並列オブジェクト協調記述言語 Produce/1 の実装

鵜林 尚靖　　大木 敦雄　　久野 靖
筑波大学 経営・政策科学研究科 経営システム科学専攻

現在の並列オブジェクト指向言語は、オブジェクト間の協調動作を明示的に表現する手段を持っていない。そのため、オブジェクト指向で記述するとシステム全体がどのように動作するのか理解しづらいという問題が指摘されている。我々は、オブジェクト間の協調動作に関わるこのような問題を解決する1つの方法として、「プロデューサモデル」とそれを記述するための言語 Produce/1 を提案しているが、今回その処理系を開発した。Produce/1 の処理系は、Solaris2.3 上で稼働する C++ 言語 (プラス Solaris2.3 マルチスレッドライブラリ呼び出し) へのトランスレータであり、C++ 言語と lex/yacc により記述されている。

アクタモデルでは、各オブジェクト群が「ネットワークトポロジの組み立てのための計算」と「メッセージ通信とそれに関わるオブジェクト内計算」を同じ枠組の中で行っていたため、システム全体の動作が理解しづらくなっていた。それに対し、Produce/1 では、これら2つの計算を明確に分離して2階層の並列化を実現し、それぞれをプロデューサ・オブジェクトと俳優オブジェクトに役割分担させている。アクタモデルが「メッセージ goto 文しかない計算モデル」だとすると、Produce/1 は「メッセージトポロジが構造化された計算モデル」と考えることができる。

通常の並列オブジェクト指向言語の処理系ではオブジェクト単位にスレッドを割り付けるのが一般的であるが、Produce/1 の処理系では、「ポートメソッドが送信と受信の2つに明確に区分されている」「ネットワークトポロジの組み立てと実際のメッセージ発火が別々になっている」という言語仕様上の特長を生かして、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、粒度の細かい並列性が実現されている。

Implementation of Produce/1, a collaboration based parallel object-oriented language

Naoyasu UBAYASHI　　Atsuo OHKI　　Yasushi KUNO
Graduate School of Systems Management, University of Tsukuba

State-of-the-art parallel object-oriented languages do not provide us with the means for expressing collaboration of objects explicitly, and make it difficult for us to understand system's behavior as a whole. We already proposed *Producer Model* and its description language *Produce/1* to solve this problem. We have developed *Produce/1* language processor. Source programs in *Produce/1* are translated to C++ source programs, in which Solaris2.3 multithread libraries are called. This translator is implemented using C++ and lex/yacc, and runs on Solaris2.3.

In *Actor Model*, not only objects *build a network topology*, but also *send message to each other*. It is difficult for us to understand system's behavior as a whole. On the other hand, in *Producer Model*, the former calculus is distinguished from the latter calculus. The former calculus is executed by a *producer* object, and the latter calculus is executed by an *actor* object. These calculus are executed in parallel. If we think of *Actor Model* as a *model only with message goto statement*, *Produce/1* is a *model structured by message topology*.

Generally, parallel object-oriented laguage processors assign one thread to one object. On the other hand, *Produce/1* language processor can assign one thread to one *send/receive port method*, since in *Produce/1* (1)*send port method* is distinguished from *receive port method*, and (2) *building a network topology construction* is distinguished from *message fire*. In *Produce/1*, granularity of parallelism is very small.

1 はじめに

現在、並列計算モデルとして、メッセージ通信に基礎を置くモデル (アクタモデル、CSP、CCS) やタプルスペースをベースとする Linda などが提案されている。これらには基本的なモデリング機能しかないため、計算主体 (本稿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複雑に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」という観点からのモデリング能力が弱い。筆者らは、このような問題を解決する1つの方法として「プロデューサモデル」とその記述言語 **Produce/1** を提案しているが [1][2]、今回その処理系を開発した。本稿では、Produce/1 の実装方式を中心にその言語特長について紹介する。

2 プロデューサモデルと Produce/1

2.1 プロデューサモデル

プロデューサモデルはアクタモデル同様、メッセージ通信に基礎をおく並列計算モデルである。アクタモデルでは「各オブジェクトのメソッド中で関係する相手へメッセージ送信する」という形でしか協調動作が表現できないため、大域的な観点から各オブジェクトの役割を理解することは難しい。プロデューサモデルでは並列システムを「演劇」に例えている。俳優オブジェクト群とそれらのコーディネータ役を果たすプロデューサ・オブジェクトから構成され、それを1つの演劇単位 (グループ) と考える。メッセージ通信の制御権をプロデューサ・オブジェクトのみに制約しているため、システム全体の協調動作はプロデューサ・オブジェクトの内容を階層的に追って行けば理解できるようになっている (図1)。

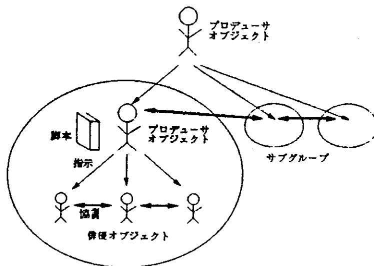

図 1: プロデューサモデル

2.2 Produce/1 の言語仕様

Produce/1 のソースプログラムは、図2に示すようなクラス定義の集合から構成される。

```
//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 2: Produce/1 のクラス定義

group: で始まるセクションにはグループのメンバとなる俳優オブジェクトを列記する。クラス定義中にこのグループ文が含まれるとプロデューサとなる。**state:** で始まるセクションにはクラスが取りうる抽象状態を列記する。**$state** は Produce/1 が管理する変数であり、状態値の代入 / 参照が可能である。**var:** で始まるセクションにはクラスの属性を記述する。**sendport:** および **recvport:** で始まるセクションには、各々送信 / 受信ポートメソッドの宣言を記述する。Produce/1 では、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実行されるようになっている。クラス名とポート名の間に存在するオペレータはメッセージ通信の方向を示しており、'<::' は送信を、'::>' は受信を意味する。

実装部には、ポートメソッドの処理手順が記述される。特に、プロデューサ・オブジェクトのポートメソッドの中には、傘下の俳優オブジェクト群の協調動作を記述するためのメッセージ接続文 (同期型と非同期型がある) が記述される。

2.3 Produce/1 における協調動作表現

Produce/1 では、プロデューサ・オブジェクトが傘下の俳優オブジェクト群の送信 / 受信ポートを接続するより、ネットワークトポロジを組み立てる。このネットワークトポロジは俳優オブジェクト間の協調関係を示しているが、接続の時点では、メッセージをやり取りするための通信路ができただけであり、実際のメッセージはまだ発火しない。メッセージが発火するのは、送信側および受信側のオブジェクト

の状態がポートメソッド宣言部に設定された事前抽象状態になったときである。

アクタモデルでは各オブジェクト群が「ネットワークトポロジの組み立てのための計算」と「メッセージ通信とそれに関わるオブジェクト内計算」を同じ枠組の中で行っていたため、システム全体の動作が理解しづらくなっていた。一方、Produce/1 では、これら 2 つの計算を明確に分離して 2 階層の並列化を実現し、各々をプロデューサ・オブジェクトと俳優オブジェクトに役割分担させている。アクタモデルが「メッセージ goto 文しかない計算モデル」だとすると、Produce/1 は「メッセージトポロジが構造化された計算モデル」と考えることができる。

3 Produce/1 によるプログラミング例

3.1 哲学者の食事問題の記述

Dijkstra の「哲学者の食事問題」は資源競合問題として有名である。以下に示すプログラムでは、哲学者をプロデューサ・オブジェクトに、フォークを俳優オブジェクトに割り付けている。実際には各哲学者の上に全体をまとめるプロデューサ・オブジェクトがもう 1 つ必要であり、何人の哲学者が食事に参加するかを記述する。全体がどのような協調動作をするかは、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。

```
//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};
}
//哲学者のクラス定義(プロデューサ)
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};
}
//哲学者の事象トレース
recvport void Phil::life()
{ while(true)
  { $state={thinking};
    p1_sprint("thinking");  p1_sleep(1);
    //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
    s_msg( self, *f1 & *f2, pick );

    $state={eating};
    p1_sprint("eating");     p1_sleep(2);
    //非同期型メッセージ接続('|'は非決定性を示す)
    a_msg( self, *f1 | *f2, release );
  }
  $state={end};
  return;
}
```

s_msg(同期型メッセージ接続文) および a_msg(非同期型) は共に哲学者から 2 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 の時であるが、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。

個々の哲学者オブジェクトの動きは上述の通りであるが、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 の言語仕様上の特長は、哲学者間の「競合関係」をほとんど意識しなくても記述できる点にある。個々の哲学者オブジェクトの動作を記述するだけで良く、競合関係の回避は、Produce/1 の処理系が、メッセージ発火の「同時性」や「非決定性」を認識して行ってくれる。

3.2 トーナメント問題の記述

Produce/1 の特長の 1 つとして、ネットワークトポロジが定義できることがある。ここでは、4 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。

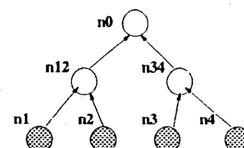

図 3: トーナメント問題のネットワークトポロジ

「トーナメント問題」のプログラムは以下のようになる。God::life で図3に示すネットワークトポロジを定義している。ネットワークを構成する各ノードは下の2つの子ノードから値を受け取って、大きい方を親ノードに渡す役割を果たす。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行き、最後に最大値が求まる。

```
// ノードのクラス定義 (俳優)
class Node
{ state:    $state=(noval, oneval, twoval);
  var:      int v1, int v2;
  sendport: void Node<::setval{twoval}(int *p){noval};
  recvport: void Node{}(int v){noval|twoval};
            void ~Node{noval}(){};
            void Node::>setval{noval|oneval}(int *p)
                                  {oneval|twoval};
            void prval{twoval}(){twoval};
}
sendport void Node::setval(int *p) //p は送信パラメタ
{ if( v1 > v2 ){ *p = v1;}
  else         { *p = v2;}
  return;
}
recvport void Node::setval(int *p) //p は受信パラメタ
{ if($state=={noval})       { v1=*p; $state={oneval};}
  else{if($state=={oneval}){ v2=*p; $state={twoval};}}
  return;
}
// 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};
}
// トーナメントの事象トレース (ネットワークトポロジを定義)
recvport void God::life()
{ $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();             // プログラム終了
  return;
}
```

アクタモデル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、1つのメソッドの中でどの子ノードから値を貰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、1つにまとめて記述しなければならない。Produce/1 ではポートメソッドに送信と受信の2種類があるため、従来型の並列オブジェクト指向言語よりも簡潔に記述することができる。

4 Produce/1 の実装方式

4.1 Produce/1 言語処理系の概要

Produce/1 の処理系は、Solaris2.3 上で稼働する 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 (図4)。Produce/1 ランタイムライブラリは、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り付けなどに関する部分を受け持っている。

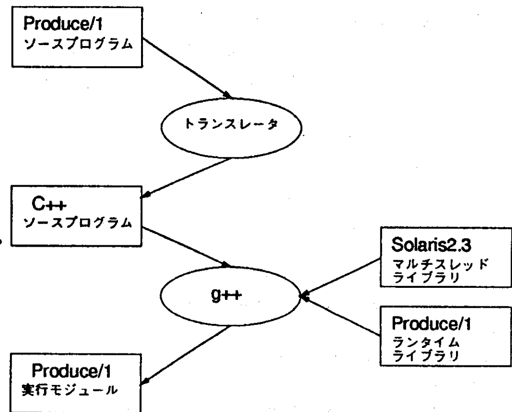

図 4: 実行モジュールが生成されるまでの流れ

4.2 並列実行とスレッドの生成過程

Produce/1 では、図5のようにプログラムの実行が開始されると、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。

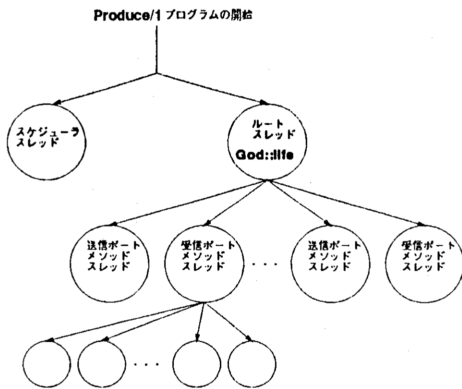

図 5: Produce/1 におけるスレッドの生成過程

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、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る (図 6)。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、スケジューラの他には 1 つしかなかったスレッドが鼠算式に増えて、各々が並列に実行されるのが Produce/1 の特長である。

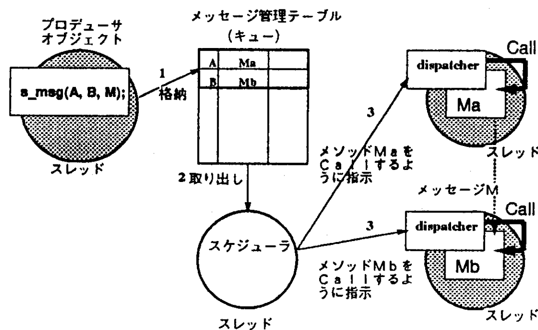

図 6: Produce/1 の実装方式

Produce/1 の処理系では、送信 / 受信ポートメソッド単位にスレッドを割付けており、粒度の細かい並列性を実現している。通常の並列オブジェクト指向言語の処理系ではオブジェクト単位にスレッドを割り付けるのが一般的であるが、Produce/1 では、「ポートメソッドが送信と受信の 2 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」という言語仕様上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ることができる。

4.3 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。いま、図 7のように、隣合った 2 人の哲学者の事前抽象状態が両方共 thinking になり、両脇にあるフォークの事前抽象状態が全て down になったとしよう。このとき、哲学者は 2 人共、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 1 つしかないので、競合してしまう。

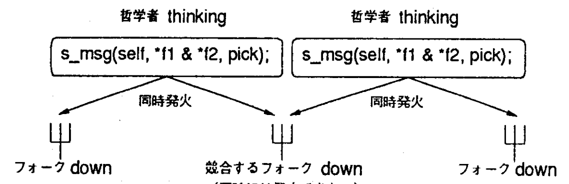

図 7: 競合問題の回避

Produce/1 の処理系では、1 つのメッセージ管理テーブルと 1 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 1 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。ただし、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。したがって、実行のたびに食事をする順番が異なることになる。

なお、プログラム規模が大きくなった場合は、メッセージ管理テーブルを n 区画に分割し、各区画に 1 つずつスケジューラを割り当てればよい。

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。どの哲学者が食事するかがデッドロックに陥ることなく「非決定的」に決まっている様子が分かる。

```
% 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
  :
  :
```

5 考察

今回は、4個のCPUを持つSMP型並列マシンSPARC1000上でProduce/1の処理系を実装した。以下は実装を通じて得られた知見である。

- 並列性の単位粒度が小さい。これは、送信/受信ポートメソッド単位にスレッドを割付けることで可能になっている。
- プログラム作成に当たって並列性を意識する必要がない。Produce/1では、ネットワークトポロジと各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。
- ネットワークトポロジの部品化とその再利用が可能である。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ネットワークトポロジがアルゴリズム上重要となるプログラムを容易にかつ分かりやすく記述できる。
- Produce/1プログラムを実行させると非常に多くのスレッドが生成され、性能上のネックになる可能性がある。単純なメッセージ接続を行うプログラムでベンチマークテストをしたところ、発生メッセージ数が100程度までは線形に処理時間が増加するが、それを越えると非線形に増加することが分かった。大規模な並列計算の処理時間については今後改善して行きたい。

6 議論と今後の課題

並列分散環境下でのオブジェクト間の協調動作は外部環境の変化によりその形態が変化するが、現状のProduce/1では動的に変化するネットワークトポロジを大域的に表現することが難しい。この問題を解決する方法として、現時点で考えている「型遷移」の構想について簡単に述べることにする。Produce/1において各オブジェクトは型を持つ(ここではクラスと同一視する)。現状では、型は外部環境が変化しても変わらないが、型遷移では1つのオブジェクトは複数の型状態を持ち、外部環境の変化により、その型状態を遷移させて行くと考える。型はProduce/1においてオブジェクト間の協調動作を規定するが、これが外部環境の変化によって変化するのである。

図8は、型遷移の概念を示したものである。クラスGは2つの型状態AとBを持つ。送信/受信ポートメソッド宣言部がvirtualになっているが、これは実際のポートメソッドが型状態によって異なることを示している。型状態がAの場合は、Gaというサブクラスに定義されたメソッドが適用される。

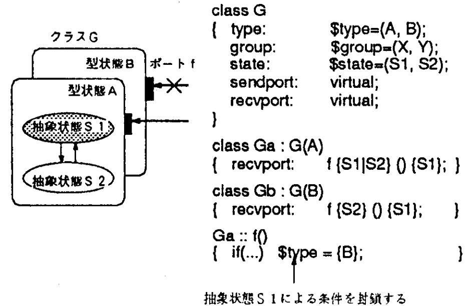

図8: 型遷移の概念

クラスGに属するオブジェクトの型状態がAでその抽象状態がS1とする。この場合、ポートfへのメッセージ受信は許可される。型状態がAの場合は、サブクラスGaに定義された受信ポートメソッドが適用され、その発火条件である事前抽象状態にS1が含まれているからである。ところが、同じオブジェクトが型状態Bに遷移すると、ポートfへのメッセージ受信は拒否されてしまう。この場合には、サブクラスGbに定義されたメソッドが適用され、その発火条件にS1が含まれていないためである。このように、Produce/1に型遷移の概念を導入することにより、簡潔に外部環境に適合するためのモデルを表現することが可能になる。

7 おわりに

本稿では、オブジェクト間の協調動作を記述する能力を持った言語Produce/1の概要とその実装方式について述べた。今後は、型遷移の概念を導入するなど、環境適合型のより柔軟なオブジェクト間の協調動作記述を実現して行く予定である。

参考文献

[1] 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, 情報処理学会研究報告, 95-PRO-1, pp.41-48(1995).

[2] 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回大会, pp329-332(1995).