ICOT Technical Memorandum: TM-0295

TM-0295

推論エンジン KORE/IE

—反駁メカニズムに基づく高速な推論エンジンの実現—

新谷虎松
(富士通)

April, 1987

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

推論エンジンKORE／IE
―反駁メカニズムに基づく高速な推論エンジンの実現―

新谷虎松

（富士通㈱国際情報社会科学研究所）

1. はじめに

KORE/IEは問題解決支援環境KORE（Knowledge Oriented Reasoning Environment）[新谷 86]における推論エンジン・サブシステム（部品）として機能する。一方、単独では、OPS5[Forgy 81]で代表される前向き推論型プロダクションシステムとして用いることができる。本論文では、KORE/IEの特長及び推論の高速化のために用いた手法について議論する。KORE/IEの主な特長は(1)KOREにおける推論エンジン機能、(2)ルールベース間の協調問題解決機能、(3)ルールベース毎の競合解消戦略定義機能、(4)高速な推論の実現等出ある。特に、論理型言語をベースにしたシステムにおいて、ルールの表現力を犠牲とせずに(4)の推論の高速性を達成することは重要な課題であり、アプリケーションプログラムを構築する上で強く要望される本質的な機能である。KORE/IEでは、プロダクションシステムにおける認識-行動サイクルの構成要素を高速化するために、論理型言語の利点（例えば、高速な反駁（Refutation）メカニズムや部分計算技法など）を素直に導入することにより、その高速性を実現する。現在、KORE/IEはProlog（C-Prolog及びQuintus Prologで約1900ステップ）で実現され、その高速性はLisp(Franz Lisp)上のOPS5に匹敵する。

2. KOREにおける推論エンジン機能の実現

2.1. 知識の制御と構造化

問題解決支援環境KORE[新谷 86]は問題解決を支援するための知識の制御機構と知識の構造化機能を提供する。知識の制御機構としてはKOREのサブシステムとして1)KORE/IE(Inference Engine subsystem)以外に、2)データ指向的制御を提供するKORE/DB(Data Base subsystem)、3)オブジェクト指向的制御を提供するKORE/KR(Knowledge Representation subsystem)、4)ネットワーク指向的制御を提供するKORE/EDEN(Extended Dependency Network subsystem)がある。これらサブシステムは、単独でも独立的に機能し、全体としてKOREに様々な問題解決支援機能を提供する（図1参照）。

KOREでは、KOREサブシステムで用いられる知識の内部表現を統一的に管理するための構造として関係テーブルを用いる。これらサブシステムは、内部表現において共通の関係テーブル表現をもつことにより、KOREとして統合化される。このような内部表現の統一性は問題解決過程において、KOREサブシステムを協調させるための手段を提供する。つまり、KOREの各サブシステムはこのような関係テーブルをコミュニケーションの場として用いることにより、他のサブシステムの問題解決過程の結果を利用したり、問題解決そのものを支援することが可能になる。

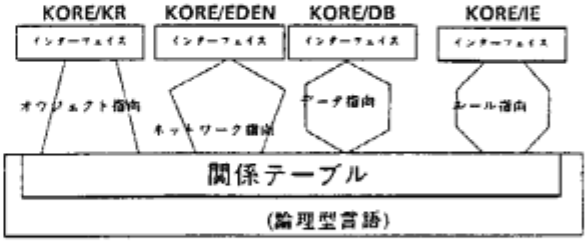

図1. 問題解決支援環境KOREの構成

関係テーブルは、Hayes-Roth［1978］等が提案した黒板モデルにおける黒板と見ることができる。しかしながら、KOREにおける関係テーブルは異なった知識制御を担うKOREサブシステム間における情報の交換場所であり、KORE/DBにより管理される関係データベースであるという点に特徴がある。具体的には、関係テーブルはPrologにおける関係表現（事実）として素直に実現される[Kowalski 77]。つまり、n項関係における引数の位置はテーブルの列に相当し、個々の関係はテーブルの行に相当する。

2.2. 推論エンジン機能の実現

前節で論じたように、KORE/IEと同様に、他のKOREサブシステムにおける内部表現（特に、宣言的情報）もサブシステム間で共通な関係テーブルで統一的に表現される。そこで、KORE/IEは、特別な制御機構を準備することなしに、他のKOREサブシステムの推論エンジン部品として協調的に利用することができる。

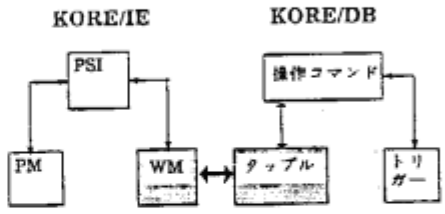

図2. KORE/IE とKORE/DB の協調

例えば、図2は、KORE/IEを関係データベースサブシステムKORE/DBの推論エンジンとして用いる協調例である。これは、KORE/DBとKORE/IEをPrologシステム上で共存させること（各システムを読み込むこと）により達成され、KORE/IEを推論エンジンとして組み込むことに相当する。つまり、KORE/DB によるタップルの追加はKORE/IE にとって

は作業記憶(WM)の変化に相当し、これによりルールが起動されるきっかけとなる。また、ルールの起動の結果、WMの変化はKORE/DB のトリガー起動のきっかけとなり関係テーブル（データベース）の変更をもたらす。これにより、KORE/DBとKORE/IEは協調的に問題解決を図ることが可能になる。

図３で示した例は、KORE/KR とKORE/IE との協調問題解決の例を示している。つまり、KORE/KR のインスタンスの生成やそのスロット値の変更をKORE/IE におけるルールのトリガー条件として用いることができ、また、KORE/IE におけるルールの実行過程（WM）を構造化された知識としてKORE/KR を用いて管理することができる。これは、KORE/IE を推論エンジンとして、KORE/KR を知識ベースとして用いたことに相当する。

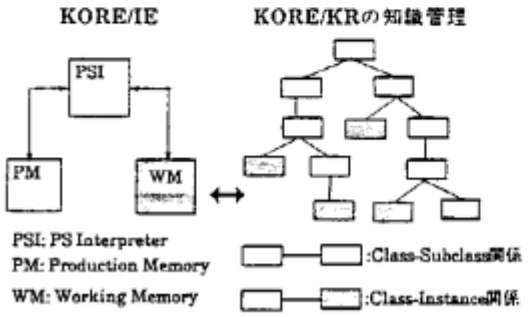

図3. KORE/IEと KORE/KRの協調

KOREにおいて、この様なサブシステム間の協調動作のための機構は、単に、各サブシステム内に他のサブシステムの有無をチェックするフラグを実現・利用することにより達成される。普通、このようなフラグを用いてシステム間の協調動作機構を実現すると、一般的には、システム全体としての実行効率を下げることになる。実行効率低下の原因は、各サブシステムを統合・制御するためのメタ制御機構が必要となるからである。一方、KOREでは、このようなメタ制御機構の必要性を回避するために、各サブシステムの制御機構は直接にPrologの計算メカニズムである反駁機構を用いるように設計されている（各サブシステムにおけるプログラミングは全てPrologのプログラムに変換される）。これにより、サブシステム統合化によるシステムの実行効率の低下を回避している。つまり、本手法は、Prologの重要な特長の一つである段階的プログラム開発の容易さを利用しており、各サブシステム及びサブシステムを用いたプログラムを全て統一的な形式でPrologプログラムに変換し利用することにより、サブシステムの統合を容易に達成する。これにより、サブシステムを組み合わせたシステムはあたかも全体で一つのシステムとしての振舞が可能になる。

3. KORE/IEの特長

KORE/IEの基本的機能はOPS5をベースにして機能拡張したものであり、OPS5的なルール指向的プログラミング環境[Brownston 85]を提供する。KORE/IEはOPS5と同様に、純粋なプロダクションシステムのメカニズムを採用しており、その構成要素として①PSI(PS Interpreter),②WM(Working Memory), ③PM(Production Memory)を意識したルール表現形式を用いている。推論（ルールの実行）は認識-行動サイクルを実行することにより達成する。認識-行動サイクルは、(1)照合(Matching),(2)競合解消(Conflict Resolution),(3)動作(Action),より構成され、これら(1)～(3)を繰り返す。(1)の照合は全てのルールの条件部（ＬＨＳ(Left Hand Side)と呼ぶ）とWMの内容を照らし合わせて、ＬＨＳを満足するルール（インスタンシエーション(Instantiation)と呼ぶ）を選び出す。インスタンシエーションの集合は競合集合と呼ばれ、(2)の競合解消時に一つ選択され、起動されるべきルールが決定される。KORE/IE の特徴は、これら構成要素のうち、WM(宣言的情報）とPM(手続き的情報）がKOREの統一的な内部表現形式である関係テーブルで表現され、PSIはKOREの制御機構としてPrologの計算メカニズムである反駁(refutation)メカニズムを直接に利用して実現されることである。これらは、KORE/IEを高速化するための重要な特長である（４節参照）。また、KORE/IEにおけるWMの操作アクション（make（WM要素の作成）, modify（WM要素の修飾）, remove（WM要素の削除）等）やトップレベルコマンドのliteralize（ルール中に使用されるクラス名、属性名を宣言する）等は表１で示すようにKORE/DB の関係テーブル操作コマンドに相当する。

機能		KORE/DB	KORE/IE	機能	
テーブル定義		create	literalize	クラス定義	
タプル操作	挿入	ins	make	追加	WMの操作
	更新	update	modify	更新	
	削除	del	remove	削除	

表1. KORE/IE とKORE/DB コマンドの対比（同一性）

3.1. ルール記述

KORE/IEにおけるルール記述は、Prologにおける項記述のシンタックスを十分に取り入れることにより、その可読性とルール表現の柔軟性を実現している。KORE/IE におけるルールは、(1)ルール名, (2)シンボル":", (3)シンボル"ｉｆ", (4)LHS(left hand side), (5)シンボル"ｔｈｅｎ", (6)RHS(right hand side), (7)シンボル". "により構成され、 次のように記述される。

```
RULE_NAME : if Condition1 & Condition2 ‥
                    then Action1 & Action2 ‥.
```

ここで、RULE_NAMEはルール名である。Condition.Actio

nは，それぞれLHSにおけるルールの条件要素（LHSパターンと呼ぶ），RHSにおけるルールのアクション（RHSアクションと呼ぶ）を示す．複数のLHSパターンやRHSアクションはシンボル"&"を用いて区切られる．LHSは、ルールが適用可能になる状態の記述であり、ルールが処理できる情報がWMの中に存在したときに真となる．例えば，ルールは次のように記述される．

```
temp:  if  weather_data(place=PLACE,temperature=TEMP)&
           work(name=measuring_temperature,place=PLACE)
       then
           modify(1,[temperature=compute(TEMP+10)]).
```

例で示されるように，KORE/IEにおけるLHSパターン記述はPrologの項表記法を素直に採用している．我々は、このように記述される情報を宣言的情報と呼ぶ．このパターン表記法は、KOREにおける宣言的な情報を記述する際の統一的な表記法である．KORE/IEのWMはこのような宣言的情報の集まりであり、これはKORE/DBが管理する関係データベースと同一である（表１参照）．パターンは、次に示すように、1)クラス名と2)いくつかのスロット記述である"属性－値"対により構成される。

クラス名（スロット$_1$=値$_1$，スロット$_2$=値$_2$，...，スロット$_n$=値$_n$）

さらに、この正のパターンに対して、負のパターンはクラス名の前にマイナス記号"－"を付加することにより表記する．スロット記述には、上例で示した値が等しい(unifyする)ことを示す"＝"以外に、"＼＝＝（等しくない）"，"＝＜（等しいか小さい）"，"＞＝（等しいか大きい）" 等のPrologで用いられる比較式を同様な意味で用いることができる．また、スロット値の記述には以下のようなand記述やor記述が可能である。

ａｎｄ記述：　スロット＝（値$_1$，値$_2$，...，値$_n$）

ｏｒ記述　：スロット＝（値$_1$；値$_2$；...；値$_n$）

これらandやor記述もPrologにおける意味と同じである．例えば、先のand記述は、"スロットの値は値$_1$，値$_2$，...，値$_n$の全てである"として解釈される．

RHSアクションとしては、特別に、WMを直接に操作するmake（WM要素の作成），remove（WM要素の削除），modify（WM要素の修飾）が特別に用意されている．それ以外のアクションはPrologの組み込み述語やユーザ定義の述語を用いる．また、KORE/IEでは、ルールの可読性を高めるために関数表記を可能としている．例えば、次のような第一引数に結果を返す述語定義は、

```
add_one(Result,Number) :- Result is Number + 1.
```

ルール記述において以下のようにあたかも関数として記述可能である。

```
slot = add_one(X)
```

関数表記は、ルール実行時には（予め、関数表記を含むルールが通常のprologプログラムへ変換されているので）元の通常な述語として評価される。

3.2. 協調問題解決機能

第２節で論じたように、KOREの各サブシステムは関係テーブルをコミュニケーションの場として用いることによる協調問題解決機能を実現している．このような協調問題解決機能はKORE/IEにおけるルールベースの間でも適用可能である．この協調機能は、KORE/IE同士の協調機能として見ることができるが、むしろ黒板モデル[Lesser 77]における協調問題解決機能に相当する．つまり、個々のルールベースは知識源と呼ばれる互いに独立な知識ベースであり、KOREにおける関係テーブル（これは、黒板に相当する）における変化に応答することにより、協調的に問題解決を行う．このようなルールベースの独立性は他のルールベースを考慮せずにルールの付加・修正を可能とし、問題領域を階層的に表現する手段を提供する。

KORE/IEにおけるルールベース間協調問題解決のメカニズムは次のように記述できる．KORE/IEでは、ルールベース間の協調動作はPrologシステムに協調させるべきルールベースをKORE/IEのローダを用いて読み込むこと（これをルールのコンパイルと呼ぶ）により達成する．ルールベース名は、この時指定したファイル名から決定される．ルールのコンパイルは、ルールをKOREにおける統一的な形式のPrologプログラムに変換することである．これにより、異なったルールベースにおけるルールは一様なPrologプログラムへと変換され、容易に協調動作が可能になる．このPrologプログラムは、ルールのLHSに対応したLHSプログラム及びRHSに対応したRHSプログラムにより構成される．プログラムはPrologの節として実現される（図４参照）．

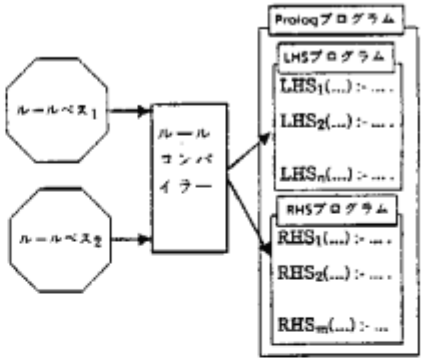

図4. ルールのコンパイル

WMの変化はLHSプログラムに対する質問へと変換され、次に、Prologの反駁メカニズムを直接に用いて認識-行動サイクルにおける照合過程が達成され（この時、LHSにおける変数束縛条件等もチェック・保存される），その副作用としてインスタンシエーションが生成される（4節で詳細に議論する）．インスタンシエーションはルールベース毎に作成され、これにより各々の競合集合が構成される．これら競合集合は、ルールベース固有の競合解消戦略（3.3節参照）を用いて競合解消され、ルールベース毎のRHS

プログラムを起動するための各々一つのインスタンシエーションが選ばれ、RHSプログラムを実行する。この実行も、LHSプログラムと同様に、選ばれたインスタンシエーションをRHSプログラムの質問へと変換されることにより、Prologの反駁メカニズムを用いて直接実行される。現在、KORE/IEではルールベース間の競合（協調時においてルールベース間で同時に適用可能なルールが生じること）解消のための制御機構を特別には提供していない。ルールベースの適用順序はルールベースがユーザによりロードされた順を採用している（もしくは、システム起動時にデータとしてその順番を与えている）。現在、今後の課題として、推論の高速性も考慮したより柔軟なルールベース間の競合解消機構の構築を検討している。

3.3. 競合解消戦略の定義

KORE/IEではルールベース毎に競合解消戦略の定義を可能とすることにより、より柔軟なルールベース間の協調問題解決機能を実現する。競合解消戦略の定義は次の(1)～(3)の条件を組み合わせて指定することにより達成する。(1)インスタンシエーションを選択するときに最新のものを選ぶか、最古の物を選ぶかを指定する。(2)インスタンシエーションのタイムタグを比較する際に、最初に比較したいタイムタグの位置を指定する。(3)(1)と(2)が等しい場合、LHSにおける未定義スロットの数の多寡を指定する。ここで、インスタンシエーションは、①ルール名、及び②そのルールのLHSを満足させたいくつか（一つ以上）のWM要素に対応するタイムタグにより構成される。(1)は、競合解消の判断基準として、このタイムタグの最新性もしくは最古性を指定することである（これにより、認知-行動サイクルの実行に一定の方向性を与えることができる）。(2)は、ルールにおけるLHSパターンの位置を指定し、それに対応するタイムタグを(1)の判断基準として用いる。

具体的には、OPS5におけるMEA戦略やLEX戦略（KORE/IEでも定義済みである）をこの機能を用いて定義可能である。例えば、mea戦略は次のステップを踏んで競合解消を行う。1)ルールの一番先頭のLHSパターンとマッチしたWM要素のタイムタグを比較する。2)1)のタイムタグが同じ場合、次に新しいタイムタグを比較して新しいものを選ぶ。3)一方が選ばれるか、他方のタイムタグがなくなるまでに次々に2)を繰り返す。4)3)で競合解消ができなければ、LHS中の未定義スロットの数の少ないものを選択する（KORE/IEでは、競合解消戦略定義のなかで未定義スロットに言及しなかった場合、デフォルトとしてこの定義が用いられる）。5)4)で決まらなかったら任意のインスタンシエーションが選択される。KORE/IEにおいて、このようなMEA戦略は次のように定義される。

```
define_strategy mea :=
        select latest instantiation
                by comparing first time tag.
```

ここで、下線を施した部分はユーザが記述する所であり、他はキーワードである。下線部は、それぞれ、戦略名、上で述べた(1)の指定（他にoldestを指定できる）、(2)の指定（他に、second, third, fourth, fifth または整数を指定できる）を表している。さらに、未定義スロットに言及した競合解消戦略定義例は次のように記述できる。

```
define_strategy tora :=
        select oldest instantiation
                by comparing second time tag
                        and more constraints.
```

ここで、最後の下線施した部分で未定義スロットの多いものを選ぶことを指定している。

以上のような競合解消戦略定義は、次のように、ルールベースに対してその戦略名を宣言することによりルールベース固有の競合解消戦略として用いられる。

```
?- strategy(<Rule_base>, <Strategy>).
```

ここで、第1引数に具体的なルールベース名、第2引数に具体的な競合解消戦略名を指定する。

4. KORE/IEにおける推論の高速化

4.1. 論理型言語による推論エンジンの概観

Prologを代表とする論理型言語は、その言語内に推論エンジン構築に必用なパターン・マッチング（ユニフィケーション）機構や解の探索のためのバックトラッキング機構を包含しており、また、それ自身、反駁メカニズム[Kowalski 79]に基づく強力な推論エンジンとして利用することができる。一般的に、推論システムを実現するには、ルール解釈実行機構の設計・構築が必用となるが、論理型言語を推論エンジンとして見なすことにより特別なルール解釈実行機構を構築することなしに容易に推論システムを構築することが可能である。この種のシステム(例えば、BUP[Matsumoto 83]など）ではルールを論理型言語のプログラムに変換し、そのプログラムの実行という形で推論システムが構成される。

一方、竹内[竹内 85]はこの種の推論の実行効率とルール・インタープリタとしてのルール解釈実行機構実現による利点（ルールの汎用性, maintenability, readability）を統合するための技法として部分計算技法に基づく推論システム構成手法を提案している。この手法は、ルールをデータとして与えることによりルール解釈実行機構をそのルールに対して特殊化し、推論を論理型言語のプログラムとして直接実行することで論理型言語上に構築した各種インタープリター般に対しての高速化を指向する（図5参照）。

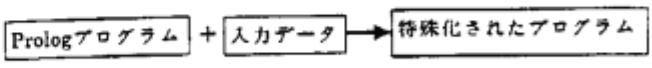

図5. Prologにおける部分計算法概略

Prologは、ユニフィケーションをベースにした計算メカニズムを採用しているので部分計算時に特殊な評価方式（例えば、遅延評価等）は必要がなく、変数の値の有無に関わらず計算できる利点を有している。この部分計算技法による高速性は論理型言語の実行環境である反駁メカニズムの高速性に帰着する。しかしながら、この手法による推論システムは、PS等で代表される推論エンジンの内部メカニズム（例えば、競合解消戦略、WMの検索・更新、認識-行動サイクル等）の特徴を積極的に利用していないので、更に高速化することが可能である。

本研究では以上のような推論システム構築技法を背景にして、論理型言語上で十分に高速な推論エンジンを実現するために、KORE/IEの推論エンジンのメカニズムの特徴を利用し、また論理型言語の反駁メカニズムの高速性を直接利用することにより（KORE/IEでは特別なルール解釈実行機構を実現していないが、ルール解釈実行機構実現による利点を損なうことなしに）、その高速化を実現する。

4.2. KORE/IEにおける推論の高速化技法

KORE/IEにおける推論動作は、OPS5等に代表されるPSと同様に、認識-行動サイクルにより問題解決を達成する。認識-行動サイクルは、(1)照合(Matching)、(2)競合解消(Conflict Resolution)、(3)動作(Action)、より構成され、これら(1)～(3)を繰り返す必要がある。そこで、推論の高速化は、(1)～(3)の各動作を高速化することにより効果的に達成できる。一般に、PSにおいて、この中で最も多くの処理時間が必用とされるのが(1)の照合であり、推論の高速性を左右する本質的な動作である。サイクル毎に全てのルールのLHSパターンとWM要素を照合すると、ルール数やWM要素数が多くなれば、照合に多くの時間が費やされることになり、実用的でない。また、照合時に必要とされる多くの変数情報を管理することも多くの時間が必要とされる原因である。

OPS5では、この様なサイクル毎に照合を行うことを避けるために、Rete Matchアルゴリズム[Forgy 82]を用いている。Rete Matchアルゴリズムは予め全てのルールのＬＨＳをマッチングパターンとしてネットワーク化（データフローネットワークの一種を形成する）しておき、サイクルに関係なく、WMの変化の分だけこのネットワーク（Reteネットワークと呼ばれる）との照合が行われ（ネットワーク内に照合に関する情報（WM要素、変数の束縛等）は全て格納される）、インスタンシエーションの更新を行う。このようなRete Matchアルゴリズムを効果的にインプリメントするにはネットワーク構造を効果的に実現するためのデータ構造（ポインタ等）が必要であり、論理型言語で効率的に構成することは困難である。

そこで、KORE/IEでは、認識-行動サイクルにおける内部メカニズムに着目し、それらメカニズムの高速化を図るために、Prologの利点である(1)高速な反駁メカニズム（Prologにおける基本的な計算メカニズムである）、(2)変数の有無にかかわらず計算する部分計算技法を用いる（図6参照）。

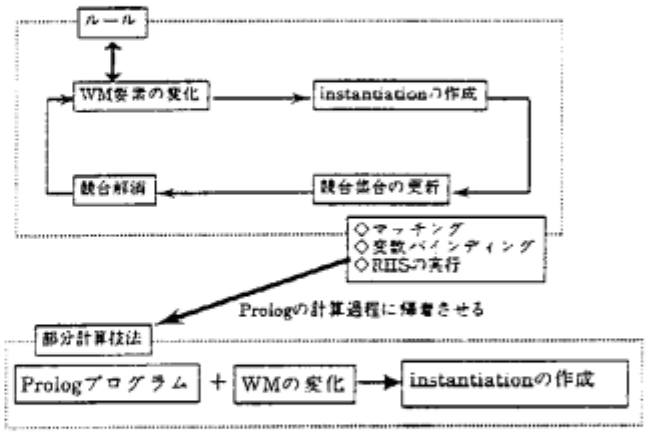

図6. KORE/IEにおける推論メカニズム

図６で示すように、具体的には、KORE/IEにおける推論メカニズムは(1)WMの変化、(2)インスタンシエーションの作成、(3)競合集合の更新、(4)競合解消のサイクルにより構成される。そこで、推論の高速化のために、このサイクルにおける各動作（特に、照合(ﾏｯﾁﾝｸﾞ)、変数束縛、RHSの実行等）をPrologの計算過程（反駁メカニズム）に素直に帰着させ、また、インスタンシエーションの作成にかかわるPrologプログラムの作成を部分計算技法を用いて達成する。具体的には、部分計算技法はルール記述からPrologプログラムを生成する際に用いられ、LHSにおける変数情報を加工してインスタンシエーションを生成するのに都合の良い形式へ変換する（4.3節参照）。この様なPrologにおける利点を導入し、効果的に推論の高速化を実現するためには、Prologシステムの特長を認識して、予め、KORE/IEのルール表現をPrologの最適化されたプログラムへコンパイルする必要がある（KORE/IEをさらに高速化するためには、Prologプログラムのための最適化手法に関連した多くの研究成果（例えば、[沢村 86]）を応用する必要があり、現在、これら研究成果の導入を検討している）。KORE/IEでは、特に、Prologシステムの特長として次の点を考慮している。①節のヘッド探索を高速化するために、ヘッドはヘッドを構成するファンクター名とそのアリティ（引数の数）よりハッシュインデックス化されている。②探索のためのバックトラッキングを有している。

①は、高速化のために、特に、重要であり、これによりReteネットワークにおけるルートノードに相当する機能とその効果を実現できる。このルートノードはReteネットワークの入口であり、WMの変化の分だけをネットワークとの照合を行わせるものである。つまり、KORE/IEではLHSに対応したLHSプログラムの節のヘッドにはLHSパターンのクラ

ス名を用いた名前付をすることにより、Reteネットワークにおけるルートノードに相当する機能を実現している。また、KORE/IEではLHSパターンはひとつのProlog節として実現されるので、Reteネットワークにおけるルートノードから終端ノードまでの一つのパスに含まれる1-入力ノード（スロットに関する照合を行う）及び2-入力（変数束縛のチェックを行う）ノードをLHSプログラムのヘッドとして、一度に表現していることに相当する。さらに、LHSパターン間の束縛変数の無矛盾のチェックはLHSプログラムの本体で行うことにより、Reteネットワークにおける2-入力ノードに相当する機能を実現している（図７参照）。

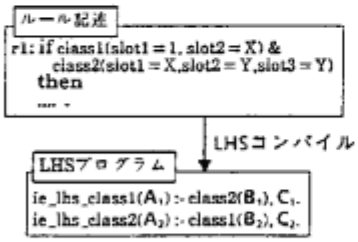

図7. KORE/IEにおけるLHSプログラム

図７において、A_iの部分は具体的なルール名，スロットの値及び変数情報を示し、B_iの部分は他に満足すべきWM要素のパターン（もしくはLHSパターン）を示し、C_iの部分はA_iとB_iより得られた束縛変数の無矛盾をチェックする。

KORE/IEにおける、この様なLHSプログラムの特長は、Reteネットワークにおける照合過程がWM要素をトークンとしたデータフローに帰着されるに対して、KORE/IEでは照合過程がWM要素の変化を質問とした反駁メカニズムにより達成されることである。このことは、KORE/IEにおける照合過程が特別な機構（プログラム）を構築すること無しに、直接、Prologの計算メカニズムである反駁メカニズムにより実現したとに相当し、Prologにおける節のヘッド探索の高速性を十分に取り入れることができる。さらに、これにより、KORE/IEにおける照合過程では、Reteアルゴリズムに潜在する欠点（ルールのネットワーク化に際し、ノード順序が処理効率に影響を与えること）は生じず効率の良いものとなっている。

②は、WMの変化にともない、幾つかのインスタンシエーションを生成する際に使われる制御として用いることができる。例えば、（図７の例を参考にして）クラスclass1に関するWM要素の変化によるインスタンシエーションの生成は次のような質問として実現できる。

?- ie_lhs_class1(...), <インスタンシエーションの作成>, fail.

ここで、例における<インスタンシエーションの作成>はie_lhs_class1(...)の質問により得られた情報をもとにインスタンシエーションを作成することを示している。これにより、WM要素変化に関連したLHSパターンが全てチェックされ、それに伴うインスタンシエーションの生成が特別な制御機構を構築することなしに実現される。

4.3. ルールのコンパイル

KORE/IEでは、ルールを部分計算技法を用いて（なぜなら、ルールには多くの変数が用いられているから）論理型言語のプログラムにコンパイル（もしくは、変換）することにより、推論の高速化を実現する。このコンパイルにおいては、認識-行動サイクルにおける各段階を高速化するために、ルールのLHSとRHSのそれぞれに対応した論理プログラムが生成される。ここで、前者をLHSコンパイル、後者をRHSコンパイルと呼ぶ。LHSコンパイルは、先に述べた認識-行動サイクルにおける照合の高速化のための論理プログラムを生成する。RHSコンパイルは競合解消段階で選ばれたインスタンシエーションをもとに高速に動作（例えば、LHSからの変数の引渡し等を含む）を実行するための論理プログラムを生成する（図８参照）。

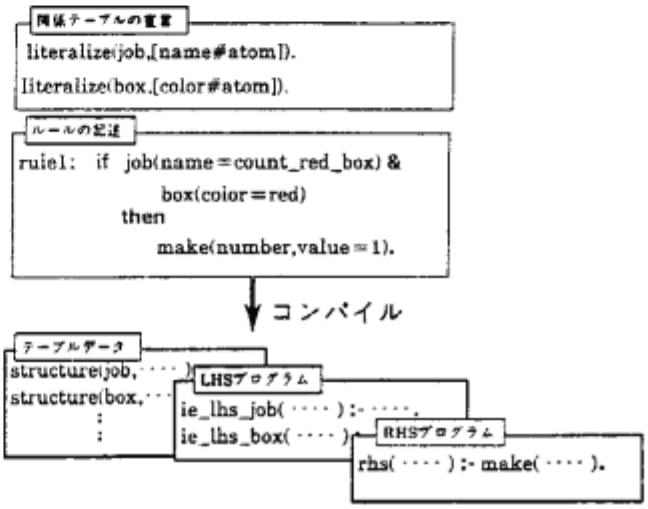

図8. KORE/IEにおけるルールのコンパイル

図８において、関係テーブルの宣言とはルール中に使用されるクラス名やスロット名を宣言することであり、KORE/IEのliteralizeコマンドを用いて実行される。このような宣言を行うことにより、ルールコンパイル後のパターンの標準化を達成する。つまり、これは、LHSパターンにおけるスロットの位置（ユーザにより任意の位置に記述される）をコンパイルすることにより固定化することである。ことによりそれ以後のLHSパターンの処理が高速化される。

例えば、図９で示したKORE/IEにおけるルール記述は、ルールコンパイルにより（図８で示したように）、①literalizeコマンドによるテーブルデータ（図１０参照）、②LHSコンパイルによるLHSプログラム（図１１参照）、③RHSコンパイルによるRHSプログラム（図１２参照）を生成する。

図１０において、データstructureにおける第１、２引数はそれぞれクラス名、テーブル名を示す。第３引数はスロットの数、第４引数はスロット名のリスト、第５引数はスロット名に対応する値のタイプのリストを示している。

KORE/IEでは、便宜のために、図で示すようにLHSパターンで現れるクラス名以外にstartというクラスがデフォルトで定義されている。データpositionはLHSパターンにおけるスロットの位置（literalizeにより宣言される）を保存する。データlhs_classはLHSプログラムを参照する際に用いられるクラス名に対応したLHSプログラムのヘッドのファンクター名を保存する。これらデータは、パターンを標準化するために用いられ、スロット値を参照の際の高速化のための情報となる。

```
literalize(monkey,[at,on,holds]) .

literalize(goal,[status,type,object_name,on,to]) .


'At__Monkey' :if
        goal+  goal(status=active,type=at,
                            object_name=nothing,to=P1) &
        monkey+ monkey(on=floor,at\==P1,holds=nothing)
then
        nl &
        write('Walk to ') &
        write(P1) &
        nl &
        nl &
        modify(monkey,at=P1) &
        modify(goal,status=satisfied).
```

図9. KORE/IEにおけるルール記述例

```
structure(goal,goal,6,[time_tag,status,type,object_name,on,to],[
structure(monkey,monkey,4,[time_tag,at,on,holds],[number,non,nor
structure(start,start,2,[time_tag,order],[number,number]).

position(start,time_tag,number,1).
position(start,order,non,2).
position(monkey,at,non,2).
position(monkey,on,non,3).
position(monkey,holds,non,4).
position(monkey,time_tag,number,1).
position(goal,status,non,2).
position(goal,type,non,3).
position(goal,object_name,non,4).
position(goal,on,non,5).
position(goal,to,non,6).
position(goal,time_tag,number,1).

lhs_class(start,ie_lhs_start).
lhs_class(monkey,ie_lhs_monkey).
lhs_class(goal,ie_lhs_goal).
```

図10. ルールコンパイルによるテーブルデータ

```
ie_lhs_monkey('At__Monkey',r1,[A,B,B3],[C,D3],1,D,A,floor,nothing) :-
        goal(C,active,at,nothing,E,B),
        A\==B.

ie_lhs_goal('At__Monkey',r1,[A,B,B3],[C,D3],1,C,active,at,nothing,E,B) :-
        monkey(D,A,floor,nothing),
        A\==B.
```

図11. ルールコンパイルによるLHSプログラム

図11で示すように、コンパイラーはルールにおけるLHSパターンの数に対応したLHSプログラムを生成する。これにより、4.2節で論じたように、WMの変化対応した分だけのLHSプログラムの呼び出しが可能になる。LHSコンパイルにより生成されたPrologプログラムは照合過程において変数束縛のチェックも同時に行う。例えば、monkeyに関連したLHSパターンに対するLHSプログラム（図11において最初のPrologプログラム）を見てみよう。LHSプログラムのヘッドにおける各引数の対応は図11-1のように示すことができる。

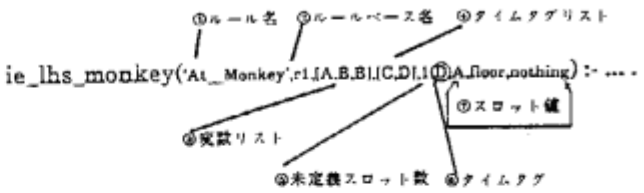

図11-1. LHSプログラムにおけるヘッド

図11-1で示した③の変数リストとは、①のルール名で示されたルールのLHSに含まれる全変数のリストである。例では、変数BがルールのLHSに現れる変数Ｐ１に対応し、変数Ａはルールの２番目のLHSパターンにおけるスロット記述"at\==P1"で用いられる仮の変数である。つまり、このスロット記述の内部表現は"at=（X \== P1）"として実現され、変数Ａはここで現れる仮の変数Xに対応している。④のタイムタグリストとは、LHSパターンと照合が成功したWMのタイムタグのリストである。この場合、このタイムタグリストはLHSプログラムの本体の第１ゴール（つまり、goal(C, ...)であり、WM要素のパターンを表現している）でその呼び出しに成功した際に決定される（この時、LHSパターン全て満足されたことになるので、このルールのLHSは満足されたことになる）。⑤の未定義スロットの数とは、literalizeコマンドで宣言したスロットの中でLHSで用いていないスロットの数を表し、競合解消時の情報となる。⑥のタイムタグとは、LHSプログラムを呼び出すときにシステムにより決定されるWM要素のタイムタグを表す。⑦のスロット値とは、literalizeされたスロットの順に沿ってそれらの値を並べたものである（未定義スロットには仮の変数が置かれる）。図11-1の例では、monkeyクラスのatスロットが未定義なので、その位置に仮の変数Ａが置かれている。また、LHSパターン間の束縛変数のチェックはLHSプログラムの本体で行われる。例では、LHSプログラム本体の最後のゴールである"Ａ \== Ｂ"である。

```
rhs('At__Monkey',r1,[A,B,B],[C,D],E,F,G) :-
        nl,
        write('Walk to '),
        write(B),
        nl,
        nl,
        modify(E,F,D,[at=B],G),
        modify(E,F,C,[status=satisfied],G),
        !.
```

図12. ルールコンパイルによるRHSプログラム

図12におけるリスト"[A,B,B]"、"[C,D]"は、それぞれ、LHSプログラムに現れた変数リストとタイムタグリストであり、その他の変数E,F,GはKOREにおける他のサブシステムと協調する際に用いられるフラグである。

LHSコンパイルされたプログラムは、WMを変化させる動作(make,modify,remove）から呼ばれる。これは、WMを変化させる動作をLHSプログラムに対する質問として実現することにより達成する。具体的には、WM要素を作成するコマンドmakeは、図13のようにPrologで定義される。

```
make(Fact):-
    ①{ Fact =.. [F|Atrs],
       Time_Tag is cputime,
    ②{ structure(F,_,_,C_|ANL],C_|ATL]),
       reforming_make(F,Atrs,ANL,ATL,Reformed),
    ④{ FACT =.. [F,Time_Tag|Reformed],
       assert(FACT),
       add_tuple_number(F),
       lhs_class(F,FF),
    ⑤  Call =.. [FF,
                 Rule_Name,Rule_Base,VL,Instantiation,NUS,
                                          Time_Tag|Reformed],

    ⑤{ (call(Call),
       strategy_rec(Rule_Base,_,Strategy),
       ass(Rule_Base,Rule_Name,Instantiation,NUS,Strategy,VL),
       fail;
       true),
       !.
```

図13. makeコマンドのprolog定義

例えば、makeは次のように用いることにより WM要素を生成する。つまり、その引数としてKOREで用いられるパターンが記述される。

make(気象情報(場所=東京, 気温=13))

図13で示すように、makeは、先ず、①の部分でタイムタグが与えられ（KORE/IEでは、CPUタイムを直接用いている）、②の部分で入力されたパターンを標準化し、③でWMに付加する。そして、④において、①～③で得られた情報に基づいてLHSプログラムに対応した質問を構成し、⑤によりWM要素の作成に応じたインスタンシエーションを作成する。

```
running(N,off,Rule_Base,0) :-
        now_runnig(Rule_Base,running),
        strategy_rec(Rule_Base,_,Strategy),
        retract(cs(Rule_Base,[[Rule_Name1,Time_Tags1,STT1,NUS1,Var_L
    ①   conflict_resolution(Strategy,CS,
                            [Rule_Name1,Time_Tags1,STT1,NUS1,Var_
                            [Rule_Name,Time_Tags,_,_,Var_List],
                            New_CS),
        asserta(cs(Rule_Base,New_CS)),
        ie_to_eden(Rule_Name,Rule_Base,EDEN_Mode),
    ②   rhs(Rule_Name,Rule_Base,Var_List,Time_Tags,off,EDEN_Mode,Time_Ta
        NN is N - 1,
        !,
        running(NN,off,Rule_Base,Hatch_Mode).
```

図14. RHSプログラムの実行

RHSプログラムの実行は、図14で示すように、①での競合解消で得られたインスタンシエーションの情報をもとに、②の部分でRHSプログラムに対する質問（ゴール）として呼び出されることにより達成する。

4.4. 性能評価

元来、PrologはLispに比べプログラムの実行速度が非常に遅いシステムであるとされている。そこで、本性能評価では、先ず、Prologシステムの中でも遅いとされるC-Prolog上のKORE/IEと、Lisp上では最も早いとされるPSであるOPS5との比較をすることにより、KORE/IEの高速性（間接的には、Prologの潜在的な能力）を示すことにする。表2にVAX11/780上でインプリメントしたKORE/IE(C-Prolog(version 1.4)を用いた)とOPS5(Franz Lisp(Opus 38.79)を用いた)の比較を示す。例題はMonkey and Bananas [Brownston 85](27ルール(文献中のテスト・ルールは除いた))である。KORE/IEのルール記述はOPS5のルール記述と一対一に対応づけた（KORE/IEでは、OPS5的なルールプログラミングを表現でき、LHS及びRHS記述を表現的に同様にした）。表2において、OPS5(Compiled)はOPS5インタプリターをLiszt(Franz Lisp Compiler)でコンパイルしたものであり、OPS5(interpreted)はOPS5インタプリターをそのままFranz Lispにコードしたものである。

計算機：VAX11/780
OPS5：Franz Lisp (Opus 38.79)
KORE/IE：C-prolog (version 1.4)

表2

	OPS5 (Compiled)	OPS5 (interpreted)	KORE/IE
ルールコンパイル時間(秒)	4.7	26.9	10.3
ルール実行時間(秒)	1.5	61.5	7.0

表２で示すようにProlog上のKORE/IEはLisp上のOPS5(interpreted)に比べ十分に早く、更に高速なPrologを用いればOPS5(Compiled)に匹敵する高速性が期待される。

計算機：SUN3/52m
KORE/IE：C-prolog (version 1.4)
Quintus Prolog (Release 1.6)

表3

	Quintus (Compiled)	Quintus (interpreted)	C-Prolog
ルールコンパイル時間(秒)	46.1	10.6	6.2
ルール実行時間(秒)	2.0	2.5	4.1

表３は、表２における例題を用いて、C-Prolog及びQuintus Prolog上におけるKORE/IEの性能を比較したものである。表３において、Quintus(interpreted)はKORE/IEのC-Prolog版をコンパイル実行したものであり、Quintus(Compiled)はKORE/IEのルールコンパイルにより得られたPrologプログラムを更にQuintus PrologのPrologコンパイラーを用いてコンパイル実行したものである。つまり、表３におけるQuintus(Compiled)におけるルールコンパイル時間にはKORE/IEにおけるルールコンパイル以外に、Prologコンパイラーによる時間も含まれている。ここで、Prologコンパイラーは、ファイルからの入力が前提とされているので、この時間にはKORE/IEのルールコンパイルで得られたPrologプログラムのファイルへの書き込みのための時間も必要とされ、他のルールコンパイル時間に比べ多くの時間を消費している。

図16は、Quintus(Compiled)とQuintus(interpreted)をルール数を増やしていくことによる性能比較を示している。テストルールは図16上で示すように、WMの要素を次々に更新して行くものである。図16下のグラフ（縦軸はルールの実行時間(CPU時間(秒))，横軸はルール数を示している）で示すようにQuintus(Compiled)はPrologにおける節のヘッド探索における高速性をそのまま実現しており、ルール数に線形比例した理想的な実行速度を達成している。

5. おわりに

本論文では、KORE/IEの主な特長として、(1)KOREにおける推論エンジン機能，(2)ルールベース間の協調問題解決機能，(3)ルールベース毎の競合解消戦略定義機能，(4)高速な推論の実現について議論した。(1)～(3)では、KORE/IEの推論システムとしての柔軟で強力な機能を示した。(4)は特に

重要な課題であり、KORE/IEではPrologシステムの特長をうまく用いることにより、ルールの記述力を犠牲とせずに推論の高速性を実現した。具体的には、論理型言語上で十分に高速な推論エンジンを実現するために、PSとして実現される推論エンジンのメカニズムの特徴及び論理型言語の反駁メカニズムの高速性を利用することにより（ルールのmaintenabilityやreadabilityを損なうことなしに）、その高速化を達成した。このために、KORE/IEでは、予め、KORE/IEのルール表現を部分計算技法を用いてPrologの最適化されたプログラム（推論に適した）へコンパイルし、PSにおける認識-行動サイクルをPrologにおける反駁メカニズムに帰着させた。これにより、Lisp上において最高速と言われるPSであるOPS5に匹敵する高速な推論システムをProlog上で実現した。

なお、本研究は、第五世代コンピュータ・プロジェクトの一環として行われたものである。

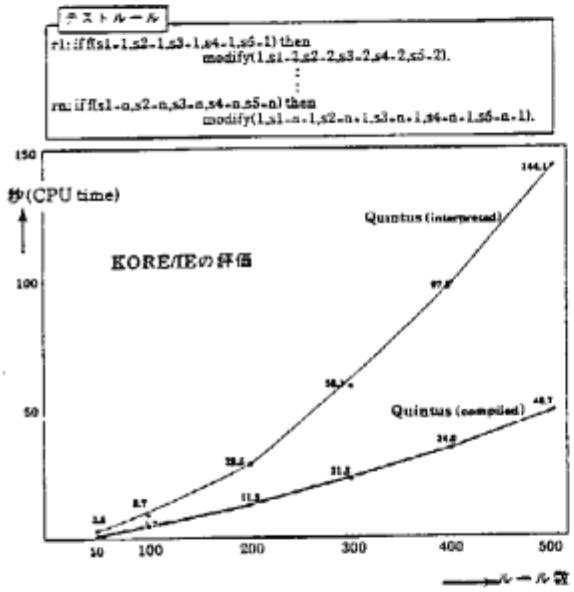

図16. Quintus PrologにおけるKORE/IEの性能比較

謝辞

日頃よりご指導頂く当研究所北川敏男会長ならびに渕本澄所長に感謝いたします。本研究を進めるに当たり、貴重な御意見を頂いた当研究所戸田光彦主任研究員及び片山佳則、平石邦彦研究員に深謝いたします。また、KORE/IEをインプリメントするに当たり、㈱富士通SSLの二神氏と多くの討論を致しました。氏の貢献は大きく、かつ重要な役割を果たされたことに感謝いたします。

参考文献

[Brownston 85] L. Brownston, R. Farell, E. Kant and N. Martin : Programming expert system in OPS5, Addison Wesley (1985).

[Forgy 81] C. L. Forgy : OPS5 User's Manual, CMU CS-81-135, July, (1981).

[Forgy 82] C. L. Forgy: Rete:A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence 19, pp. 17-37 (1982).

[Kowalski 77] R. Kowalski: Logic for Problem Solving, (Elservier North Holland, 1977).

[Lesser 77] V. R. Lesser and L. D. Erman : A retrospective view of the HEARSAY-II architecture, Proc. IJCAI 5, pp. 790-800 (1977).

[Matsumoto 83] Y. Matsumoto et al. : BUP:A Bottom-up Paser Embedded in Prolog, New Generation Computing, Vol. 1, No. 2 (1983).

[沢村 86] 沢村: Prologプログラムの最適化, ICOT Technical Report TR-154 (1986).

[新谷 86] 新谷虎松, 片山佳則, 平石邦彦, 戸田光彦 : 論理型言語によるハイブリッドな問題解決支援環境KOREの設計－知的意思決定支援システムへの接近－, The Logic Programming Conference 86, 予行集pp. 43-50 (1986).

[竹内 85] 竹内, 他 : Prologプログラムの部分計算とメタプログラムの特殊化への応用, The Logic Programming Conference 85, 予行集 pp. 155-165 (1985).