自律的エージェントからなる組織の計算モデルと分散協調問題解決への応用†

丸一威雄†† 市川正紀†† 所 真理雄††

魚の群れ，スポーツのチームプレイ，会社組織など，主体的に活動する複数の個体が協調しながら問題解決を行うシステムを表現する組織計算モデルを提案する．組織計算モデルは，個体の自律性を向上させるメッセージ・インタプリタ，および不特定多数との通信を実現する環境の概念を導入した並行オブジェクト計算モデルである．この計算モデルでは，各個体（自律的エージェントと呼ぶ）は自分の判断で主体的に仕事を処理し，個体間で生じた問題の解決は各個体の意志決定を中心に行うという特徴がある．本論文では，この組織計算モデルの分散協調問題解決への応用例と分散環境での実現方法についても述べる．

1. はじめに

動物の群れや，会社組織では，複数の主体的に活動する個体が協調しながら，群れや組織の内外で発生する問題を解決していると考えることができる．群れや組織は，動的に変化しながらバランスを保ち，複雑な状況に対応してゆくという特徴を持っているが，この性質は各個体の自律性から生じている[2),19)]．群れ，集団，組織などの社会モデルに基づいた問題解決手法は，分散 AI (DAI)[3),7)] の一分野として研究されているが，並行に動く物を中心にモデル化を行う並行オブジェクト計算の考え方が応用できる問題領域のひとつである．

本論文では，自律的に活動する個体が集まってできる群れ，集団，組織などを総称して組織(Organization)と呼び，組織に属する個体をエージェント（Agent：自律性を備えた並行オブジェクト）と呼ぶ．そして，組織における分散協調問題解決を表現する枠組みを並行オブジェクト計算[5),23)]の考え方に基づいて定義し，これを組織計算モデル（An Organizational Model of Computation）と呼ぶ．

次章より，群れ，集団，組織における問題解決を表現する上での要件および問題点を指摘し，並行オブジェクト計算モデルにメッセージ・インタプリタと環境を導入した組織計算モデルを定義する．そして，組織計算モデルの分散協調問題解決への応用，分散環境への実現方法，関連研究との比較を行う．

† An Organizational Model of Computation and Its Application by TAKEO MARUICHI, MASAKI ICHIKAWA and MARIO TOKORO (Department of Electrical Engineering, Faculty of Science and Technology, Keio University).

†† 慶應義塾大学理工学部電気工学科

2. 群れ，集団，組織における問題解決

2.1 組織を構成する個体への要件

組織を構成する個体（エージェント）は，それぞれ自分で考え自分で行動しながら各自の仕事（Task）を処理することによって組織全体を維持している．こうした自律的なエージェントを並行オブジェクトで実現する場合，次のような点が問題となる．

i）非同期通信を基本に考えると，**複数のエージェントから送られてくるメッセージの到着順序は不定**であるので，到着したメッセージは各エージェントに適した順序で処理されなければならない．

例えば，客が店から商品を買う場合，店へ直接注文書を送ると同時に，第三者の銀行へ店への送金を依頼するとする．このとき店では，注文書と代金のどちらが先に到着するかは予測できない．実世界では，どちらが先に到着しても正しく処理できるので，到着するメッセージの処理順序をコントロールできる機構が必要である．

ii）エージェントに複数のメッセージが到着した場合，それらの関係から新たな行動を引き起こしたい場合がある．

例えば，i）の例で，客から商品の注文と代金が送られてきて，さらにそれをキャンセルする通知が来ている場合，いずれのメッセージも実行されていなければ，店側の判断で3つのメッセージを取り除くことができる．このように到着したメッセージの関係から新たな行動を引き起こせる機構が必要である．

i），ii）はいずれもメッセージが送られた後のエージェントの対応が重要であり，エージェントが自らメッセージの処理方法をコントロールすることが必要で

あることを示している．本論文では，このように自分の行動（メッセージの処理方法）を自分で決定できる性質のことを自律性と呼ぶ．

2.2 組織的活動を表現するための要件

複数のエージェントの組織的活動を表現するには，次のような機能が必要であると考えられる．

i）会議，ミーティングでは，複数のエージェントと交信しながら仕事を進める場合が多く，各エージェントには不特定多数のエージェントと通信する手段が必要である．

ii）組織の中では，複数のエージェントがグループを作って仕事をしていると考えられる場合がある．したがって，エージェントのグループを表現できる方法が必要である．

i），ii）は，いずれもエージェントの存在する場所を問題としている．i)は，ある場所にいるすべてのエージェントは同じメッセージが受け取れるようにすることで，またii）は，同じ場所にいるエージェントはグループを構成していると考えることで表現することができる．こうした場所の概念は，場，世界，などと呼ばれるが，本論文ではこれを環境と呼ぶ．

2.3 並行オブジェクト計算モデルで組織を実現するときの問題点

（1） 並行オブジェクトの自律性の向上の必要性

並行オブジェクト計算では，並行オブジェクトへメッセージが渡されると，メッセージに1対1に対応したメソッドが呼び出される．受け取ったメッセージは，素直に実行されるため，レシーバはセンダに対して常に従順な行動をとる．だが，実際には2.1節で述べたように，メッセージを実行すべきかどうかの判断を加えたり，受け取った複数のメッセージの関係から新たな行動を起こすような記述が必要な場合がある．従来の並行オブジェクトにはこうした機能は用意されておらず，自律的エージェントを実現するためには受け取った複数のメッセージの実行順序やメソッド呼出しを制御する機構が必要となる．

（2） 並行オブジェクトの存在する環境の必要性

並行オブジェクト計算では，メッセージを送る相手を指定しなければメッセージは送れない．実世界では，相手を指定せずに不特定多数がグループを構成したり，それらに同時にメッセージを送っていると考えられる場合がある．これらを実現するためには，並行オブジェクトの存在する場所（環境）の概念の導入が必要である．

3. 組織計算モデル[13)]

ここでは，2.3節で述べた問題点を検討し，並行オブジェクト計算モデルに新たな機能を加えて組織計算モデルを定める．

3.1 メッセージインタプリタによる自律性の実現

並行オブジェクト計算では，オブジェクトが受け取ったメッセージに対し対応するメソッドを起動するというプログラミング・スタイルになるため，メッセージの処理を自分で考える自律的なオブジェクトを記述するのに向いていない．特に，メッセージの処理順序や複数メッセージの処理を含むプログラムは記述しにくくなる．こうしたメッセージの実行に対する自律性を高めるためには，受け取ったメッセージをどのように処理するか判断を加えられる機構が必要である．従来の研究にメソッド実行の前後で手続きが起動するFlavorsのBefore-Afterデーモン[15)]，エンカプスレータ[17)]，ConcurrentSmalltalkのセクレタリ[21)]，メタ・オブジェクト[10),20)]等があるが，これらは起動するメソッドを変更できなかったり，複数メッセージの処理やメッセージの実行順序を制御する機構が十分でない．ここでは受け取った複数のメッセージを参照しながらオブジェクトの行動の順序を管理するための機構が必要なので，これをメッセージ・インタプリタと呼びエージェントごとに用意する．メッセージ・インタプリタは自己（エージェント）を参照し，自分にとって適したメッセージを1つ以上選び出し，適当なメソッドを実行することが主な仕事である．

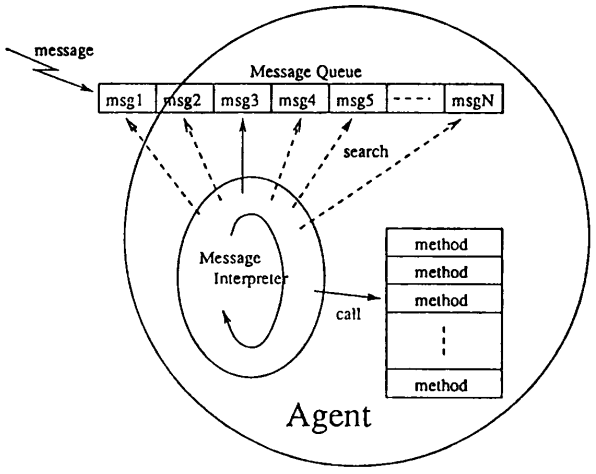


図1 エージェント・マシンの内部構造
Fig. 1 The internal structure of an agent machine.

```
(defagent cyclic-buffer-agent
        (super-agents      agent)
        (instance-variables
            (state         'empty)          ; バッファは空.
            (top           0)               ; ポインタ.
            (bottom        0)               ; ポインタ.
            (buffer-size   10)              ; バッファのサイズは 10.
            (buffer        (make-array 10))))
(defmethod put ((self cyclic-buffer-agent) value)
  (setf (aref (@ buffer) (@top)) value)     ; 値を代入.
  (incf (@ top))                            ; ポインタを進める.
  (if (>= (@ top) (@ buffer-size))          ; 循環バッファの最後に達したらポインタを
      (setf (@ top) 0))                     ; つけかえる.
  (if (buffer-full? self)                   ; バッファが一杯か?
      (setf (@ state) 'full)
      (setf (@ state) 'enough)))
(defmethod get ((self cyclic-buffer-agent))
  (prog (retval)
        (setq retval (aref (@ buffer) (@ bottom)))  ; 値を取り出す.
        (incf (@ bottom))                   ; ポインタを進める.
        (if (>= (@ bottom) (@ buffer-size)) ; 循環バッファの最後に達したら
            (setf (@ bottom) 0))            ; ポインタをつけかえる.
        (if (buffer-empty? self)            ; バッファが空か?
            (setf (@ state) 'empty)
            (setf (@ state) 'enough))
        (return retval)))                   ; 取り出した値を返す.
```

図 2 循環バッファ・エージェント・クラスの定義
Fig. 2 The definition of a cyclic-buffer-agent.

3.1.1 エージェント

エージェントが生成されると，すぐにメッセージ・インタプリタが起動される．そして，メッセージが到着すると**図 1** に示すようにエージェントごとに設けられたメッセージ・キューにそのメッセージが入る．一般に並行オブジェクトでは，順番にメッセージが実行されていくが，エージェントでは順番に関係なくメッセージ・インタプリタがキュー内のメッセージを検索，選び出した順に処理される．このようにエージェントはキューを検索しながら主体的に処理を進めてゆく点が並行オブジェクトと異なる点である．エージェントはメッセージ・インタプリタを中心に動いているので，エージェントのアクティビティは 1 である．

例えば，2 つ以上の格納場所を持つ循環バッファ・エージェントを考える．循環バッファ・クラスは，**図 2** のように定義され，:put と :get メッセージを受け付けながら仕事をする．(このプログラムで<- はエージェントの内部メソッド呼出し，@はインスタンス変数参照を意味している．) 複数のメッセージを 2 つ以上のエージェントから非同期で受け付けるとバッファが Full 状態や Empty 状態のときに，それぞれ :put, :get メッセージを実行するとエラーが生じてしまう．エージェントがエラーを起こさずに自律的に活動するにはエージェント自身に自分の仕事を管理させる必要がある．このエージェントの仕事は**図 3** に示すような状態遷移図で表すことができる．この図で円はエージェントの状態を表し (Enough は Full でも Empty でもない状態)，矢印は受けたメッセージによる状態の移り変わりを表している．このエージェントがエラーを起こさないようにするのは簡単で，Full のときに

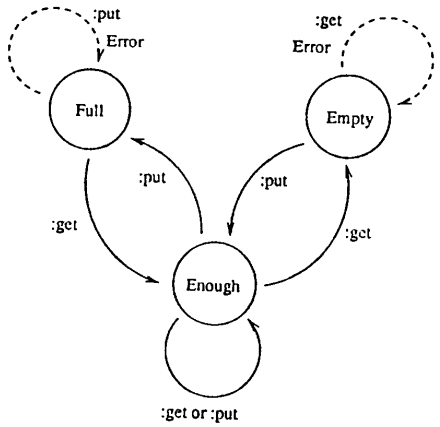


図 3 循環バッファ・エージェントの Task の状態遷移図
Fig. 3 The state transition diagram of a cyclic-buffer-agent.

:put メッセージを Empty のときに :get メッセージを実行しなければよい．こうした記述はメソッドの中で排他的記述をするよりも，エージェント自身に仕事の状態遷移に基づいてメッセージの実行を選択させる方が簡潔に表現でき，実際には図4に示すようなメッセージ・インタプリタで実現できる．このように，処理すべきメッセージの選択，メソッドの呼出し，およびメソッド実行後の処理は，分散協調問題解決を行う自律的なエージェントを実現するには極めて重要で，これらはエージェントの行動の中心となるメッセージ・インタプリタで記述される．

```
(def-interpreter buffer-interpreter ((self cyclic-buffer-agent))
  (prog ((flag t))

        (while flag
               (RECEIVE-IF
                      (AND (eval (not (eq (@ state) 'full)))
                           (put))
                      (THEN
                           (<- self :put (@ (msg-ref 0) value))))

               (RECEIVE-IF
                      (AND (eval (not (eq (@ state) 'empty)))
                           (get))
                      (THEN
                           (SEND (@ (msg-ref 0) target)
                                 (msg-def reply.
                                          (value (<- self :get)))))))))
```

図4 循環バッファ・エージェントのメッセージ・インタプリタの定義
Fig. 4 The definition of the message interpreter of a cyclic-buffer-agent.

3.1.2 メッセージ・インタプリタの機能

メッセージ・インタプリタでの記述は，メソッドでの記述と明確に区別するためメッセージ・キューの管理およびエージェントの状態管理（または Task 管理）を限定する．エージェントの生成はメッセージ・インタプリタとエージェントに与える名前を指定して次のようにして行うので，

```
(new ($$ cyclic-buffer-agent)
     #' buffer-interpreter 'name)
```

同一クラスのエージェントであってもエージェントごとに異なる性質のメッセージ・インタプリタで動かすことが可能である．組織計算モデルでは，エージェント・クラスの構造，メソッドなどは多重継承により継承することが可能であるが，メッセージ・インタプリタはエージェントごとに固有で継承はない．(ただし，同じ名前のメッセージ・インタプリタをコピーして使うことはできる．)これは，人間が機能的（メソッドとインスタンス変数）には同じであるのに，人によって個性（メッセージ・インタプリタ）が異なるのと似ている．

メッセージ・インタプリタでは，エージェントの状態に応じてメッセージ・キューを効率よく検索できることが重要で記述性を上げるために次のような基本命令を用意した．

(receive)

メッセージ・キューの先頭からメッセージを1つ取り出す．メッセージが到着していなければ，nil を返す．

(receive-block)

メッセージ・キューの先頭からメッセージを1つ取り出す．メッセージが到着していなければ，メッセージが来るまで待つ．

(receive-peek)

メッセージ・キューの先頭にあるメッセージを調べる．

(message-put-back message)

メッセージをキューの最後に戻す．

(message-queue-length)

メッセージ・キューに存在するメッセージの数を返す．

(receive-if condition action)

メッセージ・キューを検索し，condition で示した条件を満たすメッセージが存在すれば，そのメッセージをメッセージ・キューから取り出し，action を実行する．条件を満たすメッセージがなければ何もしない．condition には，複数のメッセージを組み合せた検索条件も指定できる．

(receive-if-all condition action)

receive-if 命令が1つのメッセージを対象にするのに対して，この命令は条件を満たすすべてのメッセージをキューから取り出しリストにまとめ action を実行する．条件を満たすメッセージが存在しなければ何もしない．

一般に並行オブジェクトではメソッドとメッセージが1対1に対応しているために2つ以上のメッセージを一括処理するような記述はできない．しかし，エージェントでは，複数のメッセージをまとめた処理も次のように receive-if 命令を使ってメッセージ・インタプリタ内で簡潔に記述することができる．(このプロ

グラムの THEN 部で使われている msg-ref 関数は，条件を満たしたメッセージを参照している．)

```
(RECEIVE-IF；メッセージ・キューの検索
  (AND (eval (@ product-list))；商品がある
       (注文-message  ；注文メッセージ
            (category      '注文)
            (task-id       ?taskid))
       (送金-message  ；入金メッセージ
            (category      '支払)
            (task-id       ?taskid)
            (from          'a-bank)))
  (THEN
      ；商品を発送するメソッドの呼出し
       (<- self : send-a-product-to-customer
            (msg-ref 0) (msg-ref 1))))
```

3.2 環境の導入

同時に不特定多数のエージェントと通信するため，環境 (environment) という概念を導入する．環境は，エージェントの存在する場を定めており，その環境に対して送られたメッセージは，環境に属するエージェントに必ず送られる．モデル的には，環境を介した間接メッセージ・パッシングとなるが，環境を用意することで次のような利点が得られる．

i) 互いに影響を与え合うエージェントの集合やグループを環境によって定められるので，鳥や魚の群れ[11),18)]のように不特定多数との交信の必要なグループを表現するのが容易になる．

ii) さらに，環境に動的に入ったり，出たりすることにより，グループに属す個体数が変化する問題の表現が可能になる．

環境はエージェントとは異なり受動的なオブジェクトである．環境は，環境に属すエージェントの名前を管理している．環境に対する基本命令を次に示すが，これらの命令はすべてアトミック・オペレーションとして実行されるので，他のエージェントが同じ環境をアクセスしてもそれらは順番にしか実行されない．

(new-environment environment-name)

environment-name という名前の環境を生成する．

(destroy environment-name)

environment-name という名前の環境を消去し t を返す．ただし，エージェントがこの環境に存在する場合は削除できず nil を返す．

(enter agent environment-name)

エージェント agent が環境 environment-name に入る．環境に入れた場合は t を，環境が存在しない場合は nil を返す．

(exit agent environment-name)

エージェント agent が環境 environment-name から出る．

(number-of-agents environment-name
&key : except-me)

環境に存在するエージェントの個数を返す．この関数は，キーワード：except-me を指定すると，環境に存在する自分以外のエージェントの個数を返す．

(name-list-of-agents environment-name
&key : except-me)

環境に存在するエージェントの名前のリストを返す．：except-me キーワードが指定されると自分以外のエージェントの名前のリストが返される．

また，環境へメッセージを送る Send 命令は次のように拡張される．これらもアトミック・オペレーションである．

(send-all environment-name message
&key : except-me)

環境 environment-name に存在するすべてのエージェントに対して，メッセージ message をマルチ・キャストする．この命令が実行されたときに存在していたエージェントの名前をリストとして返すので，幾つかのエージェントがこのメッセージを受け取るのがわかる．：except-me キーワードを指定すると，環境に存在する自分以外のエージェントにマルチ・キャストする．

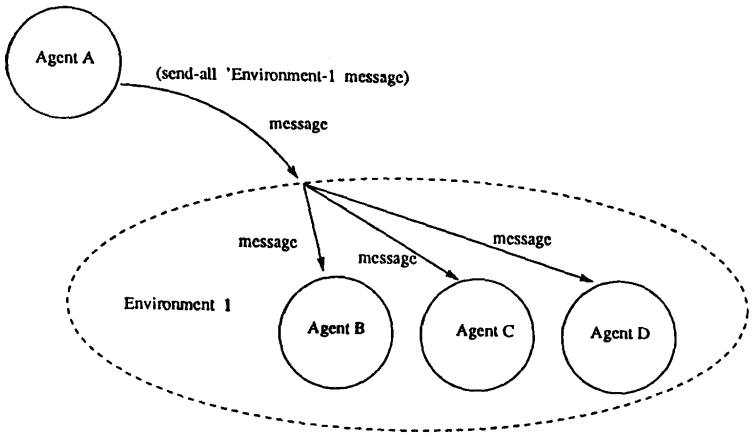
(send-to environment-name target-agent-name
message)

環境にエージェント target-agent-name が存在する場合には，メッセージを送り t を返す．また，環境にエージェントが存在しない場合には，メッセージを送らずに nil を返す．

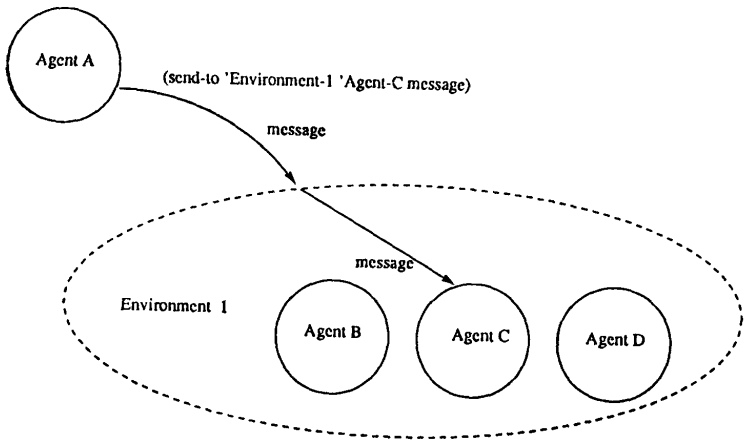
図 5 に send-all と send-to の概念図を示す．小さな円がエージェントを表し，点線がエージェントの存在する環境を表す．

3.3 メッセージの扱い

メッセージは，並行オブジェクト計算モデルと同じであると考えてよいが，次のメッセージ定義に示すようにメッセージには送り手と送り先のエージェント名が記録されており，環境へのマルチ・キャストの場合にはメッセージの送り先すべてのエージェント名がリストとして記録される．したがって，メッセージを受け



a) send-all 命令
Send-all primitive.



b) send-to 命令
Send-to primitive.

図 5 環境を介した send 命令
Fig. 5 Send operations using an environment.

取ったエージェントは，同じメッセージをだれが受け取るかがメッセージからわかる．

```
(defmessage message
   (from     sender)
   (to       receiver 1 receiver 2…)
             :
             :)
```

4. 組織計算モデルによる分散協調問題解決

組織計算モデルでは，非同期に動く複数の自律的なエージェントが互いに協調しながら問題を解決する．ここでは，こうした問題解決の例として，覆面算と分散スケジュール・ボード問題を取り上げる．

4.1 覆面算の分散協調型解法[8]

覆面算は，次のような文字で作られた数式をもとに，各文字に割り当てられた数値を当てる問題である．

```
  DONALD
+ GERALD
--------
  ROBERT
```

この例では，Dが5であるという初期値が与えられる．組織計算モデルでこの問題を表す場合，各カラム3文字分を1つのエージェントと考え，6つのエージェントを用意する．例えば，左端のカラムを表すagent-Aは，(D+G=R) という式を受け持つ．6つのエージェントは同一の環境に属しており，互いに自分の発見した情報を他のエージェントに環境を介してマルチ・キャストすることにより情報を伝える．ここで，エージェントが送出するメッセージは次の2種類である．

i) 各変数の決定値，候補値，制約条件（偶数，奇数など）のメッセージ

ii) 隣のカラムに対するキャリー，ボローに関するメッセージ

各エージェントはこれらのメッセージを受け取りながら，各カラム内で決定できる変数の値を絞り込んでいく．このとき各エージェントはPast 型の非同期通信しか使わず互いに完全に非同期で動作する．Past 型の非同期通信を使ったこのような計算方法をメッセージとメソッドが1対1に対応している並行オブジェクトで作ると，メッセージ・キューを検索できないために上記 i), ii) のメッセージを受け取る度にメソッドを動かし，そこで発見した情報を他の並行オブジェクトに送らなければならない．これに対してエージェントでは，1つのメッセージを処理した後にまだ未処理のメッセージがキューに残っているかどうか調べ，残っている場合にはメッセージを送出せずに，キュー内のメッセージを続けて処理する記述が可能となる．そして，キューが空になったらそれまでに発見した情報をメッセージとして環境にマルチ・キャストする．こうした動作は，エージェント

の状態とメッセージ・キューを管理するメッセージ・インタプリタによって実現され，メッセージ・インタプリタのメイン・ループの中の基本部分は次のような構造になる．

```
(setq flag nil); 新しい発見があるとtになる
                ; フラグ
(while (not(=(message-
              queue-length)0))
  ; メッセージがキューにあれば処理を続ける．
  (setq msg (receive))
    ; メッセージを取り
  (if (<- self : process msg)
    ; 処理
      (setq flag t)))
(if flag  ; 新しい発見があった場合だけ
          ; メッセージを送出する．
    (SEND-ALL 'Environ-
              ment newmsg))
```

この処理方法では，各エージェントの処理の進み具合いにより送出されるメッセージの数が変化するが，20回行った実験では，次のような結果が得られた．ただし，この表でエージェントはメッセージ・キューを検索しながら実行し，並行オブジェクトはメッセージ・キューが参照できないことを意味している．実際には，エージェントの実行速度がメッセージの転送速度に比べて速くなるほど，エージェントと並行オブジェクトの差が小さくなると考えられる．

メッセージの個数	最小	平均	最大
エージェント	17	17.7	19
並行オブジェクト	19	23.5	29

6つカラム・エージェントが必要なメッセージを利用している関係を**図6**に示す．(使っていないメッセージはこの図には表されていない．) 図の縦方向は時間を表し，実線部分はエージェントが問題解決に貢献している期間を表している．この図から実際の最大並列度が6であることがわかる．非同期で動作しているために，必ずしもこの順序でメッセージが伝わるわけではないが，各エージェントはどのようにメッセージが到着しても，メッセージ・インタプリタで不必要なメッセージは捨て，必要とする情報を運んできたメッセージだけを利用し必ず問題解決を完了する．このとき各エージェントは自分の分担であるカラムの変数が決定し終ったら処理が終了したことになり，その後環境から抜け出る．環境内にエージェントが存在しなくなったら覆面算が解けたことになる．

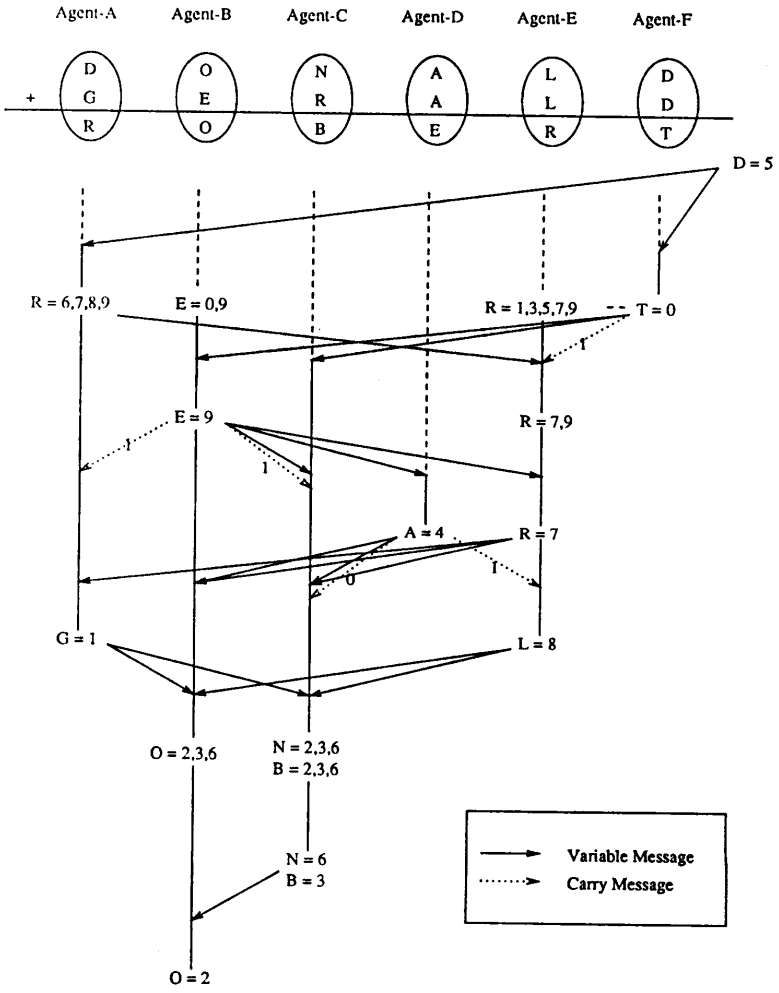


図6 覆面算を解く6つの Column-agent の協調関係
Fig. 6 Cooperation between column-agents in Crypt-Arithmetic puzzle solving.

4.2 分散スケジュール・ボード問題[12]

ネットワーク上で，各ユーザごとにスケジュール・ボードが用意されているとする．ここでは，スケジュール・ボードがユーザの指示を受けて他のスケジュール・ボードと協調することによりミーティング・

ルームを使用したミーティングの予定を立てる問題を考える．各スケジュール・ボードおよびミーティング・ルームをエージェントとして実現し，それらに相談させながらスケジュールを調節させる．こうしたオフィス・オートメーション[6),9)]は組織計算モデルで扱うのに適した問題領域の1つである．ここでは，各エージェントが他のエージェントを考慮しながら自律性を発揮しなければならない点で覆面算より難しくなっている．

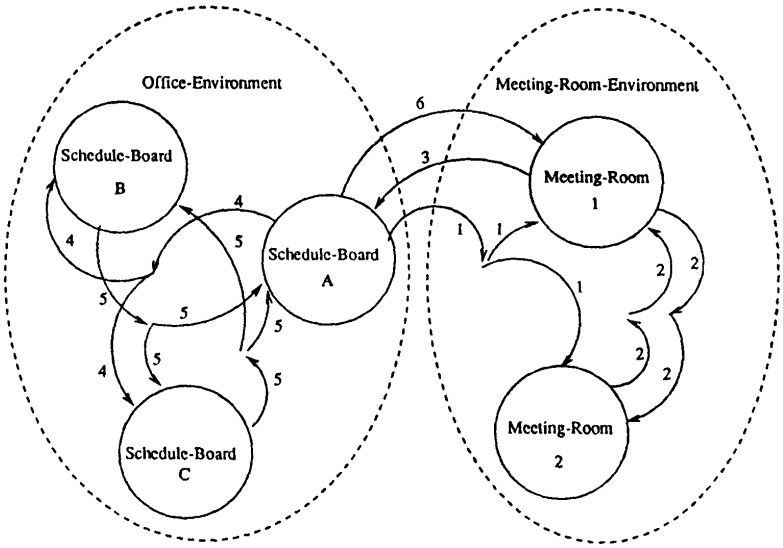


図 7 分散スケジュール・ボード問題
Fig. 7 Distributed Schedule-Boards.

さて，図7のように3つのスケジュール・ボードA，B，Cが環境（Office-Environment）に，2つのミーティング・ルームM1，M2が環境（Meeting-Room-Environment）に属している場合を考える．また簡単化のため計算中に各環境のエージェントの数は変わらないものとする．A，B，Cがどちらかのミーティング・ルームを使ってミーティングを行う場面を考える．

（1） スケジュール・ボード間での協調

例えば，「Bさん，Cさんと今週5時間のミーティングを行いたい．」とスケジュール・ボードAが指示されたとする（Taskの割り当て）．図7に示すように，スケジュール・ボードAは，まず自分のスケジュールを調べて5時間の時間を選び，その時間が空いているかどうかMeeting-Room-Environmentへメッセージをマルチ・キャストして問い合わせる(1)（以後は，iiを参照）．そしてミーティング・ルームが使えれば，B，Cへ知らせるためにOffice-Environmentへミーティングの提案を示すメッセージをマルチ・キャストする(4)．ここで，B，CがOKのメッセージを返してくれば(5)，ミーティングが成立し，Aはミーティング・ルームを予約する(6)．

このとき，BかCのいずれかが都合が悪い場合は，Aは予約する代わりにミーティング・ルームをキャンセルして(6)，A以外のBかCがリーダシップをとって図7の(1)から(6)のように繰り返すことにより新しくミーティング時間の提案を行う．このように決定権（リーダ）を順番に移動することはグループでの意志決定を公平に行うためには重要で，このコントロールはメッセージ・インタプリタ内で記述されている．この場合，リーダになった回数の少ないものでかつ各エージェントのユニーク番号の小さいものが新しいリーダになれる．これは(5)で返答を返すときに各エージェントが情報（リーダになった回数とユニークID）を環境を返して全員が交換し，1つでもNot-OKのメッセージを出しているエージェントがいるとわかったら，各エージェント内で自分が新しいリーダになれるかどうかを判定することにより実現されている．これは，同じ評価関数を持った平等なエージェントが同時に意志決定を行って競合せずに成功している例である．

ここで述べた協調動作は，各エージェントがメッセージの交換手順に従っているためにできることであるが，非同期通信では，メッセージ・インタプリタによりタスクの状態を管理しながら処理を進めるという記述が複雑な協調を成功させる点で重要である．

（2） ミーティング・ルーム間での協調

いずれかのスケジュール・ボードから「空いている時間」の確認をされたミーティング・ルームは図7に示すように，まず自分の時間が使用可能かを調べ，その情報を他のミーティング・ルームに知らせるためにメッセージをMeeting-Room-Environmentへマルチ・キャストする(2)．そして各ミーティング・ルームは他のすべてのミーティング・ルームからのメッセージを集め各自で自分が適任であるかどうかを集めたメッセージと比較することで判断する．このアルゴ

リズムはⅰ）でリーダを選出する場合と同じで，各ミーティング・ルームで独立に判断される．自分が適任であると判断したミーティング・ルームは（図7では，Meeting-Room-1 が自分が適任であると判断している．）自ら進んでメッセージをスケジュール・ボードへ返す(3)．ここでもすべてのミーティング・ルームは対等な関係にあり全体を管理する部分はない．各エージェントは環境に存在するエージェントの数を調べながら，自分が最適であるかを判断するように記述されているので，ミーティング・ルームの数が環境内で変化したとしても最適候補を選出することが可能となる．

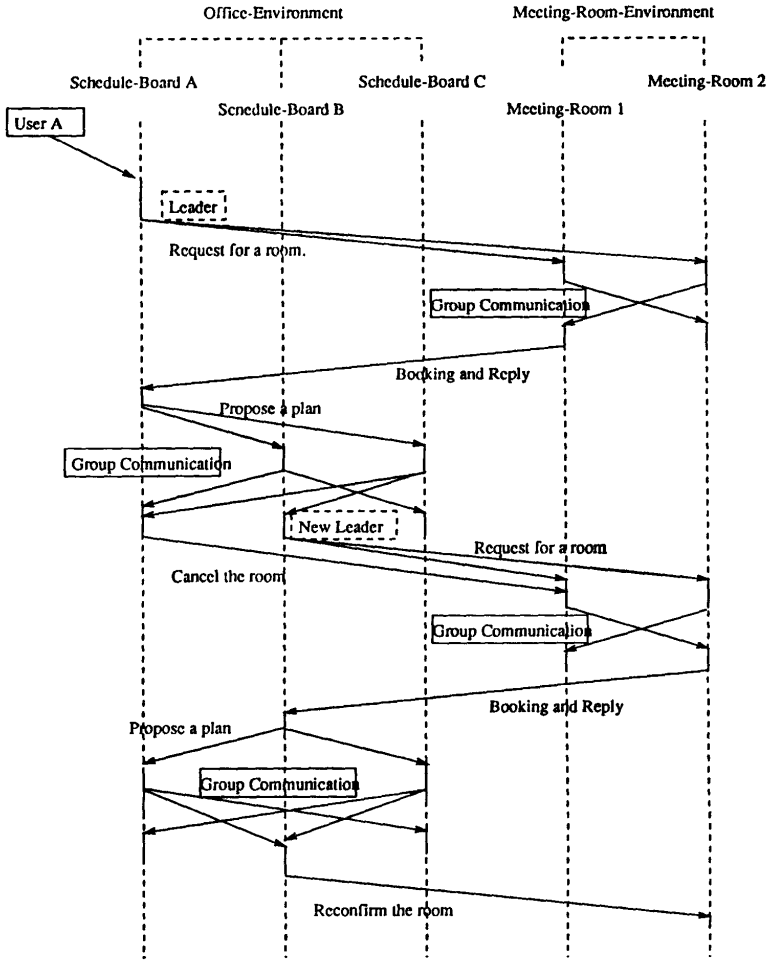


図8 分散スケジュール・ボード問題におけるエージェントの協調関係
Fig. 8 Cooperation between Schedule-Board agents in distributed Schedule-Boards.

図8は2人目のリーダの案でミーティング時間が予約されたときの各エージェント間のメッセージの伝搬を示している．この図で，Group Communication とした部分が，グループの中から任意のエージェントが1つ選び出される部分を示している．また，**図9**はここで述べたプログラムの実行例である．白抜きの時間帯は個人的なスケジュール，黒で塗りつぶした時間帯はミーティング時間を示しており，各エージェントが話合いにより同じ時間帯をミーティング時間としたことがわかる．

5. 組織計算モデルの分散環境での実現

組織計算モデルをネットワーク上の複数のワークステーションで動かすことは，会社組織など実際の問題解決に応用する上で意味がある．組織計算モデルは，次のような3種類のオブジェクトで実現されている．

ⅰ）エージェント：メッセージ・インタプリタが常に稼働している能動型オブジェクトで，プロセッサに割り当てられる．

ⅱ）メッセージ：受動型オブジェクトで，Send 系の命令によりエージェントから他のエージェントに送られる．

ⅲ）環境：エージェントから参照される受動型オブジェクトで，環境に存在するエージェントの名前を管理する．

組織計算モデルは，オブジェクト指向言語／システム PANDORA-II (KCL[24] で実装）により実現されている．分散環境での実装方法の詳細は省略するが，ソケットで N 対 N で結合された任意のホストの PANDORA-II にエージェントまたは環境を作ることができる．ただし，エージェント・クラスがそのホストに存在しなければならない．エージェントと環境はそれぞれ名前で識別されるが，これらはネットワーク上でもユニークである．環境はネットワーク上で1つ存在すればよく，その環境への Enter, Exit 命令はその環境の存在するホストまで送られて実行される．

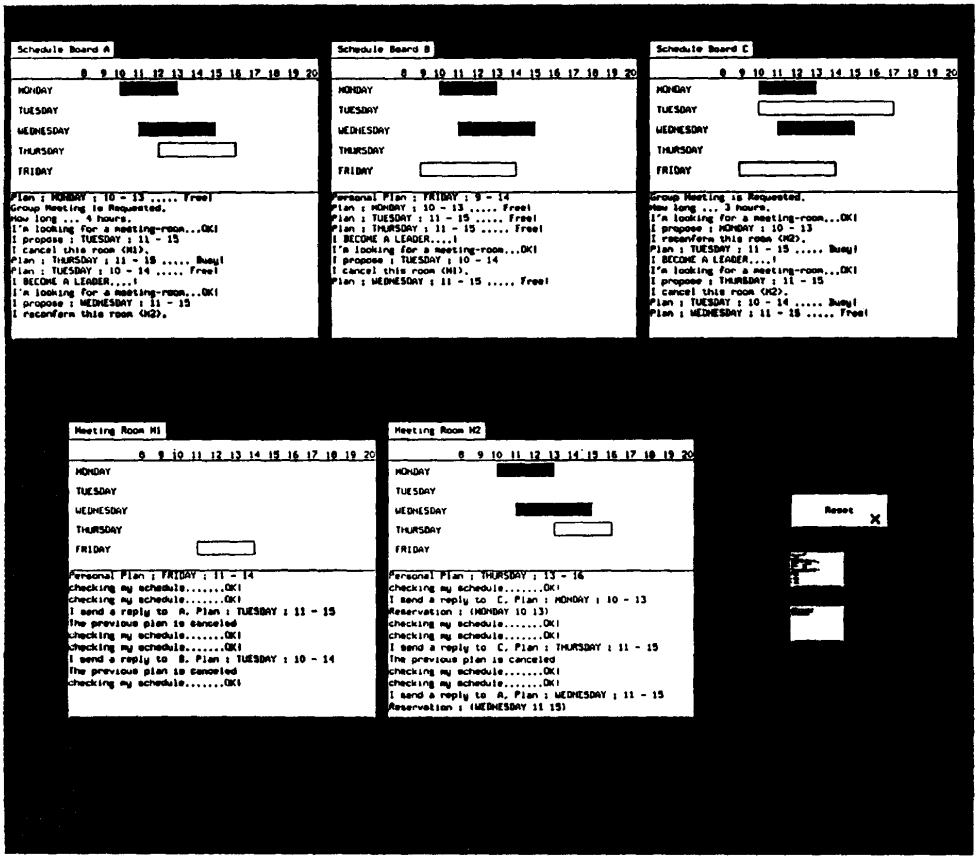


図 9 分散スケジュール・ボード実行例
Fig. 9 A screen snapshot of distributed Schedule-Boards problem.

6. 考　　察

6.1 エージェントと並行オブジェクトの比較

組織計算モデルでは，他の並行オブジェクト計算モデル (ABCM/1[22] など) と同様，同一のエージェントから送られてきた複数のメッセージは送られた順に到着する．Actor モデル[1),5)]では，同一のアクタから送ったメッセージでさえも到着順序が保証されないために必要に応じてシリアライザ (serializer) により順序を保証させる．このシリアライザやメタ・オブジェクト，セクレタリではメソッド実行の前後で付随するオブジェクトが動作するが，複数のメッセージの関係から任意のメソッドを呼び出すような機能は特に用意されていない．しかもそのオブジェクトは再帰的に存在するので実装は難しい．これに対してメッセージ・インタプリタは，複数のメッセージの関係を処理でき，かつ並行オブジェクトと組み合わされて1つのエージェントを作るため再帰的とはならず実装も簡単である．

6.2 環境と他の通信機構の比較

エージェント間での協調を成功させるには，他のエージェントと情報交換をする必要がある．本論文で述べた環境はその交信手段の1つであるが，ここではメッセージ・パッシングと共有記憶域を使う通信方法と比較することにより，組織計算モデルの特徴を明確にする．

並行オブジェクト計算モデルにおけるメッセージ・パッシングは，1対1の通信で，共有記憶域を必要としないという特徴がある．通信は一方向でかつ非同期で行われるので，並列性は高い．ただし，メッセージが到着したあとで受け手がそのメッセージを処理するという受動的な動作になるので能動的な記述には向いていない．

黒板モデル[15)]やタップル・スペース (Tuple Space)[4),14)] に代表される共有記憶域を使う通信方法は，対等な関係にある複数の知識源またはプロセス間の交信を表現できる．しかも各プロセスは能動的に共有記憶域にアクセスできるため自ら考え行動する記述

がしやすい．しかし，共有記憶域上では他のプロセスと同期をとる必要があるため並列性をあげるのは難しい．

これらをまとめると，メッセージ・パッシングは並列性が向上するが，能動的な行動は記述しにくい．一方，共有記憶域を使ったモデルは，能動的な記述に向いているが，並列性を向上させるのは難しいことになる．組織計算モデルのエージェントは，環境という共有記憶域を使うが，環境に対してはメッセージ・パッシングしか行えず，エージェント内ではメッセージ・キューの検索しかできない．これは，共有記憶域に特有な同期を排除しかつ能動的にメッセージ・キューをアクセスできるので並列性を向上させたまま，自律性が実現しやすくなるという長所がある．したがって，組織計算モデルは共有記憶域の長所とメッセージ・パッシングの長所を組み合わせた計算モデルであるといえる．

7. まとめ

組織的な問題解決を表現するには，自律的に活動するエージェントと，不特定多数との通信が必要なことから，並行オブジェクト計算モデルをベースにエージェントごとにメッセージ・インタプリタを持たせ，環境の概念を導入した組織計算モデルを提案した．そして，対等な関係にある複数の自律的エージェントが協調しながら問題を解決する例を示した．

必要なメッセージの選択，適したメソッドの起動などを記述するメッセージ・インタプリタは，分散協調問題解決におけるエージェントの自律性を実現する上で重要な機構である．また，環境を複数用意することで幾つかのグループの集合体として表される組織を表現することができるようになる．これらの特徴を持つ組織計算モデルは実際の組織的活動を比較的正確に表現できるので組織活動のモデル化手法として有効であると考えている．

謝辞 本研究を行うにあたり貴重な御助言と御討論をして頂いた慶応義塾大学理工学部の安西祐一郎教授，ぺんてる(株)の石川孝氏，ICOT の吉田かおる女史，M. I. T. Media Lab. の Steve Strassman 氏，慶応義塾大学理工学部の中島達夫氏に感謝いたします．

参考文献

1) Agha, G.: *ACTORS, A Model of Concurrent Computation in Distributed Systems*, The MIT Press (1986).
2) 青井和夫，小林幸一郎，梅沢 正：組織社会学，サイエンス社 (1988).
3) Decker, K. S.: Distributed Problem-Solving Techniques: A Survey, *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 5, pp. 729-740 (1987).
4) Gelernter, D.: Generative Communication in Linda, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 1, pp. 80-112 (1985).
5) Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages, *Artif. Intell.*, Vol. 8, No. 3, pp. 323-364 (1977).
6) Hewitt, C. and de Jong, P.: *Open Systems, Perspectives on Conceptual Modeling*, Springer-Verlag (1983).
7) Huhns, M. N.: *Distributed Artificial Intelligence*, Pitman Publishing (1987).
8) 市川正紀，丸一威雄，所真理雄：組織計算モデルによる覆面算の分散協調型解法，電子情報通信学会春期全国大会，pp. 327-328 (1989).
9) Jong, P. D.: The Ubik Configuration: A Fusion of Messages, Daemons, and Rules, *MIT AI Memo*, No. 1058 (1988).
10) Maes, P.: Concepts and Experiments in Computational Reflection, *Proceedings of OOPSLA '87*, pp 147-155 (1987).
11) Maruichi, T., Uchiki, T. and Tokoro, M.: Behavior Simulation Based on Knowledge Objects, *Proceedings of ECOOP '87*, pp. 257-266 (1987).
12) Maruichi, T., Ichikawa, M. and Tokoro, M.: Modeling Autonomous Agents and Their Groups, *Proceedings of MAAMAW '89*, pp. (1989).
13) Maruichi, T.: Organizational Computation, Ph. D. Thesis, Keio University (1989).
14) Matsuoka, S. and Kawai, S.: Using Tuple Space Communication in Distributed Object-Oriented Languages, *Proceedings of OOPSLA '88*, pp. 276-284 (1988).
15) Moon, D., Stallman, D. and Weinreb, D.: *LISP Machine Manual*, Fifth ed., MIT A. I. Lab. (1983).
16) Nii, H. P.: Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architecture, *The AI Magazine*, Vol. 7, No. 2, pp. 38-53 (1986).
17) Pascoe, G.: Encapsulators: A New Software Paradigm in Smalltalk-80, *Proceedings of OOPSLA '86*, pp. 341-346 (1986).
18) Reynolds, C.: Flocks, Herds, and Schools: A Distributed Behavioral Model, *Proceedings of SIGGRAPH '87*, pp. 25-34 (1987).
19) Ridley, M.: *Animal Behavior: A Concise Introduction*, Blackwell Scientific Publications

(1986).
20) Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language, *Proceedings of OOPSLA '88*, pp. 306-315 (1988).
21) 横手靖彦，所真理雄：並行オブジェクト指向言語 ConcurrentSmalltalk，コンピュータ・ソフトウェア，Vol. 2, No. 4, pp. 24-42 (1985).
22) 米沢明憲，柴山悦哉，Briot, J. P.，本田康晃，高田敏弘：オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1，コンピュータ・ソフトウェア，Vol. 3, No. 3, pp. 9-23 (1986).
23) Yonezawa, A. and Tokoro, M. (eds.): *Object-Oriented Concurrent Programming*, The MIT Press (1987).
24) 湯浅太一，萩谷昌巳：*Kyoto Common Lisp Report*, Research Institute for Mathematical Sciences, Kyoto University (1985).

(平成2年4月19日受付)
(平成2年9月11日採録)

丸一 威雄

1961年生．1984年慶應義塾大学理工学部電気工学科卒業．1990年同大学院博士課程卒業．工学博士．並行オブジェクト指向計算モデルの研究を経て，分散人工知能の研究に携わる．マルチエージェント・シミュレータ PANDORA-II，地図データベース RINZO の開発に従事．現在，インテリジェント・システムズ・ジャパン(株)取締役．

市川 正紀

1965年生．1988年慶應義塾大学理工学部電気工学科卒業．1990年同大学院修士課程卒業．分散人工知能の研究に携わる．同年，(株)野村総合研究所に入社．計算機，ネットワーク関連の調査，研究に従事．

所 真理雄（正会員）

昭和22年生．昭和45年慶應義塾大学工学部電気工学科卒業．昭和47年同大学院修士課程（管理工学専攻），50年博士課程（電気工学専攻）修了．工学博士．同大学電気工学科助手，専任講師を経て現在助教授．その間カナダウォータールー大学，米国カーネギー・メロン大学訪問助教授．プログラミング言語，試算機アーキテクチャ，分散システム，人工知能などに興味を持っている．共著書に「システム構成技術」(岩波書店) などがある．日本ソフトウェア科学会，電子情報通信学会，ACM, IEEE, AAAI 各会員．