数理解析研究所講究録
第525巻 1984年 315-338

Mandala : A unified system for modular programming and knowledge representation on Concurrent Prolog

古川康一 (Koichi Furukawa)
竹内彰一 (Akikazu Takeuchi)
国藤進 (Susumu Kunifuji)

(財)新世代コンピュータ技術開発機構

Abstract

Manadala is a programming system aimed for developing knowledge information processing systems in logic programming. Mandala inherits many features from Concurrent Prolog (CP), as its base language, and the system itself is written in CP. It is not only a knowledge programming system but also a basis for a hierarchical modular programming system for CP. The nature of this duality comes directly from two-facedness of logic programming, that is, both procedural and declarative interpretation. Moreover, it realizes object oriented programming using a process mechanism in CP, thus achieving objects with dynamically changing states. On the other hand, the rules represented by program codes are also changeable data. Its changing rate is slower than that of objects, but the meaning and affect of its change is more important and drastic than that of objects. These differences in updating are reflected on the different representation schema between rules and objects. Moreover, the semantics of "is-a" hierarchy and "part-of" hierarchy turned out to be much clearer by representing rules and objects distinctively. Another important concept in a knowledge programming system is its descriptive power for realizing such meta level systems as knowledge base management systems. In Mandala, a meta programming feature based on a provability concept in parallel computational environment is developed and is itself used to write its interpreter.

1．はじめに

知識表現は，人工知能の研究の大きな課題の１つである。人間の持っている知識は多種多様で，それらのすべてを適切に表現し得るような言語あるいはシステムは，これまでのところ，まだ作られていない。情報処理の研究一般に言えることだが，人工知能の研究も構成的な面があり，言語あるいはシステムを作ってそれを試すことを繰り返して研究が進められていく。その意味で，知識表現言語／システムを作る道具となる知識プログラミング・システムが最近急速に注目を浴びてきた。LOOPS [1]，GLISP [10] などがその例である。

知識は物と物との間の複雑な関係が基本的な役割を果すので，それらを表現し，操作できることは知識表現言語にとって必須である。そのための基本的な道具立てはリスト処理であるが，従来 LISP がその道具として用いられてきた。LOOPS にしてもGLISP にしても親言語が LISP であり，その上にオブジェクト指向，データ指向，ルール指向などの高度なプログラミング技法を可能とする環境が作られている。これらのプログラミング環境は，リスト処理能力と相まって，知識表現システムを作成するための強力な道具を実現するのに大きく寄与している。

しかしながら，それらの諸機能を実現するのにこれらのシステムはそれぞれ別個の仕組みを用意しており，また言語的に見てもそれぞれ異なる表現を用いなければならない。すなわち，LOOPS にしてもGLISP にしても機能的には十分であるが，それらは抽象化が不十分で必ずしもシステムとして洗練されているとは言えない。

これらのプログラミング機能に加えて，知識表現にとって欠かすことのできない機能に，知識の無矛盾性チェックなどの管理機能および不完全な知識の表現とそれらを用いた推論機能がある。そのような機能の実現を目指したシステムに，MRS [4]，KRYPTON [3] などがある。この２つはいずれも述語論理を基にした形式化を行なっているが，それはこの種の問題に対する述語論理の適性を示していると言えよう。

われわれは，LOOPS ，GLISP などのシステムのもつ知識プログラミング機能とMRS ，KRYPTON などのシステムのもつより高度な知識表現機能を合わせ持つシステムを目指して，Concurrent Prolog [11),13)] を親言語とするシステムMandala を設計し，その処理系を試作した。Mandala はConcurrent Prolog 上の階層型モジュラー・プログラミング・システムの骨格としても位置づけている。Mandala の特徴は親言語であるConcurrent Prolog のそれに負うところが多いが，記述の抽象度がより高いので，Concurrent Prologのユーザ言語と考えることができる。

一方，それ自身が知識プログラミング・システムともなっている。それは，論理プログラミング言語の特徴であるプログラムとルールの同一性に深く依存している。すなわち，Mandala は，プログラムの面から見れば階層型モジュラー・プログラミング・システムとしての色合いが強くなり，ルールの面から見れば知識プログラミング・システムとしての色合いが強くなると言えよう。

以下，本論文では，２．では，Mandala の基本構成要素を与え，基本的なプログラミング技法がどのように実現されるかについて述べる。３．では，試作されたMandala の処理系について述べる。４．では，階層型モジュラー・プログラミング・システムがMandala でどのように実現されるかを述べる。５．では，知識ベース管理システムの実現方法について述べる。そして，最後に，本論文のまとめと今後の課題について述べる。

2. Mandala の基本構成要素と基本リンク

Mandala の基本構成要素は，Concurrent Prolog の節集合から作られる単位世界(Unit World)と，Concurrent Prolog のプロセスから作られる実体(Instance)の２つであり，

(a) 単位世界　　(b) 実体

図１． 基本構成要素の表現

それぞれ図１のようなディスクおよび円を用いて図示することとする。単位世界は静的な情報を表し，プログラミング・システムとしては，管理のための最小単位であるモジュールに対応し，オブジェクト指向プログラミングでは実体を作り出すための鋳型として使われる。知識表現システムでは，単位世界は，より複雑な世界を実現するための構成要素として用いられる。 実体は，単位世界と異なって，動的な個体を表すのに用いられる。それがオブジェクト指向プログラミングで中心的な役割を果すことは言うまでもないが，その他，知識表現システムにおける知識ベース管理プログラムの実現にも用いられる。

基本構成要素の１つにプログラムの単位であるモジュールに相当する単位世界を設定することは，システムの自己記述性の点からは好ましくはない。すなわち，そのモジュールを作り出すメカニズムがMandala に備わっていないことを意味し，その結果，そのようなメカニズムを外に用意しなければならないことになるからである。Smalltalk -80 [5]などの言語では，システム記述の能力を重視して実体の中にプログラムを置くこととし，基本構成要素を１種類しか用意していない。しかし，Mandala では，知識ベース管理能力を考慮し，知識を単独で表現するための道具として単位世界を基本構成要素とした。

つぎに，４つの基本リンクinstance_of，is_a ，part_of， manager_ofを定義しよう。

（1）instance_of リンク

単位世界と実体間を関係づけるリンクがinstance_ofで，ある実体Jが，単位世界Uとinstance_ofリンクで結合されているとき，JはUの実体であると言い，図2のように波

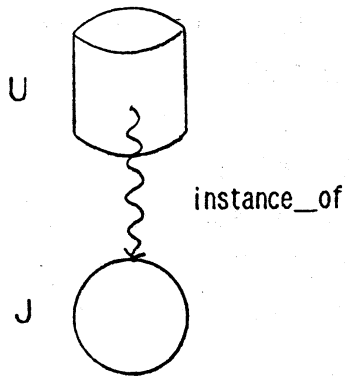

図2． instance_of リンク

線を用いて表すことにする。JがUの実体であるとき，Jの振舞いは単位世界のプログラムで規定される。実体Jは，オブジェクト指向プログラミングではメッセージを送受信して動く実体であり，単位世界UにはJが受けとるメッセージの処理の仕方を書いたプログラム（メソッド）が置かれている。

1つの単位世界はいくつもの実体を持つことができるので，instance_ofリンクは1つの単位世界から複数個張ることが可能である。しかしながら，逆に1つの実体がいくつかの単位世界につながれることはない。

（2）is_a リンク

is_a リンクは単位世界間にはられるリンクであり，単位世界間の概念の階層関係を表す。たとえば，単位世界U2が単位世界U1の特殊化になっているとき，“U2 is_a U1”となり，図3のように実線のリンクでU2をU1に結合する。is_a リンクは，概

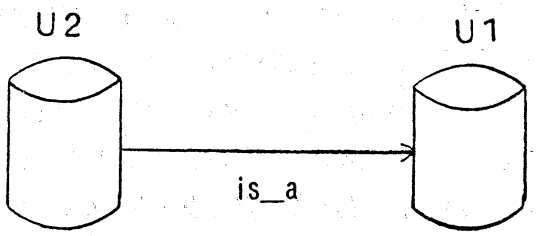

図3． is_a リンク

念階層間の性質の相続（Property Inheritance）[5),14)]を自動的に行うことを目的として張られるリンクである。性質の相続を実現する方法は，他のオブジェクト指向型のシステムと異なり，メッセージを処理するための世界の合成によって行われる。すなわち，もしある実体が受けとったメッセージの処理を，その実体が属している単位世界にあるメソッドによって遂行できなければ，is_a で結ばれた上位の単位世界を取り出し，それら２つの単位世界を結合して新たな世界を作り，そこでメッセージの処理を試みる。

ある単位世界は，その上位概念を複数個もつことができる。そして，その単位世界は，それぞれの上位概念から異なった性質を相続することになる。この多重相続（Multiple Inheritance)機能は，Concurrent Prolog のもつ手続きの非決定的選択機能を用いて実現することも可能であるが，現在のインプリメンテーションでは後で述べるように決定的に実現されている。

（３）part_ofリンク

part_ofリンクは，より小さい実体を部品として持つような複合実体を実現するためのリンクで，それは，図４のように全体から部分の向きに単位世界間に張られると同時に対

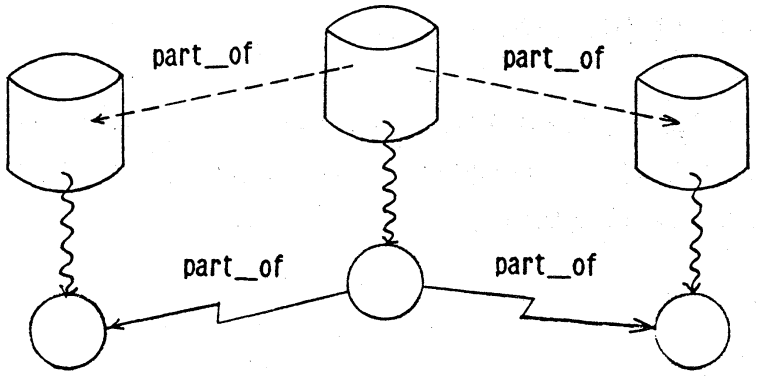

図４．　part_ofリンク

応する実体間にも張られるものである（part_ofリンクの向きは，その名前からは部分から全体への方がふさわしいが，情報のアクセス経路を反映させて逆にした）。その点が，is_a リンクとの相違でもある。すなわち，is_a リンクは，実体間には存在しない。ところで，単位世界間のpart_ofリンクと実体間のpart_ofリンクは，その役割が異なることに注意しよう。破線で表示される単位世界間のpart_ofリンクは，たとえば，目は顔の一部である[14)] といったような，一般的な事実を表すのに対して，折れ線で表示される実体間のpart_ofリンクは，特定の目を持った特定の顔を表すのに用いられる。すなわち，単位世界間のpart_ofリンクは，そのリンク自身の表す事柄が重要な意味をもつが，実体

間のpart_ofリンクは，それによって複合物を実現していることの方が重要である。この差は，それぞれのリンクの実現方法にも反映されている。すなわち，単位世界間のpart_ofリンクは，複合物を表す単位世界の方にそれが部品として持つものに対応する単位世界の識別名を登録することによって実現されるが，実体間のpart_ofリンクは，Concurrent Prolog の通信チャンネルによって実現される。そのときの通信の方向は，複合物を送信側とし，それを構成する部品を受信側とする。このような通信の方向は，外部から情報をアクセスするときの対象物の識別名の構造を反映している。たとえば，ある特定の人の目について言及するときに，その目に名前がつけられていることはまずなく，ほとんどの場合に，「誰それさんの目」のように，人の名前を経由してアクセスされるのが普通である。これは，目が顔の一部で，顔が人の一部である，というpart_ofの関係を反映している（人と目の中間に位置する部品である顔は，アクセス時には省略されることが多いが，それはシステムで自動的に補えばよいものである）。

単位世界間のpart_ofリンクは，複合物の作成時に構成部品が何であるかを調べるために辿られ，そのときに部品となる実体が同時に作られる。一般に部品には複合物よりアクセスするための名前が付けられている。この名前は複合物の鋳型である単位世界の中に記述されており，その単位世界のすべての実体で共通して用いられる。しかし，これらの名前は各実体ごとに局所的に用いられるので，複合物の別々の実体が同じ名前の部品を持ったとしても実際には別々の部品をそれぞれ持つこととなる。これにより，１つの複合物が部品として同じ単位世界の実体を２つ以上持つとき，個々の部品を区別することが可能となる。

（４） manager_ofリンク

manager_ofリンクは，instance_ofと同様に，実体－単位世界間のリンクであるがmanager_ofで結合された実体は，instance_ofで結合された他の実体（群）とは全く異なる働きをする。 manager_ofリンクは各単位世界に１つづつ付けられており，図５に示

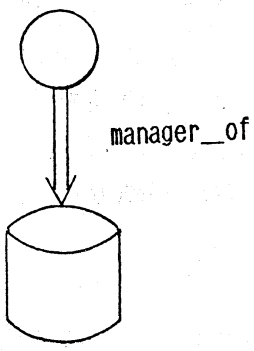

図５． manager _of リンク

すように二重線によってそのリンクを表す。 manager_ofによって結合された実体は，相手の単位世界を管理する役目を担う。

管理を行う実体（管理者）の役目は，単位世界を記述しているメソッド，すなわち Concurrent Prolog の節集合を変更することと，その単位世界に属している実体を新たに作ったり消滅させることなどである。これらの仕事は他の実体からのメッセージによって起動されるが，そのメッセージの処理手順（メソッド）は，別の単位世界に定義されており，管理者は，図6に示すようにその単位世界とinstance_ofの関係で結ばれている。

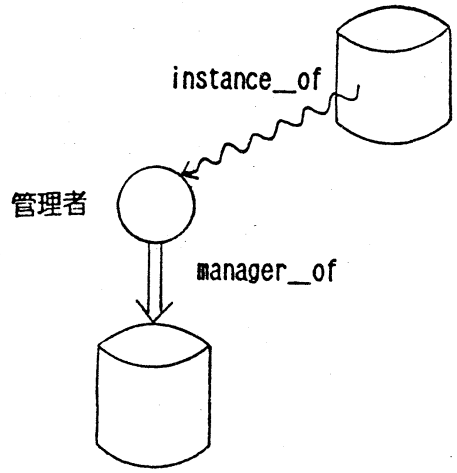

図6． 管理者

管理者の仕事と他の実体の仕事は，その性質が全く異なる点に注意しよう。一般の実体の仕事は，オブジェクト指向プログラミングでユーザが表現したい問題領域に特有の事柄に関連したものであるが，管理者の仕事はそのようなプログラムの実行を管理することである。ユーザの関心事である問題領域に関する事柄をオブジェクト・レベルと言うのに対して，管理者の扱う事柄はメタ・レベルであると言われる。

オブジェクト・レベルの処理とメタ・レベルの処理を融合させる方法がBowen & Kowalski [2]によって提案され，国藤等[7]によって，Prolog上で実現されている。それは，Prologプログラムを定理と考えたときの証明可能性を調べるdemoと呼ばれる手続きによってなされている。demoは，公理の集合，ゴール，制御，証明木の４引数を持つ述語であり，あるゴールが，与えられた公理の集合と制御から，証明木で示される通りに導かれることを示す（demonstrate ）働きをする。

我々は，demo述語を，Concurrent Prolog プログラムの証明可能性を調べる述語に拡張し，それをsimulateと名付けた[8]。simulate自身，Concurrent Prolog で書かれており，それはMandala の処理系の中核部分ともなっている。

3. Mandala の処理系

Mandala の処理系は，現在，そのプロトタイプがConcurrent Prolog 上に作成されている。本章では，まず，2．で述べた単位世界および実体の表現法，実現手段について述べ，さらにオブジェクト指向プログラミングがどのように実現されるかについて述べる。つぎに，管理者によってどのように実体が作られるのか，管理者自身のメソッドがどのようになっているのかについて述べる。本章の最後では，part_of関係の実現について述べる。

3．1　単位世界と実体の表現

単位世界は，Concurrent Prolog のガード付き節（guarded clause，以下，単に節と呼ぶ）の集合として実現され，その集合に対して1つの名前が与えられている。この名前は自分を含めたすべての単位世界およびすべての実体から見ることのできる大域的な情報である。単位世界の表現形式は次の通りである。

<単位世界名>（<節>1).
　　・
　　・
<単位世界名>（<節>n).

このように，すべての節には1つの単位世界名がつけられているが，これはその節がその単位世界に属していることを意味する。従って同じ単位世界名に結合されている節の全体がその名前で表される単位世界を構成していると言える。

実体は，Concurrent Prolog の1つのプロセス，すなわち，1つのゴールを解く過程として実現される。このゴールの形式は，常に次の形である。

```
instance (<名前>, <入力列>, <状態>)
```

ここで，<名前>はこの実体の識別名であり，<入力列>はこの実体が受け取るメッセージの時系列（メッセージ・ストリーム）を表す。第3引数の<状態>は，実体ごとの個別情報を蓄える一種のキャッシュのようなものである。たとえば，画面表示を行うプログラムであるウィンドゥ・システムで特定の窓枠fr1 を表す実体は，

```
instance (fr1 , M , [ (fr1   instance_of  frame ) ,
        (( rec ,  rectangular_area, Chan)   part_of  fr1 ) ] )
```

のような形をしている。ここで Mはメッセージ・ストリームを受け取るための変数である。本実体の<状態>はfr1 がframe の実体であることを示す節と，frame の部品である矩形領域rec が通信チャンネルChanを介してfr1 につながっていることを示す節の２つから成っている。

実体の生成については後で詳しく述べるが，単位世界はその際に鋳型として使われる。そして，１つの単位世界からいくつもの実体を作り出すことができる。すなわち，一般に単位世界はそれを鋳型とする複数の実体間で共有され，それぞれの実体はその共有されている知識（具体的には節の集合）に基づいて個々に受取ったメッセージを処理する。しかしながら，個々の実体は同じ単位世界を鋳型としていても完全に同一であることはない。このような実体間の差異は，受取って処理したメッセージの履歴を反映している。前述の述語instanceの第３引数<状態>は，このような実体ごとの履歴を吸収するために用いられる。具体的には，<状態>の内容は，メッセージを処理する過程で共通知識である単位世界内の節集合に対して追加する必要があったり，削除する必要があった節から成る。ただし，上の例で示したようなinstance_ofとpart_ofに関する情報は，実体の生成時に<状態>に置かれる。

instance（X,Y,Z ）自身，ゴールであり，そのConcurrent Prolog プログラムは，つぎのように定義されている。

```
instance(Name,[Goal    Input],State) :-
        simulate(Name,Goal,State,NewState) ,
        instance(Name,Input?,NewState?).
instance(Name,[],State).
```

ここで，第２節はメッセージ・ストリームが尽きた場合を記述しており，その場合にはその実体に当たるプロセスがその時点で消滅することを示している。さて第１節はinstanceゴールを解くプロセス，すなわち，ある実体がメッセージを受取った場合の記述であるが，プログラムにある通り，２つのプロセスsimulateとinstanceを並列に起動する。すでに述べたように実体はメッセージを受取るとそれの処理を行なうが，その実行に当たるのがsimulateプロセスである。simulateは第２引数にいま受取ったメッセージGoalを，第３引数に現在の状態State をとり，Goalを解き，新しい状態 NewState を求め，それを第４引数として返す。第２のプロセスinstanceは， NewState を状態とし，Goalに引きつづくメッセージの処理に当たるこの実体そのものを表す。

simulateプログラムはConcurrent Prolog で記述することも可能であるが，効率を上

げるため現在はConcurrent Prolog のシステム述語として実現されている。その機能を以下に要約する。

simulate (Name,Goal,State, NewState) の機能

① GoalをState （これは実際には節のリストである）を用いて解く。
② ①に失敗した場合，State 中の“Name instance_of W.”という節（Wはこのsimulateを起動した実体の鋳型になっている単位世界の名前である）を取出し，GoalをState と単位世界Wの知識を用いて解く。
③ ②にも失敗した場合，Wの中から“W is_a Vo .”という形の節を１つ取出し，GoalをState , W, Vo を用いて解く。
④ ③にも失敗した場合，Vo の中から“Vo is_a V1 .”という形の節を１つ取出し，GoalをState , W, Vo , V１を用いて解く。これにも失敗した場合，さらにis_a 節を取出し節集合をより大きくしてGoalを解こうとするが，それは一般的には次のように述べられる。
「GoalをState , W, Vo , V1 , …, Vn を用いて解くことに失敗した場合，Vn の中から“Vn is_a Vn+1 .”という形の節を１つ取出し，GoalをState , W, Vo , V1 , …, Vn , Vn+1 を用いて解く」
⑤ ④において，State , W, Vo , …, Vm を用いてGoalが解けず，かつ，Vm がis_a 節を１つも含まなかった場合には，Vm-1 において，すでに選んだis_a 節（“Vm-1 is_a Vm .”）とは別のis_a 節“Vm-1 is_a V´m .”を選び，GoalをState , W, Vo , …, Vm-1 , V´m を用いて解く。Vm が他にis_a 節を含まなかった場合には，この処理はVm-1 におけるis_a 節の選択まで逆戻りして行なわれる。
⑥ ⑤にも失敗した場合，simulateは失敗する。
⑦ 以上の過程で，Goalがadd(C)あるいはdelete(C) （それぞれ節 Cの追加，削除を意味する）であった場合には，この操作をState に対してのみ行ない，他の単位世界WやVi は変更しない。ただし，この操作ではState の直接的変更はしないで，代りに更新された NewState （これが第４引数に返される）を作る。

上の要約において，③～⑤が概念階層間の性質の多重相続（Property Inheritance)を実現していることに注意。③～⑤で述べていることを別の言葉で述べると次のようになる。一般にsimulateは実体の鋳型を根（root）として，is_a 関係で張られる単位世界からなる木を深さ優先（depth-first ），に左から右へと，Goalを解くことのできる単位世界の組合せを探索する。このとき分岐した枝のそれぞれは論理的にはOR関係にある単位世界を表わしている。図７に簡単な単位世界の例として矩形領域 rectangular_ar

```
rectangular_area(rectangular_area  is_a  'Simple_Instance').
rectangular_area(state((20,5,30,10))).
rectangular_area(clear:-state(Param) & clear-primitive(Param)).
```

図7.　記述例1：矩形領域

eaを示す。ただし，図中&はConurrent Prologに新たに導入した逐次AND であり、A&B は Aを解き、それが終了したら Bを解くことを意味する。図にあるように，一般にis_a 関係は単位世界中に節の形で埋め込まれている。

3．2　実体の生成

ある単位世界を鋳型としてその実体を生成する仕事は，その単位世界を管理している実体（管理者）が行う。一般に，単位世界の管理者自身は，図8に示す単位世界' Mana

```
'Manager' ('Manager'  is_a  'Simple_Instance').
'Manager' (number(0)).
'Manager' (create(Name,Goals) :-
          delete(number(X)) & X1 := X+1 & add(number(X1)) &
          add(instance(Name,Goals)) & Cname instance_of  Mname &
          instantiate(Cname,Name,DW)
          instance(Name, [init   Goals],DW)).
'Manager' (how-many :- number(X) & write(X) & nl).
```

図8.　単位世界：'Manager'

ger'，あるいは'Manager' をis_a 階層の上位にもつ単位世界を鋳型として作られる。管理者自身は誰が作るかが問題となるが，現在は，初めから組込みの「超管理者」を1つ作っておいてこの問題を回避している。超管理者の鋳型自身'Manager' としてもよいが，5．2で述べるように，管理者は自分が管理している単位世界を更新する能力を有するので，自分自身で自分の鋳型を更新することになってしまう。これを避けるために，超管理者用の別の鋳型を用意することとする。

管理者は，自分が管理している単位世界の実体を作る際に，同時に自分のメモとして

<状態>にその実体の名前とそこにメッセージを送るための通信チャネルの対を登録し，現在の総実体数を記憶しておく。実体の起動時に，初期化のためにメッセージ“init”を与える。“init”メッセージは，その実体が部品をもつとき，その部品を自動生成するために用いられる。

3．3　part_of関係の実現

前節で述べたように，実体はそれが生成されたときに必ずinitというメッセージを最初に受取る。このメッセージは実体の初期化処理を起動するために用いられるが，初期化処理の中で一般的なものとして，複合実体の部品となる実体の自動生成がある。現在，この自動生成は複合実体の鋳型となる単位世界のis_a 階層の上位に'Composite_Instance' という単位世界を置くことにより解決されている。この場合，複合実体に対するinitメッセージはすべてこの'Composite_Instance' 内のinit処理記述を用いて処理さ

```
'Composite_Instance'('Composite_Instance' is_a  'Instance').
'Composite_Instance'(send_to(Name,Msg) :-
    delete((Name,Class,  [Msg   New]) part_of Me) &
    add((Name,Class,New)  part_of Me)   true).
'Composite-Instance'(init :-
   get_all _parts(Ps)
  'Enumerate'({( Name,Class,Chan)   (name,class,Chan) part_of  Me},List) &
  'Map'({(Name,Class,Chan)
    instantiate(Class,Name,World) & instance(Name,[init   Chan],World)},List)).
```

図9．　単位世界：'Composite _Instance'

れる。図9にこの単位世界を示す。第2節は複合実体とその部品実体間のメッセージ交換処理の記述である。第3節がinitメッセージの処理を記述している。記述中にEnumerate,Map 等の述語を用いているが，これらはすべての単位世界間で共通して用いられる述語であり，従って単位世界間のis_a 階層の最上位に位置する'Instance'という単位世界に定義が書かれている。init処理で行なっていることを要約すれば，複合実体の部品となっている実体の鋳型である単位世界名をすべて取出し（これは複合実体の鋳型である単位世界中にpart_of関係を用いて列挙してある(図10))，それぞれの実体を生成し，それぞれにinitメッセージを与えるということを行なっている。これにより，複合実体が部品としてさらに複合実体をもつということも許されるようになる。図10

に複合実体の鋳型の例としてframe を示す。frame は４本の枠線を持つ長方形であり，図７の rectangular_areaを部品としてもつ。

```
frame(frame is_a 'Composite_Instance').
frame((rec,rectangular_area) part_of frame).
frame(draw :- send_to(rec, state(Param))   draw-lines(Param)).
frame(refresh :-  send_to(rec,clear)   draw).
```

図１０．　記述例２：frame

図１１にこれまで現れた単位世界間の関係を図示する。図中の'Simple-Instance' という単位世界は部品を持たない単位世界間のis_a 階層の最上位に位置するものである。

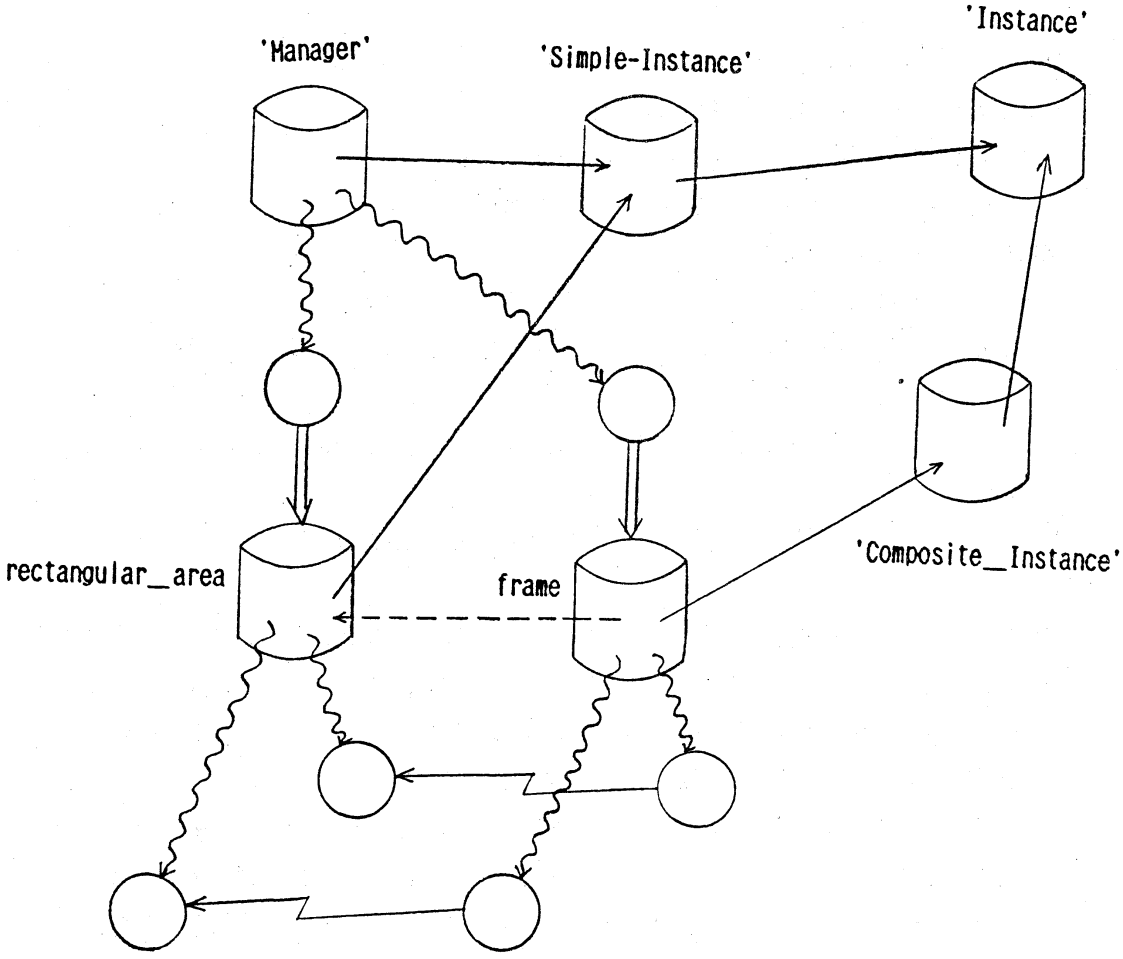

図１１．　単位世界間の関係

4. Mandala による階層型モジュラー・プログラミング・システムの実現法

Mandala は，知識プログラミング・システムであると同時に，Concurrent Prolog を親言語とする階層型モジュラー・プログラミング・システム（HMPSと略記する）の骨格を形作るものと考えることができる。

HMPSは大規模なソフトウェアの開発を容易にし，それによってソフトウェア生産性の向上を図ることを目的としている。HMPSはプログラミング・システムであるから，機能面もさることながら，十分に実用的な速度で実行されるようなプログラム・コードを作り出すことも重要である。すなわち，コンパイラ，オプティマイザなどの処理系が不可欠である。

しかしながら，本論文では，機能面に焦点をあて，Mandala で，階層化およびモジュラー・プログラミングがどのように実現されるかについて述べることにする。

4．1 Mandala による階層化の実現

HMPSにおける階層性は，データ抽象化に基づくのが普通であるが，Mandala のそれは ι（イオタ）[9] と同様，より一般的に概念の階層性に基づいている。そして，データ抽象化は，概念の階層性の一種として促えている。すなわち，Mandala における階層化は，概念の階層性を表すためのリンクであるis_a リンクによって実現される。たとえば，整数および文字列はともに順序集合であり，分類プログラムsort (X,Y,P)は，順序集合上で定義されている。ここで Pは要素の順序を判定する述語引数である。このとき，整数上および文字列上の分類プログラムは，パラメータ化されて順序集合上に定義されているプログラムから，パラメータを適当に設定することによって作り出すことができる。たとえば，整数上の分類プログラムを得るには，パラメータ化された述語引数を整数の大小比較述語とすればよい。これらの関係は，図１２に示されている。

4．2 Mandala によるモジュラー・プログラミング

モジュラー・プログラミングは，大きなプログラムをより細かいモジュールの組み合わせによって作り上げることをその目的としている。モジュラー・プログラミングを実現するための条件は，プログラミング言語が，プログラムの分割作成に適していること，モジュール間インタフェースがとり易く，しかもその自由度が大きいこと，モジュール間の独立性が強いこと，モジュールが多くのプログラムで利用できること，すなわちモジュールの一般化，などである。いま，論理プログラミング言語での節をモジュールとすると，モジュール間インタフェースは，引数間の同一化（unification ）によって，

行われるので，その記述は容易であり，しかも呼び出す側，呼び出される側のどちらに変数が現れても構わないことが示すように，その結合の自由度は大変大きい。また，プログラムは節の集合で与えられるので，その独立性は十分である（逐次実行を基本としている通常のPrologは同一名の節間に順序関係が存在し，その順序が節の選択順序となるが，Concurrent Prlogではその順序も存在しない）。すなわち，論理プログラミング言語それ自身が，モジュラー・プログラミングを実現するための条件のうち，いくつかを備えていると言える。

他の２つの条件，すなわち，プログラムの分割作成が容易な言語であることと，モジュールが一般化され得ることは，より難しい問題である。そのうち，モジュールの一般化は，前節で述べたパラメータ化によるモジュールの抽象化が有効である。プログラムの分割作成を助長する機能として，オブジェクト指向プログラミング機能が考えられる。それはとくに並列環境下において，問題の構造を反映した分割法を行うのに適している。

Mandala においては，part_ofリンクがこのモジュール分割を実現するのに用いられる。part_ofリンクはその名前が示すように，全体と部分の関係を示し，ある実体が，いくつかの部品から作られていることを表す。前章で述べたframe と rectangular_areaはこの例になっている（図１１を参照のこと）。

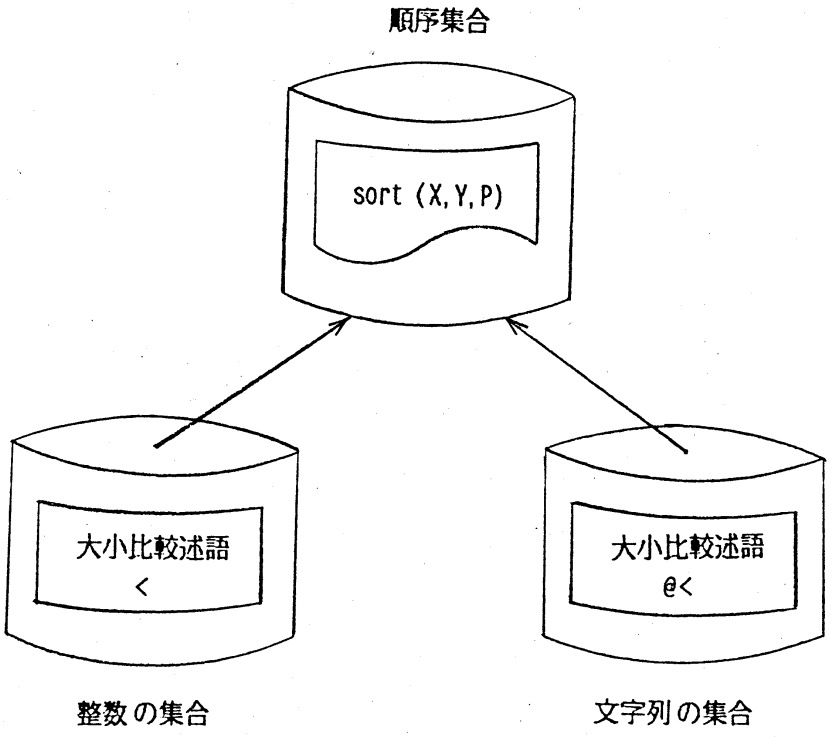

図１２．　概念の階層化

5. Mandala による知識ベース管理システムの実現法

知識プログラミング・システムMandala は，単位世界に現れるプログラムを知識と見なすことによって，知識ベース管理システムの基本的枠組を与えていると考えることもできる。

知識ベース管理システムの基本機能として，知識表現，知識利用，知識獲得などの諸機能が考えられるが，知識表現の土台は，Mandala の基本機能に含まれているので，ここでは知識利用および知識獲得の両機能がどのように実現されるかを示そう。

5．1　知識ベースの検索

知識の利用は，知識ベースからいかに必要とする知識を検索するかにかかっている。ゆえに，ここでは，知識ベースの検索機能に絞って話を進めよう。

Mandala では，あるまとまった単位の知識は１つの単位世界を用いて表される。それは，たとえば関係データベースでの１つの関係に対応すると考えればよいであろう。単位世界にある情報を取り出すためには，その単位世界専用の質問処理を行う司書を置くことが必要である。そのような司書はその単位世界の実体として実現できる。すなわち，単位世界を知識ベースとして考えたとき，instance_ofリンクで結ばれた実体は，その単位世界によって表される実体と考えるよりも，むしろその知識ベースを検索する司書と考えた方がよい。

ところで，ある場合には質問者自身が臨時にそのような司書の役割を果してよい場合もある。そのためには，検索要求の他にその検索文を発する先の単位世界名をも知っていなければならない。この情報は，もともと質問者が知っていればそれでよいが，もし知らない場合には，誰かに教えてもらわなければならない。通常は，より上位の問題解決の役割を担っている実体が，検索文と単位世界の対を下位の質問者にメッセージとして送ることになるが，問題解決者自身は，連想などの手段により適当な単位世界を見つけなければならない。

知識ベースに専用の司書を置くと，その司書に知識ベースの一時的な更新機能を持たせることができる。その更新された知識は司書自身が保有していて，元の知識ベースは変化しない。この機能を用いて，仮定に基づく推論が実現され，さらに拡張して，いくつもの司書に別の仮定を置くことによって，同時に多くの世界を仮定してどの仮定が最も妥当であるかを調べるような問題を扱うことが可能となる。たとえば，MYCIN [12] で

は，ある感染症を仮定したときの妥当性を，すべての感染症について計算し，その中で最も確度の高い仮定から順に，いくつかを結論として選択しているが，これはまさにその種の問題と言えよう。

知識ベースの局所的更新は，司書自身が持っている状態の変化として実現できる。これは，知識ベース管理者が行う全域的更新とは異なることに注意しよう。全域的更新では，実際に単位世界自身を書き替えてしまう。そのため，その種の更新の方がシステム全体に及ぼす影響はずっと大きく，厳重な検査が必要となる。次節では，この問題を取り上げよう。

5．2　知識ベース管理と知識の同化

知識ベースの管理のための知識獲得機能としては，種々のレベルのものが考えられるが，論理プログラミング言語との整合性から見て，論理としての無矛盾性および非冗長性の維持を図りながら知識を獲得していく同化の機能を[2),7)]取り上げて考えることにする。

われわれは，通常の逐次実行に基づくPrologの上で無矛盾性管理および冗長性除去を行う知識同化プログラムを実現した[7)]。そのプログラムは，知識を個々の実体に関する具体的事実を与える肯定的知識と，そのような具体的事実が満足しなければならない条件を与える否定的知識に分け，新たな肯定的知識を獲得するときに否定的知識を調べるという方法を採っている。そして，それ自身，2．で述べたdemo述語を拡張して作られている。しかしながら，demo述語の拡張による知識同化プログラムは，実行効率の点で必ずしも現実的でないことが明らかとなった。すなわち，無矛盾性について言えば，知識を獲得するたびにすべての否定的知識についてそれらが成り立っていることを示さなければならないとすれば，その実行時間は否定的知識の量に比例することになる。また冗長性除去のアルゴリズムは，1つ1つの肯定的知識の冗長性を調べるので，その実行時間は肯定的知識の量に比例する。

われわれは，Mandala 上で，より実行効率のよいアルゴリズムを実現する見通しを得た。第1に，無矛盾性の検査についていえば，肯定的知識および否定的知識を適当に分割管理することによって，1つの肯定的知識を獲得する際に調べる必要のある否定的知識の量を減らすことができる。また，各否定的知識の検査自身はPrologのゴール文の証明によってなされるがその実行自身，並列化が可能であることも指摘しておきたい[6)]。

第2に，冗長性除去について言えば，制限された範囲内であれば，各肯定的知識につ

いて独立にその冗長性を調べ，その後で冗長であると判明したすべての知識を除去することができる。いま，含意関係“→”によってグラフを作ったとき，閉じたループがないと仮定し，知識ベースTからPとQが独立に冗長であることが示されたとする。冗長性は，たとえばT－PからPが導かれることによって示されるが，含意関係のループがないとしたので，Pの証明にQが使われると同時にQの証明にPが使われるようなことはない。ゆえに，このような場合には上に述べた並列アルゴリズムによる冗長性除去が可能となる（このような条件が満たされない例としては，A，B，A→B，B→Aを考えればよい）。そして，一般に無限ループに陥らないようなPrologあるいはConcurrent Prologプログラムは，含意関係のループがないので，この手法が適用できる。

並列化による実行速度の向上は，Concurrent Prolog を並列に実行できる計算機の出現が必要条件であり，それはこれからの研究に待たなければならない。が，少くともそのような並列化の有用性を示しているものと言えよう。知識同化プログラムをMandala上で実現する方法の概略を，図１３に示す。

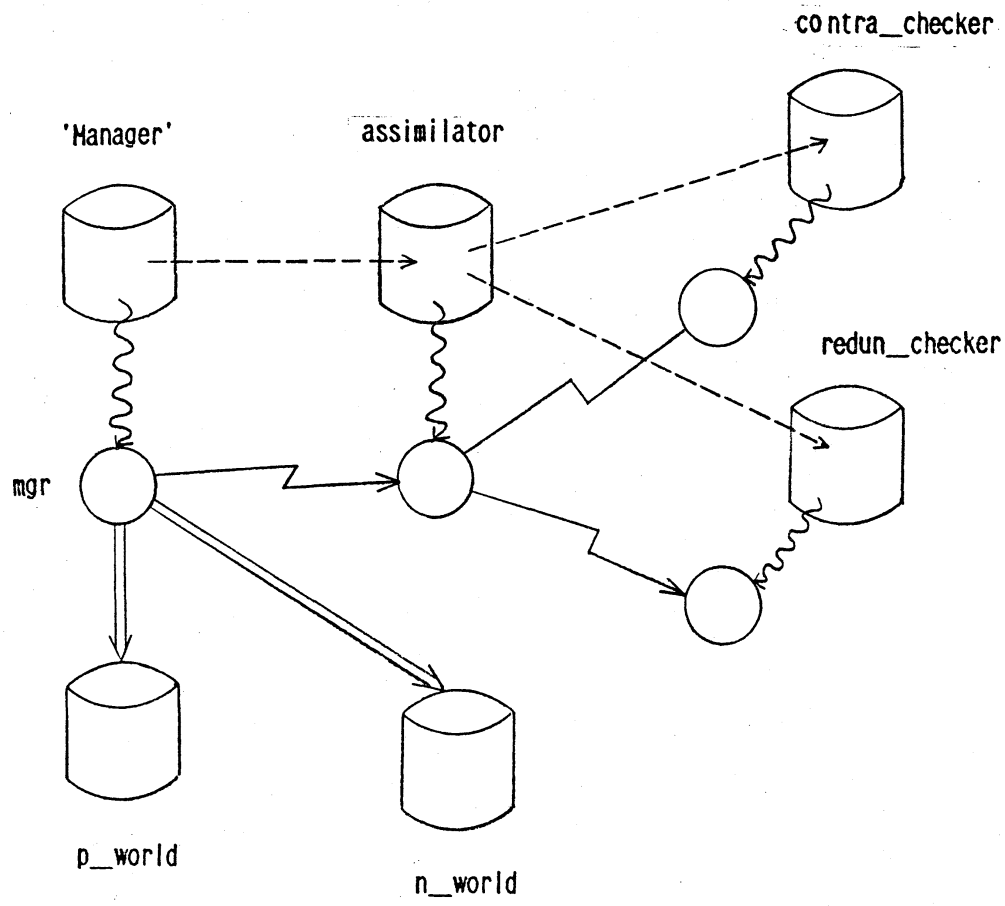

図１３．　知識同化プログラム

この図から分かるとおり，知識ベース管理プログラム'Manager' は，その部品としてassimilator を持つように拡張されている。assimilator 自身は2つの部品モジュールcontra_checker と redun_checker を持つ。'Manager' の実体mgr は， p_world および n_world と呼ばれる2つの単位世界を管理している。 p_world は肯定的知識から成り， n_world は否定的知識から成る。否定的知識は，データベースの用語で保全性制約(Integrity Constraint)と呼ばれているものであり，それは，いくつもの異なる p_world に共通して使える。それが， n_world を p_world と分離して管理する理由である。Smalltalk-80 [5]も，ここで述べた管理者の考えが使われており，単位世界に対応するクラスの1つ1つにそれを管理するためのオブジェクトが付随している。しかし，その対応は1対1であり，しかもその拡張性が乏しい。Mandala においても，原則的には，知識ベース管理者は特定の単位世界を管理することを考えているが，しかしそれは何も1つに限ったことではない。管理対象を1つに限ったとしても，図13のように，2つの異なる単位世界を参照していることになる。Mandala の優位性は，このような柔軟性にあると言ってよい。

6．まとめと今後の課題

本論文では，Concurrent Prolog 上の知識プログラミング・システムMandala の基本的な枠組とその実現方法について述べた。また，Mandala が，階層型モジュラー・プログラミング・システムおよび知識ベース管理システムの土台となることも述べた。

しかしながら，Mandala の開発は，ごく初歩的な段階にすぎない。これから，多くの研究を積み重ねて，より完成されたものにして行かなければならない。

Mandala をプログラミング・システムと見た場合，速い処理系の開発が最も必要とされる。Concurrent Prolog 自身，完成された高速の処理系が未だ存在しないので，そこから出発することが必要である。また，プログラミング環境の整備，すなわちdebug のための道具となるプログラム技法の開発，プログラムのファイル管理の整備などが必要である。とくに並列プログラムのdebug は理論的にも未開発であり，解決すべき問題は多い。

プログラミング・システムとしてのMandala 上でとくに興味があるのは，部分実行機構の導入である。それはパラメータ化あるいはコンパイルの概念とも密接に結びついている。もし，パラメータの値を早めに設定して部分実行を行わせるとコンパイルがなされることになる。その場合，プログラムの実行時におけるパラメータの動的な変化は行うことができない。プログラミング・システムとしては，部分実行のレベルを自由に設定できることが望ましい。そうすることによって，柔軟性と実行効率のトレード・オフをユーザが制御できるようになる。ある意味で，柔軟性と実行効率のどちらに重点を置くかによって，そのシステムが，知識プログラミング重視型になるか一般のプログラミング重視型になるかが分かれるところとなろう。すなわち，性能を含めてシステムを考えたときに，真に二面性をもたせるためには，部分実行機能による一方から他方へのスムーズな移行が必須であろう。

知識ベース・システムと見た場合，プログラミング・システムと同様に，知識ベースの開発を容易にするような支援システムの開発が望まれる。それは知識ベース・エディタと呼ばれるものである。知識ベース・エディタの一部分に，本論文で述べた知識同化システムが位置づけられることは言うまでもない。その他に，知識ベースの版（veision ）の管理，更新に伴う排他制御の問題なども，知識ベース・エディタの機能として考えられるべきであろう。さらに，知識ベース・エディタは，その扱う範囲を単位世界から個々の実体にまで拡げてみることが考えられる。たとえば，各実体が持っている個別の知識のupdate や，実体間の通信チャネル網の動的な変更などの機能の実現である。

知識ベース・システムとして考えたときの，もう1つの拡張のポイントは，その表現力を強化することである。われわれは，単位世界に現れる知識はConcurrent Prolog のプログラムとしたが，この制限は強すぎる。知識表現言語KRYPTON は，不確定な知識を扱うために，Horn Logicより強力である一階述語論理の文を記述できるようにしているが，Mandala の拡張の方向として，同様の機能の実現が考えられる。このためには，一階述語論理の強力な定理証明機が必要であり，かつ，それ自身，Concurrent Prolog で実現されることが望まれる。

Mandala の親言語は，現在はConcurrent Prolog であるが，将来は，第5世代コンピュータ・プロジェクトにおける並列推論マシンの機械語である KL1に移行することが計画されている。 KL1自身は現在設計中の言語であるが，それはConcurrent Prolog を中核として，それに集合の扱いおよびモジュール化機能を付加したものと想定している。Mandala の発展のためには，多くの応用プログラムを記述してみて，その経験からのフィード・バックを得る必要がある。われわれはすでにMandala によるシミュレーション・プログラムを実験的に試作し，ルール指向プログラミング，あるいは自然言語処理への応用についての検討も始めている。それらの結果については，いずれ，別の機会で報告をしたいと思う。

謝　辞

本研究は，ＩＣＯＴにおける第５世代計算機開発プロジェクトの一環としてなされたものであり，本研究の機会を与えていただいた渕一博当研究所長，ならびに，有益な議論を展開していただいたＩＣＯＴ研究員，ＩＣＯＴ Working Group２，３および４の各メンバー，知識表現タスク・グループの討論に参加された富士通，日本電気，沖電気の各研究員に感謝いたします。とくに，ＷＧ４の主査である溝口文雄東京理科大学助教授は，Mandala を生み出すきっかけを作っていただいたのみならず，その後の討論を通じて常に貴重な助言をいただきました。さらに，当研究所の招聘研究員のEhud Shapiro博士(Weizmann Institute of Science) およびKeith Clark 博士(Imperial College)との議論も有益であったことを追け加えておきます。

References

[1] D.G.Bobrow & M.Stefik : The LOOPS Manual (Preliminary Version), XEROX PARC Knowledge-based VLSI Design Group Memo KB-VLST-81-13, 1983.

[2] K.A.Bowen & R.A.Kowalski : Amalgamating Language and Meta Language in Logic Programming, School of Computer and Information Sciences, Unive r sityof Syracuse, 1981.

[3] R.J.Brachman et al. : KRYPTON : A Functional Approach to Knowledge Representation, Fairchild Laboratory for Artificial Intelligence Research, Fairchild TR No.639, 1983.

[4] M.R.Genesereth et al. : MRS Manual, Stanford University, stanford Heuristic Programming Project Memo HPP-80-24, 1980.

[5] A.Goldberg & D.Robson : Smalltalk-80 : The language and its implementation Addison-Wesley, 1983.

[6] H.Hirakawa et al. : OR-Parallel Optimizing Prolog System : POPS-Its Design and Implementation in Concurrent Prolog, SSE Symposium , 京都大学数理解析研究所 , 1983.

[7] 国藤 他 : Prologによる対象知識とメタ知識の融合とその応用, 情報処理学会知識工学と人工知能研究会資料 30-1, 1983.

[8] 国藤 他 : メタ推論とその応用－並列メタ推論用メタ述語simulateについて－, 情報処理学会, 第28回全国大会, 1984.

[9] R.Nakajima : Hierarchical Program Specification and Verification -a Many-sorted Logical Approach, Acta Informatica, 14(1980), pp.135-155.

[10] G.S.Novak Jr. : GLISP : A Lisp-based Programming System with Data Abstraction, AI magazine, Fall (1983), pp.37-47.

[11] E.Shapiro : A Subset of Concurrent Prolog and Its Interpreter, Institute for New Generation Computer Technology, ICOT TR-003, 1983.

[12] E.H.Shortliffe : Computer-Based Medical Consultations : MYCIN, America n
Elsevier, 1976.

[13] 竹内 : 論理型並列プログラミング言語－Concurrent Prolog －, ソフトウェア, vol.1, No.1 (1984). (?)

[14] 田中 : 計算機による自然言語の意味処理に関する研究, 電子技術総合研究所研究報告第 797号, 1979.