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−

鵜林 尚靖　　久野 靖

筑波大学 大学院 経営・政策科学研究科 経営システム科学専攻

オブジェクト指向言語には、「継承」「メッセージ通信」「カプセル化」などの機構により、実世界を自然にモデリングできるという極めて優れた特長がある。しかし一方では、オブジェクト間の協調動作が明示的に表現出来ないため、「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」という問題が存在している。

本稿では、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** を提示する。「プロデューサモデル」とは、複数のオブジェクト群がプロデューサ・オブジェクトの指示の元で協調動作するようなモデルである。プロデューサ・オブジェクトが発する指示をトレースすることにより、オブジェクト間の協調を明示的かつ階層的に理解することができるという利点を持つ。

Proposal on Object Collaboration Model

- **Producer Model** and its Description Language -

Naoyasu Ubayashi　　Yasushi Kuno

Graduate School of Systems Management, The University of Tsukuba, Tokyo.

Object-oriented language offes powerful abstraction mechanisms(*e.g.,* inheritance, message passing, encapsulation and so on), so that we can model real world easily. However, current object-oriented language does not provide means for expressing collaboration of objects explicitly, and make it difficult to understand systems's behavior as a whole.

In this paper, we provide **Producer Model** and its description language **Produce/1** to solve this problem. In **Producer Model** , objects collaborate each other, receiving instruction from *producer* object. The *producer* object's instruction provides means for explicit and understandable description of whole system's hierarchical collabolation structure.

1 はじめに

今日、並列計算モデルとして有望視されているものの1つとして「オブジェクト指向計算モデル」がある。「オブジェクト指向」とは、実世界に存在する「もの」に着目してモデル化する手法であり、問題対象を自然にモデル化できるという極めて優れた特長がある。しかしながら、オブジェクトの自律性を強調するあまり、オブジェクト間の相互作用や協調動作を明示的に表現することが困難になっている。そのため、オブジェクト指向計算モデルに対して、現在では以下のような問題点が指摘されている。

- 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。
- オブジェクト指向のシステムはテストが困難である。

本稿では、オブジェクト間の相互作用や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 を提示する。「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の指示の元で協調動作するようなモデルである。

2 並列システムとオブジェクト間協調

並列システムにおいて、各オブジェクトは自律的かつ独立に動作する。しかしながら、これらのオブジェクトが、お互いに全く関係を持たずに動作する状況は稀である。関係するオブジェクト間でグループを構成し、協調動作をしつつ、システムとしての機能を実現している場合がほとんどである。

このような並列システムを構築する場合、一般的に、「分散指向」と「集中指向」の2つの視点が存在する。「分散指向」とは、システムを構成するオブジェクトをなるべく少ない制約の元で自律的かつ独立に動作させることを目指した視点である。一方、「集中指向」とは、「階層性」や「グループ化」の概念を取り入れて、システムを構成するオブジェクトの振る舞いを大域的にモデル化することを目指した視点である。

一般に、並列システムを構築するプログラマは、分析／設計時に何等かの形で、「システムに求められるオブジェクトは何で、それらをどのように配置し協調動作させたら良いか」を考えているはずである。これは、「集中指向」でモデル化していると言うことができる。このようにモデル化した結果は実装する段階になって改めて、プログラマの手によって、制約の少ない「分散指向」のプログラミング言語に「翻訳」される。なぜならば、現状のオブジェクト指向言語では、「集中指向」を陽にサポートする機構を持たないからである。すなわち、分析／設計段階で考えたことと、実際のプログラミング言語で表現した内容には、ギャップが存在するのである。そして、このギャップのために、以下のような問題が生じる。

- 分析／設計段階に考えたことが、プログラミング言語に用意された機能のみでは素直に表現出来ない。そのため、プログラマに「モデル翻訳」という作業を強いることになる。
- 実装されたプログラムを見ても、階層性やグループ化の概念が明示的に表現されていないため、「そのシステムが何を計算するのか？」を、開発に携わったプログラマ以外の者が理解することは必ずしも容易ではない。

以上のような問題点を解決するためには、計算実行としては「分散指向」を、計算表現（記述）としては階層性やグループ化の概念を取り入れた「集中指向」を目指した統一的な計算モデルが求められる。

3 プロデューサモデル

3.1 アクタモデルの限界

現在、オブジェクト指向計算モデルの理論的基盤となっているものとして、C.Hewitt により提唱されたアクタモデルがある [1]。これは、なるべく少ない組み込み機能で、あらゆる現象を表現することを目指したシンプルで強力なモデルである。このモデルでは、アクタと呼ばれる計算主体 (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持つ。また、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。すなわち、アクタモデルは、前節で述べた「分散指向」の計算モデルに該当する (図 1)。

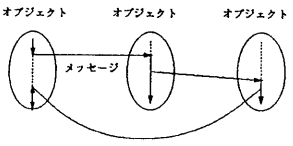

図 1: アクタモデル

このように、アクタモデルはシンプルで強力であるが、その代償として、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 (処理手続き) 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。すなわち、オブジェクト間の協調動作を表す事象トレース(メッセージ通信の時系列表現)が各オブジェクトの中に分散的に内包化されてしまっているのである。そのため、全てのオブジェクトの処理内容を順に追って行かなければ、システム全体の理解が出来ない仕組みになっている。

3.2 プロデューサモデルの発想

オブジェクト間の協調動作を表す事象トレースが各オブジェクトの中に分散的に内包化されてしまうということは、本来、オブジェクトの自律性を考えると不自然なことである。事象トレースとは、あるコンテクストの中で各オブジェクトが振る舞う行為であり、各オブジェクトの主体性そのものとは明らかに異なる。

並列システムを、各俳優がいくつか集まって演ずる「演劇」に例えてみよう。各俳優(オブジェクト)の能力は、ある特定の演劇に依存しないはずである。プロデューサの指示にしたがって1つの劇を演じているのに過ぎない。一方、プロデューサの指示の基になる「脚本」(オブジェクト間協調のためのシナリオ)も、ある特定の演劇に依存しないはずである。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、各俳優の能力と脚本とは、まったく独立の概念なのである。しかしながら、現在の計算モデルでは、これらの概念が区別されておらず、前述のような問題として表れている。

本稿で提案する「プロデューサモデル」は、この「演劇」からのアナロジーをそっとソフトウェアの計算モデルにマッピングしたものである(図2)。

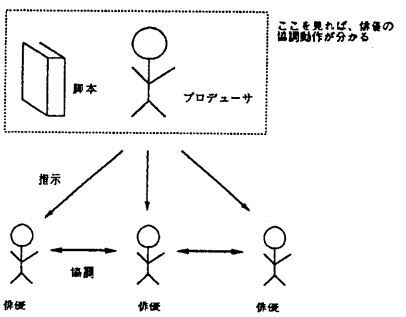

図 2: 「演劇」と計算モデル

3.3 プロデューサモデルの概要

3.3.1 集約関係によるグループ化

通常のオブジェクト指向計算モデルでは、基本的な計算要素として「継承関係」「メッセージ通信」「カプセル化」などが用意されている。これらの計算要素は、「分散指向」のモデル化には適しているが、「集中指向」の用途に対しては不十分である。プロデューサモデルは、プロデューサ・オブジェクトが、傘下にある複数の俳優オブジェクトに指示を出しながら、「演劇」というグループワイドなオブジェクト間協調の計算を行うモデルである。

このような計算を行うため、プロデューサモデルでは、新たな計算要素として「オブジェクト間の集約関係」を導入している。図3に示すように、プロデューサ・オブジェクトと各俳優オブジェクト間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ことにする。このグループは更に階層化することが可能であり、より大きな「演劇」(システム)にも対応することが出来る。また、メッセージ通信は、「演劇」の責任者であるプロデューサ・オブジェクトのみが制御できるものとする。このため、プロデューサ・オブジェクトの内容のみを追って行けば、オブジェクト間の協調を明示的に理解することが出来る。

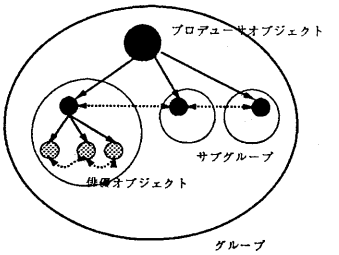

図 3: プロデューサモデル

3.3.2 協調動作とコミュニケーション

プロデューサ・オブジェクトが制御出来るメッセージ形態としては以下の3種類がある。

1. プロデューサ・オブジェクトの管理下にある俳優オブジェクト間のメッセージ通信
2. プロデューサ・オブジェクトと俳優オブジェクト間のメッセージ通信
3. 契約オブジェクトとのメッセージ通信

モデルの基本は1および2のメッセージ形態である。3は、システム内で共通に利用できるオブジェクト部品(プロデューサモデルでは「契約オブジェクト」と呼んでいる)の機能を利用したい場合に用いる。3のメッセージ形態を用意したのは、共通オブジェクトにアクセスするのに、プロデューサ・オブジェクトの指示を仰がなければならないのは非効率であると考えたからである。

なお、プロデューサ・オブジェクト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ことになっている。これにより、計算実行としての「分散指向」性が保障される。

```
//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:  相手先クラス名 :: ポート名 { 事前抽象状態 }(引数){事後抽象状態 };
    recvport:  相手先クラス名 :: 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 4: Produce/1 のクラス定義

4 記述言語 Produce/1

4.1 Produce/1 の言語仕様

本節では、プロデューサモデルに基づくプログラミング言語 Produce/1 について述べる。Produce/1 は、実用プログラムの記述に耐えることを目標にしたC++ライクなコンパイラ型の言語である。現在は、その言語仕様を開発している段階である。以下に述べる仕様は、プロデューサモデル特有なものに限定した。したがって、継承機構など一般の並列オブジェクト指向言語に備わっているものは含まれていない。

4.1.1 プログラムの構成

Produce/1 のソースプログラムは、通常のオブジェクト指向言語と同様、クラス定義の集合から構成される。プロデューサモデルにおけるプロデューサ・オブジェクトと俳優オブジェクトの区分は言語仕様上はなく、すべて同じクラス定義の構文にしたがって記述される。両者を区別するのは、クラス定義中にグループ定義文があるかないかだけである。また、グループ定義が階層化されることにより、俳優オブジェクトは、傘下のグループに対しては、プロデューサ・オブジェクトとして振舞うことが出来る。図4に、クラス定義の形式を示す。各構文の意味は、次節以降、詳細に述べる。

4.1.2 グループ化機構

グループ定義は、group: で始まるセクションに記述する。ここには、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、そのクラスは、プロデューサとなる。あるクラスが、傘下に actor1、actor2、actress1 という3つのメンバを持つ場合、グループ定義文は以下のように記述される。ここで、Actor、Actress は、メンバが所属するクラス名を表すものとする。

```
group:    $group=(Actor actor1, Actor actor2,
                  Actress actress1);
```

また、グループ中の構成メンバ数がプログラムの実行と共に変化する場合は、以下のようにポインタを使用すればよい。この場合は、Actor および Actress に所属するメンバ数が可変の場合にも対応出来る。

```
group:    $group=(Actor *actor, Actress *actress);
```

4.1.3 抽象状態管理機構

Produce/1 では、オブジェクトの状態を抽象化して管理する機構(抽象化状態)を備えている。これにより、並列オブジェクト間の同期を図ることが可能になると共に、状態変化に対するエラー検証を静的または動的に行うことが出来る。抽象化状態は、俳優オブジェクトの場合は、そのオブジェクト単独の状態を表し、プロデューサ・オブジェクトの場合は、グループ全体の状態(大域状態)を表している。Produce/1 には、前述のように「グループ化機構」が備わっているため、システム全体の状態を階層的かつ抽象的に表現することが出来る。

抽象化状態は、state: で始まるセクションに記述する。ここには、そのクラスが取りうる状態を列記する。あるクラスが、exec、ready、wait の3状態を持つ場合は、以下のように記述される。

```
state:    $state=(exec, ready, wait);
```

ここで、$state は Produce/1 が管理する変数であり、プログラム中ではカレントの状態を表している。$state は、通常の変数と同様に、値(この場合は状態値)を代入したり、参照したり出来る。

4.1.4 ポート機構

Produce/1 では、全てのメッセージ通信は「ポート」経由で行われる。ポートとは、メッセージ送受信の出入り口を指しており、ポートにメッセージが届くと、そのポートに対応するポートメソッド(ポート名と同一の名前を持つ)が起動される。すなわち、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ことになる。受信ポートメソッドは、通常のオブジェクト指向言語におけるメソッドと同様で、メッセージを受け取った後の処理を記述したものである。一方、送信ポートメソッドは、メッセージ送信する前に行う処理を記述したものである。

通常のオブジェクト指向言語では、(受信)メソッドの中で依頼処理を行ったり、関係するオブジェクトにメッセージを送ったりするが、Produce/1 では、メッセージの受信と送信は明確に分かれている。これは、プロデューサモデルの考え方から必然的に導かれる性質である。(受信)メソッドの中で、メッセージ送信を行うことは、第3節でも述べたように、「オブジェクト間の協調動作を表す事象トレースを各オブジェクトの中に分散的に内包化させてしまう」ことになるからである。プロデューサモデルでは、プロデューサ・オブジェクトが、傘下の俳優オブジェクト間のメッセージ制御を行うので、送信ポートメソッドと受信ポートメソッドは明確に分けておく必要がある。メッセージ制御のメカニズムについては、次節で更に詳細に述べることにするが、Produce/1 では、メッセージ通信は、同じポート名間で送受信される。したがって、メッセージ名とポート名は同じ名前で代用している。

ポートは、sendport:(送信ポート)、recvport:(受信ポート)で始まるセクションに、それぞれ以下のように記述する。

- 送信ポート: 相手先のクラス名(複数指定可能)、ポート名、メッセージと共に渡される引数、ポートメソッドが起動されるための事前抽象状態と事後抽象状態を列記する。
- 受信ポート: 送信ポートの場合と同様であるが、相手先のクラス名は省略することができる。特定の相手からしかメッセージ受信をしない場合のみ記述すればよく、さまざまな相手からメッセージ受信をする場合は記述する必要はない。

Actress クラスに属するオブジェクトが、Actor クラスに属するオブジェクトから、stop というメッセージを受け取る場合は、クラス定義中に以下のように記述する。ただし、受信側の事前抽象状態は exec または ready とする。

```
recvport:  Actor::stop {exec | ready} ( ) {wait} ;
```

4.1.5 メッセージ制御機構

第3節で述べたように、プロデューサモデルでは、グループ内のメッセージは契約オブジェクトとの通信を除き、すべてプロデューサ・オブジェクトが制御する。ここでは、通常のメッセージ形態(俳優-俳優間およびプロデューサ-俳優間)の制御機構について述べることにする。

Produce/1 では、メッセージ制御の内容(オブジェクト協調を表す事象トレース)はプロデューサ・オブジェクトの受信ポートメソッドとして記述される。プロデューサ・オブジェクトが他のオブジェクトからある機能処理を行うようにメッセージを受けた場合に、その事象トレースの内容が実行されるのである。

メッセージ通信は、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ポートと受信ポートを接続することにより行われる。ただし、第3節でも述べたように、プロデューサ・オブジェクト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5)。

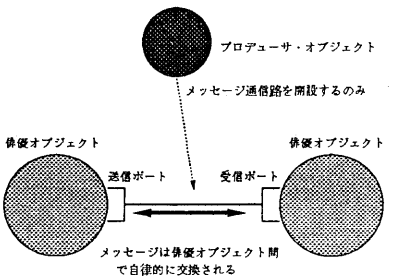

図 5: メッセージ制御機構

Produce/1 がサポートするメッセージ接続形態には、以下の2つがある。

- 同期接続: この場合は、送信側オブジェクトは受信側オブジェクトからの返答を必要とする。返答を貰うまでは事象トレースの次のステップに進むことが出来ない。
- 非同期接続: この場合は、送信側オブジェクトは受信側オブジェクトからの返答を必要としない。したがって、メッセージ接続が完了し次第、事象トレースの次のステップに進むことが出来る。

Produce/1 では、同期接続の場合は s_msg 文を、非同期接続の場合は a_msg 文を使用する。構文は両者とも同じで、送信オブジェクト名、受信オブジェクト名、ポート名（メッセージ名）を列記する。s_msg 文の返却値は受信オブジェクトからの返却値と同じである。a_msg 文の場合は返却値はない。以下に示すのは、図 6 に示す事象トレース図を Produce/1 で記述したものである。このように、プロデューサ・オブジェクトの中に記述されたメッセージ制御のみを追って行けば、オブジェクト間の協調を明示的に理解することが出来るようになっている。

```
s_msg( actor1, actor2, M1 );
a_msg( actor2, actress1, M2 );
a_msg( actress1, actor1, M3 );
```

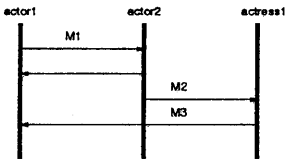

図 6: 事象トレースの例

4.1.6 ネットワーク配線とメッセージ発火

ここでは、プロデューサ・オブジェクトが行うネットワーク配線作業とその配線上でやり取りされる俳優オブジェクト間のメッセージ通信の関係について、もう少し堀下げて述べることにする。

まず、プロデューサ・オブジェクトは俳優オブジェクト間にメッセージ通信路を開設することにより、システムのネットワークトポロジを組み立てる。この時点では、メッセージをやり取りするための通信路が出来ただけであり、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、以下に示す条件が、送信側の俳優オブジェクトと受信側の俳優オブジェクトの双方で整った時である。

- 送信側および受信側のオブジェクトの状態がポート記述部に設定された事前抽象状態になった。
- 送信ポートのパラメータが具現化された。

プロデューサモデルの特長は、ネットワークトポロジの組み立てとメッセージの発火が分離されているところにある。計算の進行と共に、プロデューサ・オブジェクトが先導してシステムのネットワーク配線の接続や繋ぎ変えを行い、俳優オブジェクトは発火条件が満たされた時点で、分散かつ自律的にメッセージ通信を行うのである (図 7)。

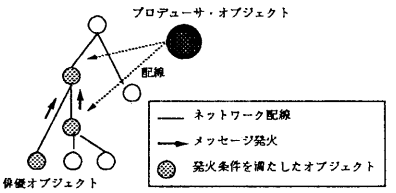

図 7: ネットワーク配線とメッセージ発火

従来のアクタモデルでは、各アクタ (オブジェクト) 群がネットワークトポロジの組み立てとメッセージ通信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、プロデューサモデルでは、「ネットワークトポロジ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」とを明確に分離して 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。このことにより、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。

Produce/1 によるメッセージ発火の仕組みを以下の例に沿って説明する。Actor クラスに属するオブジェクトから Actress クラスに属するオブジェクトへの stop というメッセージが発火するのは、送信側の抽象状態が exec、受信側の抽象状態が exec または ready になり、更に送信パラメータ sleepTime1 が具現値 (例えば 10 など) を持ったときである。メッセージが発火すると、送信パラメータの具現値は、受信側のパラメータ sleepTime2 にバイディングされる。

```
(Actor クラスでの定義)
  sendport:  Actress::stop  {exec}  (int sleepTime1 )
                                    {exec} ;
(Actress クラスでの定義)
  recvport:  Actor::stop  {exec | ready}  (int sleepTime2 )
                                    {wait} ;
```

4.2 Produce/1による記述例

ここでは、Produce/1で記述したプログラム例を紹介する。

4.2.1 MVCモデルの記述

MVCモデルは、Model、View、Controllerという3つのクラスから構成されるユーザインタフェース・システムであり、フレームワーク部品の典型的な事例とも言える。以下は、BasicMVC[3]をProduce/1で記述した例である。ここでは、Controllerをプロデューサ・クラス、ModelとViewをその傘下の俳優クラスとした。メッセージ接続は、ControllerからModelへのbuttonAction、Viewへの displayViewとdisplayの3つである。メッセージ接続文中のselfは、自分自身（この場合はController）からのメッセージであることを示している。例では、プロデューサ・クラスのみ記述し、俳優クラスについては省略した。

```
// BasicMVCのクラス定義
class Controller
{  group:     $group=(Model M, View V);
   state:     $state={start, loop, activity, update, stop);
   sendport:  M::buttonAction {activity} () {update};
              V::displayView {start} () {loop};
              V::display {update} () {loop};
   recvport:  life {start} () {stop};
              isControlActive {loop} () {activity | stop};
}

// BasicMVCの事象トレース
Controller::life()
{  $state = {start};
   s_msg( self, V, displayView );
   $state = {loop};
   while( $state=={loop} )
   {  if( s_msg( self, self, isControlActive )==TRUE )
      {  $state = {activity};
         s_msg( self, M, buttonAction );
         $state = {update};
         s_msg( self, V, display );
         $state = {loop};
      }
      else
         $state = {stop};
   }
}
```

4.2.2 輪の王問題の記述

「輪の王」問題とは、ネットワークを介してリング状に結合された複数のノード（固有の値を持つ）の中から、最も大きな値を持つノード（王）を選び出す問題である。これは、単一のクラスに属するオブジェクトが複数集まって協調動作をする例である。以下のプログラムでは、あるノード（首謀者）が自分自身が王かどうか見極めることができるようになっている。ここでは、Godというプロデューサが、複数の俳優（Node）に指示を出している。ただし、GodはNode間のメッセージ通信路を設定するだけで、Node自身がお互いに相談しながら自律的に計算を進めるようになっている。プログラム中、Node間でchkvalというメッセージが交換されるが、これは送信側ポートと受信側ポートを接続し、送信側のパラメータ（sendval）を受信側のパラメータ（val）にバインディングすることにより行われる。

```
// 輪の王のクラス定義
class God    // プロデューサ
{  group:     $group=(Node *N1, *N2);
   state:     $state={start, process, end};
   sendport:  Node::start {start} () {process};
   recvport:  life {start} () {end};
}

class Node   // 俳優
{  state:     $state={};
   var:       int myval, sendval;
              int leader=FALSE;  // 首謀者
              Node *right;
   sendport:  Node::chkval {} (int sendval) {};
   recvport:  Node::chkval {} (int val) {};
              {  sendval = max( val, myval );
                 if( leader==FALSE )  return right;
                 else                 return NULL;
                                      // val==myval-> 王
              }
              God::start {} () {};
              {  leader = TRUE;
                 sendval = myval;
                 return right;
              }
}

// 輪の王の事象トレース　－首謀者Nを指定して処理を開始－
God::life( Node N )
{  Node *tmp;
   $state = {start};
   N1 = &N;   N2 = s_msg( self, *N1, start );
   $state = {process};
   while( $state=={process} )
   {  tmp = N2;   N2 = s_msg( *N1, *N2, chkval );  N1 = tmp;
      if( N2==NULL ) $state = {end};
   }
}
```

5 議論

プロデューサモデルには、以下に示すような特長がある。これらの多くは、従来の「分散指向」の計算モデルおよびそれをベースとしたプログラミング言語では実現が困難であったものである。

- オブジェクト間の協調動作が明示的に表現される。
- 協調動作に対するテスト戦略を与える。
- フレームワーク部品記述のための言語的枠組を提供する。

フレームワーク部品については、オブジェクト指向分析や設計手法においも、同様ものが提案されて

いたが、あくまでも分析／設計上の概念であり、プログラミングレベルで取り扱うのが困難という問題が存在していた。プロデューサモデルでは、「グループ化機構」により、このフレームワーク部品をプログラミング言語できっちりと記述することが可能となっている。また、「階層化」の考え方により、単体のフレームワーク部品だけでなく、それらを結合した部品を記述することも出来る。

一方、現状のプロデューサモデルには、ネットワークトポロジの変化を柔軟に表現することが難しいという課題がある。今後、以下のような概念を取り入れて、解決を図って行く予定である。

- 開設されたメッセージ通信路は1回だけでなく、条件を満たせば複数回あるは永久的に利用出来るようにする。
- プロデューサ・オブジェクトが俳優オブジェクトに、一度指示すればあるパターンの範囲内でネットワークトポロジを自律的に組み変えることが出来るようにする。

6 関連研究

オブジェクト間の協調動作の記述は、オブジェクト指向分析／設計技法や仕様記述言語の立場からも研究されているが、以下に述べるように、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している。なお、本研究のように計算モデルやプログラミング言語の側面から研究した例は少ない。

6.1 分析／設計手法からのアプローチ

オブジェクト指向分析／設計手法として、各種、提案されているが、代表的なものとして、RumbaughらのOMT法やBooch法などがある。これらの手法に共通している考え方は、問題領域を、「オブジェクトモデル」「動的モデル」「機能モデル」の3側面からモデリングするアプローチをとっていることである。このうちオブジェクト間の協調に関わる表現は、「動的モデル」で行うことになっており、事象トレース図などによって表現する場合が多い。事象トレース図は、視覚的で分かり易いという長所を持つ反面以下のような問題を抱えている。

- 事象トレースで記述されるのは、あくまでもある1つの機能に対する協調動作の表現に過ぎない。したがって、事象トレース図はすべての機能に対して記述する必要があり、その数は膨大なものにならざるを得ない。
- 形式性の面で劣るため、十分な協調動作記述が出来ない。そのため、プログラミング言語への変換を行うための理論的基盤にはならない。

6.2 仕様記述言語からのアプローチ

前節で述べたオブジェクト指向分析／設計手法の非形式性を解決する試みとして、仕様記述言語によりオブジェクト群の相互作用や協調動作を形式的に記述しようという研究がいくつか行われている[2][3][4]。例えば、Helm[2]らは、オブジェクト間の処理の流れに着目したContractという言語要素を中心として設計を進めるアプローチを提唱している。

仕様記述言語に共通している点は、いずれの手法でも、「複数オブジェクトから構成されるグループを有限状態オートマトンで記述する」というアプローチを採っていることである。ただし、このアプローチは、考慮しなければならない状態の数が非常に多くなるので、仕様の記述内容がかなり複雑になるという問題を持っている。

7 おわりに

本稿では、「プロデューサモデル」とそれを記述するための言語Produce/1について述べた。プロデューサモデルは、複数の並列オブジェクト群の協調動作を表現するのに適した計算モデルであると同時に、フレームワーク部品記述のための言語的枠組をも提供する。今後は、このプロデューサモデルについて、意味論や自己反映計算(リフレクション)の面から更に研究を行う予定である。

参考文献

[1] Agha.G.: *Actors: A Model of Concurrent Computation in Distributed Systems.*, The MIT Press(1987).

[2] Helm.R., Holland,I.M. and Gangopadhyay,D.: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, *Proc. ECOOP/OOPSLA '90*, pp.169-180(1990).

[3] 中島震: オブジェクトの集団的振舞いの設計, 情報処理学会研究報告, 93-SE-95, pp.39-46(1993).

[4] 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, 情報処理学会論文誌, Vol.33, No.8, pp.1052-1063(1992).