きメソッドやガード宣言などを用いて受信側で同期をとることが多いが，Produce/1 では，上記のように，抽象状態のみでメッセージ通信の同期をとる方式を採用している．スケジューラがメッセージ管理テーブルを走査する場合，発火条件が単なる値（= 抽象状態）か評価式（= 条件付きメソッドやガード宣言）かでは前者の方が実装上のオーバヘッドが少なくて済むからである．抽象状態は排他ビット列で表現でき，オブジェクトが該当する抽象状態になったかどうかの評価はビット列の論理積をとればわかる．また，ガード宣言には，オーバヘッドのほかに，いつ評価すべきかが難しい（何回も評価しなくてはならない），評価のために同じオブジェクトをアクセスする別の計算と同期をとらなければならない，評価のための環境（変数のスコープなど）を整えるのが面倒などの弱点がある．これに対し，抽象状態同期にはハードウェアへの写像（例えばプログラム可能なバリア機構など）が容易だという利点がある．

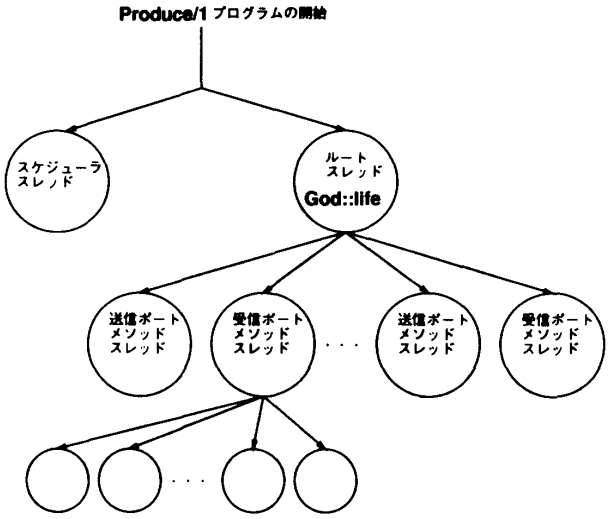



図 8 Produce/1 におけるスレッドの生成過程
Fig. 8 Thread creation in Produce/1.

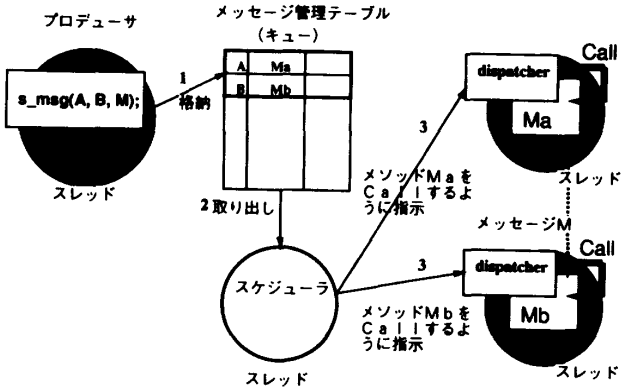



図 9 Produce/1 の実装方式
Fig. 9 Implementation of Produce/1.

本処理系では，「ポートメソッドが送信と受信の二つに明確に区分されている」「メッセージ通信路の設定と実際のメッセージ発火が別々になっている」という Produce/1 の特長を生かして送信/受信ポートメソッド単位にスレッドを割り付けている．そのため，オブジェクト単位にスレッドを割り付ける方式と比較して，粒度の細かい並列性が実現されている．今回は 4CPU の対称型共有メモリマルチプロセッサ上での実装であるが，多数の CPU とメモリがネットワーク状に分散して配置される MPP (Massive Parallel Processing) 型並列マシン [15] と相性が良いと思われる．但し，この場合，スケジューラを複数に分割し，それらを MPP 上に効率良く分散配置するなどの工夫が必要となる．これについては次節で詳細に述べる．

### 5.3 競合問題の回避方法

並列プログラムでは競合問題に留意する必要がある．哲学者の食事問題の場合では，各哲学者は両脇にあるフォークを同時に pick しなければ食事ができない．今，図 10 のように，隣り合った 2 人の哲学者の事前抽象状態が両方とも thinking になり，両脇にあるフォークの事前抽象状態がすべて down になったとしよう．このとき，哲学者は 2 人とも，両脇にあるフォークを pick して食事ができる状態になっている．ところが，2 人の間にあるフォークは一つしかないので，競合してしまう．

4. で示した Produce/1 プログラムには哲学者間の競合回避を明示的に記述した部分はなく，個々の哲学者の動作しか記述していない．Produce/1 では，競合回避は処理系で行うようにしている．

Produce/1 の処理系では，一つのメッセージ管理テーブルと一つのスケジューラしかないので，このような競合問題は回避される．すなわち，スケジューラは 1 度に一つのキューエントリしか取り出すことができないので，2 人の哲学者のうち 1 人だけにしか食事をさせることができない．但し，2 人の哲学者のどちらが食事をするかはスケジューリングの状態によって定まる．従って，実行のたびに食事をする順番が異なる．

現在の処理系は，小規模な対称型共有メモリマルチプロセッサマシン上での実装のため，一つのスケジューラしかもたせていない．しかし，MPP マシンなどの上で大規模な並列計算を行う場合はメッセージ発火数が多くなり，このことが性能上のボトルネックになる可能性がある．この場合，メッセージ管理テーブルを n 区画に分割し各区画に一つずつスケジューラを割り当てる方式が考えられるが，区画をまたがるオブジェクトが存在する場合は競合するメッセージが同時発火する可能性がある．例えば，図 10 で，2 人の哲学者が設定したメッセージ管理情報は別々の区画に登録されるかもしれない．そのとき，各スケジューラが単純にメッセージ発火条件を判断すると，前述のような競合が起きてしまう．このような同時発火の問題を回避するには，スケジューラは対象オブジェクトが複数の区画にまたがっていないかメッセージ管理テーブルの情報をたどって確認しなければならない．

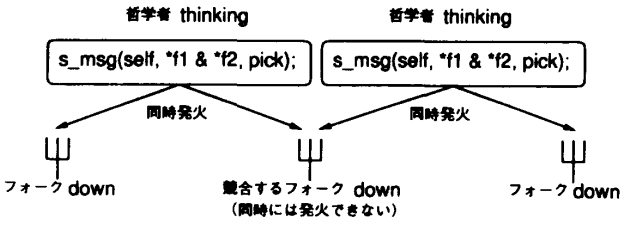



図 10 競合問題の回避
Fig. 10 Avoiding of race condition.

このような複数スケジューラを MPP マシン上に効率良く実装する方法として，次のような方式が考えられる．具体的な実装については今後の課題としたい．

（1） オブジェクトを頂点，通信路を辺とするグラフを考え，なるべく辺が計算機のノードにまたがらないように，オブジェクトを各ノードに配置する．

（2） 各ノードにスケジューラをおき，ノード内でしか通信しないオブジェクトは今回と同じに競合を気にせずにスケジューリングする．

（3） ノードをまたがって通信するオブジェクトについては，それらのノードのスケジューラ同士で通信して競合を調整する．

### 5.4 ベンチマークテスト

今回の実装では，メッセージが発火するたびに，送信/受信ポートメソッド単位にスレッドを割り付けている．そのため，Produce/1 プログラムを実行させると多くのスレッドが生成され，性能が低下することが予想される．そこで，どの程度，処理性能が低下するかを測定するため，単純にメッセージ接続のみを行うプログラムでベンチマークテストをした．

ベンチマークテストでは，発生メッセージ数を，1, 10, 50, 100, 200, 500 と変化させて，どのように処理時間が変化するかを Solaris2.3 の time コマンドを使用して調べた（表 1）．複数 CPU をもつため，real time（全体の実処理時間）の方が，user time（ユーザプログラムの実行に要した時間）と system time（システム側の実行に要した時間）の合計より値が小さくなっている．

処理時間中，user time はユーザプログラムとスケジューラの実行に要した時間の合計を，system time は Solaris2.3 のマルチスレッド処理に要した時間を表している．表 1 の 1 メッセージ当りの処理時間でこれらを比較してみると，user time はメッセージ数が増え

表 1 1 メッセージ当りの処理時間（単位：秒）
Table 1 Time required per message.

| メッセージ数 | 1 | 10 | 50 | 100 | 200 | 500 |
|---|---|---|---|---|---|---|
| real time | 0.1 | 0.01 | 0.01 | 0.012 | 0.0145 | 0.0204 |
| user time | 0 | 0 | 0.002 | 0.003 | 0.004 | 0.0036 |
| system time | 0 | 0.01 | 0.01 | 0.011 | 0.0175 | 0.0306 |

てもほとんど変化していないのに対し，system time の方はメッセージ数が 100 を超えたあたりから値が大きくなっている．今回のベンチマークテストに関する限り，Produce/1 プログラムの処理時間は，Solaris2.3 のマルチスレッド処理の性能に大きく依存していると言える．

生成スレッド数が多いということは並列度が高いことを意味するが，CPU を数個しかもたない通常の対称型共有メモリマルチプロセッサマシンでは，上で述べたように，ある水準（今回の場合は発生メッセージ数が 100）を超えるとシステム側のオーバヘッドの方が大きくなってしまう．CPU が数個のマシン上で規模の大きな並列計算を行う場合は，今後，処理方式を改善していく必要がある．例えば，発生メッセージ数が 100 になると，メッセージ発火を一時停止させ，発生数が 100 以下になるまで待つようなスケジューリング方式を採用することが考えられる．この場合，同時発生メッセージ数が常に 100 以下に押さえられるので，Solaris2.3 のマルチスレッド処理による性能低下を回避できる可能性がある．

## 6. 考　　察

### 6.1 従来型言語との相違点

Produce/1 では，メッセージトポロジーの組立てとメッセージ発火の二つを明確に分離しているため，並列オブジェクト間の協調動作を容易かつ大域的に表現できる．しかし，従来型言語でも，一定の範囲内で，これらを分離して表現することは可能である．ここでは，従来型言語を使用した場合の主な協調動作表現方法とそれらの問題点について述べる．また，それらと対比させながら，記述面から見た Produce/1 の言語としての優位性について考察する．

従来型言語でオブジェクト間の協調動作を表現する際に用いられる最も一般的な方法は，一つの管理オブジェクトと複数の計算オブジェクトからなるマスタスレーブ的なプログラミング方式である．通常，これは，管理オブジェクトから計算オブジェクトのメソッドを呼び出す方式をとる．しかし，管理オブジェクトで表現できるものは，通常，管理オブジェクトと計算オブジェクトの間の協調動作（メッセージ通信）であり，計算オブジェクト同士間の協調動作表現は，多少，工夫が必要である．

例えば，従来型言語を使用して計算オブジェクト同士間の協調動作を表現する方法として，図 11 のように，管理オブジェクトの中でメッセージトポロジー情報を組み立て，これを計算オブジェクト群にリレー的に受け渡す方式が考えられる．メッセージトポロジー情報は，メッセージをリレーする計算オブジェクト名とそのときのメソッド名から構成される．メッセージ送信に際して，メソッドのパラメータとして通信相手を陽に指定しなければならないような従来型言語であっても，各オブジェクトの通信相手をオブジェクト定義の中で固定せず，それぞれのオブジェクトの内部変数やパラメータとして保有できるので，プログラミング上のテクニックとして，ここで述べたような方法を用いることは可能である．しかし，この方法には次のような問題がある．

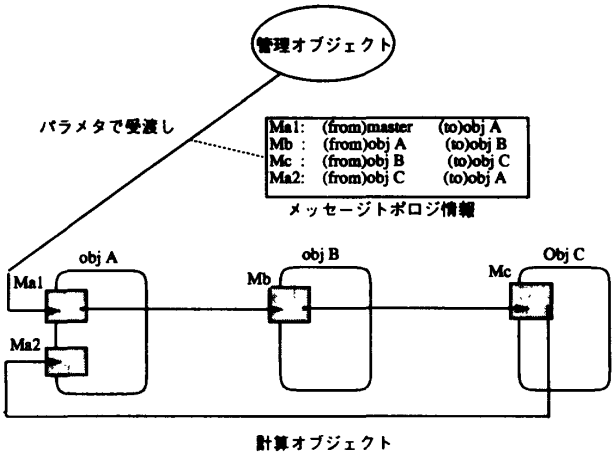



図 11　従来型言語における協調動作表現
Fig. 11　collaboration representation using traditional language.

（1） 各計算オブジェクトに対してメッセージトポロジー情報をパラメータとして与えなければならないため，メソッドのインタフェースが複雑になる．

（2） メソッド起動の連鎖であるため，逐次的な協調動作しか表現できない．Produce/1 のように，発火条件を満たしたものが並列に動作するといったような記述ができない．

一方，オブジェクト（プロセス）のポート間の接続を行うための構文を有するため，オブジェクトの定義と通信トポロジーが分離された言語も存在する．例えば，Fortran M [4],[5] では，process 文の中で定義された inport/outport の接続は channel 文で行っている．このような記述方法は，CSP や CCS などのプロセス代数をベースとする言語に多く見られる．しかし，オブジェクト間の協調動作表現という観点で見た場合，

この方式には次のような問題がある．

（1） channel 文では inport/outport を接続するという事実しか記述できない．どのような協調動作のシナリオの中でそれらが接続されるのか，接続が可能（発火）になるのはどのような条件の場合か，などが記述できない．オブジェクト間の協調動作記述で重要なのは，むしろ，この部分である．

（2） 協調動作の実際のシナリオは，個別オブジェクトの中で，メッセージ送受信を示す send/receive 文として記述される．従って，現実には，個々のオブジェクトの中身を追っていかなければ，全体のシナリオは理解できない．

Fortran M が，オブジェクト間の協調動作に関して，Produce/1 と同等の記述能力をもつためには，channel 文の中でメッセージ通信路の接続条件が書けるようになる必要がある．現状では，この条件が書けないため，channel 文はメッセージ通信路の存在を示すのみで，大域的なメッセージ通信路の変化は記述できない．

以上見てきたように，従来型言語でもある程度，メッセージトポロジーを分離して表現することは可能であるが，プログラマの任意性に依存すると同時に，その記述性も十分ではない．Produce/1 では，計算オブジェクト（俳優オブジェクト）同士間の協調動作を，プロデューサの中に構文レベルで明示的に記述できるため，このような問題を回避できる．

### 6.2 並列言語としての Produce/1 の特長

Produce/1 はオブジェクト間の協調動作を表現することを目的に開発した言語であるが，並列言語としても，興味深い特長をもっている．以下は Produce/1 の実装とプログラミング体験を通じて得られた知見である．これらの多くは，従来の並列言語ではあまり見られなかった特長である．

（1） 並列性の単位粒度が小さい．

（2） プログラム作成にあたって並列性を意識する必要がない．Produce/1 ではメッセージトポロジーと俳優オブジェクトに付与された発火条件（事前/事後抽象状態）により並列に実行されるが，プログラマは記述にあたって並列実行のことを意識する必要はない．

（3） メッセージトポロジーの部品化とその再利用が可能．部品を再利用することにより，ソートプログラムのようにメッセージトポロジーがアルゴリズム上重要となるプログラムを容易にかつわかりやすく記述できる．

## 7. むすび

本論文では，並列計算モデルや言語の側面からオブジェクト間の協調動作記述について述べた．アクタモデルなどの従来型並列計算モデルと比較すると，今回提案したプロデューサモデルおよび Produce/1 ではオブジェクト間の協調動作を大域的に表現することができる．残された課題としては以下のものがある．

第 1 の課題は外部環境の変化に対応可能な協調動作の記述である．一般的に並列システムを構成するオブジェクト群は外部環境に応じてお互いの協調関係を変化させる．ところが，現状の Produce/1 では動的に変化する協調動作を大域的に表現できない．クラスの表現形態が固定的なためにポート間の接続形態を変更できず，外部環境の変化に動的対応することが難しい．この問題を解決するには，動的な型をもつクラスを導入する必要がある．

第 2 の課題は契約ネット型の協調動作の記述である．現状の Produce/1 では，俳優オブジェクトはあらかじめプロデューサが決めた脚本どおりにしか動作できない．プロデューサが俳優オブジェクト群にお互いに協調し合って実行してほしい機能を提示すると，そのときの状況に応じて協調動作に参加する俳優オブジェクトが選択されるような機能が必要である．

第 3 の課題はプロデューサがもつ役割の拡大である．より高度なオブジェクト間の協調動作を考えると，プロデューサにメッセージ通信路の設定以外にも，並列システム全体の状態監視や制御など，さまざまな役割をもたせることが重要となる．ABCL/R2 [12] などリフレクションの機能を有する並列オブジェクト指向言語では，メタオブジェクトの概念を導入することにより，全体監視など機能をもたせており，Produce/1 でもこのような方向での検討が必要とされる．

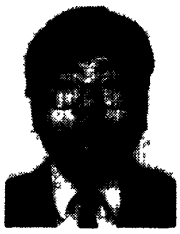
今後は，以上述べたような課題を解決し，より柔軟なオブジェクト間の協調動作記述を実現していきたい．

## 文　献

[1] G. Booch, “Booch 法：オブジェクト指向分析と設計第 2 版,” アジソンウェスレイ, 1995.
[2] N. Carriero and D. Gelernter, “Linda in context,” Communications of ACM, vol.32, no.4, pp.444–458, 1989.
[3] E.W.N. Dijkstra, “Acta Informatica 1,” p.160, 1971.
[4] I. Foster, et al., “Fortran M: A Language for Modular Parallel Programming,” J. Parallel and Distributed Computing, vol.25, no.1, 1995.
[5] I. Foster, “Designing and Building Parallel Programs,” Addison-Wesley, 1995.

[6] 何　克清，渡辺慎哉，宮本衛市，"並行オブジェクト群による協調動作に対する型の定義，" 情処学論，vol.36, no.2, pp.443-452, 1995.
[7] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," Proceedings of the 3rd International Joint Conference on Artificial Intelligence, 1973.
[8] R. Helm, I.M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," Proc. ECOOP/OOPSLA '90, pp.169-180, 1990.
[9] C.A.R. Hoare, "Communicating Sequential Processes," Prentice-Hall, 1985.
[10] I. Jacobson, "オブジェクト指向ソフトウェア工学 OOSE," アジソンウェスレイ，1995.
[11] 久野　靖，"抽象状態：モジュールのもう一つの可能性，" 情処学 RPG 研報，93-RPG-14-4, 1993.
[12] 増原英彦，松岡　聡，渡部卓雄，"自己反映並列オブジェクト指向言語 ABCL/R2 の設計と実現，" コンピュータソフトウェア，vol.11, no.3, pp.15-32, 1994.
[13] R. Milner, "Communication and Concurrency," Prentice-Hall, 1989.
[14] 水野忠則，"プロトコル言語，" カットシステム，1994.
[15] H.S. Morse, "Practical Parallel Computing," AP PROFESSIONAL (1994).
[16] 元木　誠，中島　震，"オブジェクトの集団的振舞いの設計と検証のための高レベルペトリネット，" 情処学論，vol.36, no.5, pp.1163-1172, 1995.
[17] 中島　震，"オブジェクトの集団的振舞いの設計，" 情処学 SE 研報，93-SE-95, pp.39-46, 1993.
[18] J. Rumbaugh, "オブジェクト指向方法論 OMT," トッパン，1992.
[19] 酒井博敬，"オブジェクトの振舞いに関するコントラクトの設計について，" 情処学論，vol.33, no.8, pp.1052-1063, 1992.
[20] 酒井博敬，堀内　一，"オブジェクトの協調の抽象化について，" 情処学 DBS 研報，93-DBS-94, pp.135-144, 1993.
[21] 鵜林尚靖，久野　靖，"オブジェクト間協調動作表現モデルの提案——「プロデューサモデル」とその記述言語について，" 情処学 PRO 研報，95-PRO-1, pp.41-48, 1995.
[22] 鵜林尚靖，大木敦雄，久野　靖，"並列オブジェクト協調記述言語 Produce/1 について，" 日本ソフトウェア科学会第 12 回全国大会，pp329-332, 1995.
[23] 鵜林尚靖，大木敦雄，久野　靖，"並列オブジェクト協調記述言語 Produce/1 の実装，" 情処学 PRO 研報，95-PRO-5, pp.1-6, 1996.

(平成 8 年 2 月 29 日受付，6 月 7 日再受付)

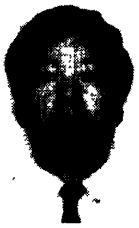


**鵜林　尚靖**　(正員)

1982 広島大・理・数学卒．1996 筑波大大学院経営システム科学専攻修士課程了．同年，東大大学院総合文化研究科広域科学専攻広域システム科学系博士後期課程入学．1982 年より (株) 東芝に勤務．現在，同社にてソフトウェア生産技術関係の業務に従事．ソフトウェア工学，プログラミング言語，並列計算モデルなどに興味をもつ．情報処理学会，日本ソフトウェア科学会各会員．

**大木　敦雄**

1983 筑波大大学院理工学研究科修士課程了．同年静岡大工学部情報工学科助手．1989 年 3 月筑波大大学院経営システム科学専攻助手，現在に至る．プログラミング言語，プログラミング環境，オペレーティングシステム等に興味をもつ．著書に「Mule 入門」(アスキー) 等がある．情報処理学会，日本ソフトウェア科学会各会員．



**久野　靖**

1984 東工大大学院理工学研究科情報科学専攻博士課程単位取得退学．同年東工大理学部情報科学科助手．筑波大大学院経営システム科学専攻講師を経て，1990 年より同助教授．理博．プログラミング言語，プログラミング環境，ユーザインタフェースなどに興味をもつ．著書に「言語プロセッサ」「UNIX の基礎概念」等がある．情報処理学会，日本ソフトウェア科学会，ACM, IEEE-CS 各会員．