ICOT Technical Report: TM-0028

TM-0028

Mandala（曼陀羅）
A Knowledge Programming Language on Concurrent Prolog

古川康一、竹内彰一、國藤　進

October, 1983

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Mandala （曼陀羅）
A Knowledge Programming Language on Concurrent Prolog

古川康一、竹内彰一、溝口 理 （ICOT）

1. 概要

対象指向プログラミングは、表現力が豊かであることから、最近になって急速にその重要性が認識されてきている。とくに知識表現の観点から見ると、刻々と共に変化する個々の対象を自然に記述できることから、従来の静的な事実の集合のみを記述する枠組に比べて、その応用範囲は大きく広がってきたと言える。

ところで、これまでの対象指向プログラミングは、その評判とは裏腹に、実際にはあまりその技術が展開されていない。その理由の1つは処理系が広く出回っていない点である。現在よく知られている対象指向プログラミング言語／システムには、 Smalltalk, Flavor, LOOPS [1] などがあるが、日本国内でそれが利用できる環境はごく限られている。

対象指向プログラミングが普及しない理由は、もう1つあるように思われる。それは、言語自身あまり使い勝手が良くないのではないか、という点である。あるいは、言語が十分に洗練されていないので、概念整理をする際に言語を使いこなすのに苦労をするのではないかと思われる。

我々は、Concurrent Prolog の上に対象指向に基づく知識情報処理用プログラミング言語／システム Mandala （曼陀羅）を作り上げる作業[2,3] を開始したところである。Concurrent Prolog 自身、並行事象を記述する言語であり、並読であること、Prologとの関連が明確であることなどのいくつかのすぐれた性質を備えている。そのようなすぐれた性質は、その上で対象指向プログラミング言語／システム[2] を作り上げる上で、大きな利点となっている。

本報告では、2章でConcurrent Prolog の概略を紹介し、つぎに3章で Concurrent Prologでどのようにしてオブジェクトを実現するのかを示す。4章では、LOOPS に現れる基礎概念が Mandalaでどのように表現されるか、その対応関係について述べる。そして最後に基本的な部分のプログラムと、その実行例を示す。

2. Concurrent Prolog

Concurrent Prologは、並行事象の記述のための論理型プログラミング言語で、Prologとは、図1のような関係にある。

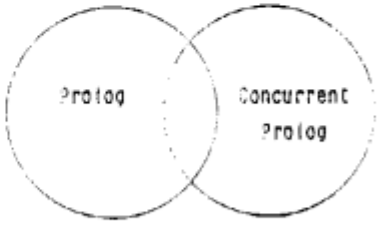

図1. PrologとConcurrent Prolog の包含関係

すなわち、Concurrent Prolog は、Prologの一部の機能をその中に含むと同時に、並行事象を記述する拡張部分をもつ。PrologにあってConcurrent Prolog にない機能は、深い後戻りの機能（deep badktracking)である。実際、並行事象の記述では、2つ以上のプロセスが互いに通信し合いから動くプログラムを作るが、1つのプロセスが他のプロセスに通信した後で失敗した場合、通信自身も取り消さなければならなくなる。このような状況は、それを実現するのも困難であるし、そのようなプログラムに意味を持たせるのもむずかしい。そのため、Concurrent Prolog では、一旦他のプロセスへの通信が行われれば、その通信自身にまで及ぶ後戻りは禁止している。

しかしながら、Concurrent Prolog 自身、そのlogic partはHorn logicであり、control partが通常のPrologのそれと異なる。すなわち、つぎの式が成り立つ。

Horn Logic ＋ 深さ優先探索 ＝ Prolog

Horn Logic ＋ ? ＝ Concurrent Prolog

この式の？部分は、つぎのような実行規則である。

(1) Process 間の通信は、共有論理変数によって行われる。

(2) 通信メッセージは、論理変数のとる値である。

(3) メッセージは、送り手と受け手が区別される。とくに受け手は、X？のように変数に？が付加される。X？をもつプロセスは、その変数の値を定めてはいけない。もし、具体的な値との同一化が行なわれたとき、その変数の値が定まっていなければ、その同一化は、一時中断される。

(4) メッセージの送出は、送り手が通信用論理変数の値を定めることによって行なわれる。その値を決定するときにいくつかの選択があるとき、送り手はその決定を"|"によって指示する。"|"はPrologのCut に似た働きをする制御命令でCommitと呼ばれる。

3. Concurrent Prolog によるオブジェクトの実現

Concurrent Prolog を用いてオブジェクトを表すには、プロセスを使うのがよい。特に、プロセスをtail recursion programとすると、同一の名前のプロセスが生きつづける。そして、オブジェクトの保持する状態は、そのプログラムの引数に持たせることができる。

［例］ 計数器

```
counter([clear | S],State):- | counter(S?,0).
counter([up  | S],State):-
  NewState is State + 1  | counter(S?,NewState).
counter([down  | S],State):-
  NewState is State - 1  | counter(S?,NewState).
counter([show(State) | S],State):-
                        | counter(S?,State).
counter([],state):- | .
```

この例で、手続きcounter の第１引数は通信用論理変数で、第２引数は、カウンタ・オブジェクトの状態である。counter プログラムは、最後のclause以外は、すべてtail recursiveなプログラムになっている。そして、メッセージ"clear" によりState は 0になり、"up"あるいは"down"メッセージにより Stateは+1あるいは -1 変化する。

オブジェクトの生成は、単にその定義プログラムを呼び出すことによってなされる。すなわち、

```
?- instream(X),use_counter(X?,C1),
       counter(C1?,0).
```

により、３つのオブジェクトinstream, use_counter, counter が作られる。ここで、instreamは端末からの入力を受けつける組込みオブジェクトで、 use_counter はcounter の値を参照して走るプログラムである。

use_counter プログラムの一例を次に示す。

```
use_counter([show(Val) | Input] ,
            [show(Val)"| Command]):-
        | use_counter(Input?,Command),
          wait(Val) & print(Val).
use_counter([X | Input], [X | Command]):-
        | use_counter(Input?, Command).
```

4. 対象指向プログラミングシステムとしての Mandala（曼陀羅）

Concurrent Prolog 上の対象指向プログラミングを行うときの基本は、前節で述べたオブジェクトである。本節では、このオブジェクトの囲りにいかにして対象指向プログラミングの諸機能を実現していけばよいか、その基本的な考えを示す。

(1) instance object

LOOPS のinstance object は、Concurrent Prolog のオブジェクトが対応する。

(2) class object

LOOPS ではclass もinstance object と同様に定義されるが、Concurrent Prolog では、class object は状態を持たない静的なプログラム（データベース）が対応する。

(3) property と method

propertyおよびmethodは、Concurrent Prolog では全く同一に扱われ、それぞれ、オブジェクトあるいはclass を表すプログラム定義中の引数上で実現される。そして、それらはそのオブジェクトに付随したlocal world を定める。すなわち、propertyやmethodは節集合となっている。

(4) is_a hierarchy によるproperty inheritance

is_a hierarchy は、local world のhierarchy を

作り出す。そのhierarchy は、1つのオブジェクトに対して親が2つ以上存在してもよい。すなわちmultiple inheritanceが可能である。その場合のサーチ・ルールは、適当なdemo programを作ることによって、容易に変更が可能である。

(5) part_of hierarchyによるproperty inheritance

part_of hierarchyは、それ自身がclass level と instance levelの両方に現れる。part_of hierarchyによるproperty inheritanceは、行なってよい場合とそうでない場合があるが、その基準が明確であればdemo programに組み込むことができる。instance levelのpart_of hierarchyはプログラムの階層的モジュール化に対応する。上の階層のプログラムは、下の階層のモジュールを部品として用いる。そしてその間の関連は共有変数を介したオブジェクト間通信によってとられる。

(6) meta object

LOOPS では、class を管理するものとしてmeta classを用いるが、Concurrent Prolog では、class objectを管理するためにmeta object を作る。meta object は、動的なオブジェクトが対応し、class variableはそこに置かれる。

以上が主要な特徴であるが、以下に、設計上考慮した問題点について述べる。

(a) 何故、class は静的なプログラムとして表現するのか

class の表現方法には、静的なプログラムの他に、動的なオブジェクトを用いる方法も考えられる。しかし、動的なオブジェクトでclass を表現すると、いくつかの不都合な点が生じる。その1つは、is_a hierarchy によるproperty inheritanceの実現が困難な点である。default reasoning の自然な実現法は、demoを用いて「失敗」時にis_a hierarchy を遡上るやり方であるが、動的なオブジェクトでは「失敗」の扱いが困難である。第2に、instance object とclass objectを共有変数を介して結合することを考えると、複数のinstanceからの通信のmerge が必要となり、それは非実現的である。

静的なプログラムとする積極的な理由としては、class が表すものは一般的な概念であり、それ自体計算の進行と共に変化する状態をもつのが不自然な点である。こうするとclass variableが実現できないが、それは、そのclass に相当するmeta object を動的なオブジェクトとして作り、そこに持たせることにする。そのmeta object は、そのclass に属するinstanceの管理を行なう。

(b) 何故、状態はlocal world か

instance object の状態は、local world と考えなくとも良いが、demo述語を考えるとき、上のclass のlocal world と考え方を合わせた方がよい。instance object の状態はそのobjectを表すリテラルの引数上に現れ、それをlocal world すなわち節集合と考えると、そのobject自身メタ的な性質を持つようになることに注意したい。ただし、instance object のlocal world に現れる節は、通常はunit clause で、ルールの形をしたものがそこに置かれることはないであろう。

(c) property inheritanceに何故demo述語を用いるか

property inheritanceは、対象指向プログラミングの要であり、その高速処理が大切な点であるが、ここでは、あえてdemo述語を用いたやり方を採用した。それは、property inheritanceの戦略自身未だ十分に煮つまっておらず、いろいろな戦略を試してみることが必要であると思われるからである。なお、この点については、ハードウェアによる高速化の検討も必要となるであろう。そのような高速化機構は、知識ベース・マシンの一構成要素を成すものと思われる。

［謝辞］ 本研究を行なう機会を与えていただいた渕一博当研究所所長に感謝致します。また WG4におけるディスカッションは本研究を始めるきっかけとなりました。溝口文雄主査を初めとする各委員に感謝致します。

References

[1] D.G.Bobrow and M.Stefik: The LOOPS Manual - A Data and Object Oriented Programming System for Interlisp (Preliminary Version), Memo KB-VLSI-81-13 (working paper), Aug. 1981.

[2] A.Takeuchi, K.Furukawa, E.Y.Shapiro:
Object Oriented Programming using Concurrent Prolog, Proc. of the Logic Programming Conf. '83, Tokyo, March 1983. (in Japanese).

[3] E.Y.Shapiro: A Subset of Concurrent Prolog and its Interpreter, ICOT Technical Report TR-003 (1983).

```
:- ['mandala.sys'].

%-----------------------------------------------------------------
%       Description about static entities
%-----------------------------------------------------------------
%
%       The world <class> keeps the common properties of all class.
%

class((number(0) -> true)).
class((create(Name,Goals) ->
        delete(number(X)),X1 is X+1,add(number(X1)),add(instance(Name,Goals)),
        Cname instance_of Mname,instantiate(Cname,Name,DW),
        create_process(instances(Name,Goals?,DW)))).
class((how_many -> number(X),write(X),nl)).
class((list(db) -> C instance_of M,
        call((world(C,Axioms),writeln1(Axioms),nl)))).
class((kill(Name) -> delete(number(X)), X1 is X-1, add(number(X1)),
                     delete(instance(Name,[])))).
class((list(self) -> process_status(S),call((writeln1(S),nl)))).


%%%---- Sample Description of a Counter World ----%%%
%       The world <counter> is one of classes.
%       The world <counter> keeps the common properties of all counters.
%

counter((state(0) -> true)).
counter((clear -> delete(state(X)),add(state(0)))).
counter((up -> delete(state(X)),X1 is X+1,add(state(X1)))).
counter((down -> delete(state(X)),X1 is X-1,add(state(X1)))).
counter((show -> state(X),write(X),nl)).
```

```
/*      MANDALA.SYS      */
%%%---- Concurrent Prolog Program ----%%%

%=============================================================================%
%       Knowledge Representation Language in Concurrent Prolog                %
%                                                                             %
%               K.Furukawa  &  A.Takeuchi  &  S.Kunifuji                      %
%                                                                             %
%                               August, 1983                                  %
%-----------------------------------------------------------------------------%
%                                                                             %
%       active actors           ---     processes                             %
%                                                                             %
%       class description       ---     databases                             %
%                                                                             %
%       metaClass       ---     meta level programs                           %
%                                                                             %
%-----------------------------------------------------------------------------%

:- op(700,xfx,is_a).
:- op(700,xfx,instance_of).
:- (value(load(mandala),true) ;
        compile('mandala.lib'),set(load(mandala),true)).

%=============================================================
%       Description about dynamic constructs.
%-------------------------------------------------------------
%  Meta Description about active instances of some things.
%
%       <instances_of_class, instances_of_metaclass, instances_of_MetaClass>
%

instances(Name,[Goal|Input],World) :-
        trace(Inh_demo,inh_demo(Name,Goal,World,NewWorld)) |
        inh_demo(Name,Goal,World,NewWorld),
        (wait(NewWorld) & instances(Name,Input?,NewWorld)).
instances(Name,[],World) :-
        trace(Termination,terminate(Name))|true.

%=============================================================
%       User Interface  <distribute>
%-------------------------------------------------------------

distribute([(Class,create(Name))|Input],AllThings) :-
        send(AllThings,Class,create(Name,Goals),NewAllThings) |
        distribute(Input?,[(Name,Goals)|NewAllThings]).
distribute([(Name,Msg)|Input],AllThings) :-
        send(AllThings,Name,Msg,NewAllThings) |
        distribute(Input?,NewAllThings).

send([(Name,[])|R],Name,die,R).
send([(Name,[Msg|Y])|R],Name,Msg,[(Name,Y)|R]).
send([H|R],Name,Msg,[H|T]) :- send(R,Name,Msg,T).


%%  Demo with Inheritance

inh_demo(Class,Goal,World,NewWorld) :- inh_demo(Class,Goal,World,NewWorld,[]).

inh_demo(Class,Goal,World,NewWorld,Default) :-
        find_method(Class,Goal,World,Default,NewDefault) |
        simulate(Goal,World,NewWorld,NewDefault).

simulate(true,W,W,_).
```

```
simulate((create_process(Process),C),World,NewWorld,Default) :-
        trace(instantiation,instantiate(Instance,Name)) |
        Process, simulate(C,World,NewWorld,Default).
simulate((P,C),World,NewWorld,Default) :-
        simulate(P,World,World1,Default) & simulate(C,World1,NewWorld,Default).
simulate(create_process(Process),World,World,Default) :-
        trace(instantiation,instantiate(Instance,Name)) |
        true, Process.
simulate(process_status(World),World,World,_).
simulate(add(C),World,[(C -> true)|World],_).
simulate(delete(C),World,NewWorld,Default) :-
        delete(C,World,NewWorld,Default).
simulate(G,W,W,_) :- system_pred(G) | call(G).
simulate(G,World,NewWorld,Default) :-
        axiom(G,B,World,Default) |
        simulate(B,World,NewWorld,Default).
```

```
/*      MANDALA.LIB      */
%%%     Mandala Library    %%%

%=============================================================================%
%       Knowledge Representation Language in Concurrent Prolog                %
%                                                                             %
%               K.Furukawa  &  A.Takeuchi  &  S.Kunifuji                      %
%                                                                             %
%                                 August, 1983                                %
%-----------------------------------------------------------------------------%


:- op(700,xfx,is_a).
:- op(700,xfx,instance_of).
:- public
        delete/4,
        system_pred/1,
        axiom/4,
        find_method/5,
        world/2,
        instantiate/3,
        go/0,
        trace_mandala/0.

%%% Sequential Prolog Program %%%

system_pred(P) :-
        system(P) ; clause(P,incore(P)).

find_method(Class,Goal,World,Default,Default) :-
        has_method(Goal,World,Default).
find_method(Class,Goal,World,Default,NewDefault) :-
        extend_world(Class,World,SuperClass,Default,NDefault),
        find_method(SuperClass,Goal,World,NDefault,NewDefault).

has_method(Goal,World,Default) :-
        axiom(Goal,_,World,Default).

extend_world(Class,World,SuperClass,Default,EnhancedWorld) :-
        (member(((Class is_a SuperClass) -> true),World) ;
         member(((Class is_a SuperClass) -> true),Default)),
        world(SuperClass,SuperWorld),
        enhance_world(Default,SuperWorld,EnhancedWorld).

delete(C,[(C -> true)|World],World,_).
delete(C,[D|World],[D|NewWorld],Default) :-
        delete(C,World,NewWorld,Default).
delete(C,[],[],Default) :- find_axiom(C,Default).

axiom(Goal,Body,World,Default) :-
        (axiom2(Goal,Body,World) ; axiom2(Goal,Body,Default)).
axiom2(Goal,Body,[Axiom|_]) :- copy(Axiom,Ax),Ax = (Goal -> Body).
axiom2(Goal,Body,[_|As]) :- axiom2(Goal,Body,As).
copy(X,Y) :- assert(temp(X)),retract(temp(Y)),!.

find_axiom(C,[(C -> true)|_]).
find_axiom(C,[_|R]) :- find_axiom(C,R).

world(SuperClass,SuperWorld) :-
        Term =.. [SuperClass,X], setof(X,Term,SuperWorld).
enhance_world(Def,Super,Enh) :- append(Super,Def,Enh).

instantiate(Name,Insname,Insworld) :-
        world(Name,Insworld0),
```

```
        append([((Insname instance_of Name) -> true)],InsworldO,Insworld).

go :-
  instantiate(class,counter,NDW),
  instantiate(class,class,CLW),
  solve((instream(X),
         distribute(X?,[(counter,C),(class,L)]),
         instances(class,L?,CLW),
         instances(counter,C?,NDW))).

trace_mandala :-
        set(traceset,[instantiate(_,_),terminate(_),inh_demo(_,_,_,_)]),
        set(trace,on).
```

実行例

```
| ?- go.
|: (counter,create(c1)).
|: (c1,up).
|: (c1,up).
|: (c1,show).
2
|: (counter,create(c2)).
|: (counter,list(db)).

clear->(delete(state(X)),add(state(0)))
down->(delete(state(Y)),ZisY-1,add(state(Z)))
show->(state(W),write(W),nl)
up->(delete(state(Y1)),Z1isY1+1,add(state(Z1)))
state(0)->true

|: (c2,show).
0
|: (c2,up).
|: (c2,show).
1
|: (counter,list(self)).

instance(c2,[show,up,show|X])->true
number(2)->true
instance(c1,[up,up,show|Y])->true
counter instance_of class->true
how_many->(number(Z),write(Z),nl)
kill(V)->(delete(number(W)),X1isW-1,add(number(X1)),delete(instance(V,[])))
list(db)->(V1 instance_of W1,call((world(V1,Y2),writelnl(Y2),nl)))
list(self)->(process_status(U2),call((writelnl(U2),nl)))
create(W2,X3)->(delete(number(Y3)),Z3isY3+1,add(number(Z3)),
                add(instance(W2,X3)),Y4 instance_of Z4,instantiate(Y4,W2,W4),
                create_process(instances(W2,? X3,W4)))

|: (class,list(db)).

how_many->(number(X),write(X),nl)
kill(Z)->(delete(number(U)),VisU-1,add(number(V)),delete(instance(Z,[])))
list(db)->(Z1 instance_of U1,call((world(Z1,W1),writelnl(W1),nl)))
list(self)->(process_status(Y2),call((writelnl(Y2),nl)))
number(0)->true
create(U2,V2)->(delete(number(W2)),X3isW2+1,add(number(X3)),
                add(instance(U2,V2)),W3 instance_of X4,instantiate(W3,U2,U4),
                create_process(instances(U2,? V2,U4)))
|:
```