プログラミング-言語・基礎・実践- 8－8
(1992. 8. 20)

エージェント間の共有知識を実現したPrologに基づく協調処理言語

川村尚生* 斎藤善徳** 金田悠紀夫*

*神戸大学大学院 自然科学研究科 知能科学専攻
**神戸大学大学院 工学研究科 システム科学専攻

本稿では，エージェントグループを規定するためのフィールドに知識を置いてメンバー間で共有する協調処理モデルを提案する．共有知識は，グループで共通なルール・データの設定やメンバー間の通信に使用される．また，本モデルでは，フィールドは一種の知識ベースとなるので，エージェントとフィールドの対応を実行時に変えることにより，エージェントの動作を動的に変更するといった目的にも利用できる．このモデルに基づき，Prologをベースとした協調処理言語を設計・試作した．例としてトランプゲームの記述を行ない，本モデルおよび言語の適用性を確認した．

A Prolog-based Cooperative Language with Shared Knowledge among Agents

Takao Kawamura * Yoshinori Saito ** Yukio Kaneda *

* Division of Intelligence Science, Graduate School of Science and Technology, Kobe University
** Department of Systems Engineering, Faculty of Engineering, Kobe University
1-1 Rokkodai, Nada-ku, Kobe 657, JAPAN

We propose a computation model for cooperative agents. In this model, we introduce a *field* as an abstraction of the group of agents. Each agent can communicate to the other agents in the same group through a field. The field also enables us to specify the action of the group, i.e., it is used for a knowledge base to keep rules and data shared by agents in a group. We implemented a cooperative language by extending Prolog based on the model. We also present an example written in the language to demonstrate the effectiveness of our model.

1 はじめに

近年，協調処理モデルが，大規模分散システム，人工知能，グループウェアなど，さまざまな分野で注目されている[1]．ここでいう協調処理とは，複数の動作主体（以下，エージェント）が，互いに協力や調停を行なうことにより，集団として目的を達成するような処理形態のことである．

協調処理モデルにおいては，複数のエージェントから構成されるグループが重要な動作単位となる．エージェントグループを表現するために，フィールドや環境と呼ばれる概念が提案されている．すなわち，同じフィールドに属するエージェントはグループを形成していると考える．これまで，フィールドを通信媒体とすることにより，次のような機能が実現されてきた[2,3]．

(1) グループ内の全エージェントへの放送

(2) グループ内の未知のエージェントとの通信

しかし，グループは目的や状況を同じくするエージェントの集まりであるということを考えれば，グループを対象とした通信手段を用意するだけでは十分ではない．グループとしてまとまって仕事をするには，グループ内でのデータやルールの共有など，メンバー間のより緊密な関係が必要であろう．

本稿では，フィールドに通信媒体としての役割だけではなく，知識ベースとしての役割も持たせることにより，エージェントグループ内での共有知識を実現した協調処理モデルを提案する．

また，知識ベースとしてのフィールドは，エージェントグループという概念を離れても，知識のモジュール化や，エージェントのふるまいを柔軟に変化させるといった目的のためにも使用できる．

以下，第2章で本モデルについて説明し，第3章でPrologを使って本モデルを実装した処理系について述べる．第4章では本処理系を用いた簡単な例を示す．第5章はまとめと今後の課題である．

2 協調処理モデル

本モデルの基本要素は，エージェントとフィールドの二つである．エージェントは他のエージェントと互いに協調して問題の解決にあたる自律的な動作主体である．一方，フィールドは一種の知識ベースであり，エージェントグループを表現する役割を持つ．

2.1 エージェント

エージェントの型定義は次のように静的に行なう．

```
def_agent agent_type
  プログラム
enddef.
```

エージェントの生成は次の述語によって動的に行う．

```
create_agent(agent_type,Agent).
```

agent_typeで与えられた型のエージェントが生成され，第2引数Agentに大域的に管理されるエージェントIDが返される．

生成されたエージェントは型定義で与えられたプログラムに従って動作する．Kamui88[4]のようなオブジェクト指向言語のオブジェクトが，メッセージの読みだしや，そのメッセージに応じたメソッドの実行といった処理を暗黙のうちに行うのと異なり，エージェントのふるまいは完全にプログラマブルである．

2.2 フィールド

フィールドは一種の知識ベースであり，エージェントがフィールドに属すると，そのフィールド内の知識にアクセスできるようになる．

エージェントからは，自らの知識と，属しているフィールドが提供する知識をまったく区別することなく利用できる．ただし，知識の検索は，まずエージェント固有の知識，次いで属しているフィールド中の知識の順に行なわれる．また，エージェントは複数のフィールドに属することができるが，この場合はフィールド内の知識は属した順番に検索される．

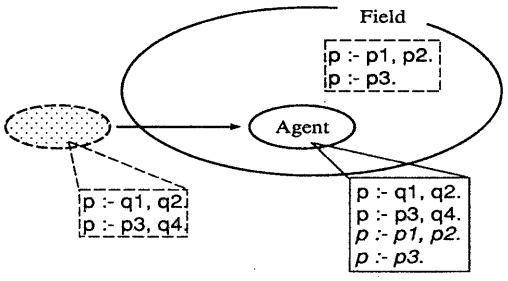

図 1: エージェントとフィールド

エージェントはすべての知識を自ら持つのではなく，必要な知識を提供してくれるフィールドに適宜属することによって問題の解決にあたる．こうすることにより，状況や目的が変化した場合には，属するフィールドを変更することで柔軟に対応できる．

例えば，状況に応じてエージェントの動作を変化させるには，次のようにする．まず，状況ごとにフィールドをいくつか用意し，それぞれのフィールドにおいて，同じ名前（例えばｐ）で手続きを記述しておく．エージェントが手続きｐを呼び出したとき，実際に実行されるのはそのときエージェントが属しているフィールドの手続きなので，状況に応じてフィールドを選ぶことによって適切な手続きが実行されることになる．この様子を図 2に示す．

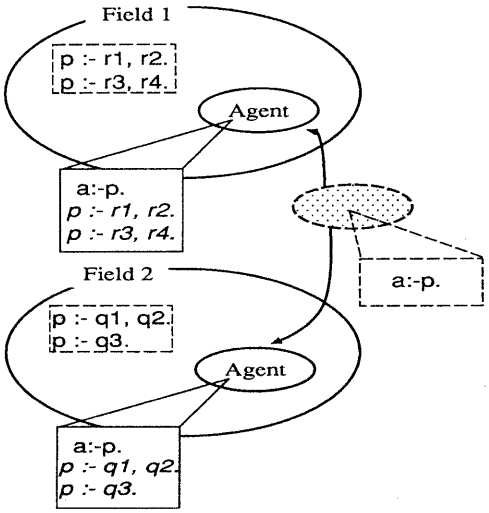

図 2: 状況に応じて動作を変えるエージェント

フィールドの型定義は次のように行なう．

```
def_field field_type
  初期知識
enddef.
```

エージェントはフィールドに関して次の４つの操作を行なうことができる．

(1) フィールドの生成
`create_field(field_type,Field).`
field_type で与えられた型のフィールドが生成され，第２引数 Field に大域的に管理されるフィールド ID が返される．

(2) フィールドに入る
`enter_field(Agent,Field).`

(3) フィールドから出る
`exit_field(Agent,Field).`

(4) フィールド内の知識を変更する
`share(Field,Assert,Retract).`
Field には対象フィールド ID を，Assert には追加したい節を要素とするリストを，

Retract には削除したい節を要素とするリストをそれぞれ指定する．

また，すべてのエージェントは生成時に自動的に common フィールドに属するものとする．

2.3 共有知識

同じフィールドに属するエージェント群は，図3のように，エージェントグループを形成していると考えることができる．

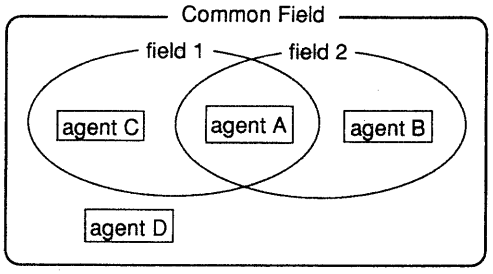

図 3: エージェントグループ

フィールド内の知識は，フィールドに属するすべてのエージェントによって共有されるので，グループで共通なルールやデータを設定するのに用いたり，メンバー間の通信手段として利用できる．

2.4 メッセージ

本モデルにおけるエージェント間の主な情報交換手段は共有知識である．しかし，グループが異なれば共有知識は利用できないし，同じグループに属するエージェント間でも，1対1の通信の方が都合がよい場合も考えられる．このような場合は，メッセージを用いて情報を伝達する．

(1) 送信

```
send(To,Message).
```

To には宛先エージェント／フィールドを，Message にはメッセージの内容をそれぞれ指定する．宛先にフィールドを指定した場合，メッセージはフィールドに属するすべてのエージェントに送られる．送信は非同期的に行われ，send/2 は送り先のエージェントの受信行為を待たずに終了する．

(2) 受信

```
receive(From,Message).
wait(From,Message).
```

receive/2 は，ユニファイ可能なメッセージが届いていなければ失敗し，wait/2 は，ユニファイ可能なメッセージが到着するまで待つ．

到着したメッセージはいったんエージェント内のメッセージプールと呼ばれる領域に格納され，受信述語の実行を待つ．メッセージプール内のメッセージはユニフィケーションによって検索されるため，必ずしも到着順に処理されない．ただし，ユニファイ可能なメッセージの中では，最も早く到着したメッセージが選ばれるものとする．

3 Prolog に基づく言語処理系

前章で述べた協調処理モデルに基づくプログラミング言語を Prolog をベースにして設計・試作した．

ベース言語として Prolog を選んだ最大の理由は，Prolog にはデータとプログラムの区別がなく，フラットな節形式によって知識を表現するようになっていることである．このような性質がないと，フィールドによる共有知識を実現することは非常に困難であると考えられる．

また，他にも，Prolog には協調処理に適した次のような性質がある．

(1) プログラムが宣言性や制御の非決定性を持つので，情報や制御の流れが動的に変化するシステムの記述に適している．

(2) 変数に型がないので，さまざまな形式を持つメッセージを統一的に受信できる．

(3) メッセージの取捨選択に必要なユニフィケーション機能を備えている．

3.1 エージェント

エージェントはメッセージ送受信機能つきのPrologインタプリタであり，図4に示すように，Prologモジュール，メッセージの送受信を司る通信モジュール，およびメッセージプールから構成される．

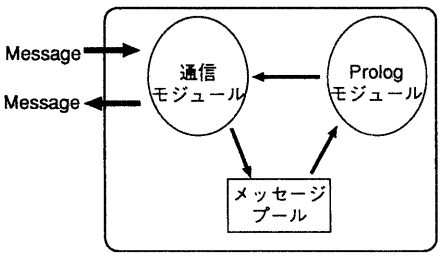

図4: エージェント

Prologモジュールと通信モジュールは並行に動作しており，他のエージェントから届いたメッセージは，Prologモジュールの動作とは無関係に，通信モジュールによってメッセージプールにたくわえられる．

Prologモジュールは，型定義で与えられたプログラムに従って動作するが，これには，エージェントが存在を開始したとき，まず最初に何をするかという情報が含まれている必要がある．ここでは，最初にmainという名前の手続きを実行することにした．また，Prologにはプログラムとデータの区別がないので，エージェントのプログラムは定義時に与えるだけでなく，動的に変更することもできる．例えば，図5のように，メッセージとして送られたプログラムを登録することが考えられる．

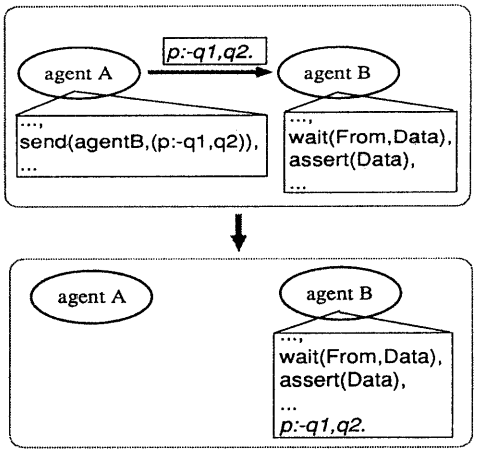

図5: エージェントの知識の変化

なお，プログラム実行の開始時は，start_upという型名のエージェントが自動生成されることとした．

3.2 フィールド

図6に示すように，フィールドは管理モジュールと通信モジュールから構成される．管理モジュールはエージェントからの要求に応じてさまざまな処理を行なう．

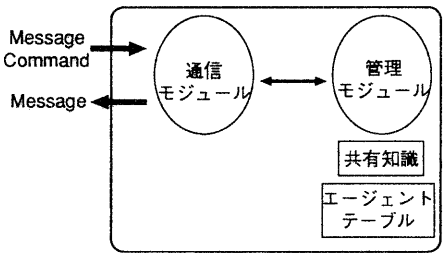

図6: フィールド

3.3 処理系の実装

処理系の実装を米 Sequent Computer Systems 社製の並列計算機 Symmetry 上で行った．OS は Mach2.5，記述言語は C を用いており，基本的な構造は次のようになっている．

(1) エージェント（フィールド）ひとつにつきタスクをひとつ割り当てる．

(2) エージェント（フィールド）内のモジュールには各々異なるスレッドを割り当てる．ただし，通信モジュールは送信部，受信部に分け，各々異なるスレッドを割り当てる．

(3) エージェントとフィールドの ID を管理するために ID 管理タスクを設ける．
create_agent/2（create_field/2）は ID 管理タスクに対して生成要求を出す．ID 管理タスクは，その要求を受けてエージェント（フィールド）を生成し，生成したエージェント（フィールド）の ID を返す．

共有知識は，フィールドに属しているエージェントすべてにコピーを持たせることで実現している．

4 例題

本言語を用いた簡単な例として，トランプゲームのババ抜きを考える．この問題では，エージェントはプレイヤーエージェントを，フィールドはババ抜きフィールドと，ディーラーフィールドを用意する．

プレイ方法に関する知識は各プレイヤーが個別に持つ．

ディーラーフィールドはカードの配り方を提供するフィールドで，同時にはひとつのエージェントしか属さない．これは，知識のモジュール化の手段としてのフィールドの例になっている．

一方，ババ抜きフィールドはプレイを行なう場を規定しており，プレイヤーの配置情報を共有知識として提供する．

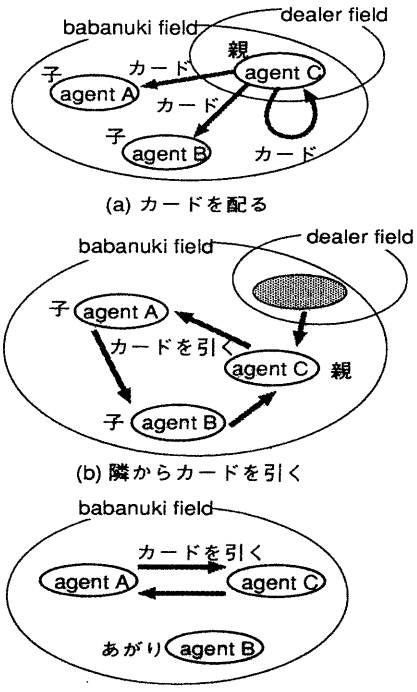

図 7: ババ抜きのモデル

5 まとめ

エージェントグループ内で知識を共有することを目的として，フィールドに通信媒体としての役割だけでなく，知識ベースとしての役割も持たせた協調処理モデルを提案した．知識ベースとしてのフィールドは，グループという概念から離れても，知識のモジュール化や状況に応じてエージェントのふるまいを動的に変化させるといった目的にも利用できる．このモデルに基づき，Prolog をベースとした協調処理言語を設計し，並列計算機上に試作した．また，例題を通じて本モデルの適用性を確かめた．

本モデルおよび言語はまだまだ未完成であり，試作処理系での実験を通じて改良していく予定である．また，協調処理は分散環境において本

領を発揮すると思われるので，分散環境での処理系の実装を検討している．

参考文献

1) 中島秀之, 久野巧, 木下佳樹, 半田剣一, 松原仁, 石川裕, 車谷浩一, 大澤一郎：協調アーキテクチャ関連研究の現状, 調査報告 221, 電子技術総合研究所 (1991).

2) 丸一威雄, 市川正紀, 所真理雄：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, 情報処理学会論文誌, Vol. 31, No. 12, pp. 1768–1779 (1990).

3) 吉田紀彦, 楢崎修二：場と一体化したプロセスの概念に基づく並列協調処理モデル Cellula, 情報処理学会論文誌, Vol. 31, No. 7, pp. 1071–1079 (1990).

4) 渡辺慎哉, 原田康徳, 三谷和史, 宮本衛市：場とイベントによる並列計算モデル — Kamui 88, コンピュータ・ソフトウェア, Vol. 6, No. 1, pp. 41–55 (1989).

付録 ババ抜きのプログラム

```
% 一部の手続きの定義は省いている
def_field common
  :- op(500,xfx,$).
enddef.
def_field babanuki
enddef.
def_field dealer
% プレイヤーの配置を設定し，カードを配る
  dealer(N,Dealer) :-
    bunkatu(N,Cards),
    players([H|T]),set_players_pos(H,[H|T]),
    send_card(Cards,[H|T]).
  cards([s$1,h$1,k$1,d$1,...,joker$0]).
enddef.

def_agent start_up
  main(Self) :-
% フィールドとエージェントの生成
    create_field(babanuki,Fl),
    create_field(dealer,D),
    create_agent(player,Pl1),
    create_agent(player,Pl2),
    create_agent(player,Pl3),
    enter_field(Pl1,Fl),
    enter_field(Pl2,Fl),
    enter_field(Pl3,Fl),
% 共有知識の初期設定
    share(D,[],[players([Pl1,Pl2,Pl3])]),
    share(Fl,[],[b_id(Fl),d_id(D),
      start_up(Self),player(3)]),
    play.
enddef.

def_agent player
  main(Self) :-
% 最初はジャンケンで親を決める
    player(N),janken(N,Result),
    play(Result,Self).
% 親: ディーラーになってカードを配る
  play(kachi,Self) :-
    d_id(Dealer),enter_filed(Dealer),
    player(N),dealer(N,Dealer),
    exit_field(Dealer),
    wait(_,card(List)),seiri(List,List1,N),
    play(sender,List1,N,Self).
% 子
  play(make,Self) :-
    wait(_,card(List)),seiri(List,List1,N),
    play(receiver,List1,N,Self).
% カードがなくなったら勝ち
  play(_,[],0,Self) :-
    win(Self).
% 自分一人だけになったら負け
  play(_,_,_,Self) :-
    next(Self,Self),lose(Self).
% リクエストを待ってカードを送る
  play(sender,List,N,Self) :-
    wait(Player,request),
    random(Num),select(List,Num,Card,List1),
    send(Player,answer(Card)),
    N1 is N - 1,
    play(receiver,List1,N1,Self).
% 隣にリクエストしてカードを受けとる
  play(receiver,List,N,Self) :-
    next(P1,Self),send(P1,request),
    wait(P1,answer(Card)),
    seiri2(Card,List,N,List1,N1),
    play(sender,List1,N1,Self).
% 負け: start_up に報告する.
  lose(Self) :-
    start_up(Startup),send(Startup,lose),
    play(kachi,Self).
% あがり: プレイヤーの配置情報を書き換える
  win(Self) :-
    f_id(Fl),
    share(Fl,[next(P,Self),next(Self,N)],
      [next(P,N)]).
    play(make,Self).
enddef.
```