# Distributed Object Connection: オブジェクト間の関係に基づいた記述モデル

中田 秀基, 小池 汎平, 田中 英彦
{nakada,koike,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部

**概要**

高並列処理においては問題のもつ並列度を、いかに抽出し記述するかが問題となる。本論文では高並列処理の記述に適した、オブジェクト間の関係に基づいた記述モデル Distributed Object Connection (DOC) を示し、その記述法に関して論じる。

DOC の特徴は、オブジェクトの内部状態を廃した点にある。記述対象の系の状態は、オブジェクトとオブジェクトの関係によって表現される。

# Distributed Object Connection: Discription Model Based on Connection between Objects

Hidemoto Nakada, Hanpei Koike, Hidehiko Tanaka
{nakada,koike,tanaka}@mtl.t.u-tokyo.ac.jp
Faculty of Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN

**Abstract**

For highly parallel processing, it is important that how to extract and describe parallelism of a problem. This paper proposes a description model based on connections between objects, called DOC (Distributed Object Connection). DOC model is suitable for highly parallel processing.

In this model, state of the execution is represented by connections between objects, instead of inner status of objects.

## 1 はじめに

並列計算機のプロセッサ台数は年々増加しており、1万のオーダを越える要素プロセッサを実装するマシンが議論の対象となっている。これらの高並列マシンを活用するには、対象となる問題のもつ並列性を充分に抽出し記述することのできる言語が必要である。

並列計算のための計算モデルとして、Actor[1] がある。Actor は自律的に動作するオブジェクトを主体として、それらの間のメッセージパッシングで計算を行なうモデルである。

Actor モデルは簡潔で強力な枠組であるが、より高並列な処理の枠組としてはオブジェクトの粒度が大き過ぎ、充分に問題の並列度を抽出できないことが予想される。

本稿では、より高並列な処理に向いた枠組として、オブジェクト間の関係に基づいた記述モデル DOC ( Distributed Object Connection ) を示し、その記述法と並列性に関して論じる。

2章で、DOC のモデルについて論じる。3章で、DOC の記述方式についてのべる。4章で、記述例を示す。5章で、処理系による実行例を示す。

## 2 DOC モデル

### 2.1 Actor

一般にオブジェクト指向に基づくプログラミングでは、プログラミングの対象となる世界を、いくつかのオブジェクトとしてとらえる。

対象世界をオブジェクト単位に切り分けてとらえることで、対象の認識と記述を容易にするわけである。このようなアプローチではオブジェクトはそれぞれ内部状態を持ち、対象世界全体の状態は、系全体に存在するオブジェクトの内部状態の総和として表現される。

計算とは、対象世界の状態をあるシーケンスに基づいて変化させることであるから、状態がオブジェクト単位に切り分けられた世界での計算は、個々のオブジェクトの状態を変化させることで行なわれる。オブジェクトの状態に基づいて、オブジェクトの状態を更新するわけである。これを、オブジェクト間の通信という形でとらえる。すなわち、メッセージパッシングである。計算の実行シーケンスは、オブジェクトのメッセージの解釈と他のオブジェクトへのメッセージ発行、という形で行なわれる。ここで、オブジェクトが実行主体というイメージができる。

並列計算にこのアプローチを適用すると、それぞれのオブジェクトを並行して走るプロセスとしてとらえ、それらのプロセス間の非同期なメッセージパッシングによって計算を行なうモデルが得られる。これが Actor である (図 1)。Actor モデルは、簡潔で自然なモデルであり、充分な記述力がある。

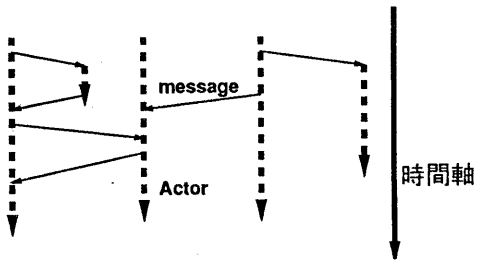


図 1: Actor の実行

Actor では計算の並列度は、その系に存在するオブジェクトの数に依存する。系に存在するオブジェクトの数は、対象となる世界によって異なるが、一般にそれほど大きい数にはならない。Actor で記述される計算の並列度は、問題固有の並列度ではなく問題の認識単位であるオブジェクトによって制限される。

これは、記述の為の単位であるオブジェクトが実行主体でもあるために、記述対象が複雑になるとオブジェクトの粒度すなわち、実行の粒度が大きくなってしまい、結果として並列度が下がっていると考えられる。

したがって、充分な並列度を生じさせるためには、オブジェクトに対するアプローチから見直す必要がある。

### 2.2 CCL

論理型言語を祖とする並列言語として、GHC や Fleng[2] などの Commited Choice Language(CCL) がある。これらの言語では、プログラムの対象となる世界は、項として表現される。項は、変数と変数の間の関係を表現するものである。プログラムは、項のリダクションルールとして記述される。計算は、ゴールのリダクションルールへのマッチングと、リダクションによっておこなわれる。プログラムの実行単位は各項 (ゴール) のリダクションであり、プログラマは実行の制御を意識する必要はない。このため、対象となる問題の並列性が無理なく抽出でき、高い並列度が得られる。

しかし、これらの言語では、言語の提供する枠組はプリミティブな表現である項のみで、抽象度の高

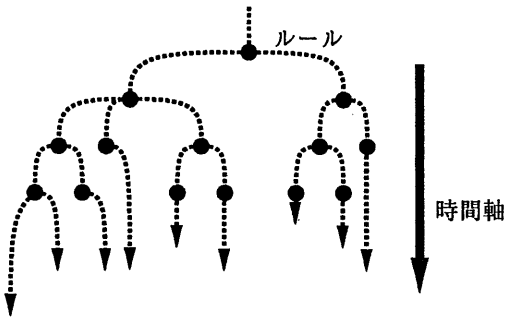


図 2: CCL の実行

いプログラムは容易には書けない[3]。また、ルールはホーンクローズであり、ルールの前提部には、単一の項しか書けない。したがって制御ののフローはツリー状になり(図2)、これ以外の構造の制御に関しては、データの依存関係などでプログラマが表現しなければならない。このため、問題の持つ並列性を自然に表現するのは難しい。

CCL をオブジェクト指向化する研究もいくつかなされているが、これらの多くはテイルリカージョンによって表現されるパーベチュアルプロセスをオブジェクトとしてみなすことで Actor モデルを実装したものである[4][5]。これらは、CCL が本来持つ高並列性とひきかえに記述力を得ようとする試みであり、Actor と同様に、問題の並列度を充分には表現できない。

## 2.3 DOC

Actor モデルは表現力はあるがオブジェクトに情報を持たせているため、記述はしやすいが並列性が出しにくい。CCL は、項にしか情報を持てないため、問題の記述が難しい。

以上のような考察に基づき、本稿ではオブジェクトにもたせる情報を最低限にし、情報をオブジェクト間の関係としてもたせることで、オブジェクトによる抽象度の高い記述を許しながら、並列度の高い実行を可能とするモデル DOC を示す。

従来のオブジェクト指向言語におけるオブジェクトは、実行制御の為の型情報と、内部状態をもっている。一般に、オブジェクトの内部状態は変数名とその変数にバインドされるオブジェクトの対であり、あるオブジェクトを主体として考えると、そのオブジェクトが、他のオブジェクトを何として認識しているかである。つまりオブジェクトの内部状態はそのオブジェクトを中心としたオブジェクト間の関係である(図3)。

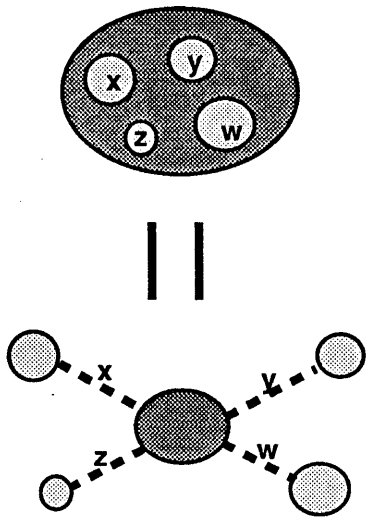


図 3: 内部状態＝オブジェクト間の関係

従って、オブジェクト間の関係を陽に記述する方法があれば、オブジェクトの内部状態が記述できる必要はないと考えられる。このため、DOC におけるオブジェクトには内部状態はない。オブジェクトは型情報(クラス)のみを持つ単なる ID である。

オブジェクト間の関係記述する方法としてコネクションを導入する。コネクションは、関係の意味を示すシンボルと、関係づけられる複数のオブジェクトのリストで表現される。 DOC によって表現される対象世界の状態は、このコネクションの集合によって定められる。

計算は、コネクションによって表現された系の書き換えによって行われる。コネクションの書き換えはリダクションルールとして定義できる。

ルールは条件が整うと自発的に発火し、新たなコネクションを系に加える。この自発的発火がプログラムの実行に相当する。CCL と異なり、複数のコネクションを前提として発火するルールが記述できるので、複雑な実行制御に関しても意識せずに記述できる。問題のもつ並列度を損なうことなく自然な記述を行なうことができる[6][7]。

コネクションの集合に対して、あるルールが発火するかどうかは、そのルールの指定するコネクション中のオブジェクトのタイプが存在するコネクション中のオブジェクトに一致するかどうかで定まる。また、複数のコネクションが同一のオブジェクトを共有することも条件とすることができる。

それぞれのルールは発火の前提となるコネクションを共有しない限り、独立に発火するので、記述された並列性を充分に生かして実行することができ

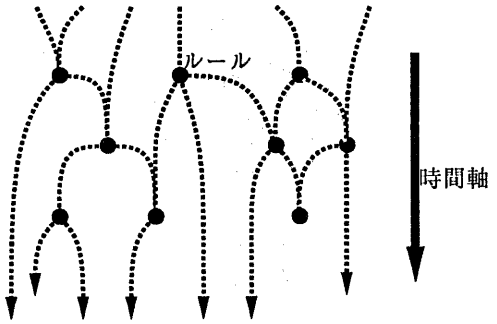


図 4: DOC の実行

る。

オブジェクトが内部状態を持たないため、オブジェクトの構造の定義は必要ない。このため、一般の言語で行なわれるような集中したクラス定義は行なわない。クラスの特徴は、そのクラスのオブジェクトに言及するすべてのルールに分散して存在する。逆にいえばルールはルール中で言及した、すべてのオブジェクトの定義の一部を担うことになる。これにより、複数のクラスに関する記述が自然に行なえる。また、インクリメンタルなプログラミングも容易になる。

Actor などで行なわれるオブジェクト間のメッセージパッシングは、メッセージを送るオブジェクトと受けるオブジェクトと、メッセージを表現するオブジェクトとを一時的に関係付けることである、と考えることができる。DOC では、メッセージパッシングに相当するオブジェクト間の相互作用も、コネクションで表現する。

次章で DOC モデルに基づく記述法に関して詳しく見る。

## 3 DOC による記述と実行

### 3.1 DOC による記述

DOC は、以下の要素で記述される。

- オブジェクト: 型のみを持つ、単なる ID
  プリミティブなオブジェクト (symbol, number 等) は値を 持つが、値の更新はできない。
- コネクション: オブジェクトとオブジェクトの関係を表現する
  関係を示すシンボルと、オブジェクトのリストで表現する。 オブジェクトは、変数名と型を：で区切って書く。

```
(symbol Object1:type1 Object2:type2 ...)
```

- ルール: コネクションとコネクションの関係を記述する
  前提となるコネクションと結果となるコネクションを ":-" で 区切って書く。また、発火の条件をガードとして記述できる。

```
con1, con2,...
        | guard :-
              conA, conB.
```

ガード には、Flat GHC と同様に、組込み述語だけが記述できる。
ルールの前提となるコネクションに存在しないオブジェクトが 結果のコネクションで必要な場合は、ルールの発火時に 新たなオブジェクトとして生成する。
また、記述を容易にするために 記号 "@" を導入する。 ルール発火の前後で変化しないコネクションに付加する。

```
con1, @con2:- conA.
```

は、

```
con1, con2:- conA, con2.
```

と同義である。

例として、下のルールを考えてみる.

```
@(aConnection X: test N:number),
 (bConnection X: test M:number)

    :- (cConnection X Y:test2).
```

これは、test 型の オブジェクト X を共有する、aConnection と bConnection に対して発火し、その際新たに test2 型のオブジェクト Y を生成し、cConnection を、X と Y の間に付加するルールである。(図 5)。

このように定義された DOC によって表現される系の状態は、オブジェクトの中にではなく、系全体の空間に分散して記述され、計算は分散したコネクションの付け換えで行なわれる。

### 3.2 DOC の実行

DOC の実行は以下のように行なわれる。

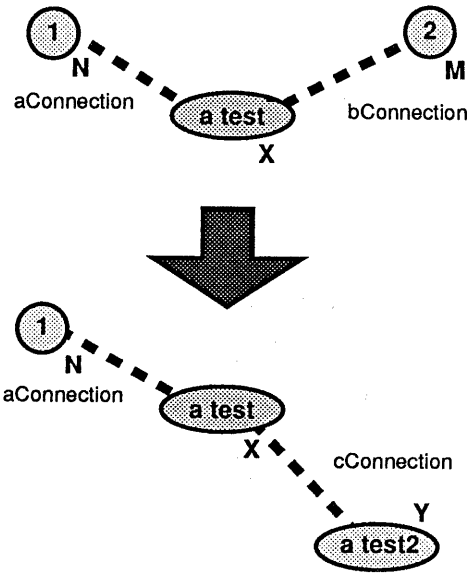


図 5: ルールの例

1. 系に存在するコネクションが、あるルールをみたすかどうか調べる。
2. 満たすならば、そのルールを発火する。
   発火は、以下のように行なわれる。
   (a) ガードを評価する。真ならば以下に進む
   (b) 発火の前提となったコネクションを系から取り除く。
   (c) 発火の結果のコネクションを系に加える。

このプロセスは、すべてのルールに関して同時に行なうことが可能である。

ここで、問題となるのは、コネクションを系から取り除く過程である。あるコネクションを系から取り去ることは、そのコネクションを前提とする他のルールの発火を制限することになる。

例えば、

```
(A x), (B x) :- (C x). %ルール1
(A x), (D x) :- (E x). %ルール2
```

というルールがあり、(B x), (D x) が存在するとき、(A x) が新たに加わったとする。この時には、ルール 1、2 双方が発火可能になるが、(A x) は発火すると消えてしまうので、両方が発火することはできない。

このため、発火が可能になった時に、前提となったコネクションにロックをかけ、コネクションを確保したルールだけが発火するように制御する必要がある。しかし、ロックを単純に行なうと、デッドロックの可能性が生じる。

下のように 2 つの前提を共有するルールを考えてみよう。

```
(A x), (B x) :- (C x). %ルール1
(A x), (B x) :- (E x). %ルール2
```

双方が発火可能になった際に、例えばルール 1 が (A x) に、ルール 2 が (B x) に、先にロックを掛けてしまうと、双方とも前提となるコネクションを確保できず、発火できなくなってしまう。

このため何らかの排他制御を、ルールに跨って行なう必要がある。この点に関しては現在検討中である。

## 4 記述の例

DOC によって実際にどのように問題が記述できるかを例を挙げて示す。

### 4.1 Point

下のような c++ で書かれたオブジェクトを考えてみよう。2 次元平面上の点で、x と y という、内部変数を持ち、move と projectionX という、メッセージを受けとることのできるオブジェクトである。

```
class Point{
  int x, y;
  void move(int x0, int y0)
        {x += x0, y += y0}
  void projectionX()
        {y = 0}
};
```

このオブジェクトは、DOC では、下のようなルールで表現できる。

```
(move P: point X0: number Y0: number),
(x    P: point  X: number),
(y    P: point  Y: number)
        :-
                (x P  X + X0),
                (y P  Y + Y0).

(projectionX  P:point),
(y  P:point  Y:number)
        :-
                (y P  0).
```

このように、DOC ではメッセージも内部状態もコネクションという同じ枠組で表現する。

ここで、point に関する宣言が陽には存在しないことに注意していただきたい。point に関する記述は、複数ののルール上に分散して存在し、一箇所にまとまった宣言部などは存在しない。point に言及するルールすべてが、point の定義なのである。

## 4.2 Best Path

ここに示すのは、重みつき無向グラフの各ノードの、スタートノードからの最短経路による距離を求めるプログラムである。

各エッジの重みは、link というコネクションで表現される。各ノードのスタートからの距離を val というコネクションで表す。

```
@(link N: bpn   M: bpn  L: number),
 (val  N: bpn  Vn: number),
@(val  M: bpn  Vm: number)

     | (L + Vm) < Vn  :-

         (val N Vm + L).
```

このただ1つのルールでベストパスを記述している。隣のノードのスタートからの距離と、そのノードとの間の重みの和(すなわち、そのノード回りの経路の距離)が、現在の距離よりも短ければ、距離を更新する、というルールである。

これに初期値として、例えば

```
(link Start: bpn  M: bpn 10)
(link Start: bpn  L: bpn 20)
(link L    : bpn  O: bpn 15)
(link M    : bpn  L: bpn 6)
(link M    : bpn  O: bpn 8)

(val Start 0)
(val M 10000)
(val L 10000)
(val O 10000)
```

を与えると、ルールが発火し val コネクションを更新して、計算が行なわれる。M,L,O に val で、適当な初期値を与えている。

発火がすべて終了した時点での、val で結ばれている値が各ノードのスタートからの最短距離である。

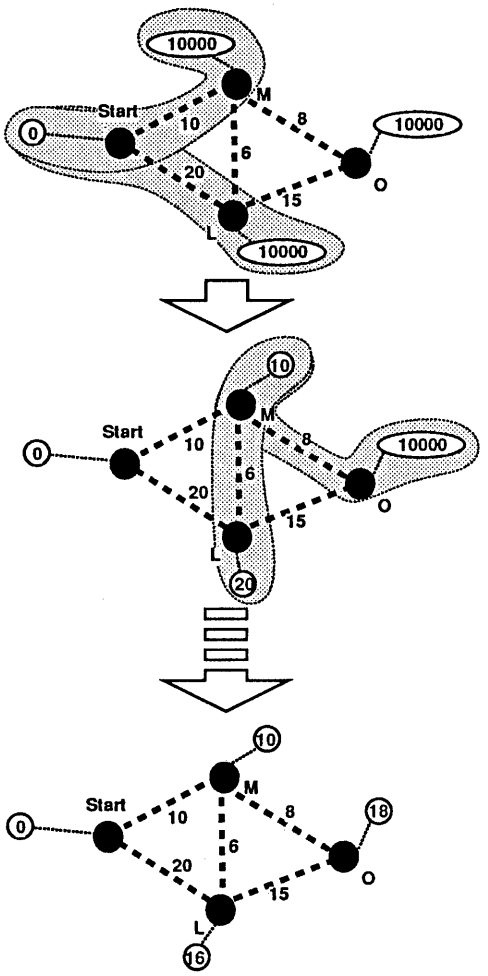


図 6: ベストパスの実行

### 4.3 Sort

ソートには、さまざまなアルゴリズムが存在する。一般に、ある値を中心に分割してソートするクイックソート、適当に分割し後でマージするマージソートなどの複雑な手法が広く用いられている。

しかし、人間が手で並び換えるときは、適当なすき間をあけて順番に並べていって、ある要素とある要素の間に入るものがあればそれをそこに移動する、というような非常に直観的なアプローチをとる。

DOCではこの人間が自然に行なうようなナイーブなアプローチを、以下の2つのルールで記述することができる。

```
@(have S:sorter N:number),
@(have S:sorter L:number)
      | N>L :-
        (mayBeNext S N L).          %ルール1

@(mayBeNext S:sorter N:number L:number)
(mayBeNext S:sorter N:number M:number)
      | L>M :-.                     %ルール2
```

これらに、以下のように初期状態をあたえる。

```
(have S:sorter 1)
(have S 9)
(have S 8)
(have S 3)
(have S 5)
```

始めのルールが、すべての要素の間に順序関係を発生させ、2つめのルールが、順序関係を比較しより近いもののみを生き残らせる。その結果、もっとも近い順序関係のみが生き残り、そのリンクが要素間の順序を示す(図7)。

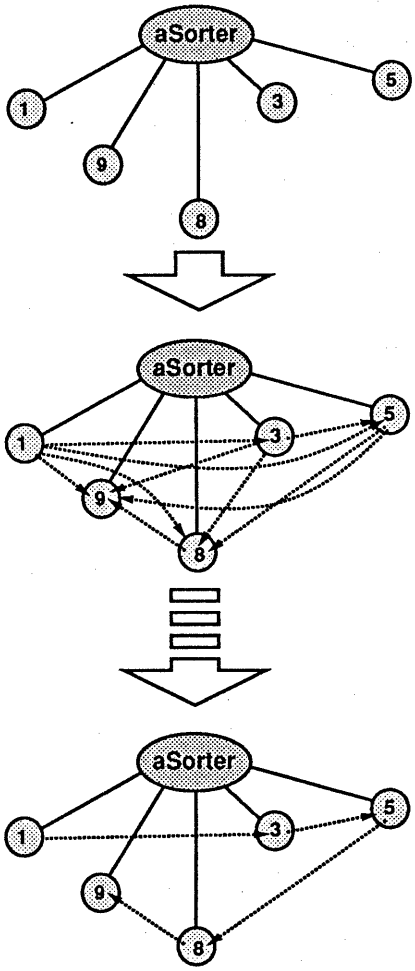


図7: ソートの実行

### 4.4 素数を求める

素数を求めるアルゴリズムとしては、エラトステネスのふるいがよく知られているが、ここではよりナイーブに、その数以下のすべての数で割って、割り切れないことを確かめている。

```
(prime N:number)                    %ルール1
    :- (makeList P:primeMaker N).

(makeList P:primeMaker N:number)
```

```
| N>1 :-
    (makeList P N - 1),        %ルール2
    (member P N).

@(member P:primeMaker N:number) ,
 (member P:primeMaker M:number)
    | (M>N) & (M % N=0)
          :-.                  %ルール3
```

初期状態として

```
(prime 10)
```

と与えると、ルール1が発火し、primeMaker オブジェクトを一つ生成し、makeList というコネクションを付加する。makeList は、ルール2によってリダクションされ、リカーシブに より少ない整数に対して makeList を作る。この時 素数であるかも知れないことを示す member というコネクションをそれぞれの数に対して付加していく。このようにして、2から10までのすべての値が member で primeMaker とつながれる。

この member コネクションは、ルール3に従って互いに消し合う。すなわち、ある数が"素数かも知れない数"で割り切れたら、その数は素数ではないので member コネクションを消去する。

最後まで残った member コネクションによって primeMaker に接続している数が素数である (図8)。

以上に示したように、オブジェクト間の関係に基づいて記述する方法では、人間がすぐに思いつくようなナイーブな方法をそのまま記述できる。

人間が世界を認識する際に得た、並列度の高い対象表現をその並列性を損なわずに記述することができる。

## 5 実行例

逐次処理によるシミュレータが Smalltalk-80 上に実装されている。

DOC の逐次実行は以下のように、コネクションに世代を設けて行なった。

コネクションは世代ごとに、系に加えられる。初期状態のコネクションが第一世代になる。

1. 世代の複数のコネクションを順に一つ一つ系に加えていく。この時、以下のように行なう。
   - 一つのコネクションを新たに加えたことによって 発火可能になったルールを順に発火する。
   - 排他的にしか発火できないルールはたまたま 先にキューにのったものを発火させる。

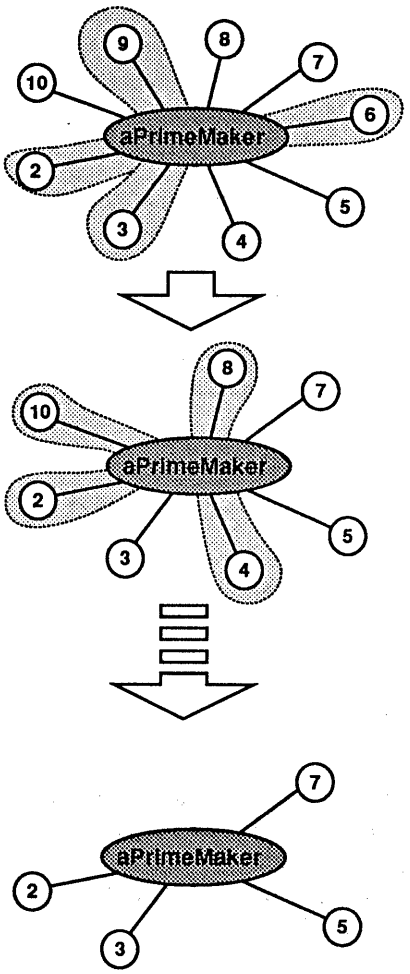


図 8: prime の実行

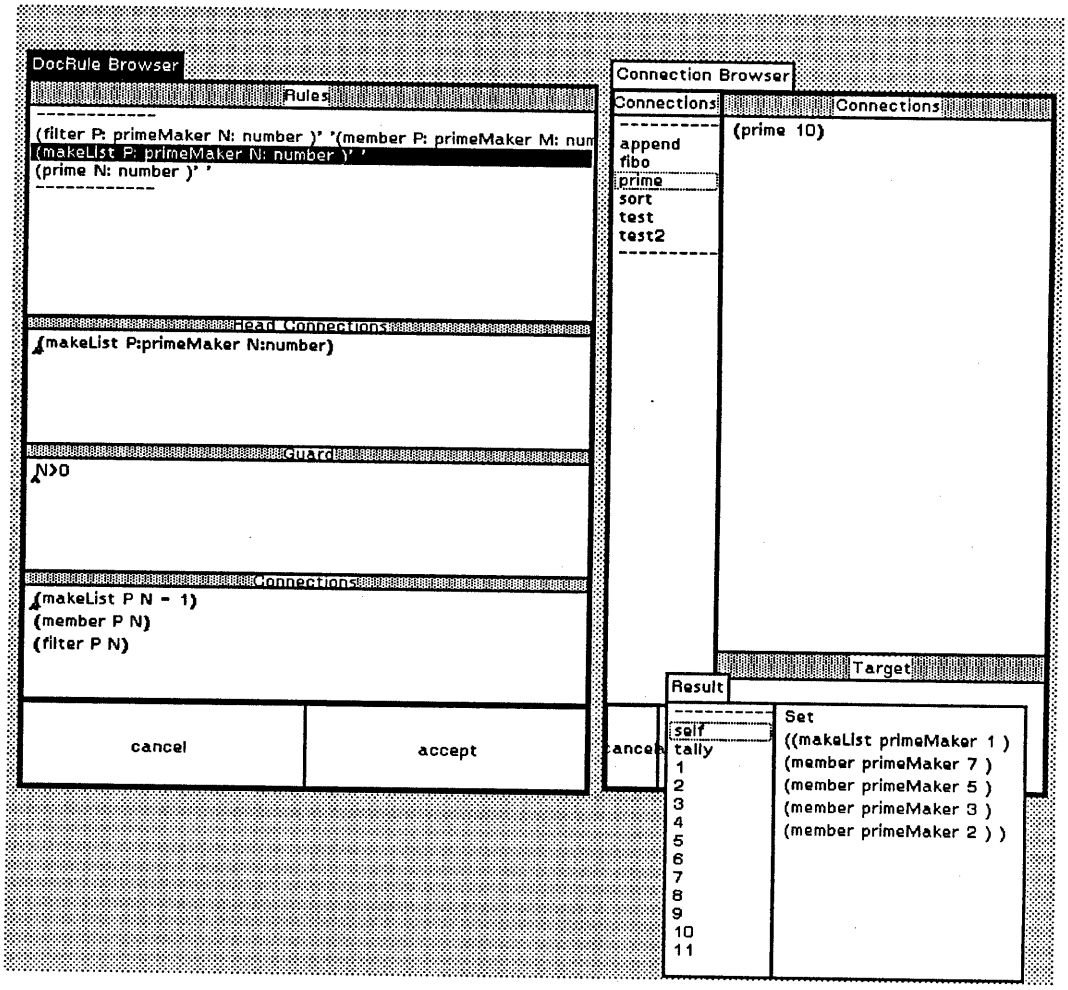


図 9: シミュレータ

- ルールの発火によって新たに生じたコネクション は、次の世代用のセットに蓄える。
- これを、この世代のコネクションがなくなるまで繰り返す。

2. 次の世代のコネクションがなくなるまで、1 を同様に行なう。

この実行法は、本来の実行法からは程遠いが、モデルのセマンティクスを損なってはいない。この実行法においては、各世代において発火したルールの数が、その世代での並列度を示す、と考えて良い。

図 9は、ソートの実行をしたところである。左のブラウザが、ルールを記述するブラウザ、右のブラウザが、初期状態を設定するブラウザである。一番右のウインドウに結果が表示されている。

現在、より本来の実行法に近い疑似並列版を検討中である。

## 6 おわりに

本稿では、状態をオブジェクト間にのみ持たせるモデル、DOC についてのべ、その記述法に関して論じた。

DOC は、現在検討中のモデルであり、言語としての機能も充分ではない。今後、このモデルに基づいて、より言語としての機能を整備していく予定である。

## 参考文献

[1] G.Agha:
Actors: *A Model of Concurrent Computaion*

*in Distriuted Systems*,
The MIT Press, 1987.

[2] M. Nilsson and H. Tanaka:
Massively Parallel Implementation of Flat GHC on the Connection Machine,
*Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pp.1031-1040, 1988.

[3] 赤間清:
"Object + Relation ≒ Program"
日本ソフトウェア科学会第7回大会論文集, pp.41-44, 1990.

[4] 中村宏明 他:
"並列論理型言語 fleng に基づいたオブジェクト指向言語 Fleng++"
WOOC'89, 1989.

[5] 吉田実 他:
並列オブジェクト指向言語におけるオブジェクト内並列性,
JSPP'91, 1991.

[6] 中村克彦:
"単位消去法：単位融合にもとづく論理プログラミングの計算方式",
PROCEEDINGS OF THE LOGIC PROGRAMMING CONFERENCE '90, pp.55-63, 1990.

[7] 戸沢義夫:
"OPS5 が提供するプログラミング・パラダイムと実行効率について"
第27回プログラミングシンポジウム 予稿集,
pp.1-12, January 1986.