ICOT Technical Memorandum: TM-0655, 0695

TM-0655, 0695

知識工学と人工知能研究会での発表論文

March, 1989

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

TM 0655　知識処理システムのための意味ネット変換規則による
意味解析方法
杉山高弘，阪田全弘(日本電気)，吉田宗宏，西谷泰昭

TM 0695　集合の概念に基づくフレームの拡張
吉良賢治，三石彰純，辻　秀一(三菱)

知識処理システムのための意味ネット変換規則による意味解析方法

杉山高弘　　阪田全弘　　吉田宗宏 *　　西谷泰昭 **
日本電気 (株) ソフトウェア生産技術開発本部
* 日本電気技術情報システム (株)　　** 群馬大学

本稿では、意味ネットワーク上のパターンマッチングに基づいた変換ルールによる意味解析方法を提案する。一般に、知識ベースを構築する際、入力データから生成される表層レベルの意味表現と深層レベルの意味表現とは大きなギャップが存在する。そこで、知識ベースの操作とユーザによる定義が簡単な変換ルール記述言語を提案し、そのルールを用いて変換する。ルールの適用順序を特にきにしなくてもデッドロックや無限ループに陥らない効率的なルール適用制御をする。プログラム自動生成システムの自然言語による仕様からプログラムを生成する意味解析部に、この方法を適用した例をあわせて示す。

"A Semantic Analysis Method Using Transformation Rules on a Semantic Network for Knowledge-Based Systems."

by Takahiro SUGIYAMA,　Masahiro SAKATA,　Munehiro YOSHIDA * and Yasuaki NISHITANI **

Software Engineering Development Laboratory, NEC Corporation
(11-5, Shibaura 2-chome, Minato-ku, Tokyo 108, Japan)
* NEC Scientific Information System Development Ltd.
(13-11, Shibaura 2-chome, Minato-ku, Tokyo 108, Japan)
** GUNMA University
(1-5-1, Tenjin-cho, Kiryu, Gunma 376, Japan)

This paper proposes a semantic analysis method using transformation rules which are based on a pattern matching with a semantic network. There is a gap between an initial semantic network generated from input data, and a semantic one representing a profound knowledge, When a knowledge database is constructed, To solve it, we propose transformation rule description language, which allows users to manipulate their knowledge base and to define a rule. We also describe an automatic program generation system as an application example of this method.

1．はじめに

エキスパートシステムに代表されるような知識データベースを活用するシステムが増えるに従い、知識表現処理機構の汎用化の必要性が強まる。つまり、知識表現する対象分野に依存しない表現形式と知識表現自身を書き換えるメタ知識による推論システムである。これまで開発されてきたものに、プロダクションシステム、フレーム、意味ネットワーク、述語論理に基づく知識表現システムなどがよく知られている。

フレームシステムの代表として[Minsky75]がある。柔軟かつ強力な構造表現が可能であるが、その上に推論系を作ることは面倒で大幅な性能向上が難しいとされている。ＣＤ理論[Schank75]を用いた知識表現システム[Lehnert&Wilks79]では、興味深い結果を出しているが、入力データの形式と最終的に得たい深層概念レベルの意味表現とは、大きなギャップがあり、どの分野にでも簡単に応用できるわけではない。このギャップを埋める工夫として、状況意味論[Barwise&Pery83]のコンストレインツによって意味表現を順次書き換え最終的に必要とする概念表現を得る方法があるが、関係論理に基づく意味表現は複雑で柔軟な構造表現とはあまり言えない。そこで柔軟かつ強力な構造表現の上に性能のよい推論系を取り入れることが課題となる。

本稿では、上記課題の解決を目指して、フレーム表現と効率的なルール適用を考慮したプロダクションシステムの融合アプローチを述べる。柔軟な構造表現のフレーム方式でクラス（意味モデル）を定義し、そのインスタンスとして意味ネットワークを構成する。これにより、意味表現したい分野への意味モデルのカスタマイズを容易にし、移植性を高める。推論システムは、意味ネットワーク上でネットワークパターンマッチング形式のプロダクションシステムを用いることによって、表層レベルの意味表現と深層概念レベルの意味表現とのギャップを埋めることができる。このとき、フレーム表現のクラスをMモデル、意味ネットワークをMネット、そしてプロダクションシステムの規則を変換ルールと呼ぶ。

また、プロダクションシステムの性能向上のために、変換ルールのユーザによる定義のしやすさと効率のよいルール適用をする。そこで、可読性が高く定義しやすい変換ルール記述言語を提案する。効率のよいルール適用においては、データフローモデルのルール制御をする。以下にそれらの特徴をあげる。

[変換ルール記述言語]

- 知識表現へのアクセス、探索等ををインタプリタに任せ、変換に本当に必要な部分だけを形式的に抽象レベルで記述できるので、知識表現の操作性がよく、変換ルールの定義が簡単
- 意味解析の知識をプログラムから独立して形式的にルールベース化が可能、ルールの体系化・再構成も容易
- 変換ルールの可読性が高いため、ルール定義者以外の人でも保守や拡張可能
- 変換ルール定義構造エディタ

操作対象のネットワークの変換イメージを構造エディタの図形描画によって簡略的に変換ルールを定義する。構造エディタは、ルール定義者に対して描画の際の基本図形を用意したり、各基本図形の名前付けや内部記述の構造的管理支援をする。定義されたイメージ的描画から形式的な変換ルール記述言語に自動変換する。

[ルール適用制御]

- データフロー方式のルール制御

Ｍノードがデータとしてランダムに取り出されルール制御システムを流れていくことによって対応するルールが起動される。ルールのパターンマッチ部では、バインディングするＭノードを順次取り出す。一度取り出されたノードは変換実行が終わるまで、他のルールからのアクセスを禁止する。

- ルール適用順序

効率的なルール適用順序として考えられるのが、一度マッチングが失敗したルールを再度同じノードに適用しないことである。失敗の原因となったノードに書換が生じないかぎりルールの再適用を中止する。これを実現するためにfail-waitテーブルとtrapテーブルを導入する。さらに、次に適用するルールを指定したほうがより、効率よく変換実行がいく場合は、ランダムなルール適用制御の枠組みを壊すことなく逐語的ルール適用の機能をＭモデルのデーモンをうまく使うことによって実現する。

最後にプログラミング自動生成システムの意味解析に本方式を採用することによって有効性を実証する。

2．ネットワーク記述言語ＮＲＬ

本ネットワーク変換ルールシステムの操作対象となるネットワーク記述言語ＮＲＬについて説明する。ＮＲＬは、基本的にフレームシステム[Minsky75]の形式をとっており、基本構成要素としてノードとアークからなる。各ノードの内部には、値を設定できるスロットが存在する。図2.1で示すように、ノードは、クラスとしてのＭモデルノードとそのインスタンスのＭネットノードから構成されている。Ｍモデルは、概念レベルでの意味表現の基本構成要素であり、ＭネットはＭモデルのインスタンスとして相互にアークによって結合しネットワークを構成する。Ｍモデルのインスタンスとしてもう一つＮネットを用意する。これは、入力が特に自然言語のとき、構文解析結果をネットワーク形式で表現したものである。ＮネットとＭネットを区別するのは、入力データの分野や形式に捕らわれず本意味解析方式を独立させることに

や形式に捕らわれず本意味解析方式を独立させることによって複数のアプリケーションシステムへのポータビリティを高める。ＮＲＬは以下の特徴を有する。

・上位ノードからの継承アクセス

継承スロットによりスロット値を上位ノードで一括管理することによって記述量の減少がはかれる。必要のないスロット値を常にノードに保持しておく必要がない。

・ノード接続時の自動チェック機能

ＮＲＬのノード接続関数は、アークスロットに記述されているクラス名と接続しようとする子ノードのクラス間に共通部分が存在すれば接続を許可するが、存在しなければ何も実行しない。

・デーモン関数の起動

ネットワークのノードをアクセスした際に、必要に応じてデーモンが自動的に起動されるように、対応するＭモデルにデーモン関数を記述できる。これは、ＮＲＬを単なる意味表現形式から能動的な処理も実行できる記述にする。

・曖昧さの表現

解析結果の途中で意味が複数存在し現時点では一意に解釈が不可能なとき、Ｍネットは一つのアークスロットに複数のノードを接続することによって曖昧さを表現する。意味解析時に適当なフェーズで一意に決定される。

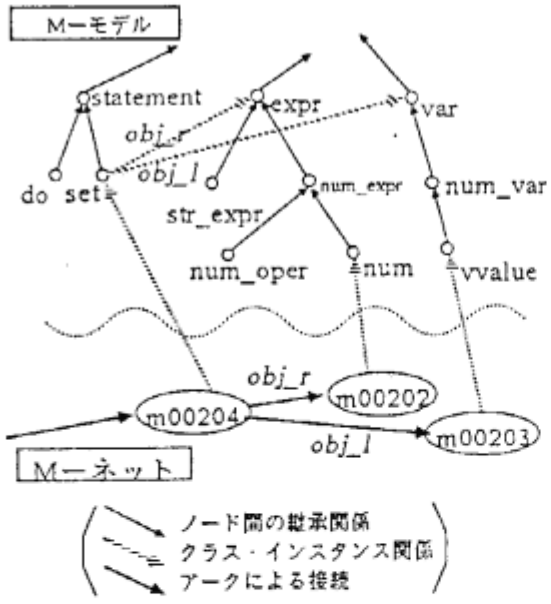

図2.1　　ＭモデルとＭネット

図2.2で示すようにＭモデルは自己スロット、継承スロット、アークスロット群からなる。自己スロットは、自分自身のノード内に常に値を保持している。継承スロットは、ノード内にスロット値が設定されていなければ、上位クラスのノードから値を継承して取ってくる。アークスロットは、子ノードの接続可能なクラスを記述する。さらに、継承スロットとアークスロットには、facetによりスロット値へのアクセスが生じたときに自動的に起動するデーモン関数を記述できる。以下にfacetの種類と、内容を示す。

(1)　Value:

スロットの値そのものを設定する。

(2)　ValueClass:

スロットに設定できる値のクラスを記述する。

(3)　IfNeeded:

スロットに値が設定されていないとき、値を得るために実行されるデーモン関数名を記述する。

(4)　IfAdded:

スロットに値が設定されたとき実行されるデーモン関数名を記述する。

(5)　IfRemoved:

スロットから値が削除されたとき実行されるデーモン関数名を記述する。

Ｍモデル記述例

```
(<MModelname>
   (InstOf: Muode1)
   (Ako:    <UpperMModelName>)
   (<SelfSlotName> <Value>)
   ...
   (InhSlots:
     (<InhSlotName> (Value: <Value>)
                    (ValueClass: <Value>)
                    (IfNeeded: <Daemon>)
                    (IfAdded: <Daemon>)
     ...)           (IfRemoved: <Daemon>)...)
   (ArcSlots:
     (<ArcSlotName> (Value: <Value>)
                    (ValueClass: <Value>)
                    (IfNeeded: <Daemon>)
                    (IfAdded: <Daemon>)
     ...))          (IfRemoved: <Daemon>)...)
```

図2.2　ＮＲＬのＭモデルの一般形式

このＭモデルのインスタンスのＭネットを図2.3に示す。Ｍネットの主なスロットについて内容を示す。

(1)InstOf:スロット
　ノードがどのＭモデルのインスタンスか示す。
(2)Reverse:スロット
　ノードの親ノード番号を保持する。
(3)Akos:スロット
　継承クラス名を保持する。
(4)Rule:スロット
　変換ルールの先頭がこのノードクラスであるルール名をすべて保持する。
(5)FailWaitTB:スロット
　FailWaitテーブルを保持する。
(6)TrapTB:スロット
　Trapテーブルを保持する。

```
(m00002
  (InstOf:     (Set))          ←自己スロット
  (Reverse:    (m00001)))
  (Akos:       (Set Expr))     ←継承スロット
  (Rule:       (SetLpara SetRpara SetLRpara)
  (FailWaitTB: <FailWaitTable>)
```

```
(TrapTB:     <TrapTable>)
(ObjL:       (m00003))      ←アークスロット
(ObjR:       (m00004)))
```

図2.3 Mネットノードの内部表現

3. ネットワーク変換ルール記述

3.1 変換ルール内部記述

(1)パターンマッチ(IF)部

変換ルールのIF部では、ルールに記述されているアークスロットにそってネットワークを順次探索し、変換ルール中のスロット名や定数項とパターンマッチングしながら$変数にふさわしいスロット値をペアにしてバインディング情報を生成する。このとき、大きく2つに分けて、ネットワーク中のノードと$変数をバインディングする［ノードに関するパターンマッチング］とネットワークの探索を進める［スロットに関するパターンマッチング］が存在する。

［ノードに関するパターンマッチング］

①ノードのパターンマッチング

(~node <$変数> <ｽﾛｯﾄﾊﾟﾀｰﾝ>...)

ネットワークを探索し、現在たどりついたノードの集合の中から、それぞれのノードのスロットに対して、スロットパターンがすべてパターンマッチ成功したノードのみパターンマッチを成功とする。このとき、$変数名とパターンマッチが成功したすべてのノードをペアにしてバインディング情報に付け加える。

②ノードのorパターンマッチング

(~node <$変数> (~or <ｽﾛｯﾄﾊﾟﾀｰﾝ>...))

ネットワークから取り出したノード集合の中から、それぞれのノードのスロットに対して、スロットパターンが一つでもパターンマッチ成功したら、orパターンマッチを成功とする。このとき、$変数名とorパターンマッチが成功したすべてのノードをペアにしてバインディング情報に付け加える。

③ノードの否定パターンマッチング

(~node <$変数> (~not <ｽﾛｯﾄﾊﾟﾀｰﾝ>...))

ネットワークから取り出したノード集合の中から、それぞれのノードのスロットに対して、スロットパターンのうちどれか一つでもパターンマッチが失敗すれば、$変数名とパターンマッチが失敗したすべてのノードをペアにしてバインディング情報に付け加える。

［スロットに関するパターンマッチング］

スロットパターンはExistパターン、Auxパターン、NonExistパターンからなる。

④スロットのExistパターンマッチング

(~E <ｽﾛｯﾄ値ﾊﾟﾀｰﾝ>...)

すべてのスロット値パターンがパターンマッチ成功したとき、Existパターンのパターンマッチを成功とする。

⑤スロットのAuxパターンマッチング

(~A <ｽﾛｯﾄ値ﾊﾟﾀｰﾝ>...)

スロット値パターンがパターンマッチ成功するか、スロット値パターンに記述されているスロットが存在していなければ、Auxパターンマッチを成功とする。

⑥スロットのNonExistパターンマッチング

(~N (ｽﾛｯﾄ名 ...))

ルールに記述されているスロット名が現在のノードに一つも存在していなければ、NonExistパターンマッチを成功とする。

⑦スロット値パターンのパターンマッチング

```
<ｽﾛｯﾄ値ﾊﾟﾀｰﾝ> ::= (ｱｰｸｽﾛｯﾄ名 <ﾉｰﾄﾞﾊﾟﾀｰﾝ>) |
                   (ｽﾛｯﾄ名 ｽﾛｯﾄ値)              |
                   (ｽﾛｯﾄ名 (~elem <$変数> ｱﾄﾑ))
```

スロット値パターンに記述されているスロットがアークスロットのとき、現在のノードからアークスロットをたどった先のノード集合からノードパターンのパターンマッチングを繰り返し実行し成功したらスロット値パターンのパターンマッチを成功とする。スロットがアークスロットでないとき、現在のノードからそのスロットに保持されているスロット内容を抽出し、その値がルールに記述されているスロット値と一致するか、アトムが抽出したスロット値の要素(~elem)であればスロット値パターンのパターンマッチを成功とする。

(2)書換実行(THEN)部

変換ルールのTHEN部では、ルールと生成されたバインディング情報からネットワークを書き換える。ルールの書換実行パターンは、生成、置換、操作、関数実行パターンからなる。既に、ネットワークが存在している部分をそのまま保持しつつ、変更が生じる部分のみを記述すればよい。そのために必要なルール記述を以下に示す。

①生成パターン

(~Create (<$変数> <ﾓﾃﾞﾙ名>)...)

新規にノードを生成する。新しく生成されたノード番号を$変数にバインドしそのノードのInst0fスロットにMモデル名を設定する

②置換パターン

(~Replace <$変数1> <$変数2>)

$変数1にバインドされているノードの親ノードからそのノードを削除し、$変数2にバインドされているノードに置き換える。

③操作パターン

```
(~Nodes (<$変数>
          (<ｽﾛｯﾄ名> <ｽﾛｯﾄ値操作ﾊﾟﾀｰﾝ>)...)...)
```

指定したノードのスロットの値以下のオペレーションによって操作する。

<スロット値操作ﾊﾟﾀｰﾝ> ::=
　　<+ﾊﾟﾀｰﾝ> | <++ﾊﾟﾀｰﾝ>
　| <-ﾊﾟﾀｰﾝ> | <--ﾊﾟﾀｰﾝ>
　| </ﾊﾟﾀｰﾝ> | <-All ﾊﾟﾀｰﾝ>

＋＋、－－パターンは、アークスロットに値を設定、削除、置換するときに用いられる。

- ＋パターン　(+ <$変数>･･･)
  $変数にバインディングされている値を指定されたスロットに設定する。
- －パターン　(- $変数･･･)
  $変数にバインディングされている値を指定されたスロットから削除する。
- ／パターン　(/ <$変数1> <$変数2>)
  指定されたスロットから直接ｐｕｔされた内容と左の$変数の内容が一致すれば、右の$変数の内容に置き換える。
- ＋＋パターン　(++ <$変数>･･･)
  $変数にバインディングされているノードを指定されたアークスロットに設定する。
- －－パターン　(-- $変数･･･)
  $変数にバインディングされているノードを指定されたアークスロットから削除する。
- －ａｌｌパターン　(-all)
  指定されたスロット以下をすべて削除する。

④関数実行パターン

(~Function <lisp関数名> <引数>...)
　　定義されたｌｉｓｐ関数に引数を適用して実行する

□

```
<変換ﾙｰﾙ> ::= ((IF (Pattern <ﾉｰﾄﾞﾊﾟﾀｰﾝ>)
                (THEN <実行ﾊﾟﾀｰﾝ>))
 [パターンマッチ(IF)部]
<ﾉｰﾄﾞﾊﾟﾀｰﾝ> ::=
  (~node <$変数> <ｽﾛｯﾄﾊﾟﾀｰﾝ>...)
  (~node <$変数> (~or <ｽﾛｯﾄﾊﾟﾀｰﾝ>...))
  (~node <$変数> (~not <ｽﾛｯﾄﾊﾟﾀｰﾝ>...))
<ｽﾛｯﾄﾊﾟﾀｰﾝ> ::=
  (~E <ｽﾛｯﾄ値ﾊﾟﾀｰﾝ>...) |
  (~A <ｽﾛｯﾄ値ﾊﾟﾀｰﾝ>...) |
  (~N (<ｽﾛｯﾄ名>...))
<ｽﾛｯﾄ値ﾊﾟﾀｰﾝ> ::=  (ｱｰｸｽﾛｯﾄ名 <ﾉｰﾄﾞﾊﾟﾀｰﾝ>) |
                (ｽﾛｯﾄ名 ｽﾛｯﾄ値)                |
                (ｽﾛｯﾄ名 (~elem <$変数> ｱﾄﾑ))
 [実行(THEN)部]
<実行ﾊﾟﾀｰﾝ> ::=
  (~Create (<$変数> <ﾒﾃﾞﾙ名>)...)   |
  (~Replace <$変数1> <$変数2>)      |
  (~Function <lisp関数名> <引数>...) |
  <操作ﾊﾟﾀｰﾝ>
<操作ﾊﾟﾀｰﾝ> ::=
  (~Nodes (<$変数>
          (<ｽﾛｯﾄ名> <ｽﾛｯﾄ値操作ﾊﾟﾀｰﾝ>...)...)...)
<ｽﾛｯﾄ値操作ﾊﾟﾀｰﾝ> ::= (+ <$変数>...)
                      (- <$変数>...)
                      (/  <$変数1>  <$変数2>)
                      (++ <$変数>...)
                      (-- <$変数>...)
                      (-all)
<関数実行ﾊﾟﾀｰﾝ> ::=
       (~Function <lisp関数名> <引数>...)
```

図3.1　変換ルールシンタクス

3.2　変換ルール定義構造エディタ

変換ルール内部記述を定義する際に、ネットワークのイメージを重視した定義方法を支援する変換ルール定義構造エディタを用いる。本エディタは、［暦本他89］のアプリケーションシステムとして構築されたもので、UNIXワークステーション上、X-Window Version 11 Release 2のクライアントプロセスとして動作する。図3.2で示すようにネットワークのパターンマッチと書換え操作をノードとアークを疑似的にイメージで表現している基本図形によって描画する。構造エディタは、IFpartとTHEN Partの定義部からなる。

IFpartでは、パターンマッチングさせるネットワークの疑似構造を作図するための支援をする。基本図形のボックスによってネットワークのノードを表し、ボックスとボックスを結ぶ有向辺によってアークを表す。このとき、ボックスの左上に記述されている変数はネットワーク中のノードIDがバインドされるノード変数を、ボックス内の記述はそのノードについてのスロット情報を、有向辺上に記述されているラベルはアークスロット名をそれぞれ示している。各スロットのマッチング条件として以下の記法を用いる。

エディタ内記述	形式的ルール定義
(ｽﾛｯﾄ名:　内容)	→　(~E　(ｽﾛｯﾄ名:　内容))
[ｽﾛｯﾄ名:　内容]	→　(~A　(ｽﾛｯﾄ名:　内容))
(ｽﾛｯﾄ名:　Nothing)	→　(~N　(ｽﾛｯﾄ名:))
(not (ｽﾛｯﾄ名:　内容)...)	→　(~not(ｽﾛｯﾄ名:内容)...)

THENpartでは、IFpartと同様な定義方法に付け加えて、スロット名とスロット内容の間に値をセットする時は＋を削除する時は－をそしてスロット内容をすべて削除するときは-allを挿入する。ただし、既にネットワークとして存在する構造を保存しつつ新たに書換が生じる部分だけ付加、削除、置換等の書換操作を記述する。

構造エディタは、上記の方法で定義されたノードとアークのイメージから、詳細な内部記述形式に自動的に変換する。

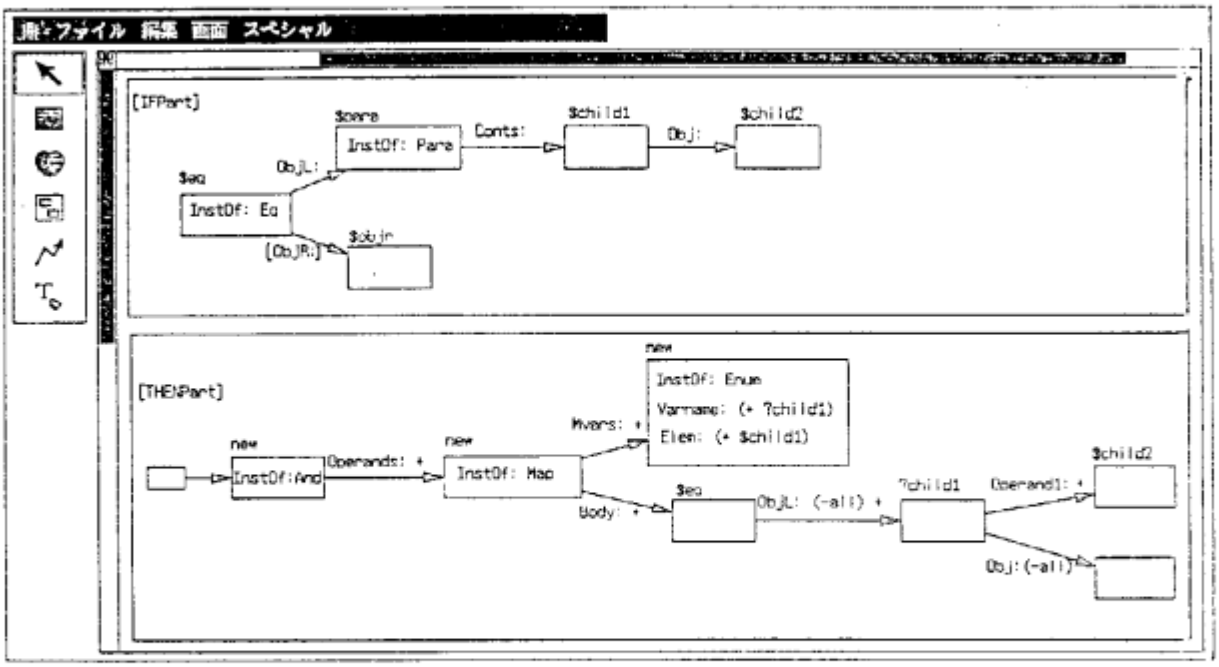

図3.2　変換ルール登録画面

4．変換ルールインタプリタ

4.1　変換ルール制御

変換ルールインタプリタは、変換ルールのパターンマッチ部の先頭のノードから順次たどって、ルールに記述されているスロット内容がマッチしたノードIDをノード変数にバインドし、パターンマッチ部がすべて正しくマッチしたら、書換実行部の記述とパターンマッチ部でバインドしたバインディング情報（図4.1参照）をもとにネットワークを書き換える。（図4.2参照）本方式の特徴を以下に示す。

- 任意のMノードに対してルールを適用し変換を実行でき、制御情報は各Mノード内に完全に独立して記述できるので並行処理への拡張が容易である。。
- 1度パターンマッチが失敗したルールを状況が変化しないまま繰り返し適用しない。
- MモデルのIfNeededデーモンと組み合わせることによって、制御方式を変更することなく連鎖的にルールを実行し、パターンマッチの失敗の回数を減らすことによって効率的なルールの適用が可能。

□

<ﾊﾞｲﾝﾃﾞｨﾝｸﾞﾘｽﾄ> ::= (<$変数名> <ﾉｰﾄﾞﾘｽﾄ>...)
<ﾉｰﾄﾞﾘｽﾄ> ::= (<ﾉｰﾄﾞ番号>) |
(<ﾉｰﾄﾞ番号> <ﾊﾞｲﾝﾃﾞｨﾝｸﾞﾘｽﾄ>...)

上記シンタックスでノードリストの並列は、$変数に同時に複数のノードがバインドすることを示し、バインディングリストの並列は、上位ノードから別々のアークで接続されているノードのバインドを示す。

図4.1　バインディングリストの形式

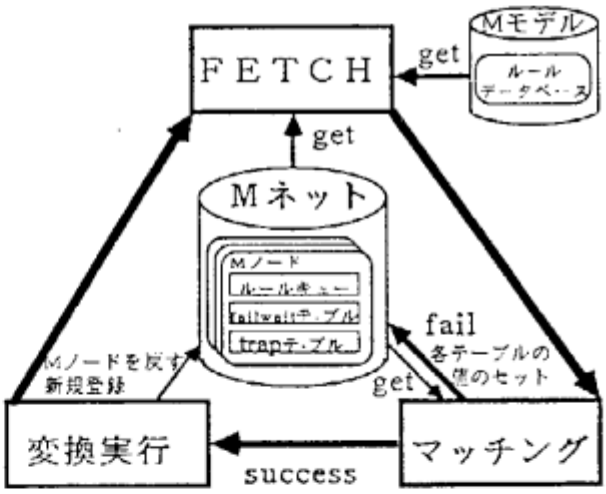

図4.2　変換ルール制御図

4.2　trapテーブルとfail-waitテーブル

trapテーブルは、各MノードのTrap:スロットに保持されている。どのMノードのどのルールがそのMノードが書き変わったかどうかを監視しているかを教えるテーブルである。図4.3のようにノード名とルール番号をペアにして管理する。自分自身のノードにルールが適用され書換が生じたとき、trapテーブルに登録されているすべてのノードのfail-waitテーブルを探索し、その中のふさわしいルール番号のFLAGを"go"にする。

ノード	ルール
node1	SetLParaRule
node2	EqNtomRule

図4.3 trapテーブル

fail-waitテーブルは、各MノードのFailWait:スロットに保持されている。fail-waitテーブルを保持しているノードから現在どのルールが発火可能かを教える。図4.4のようにルール番号と" wait" または"go"のFLAGをペアにして管理する。1度ノードに適用して失敗したルールを"wait"モードで登録する。一度失敗したネットワークの状態に変化が起こったとき再適用の可能性があることを示すためにFLAGを"go"にする。

ルール	FLAG
SetLParaRule	"go"
SetRParaRule	"wait"

図4.4 fail-waitテーブル

4.3 インタプリタ

インタプリタは以下の手順にしたがって変換操作を制御する。

[初期化]

Mネットデータベースの各Mノードのfail-waitテーブルとtrapテーブルをクリアする。各Mノードに対応するMモデル中に記述されているルールをすべて取り出し、Mノードのルールキュースロットにセットする。

[step1]

FETCH部で、MネットデータベースからMネットを構成しているMノードを任意に取り出す。このとき、すべてのMノードのルールキューが空で、fail-waitテーブルのルールが"wait"のとき処理を終了する。

[step2]

取り出されたMノードのルールキューの先頭のルールから適用する。このときルールキューが空ならば、Mノードのfail-waitテーブルに記述されているルールの中でフラッグが"go"のものを取り出して適用する。このfail-waitテーブルから、取り出したルール番号を削除する。

[step3]

マッチング部では変換ルールのIF部の記述にしたがってパターンマッチする。パターンマッチが成功したら[step4]へ。失敗したら[step6]へ。

[step4]

パターンマッチにより作成されたバインディング情報とルールの書換実行部の記述からネットワークを書き換える。適用したルールをルールキューの最後部に付ける。現在のノードが保持しているtrapテーブルに登録されている各ノードのfail-waitテーブルに登録されているふさわしいルール番号のFLAGを"go"にする。そしてtrapテーブルを空にする。

[step5]

新しくできたノードと書き換えが終わったもとのノードをMネットデータベースに戻す。不必要になったノードは廃棄する。[step1]に戻る。

[step6]

[step3]で失敗する原因となったノードのtrapテーブルに[step1]でFETCHされたノード名と適用して失敗したルール番号をペアで登録する。(図4.5参照)さらに、FETCHされたノードのfail-waitテーブルに失敗したルール番号と"wait"モードを登録し、ノードをMネットデータベースに戻す。[step1]へ。

□

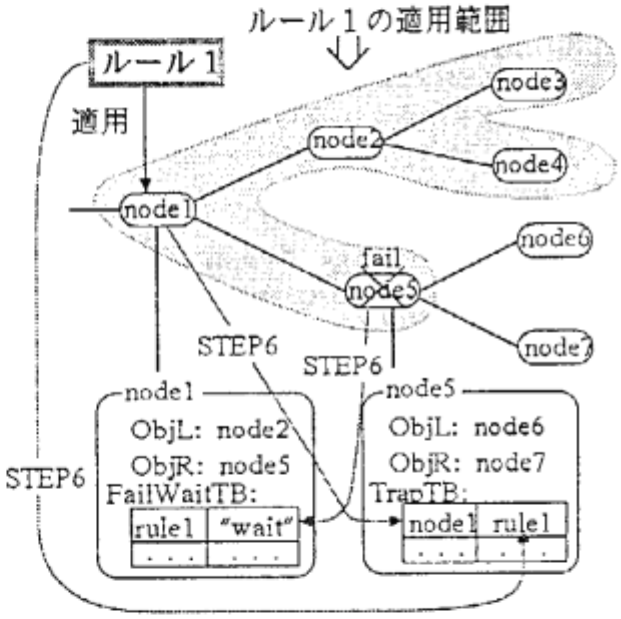

図4.5 テーブル設定

4.4 変換ルール制御の拡張

現在の制御方式ではFETCH部で1つのノードをとってきてそのノードの処理が終わるまで次のノードの処理は行なわないので、Mノードが複数FETCHされることによって処理がデッドロック状態になることはない。並行処理に拡張する場合は、このデドロックの可能性が生じる。そこで、FETCH部では変換ルールの処理の性質を考慮して、ネットワークのリーフ部分からボトムアップ的にノードを取り出す仕組みを組入れる必要がある。

パターンマッチの失敗の回数を減らすことによって効率的なルールの適用を可能とするために、ルールの適用

順序を制限したいという要求が生じる。ランダムにルールを適用している制御方式に適用順序を制限する仕組みを付加することは、一般的に独立した分散制御に全体をコントロールする制御を付加することになり、望ましくない。そのために、ここではMモデルのIfNeededデーモンにふさわしいルールを実行する関数を記述することによって、制御方式を変更することなく連鎖的にルールを実行できる。

親のノードについて直下の子ノードがすべて書き換えられてから親のノードが書き換えられなければならず、ネットワークのすべてのノードについてボトムアップに処理が実行されなければならない場合、ランダムにルールを適用して行くよりもIfNeeded:スロットのデーモン関数を併用したほうが無駄なパターンマッチの回数が約1／2に減少できる。

5．集合演算Ｍａｐ表現の導入

5.1 Ｍａｐノードの定義

ＮＲＬでは、複数のノード集合に同一の変換処理を一括して適用する要求が生じる。そこで、ネットワークの共通部分をまとめて保持し、複数のノード集合に適用するMap表現を導入する。Map表現は、以下のようにMネット上でMapノードとして表現され、ネットワークについてのマクロとなっている。

Ｍａｐノード

ネットワークの共通部分をまとめて表すノード。Enumノード、または、Intervalノードによって一時変数を定義し、Body:スロットに共通なネットワークを記述する。一般的な構造を以下に示す。

```
(Map (Mvars: (Enum (Varname: <?変数>...)
                   (Elem: <binding-list>))
             ...
             (Interval (Varname: <var2>)
                       (FromTo: <lexpr2>))
             ...)
     (Body: <network(<?変数>...)>))
```

?変数は、Ｍａｐマクロ展開時に対応するバインディング変数（$変数）にバインドされている要素を先頭から一つずつ割り当てる制御変数。<network(<?変数>...)>は、?変数を含む変換ルールの書換実行部を記述する。

Ｅｎｕｍノード

Varname:スロットとElem:スロットをもつ。Varname:スロットには、Ｍａｐ表現Body:スロット内部で用いられる一時変数名が代入される。Elem:スロットは、Varname:スロットに代入した一時変数で始まるバインディングリストが代入される。

Ｉｎｔｅｒｖａｌノード

Varname:スロットとFromTo:スロットを持つ。Varname:スロットは、Ｍａｐ表現内部で用いられる一時配列型変数名が定義される。FromTo:スロットは、配列引数の始まりと終わりを表現する。繰り返し演算を表現できる。

5.2 Ｍａｐマクロ展開機能

Ｍａｐノードによってネットワークの共通部分をまとめた表現を適当なタイミングで展開しなければならない。共通部分は、ネットワークを作成するルールとして表現されている。そのルールは、Ｅｎｕｍノード、または、ＩｎｔｅｒｖａｌノードのBverName:スロットで定義された一時変数を用いて記述されている。ルール展開時に、Elem:スロット、または、FromTo:スロットで指定されている各要素を一つづつ取り出し、一時変数に代入し、繰り返し展開する。以下にフレームネットワークとそれを変換する変換ルールを示し、それらから生成されるバインディングリストと実際にマクロ展開機能によって展開される。Map表現の例を図5.1にしめす。

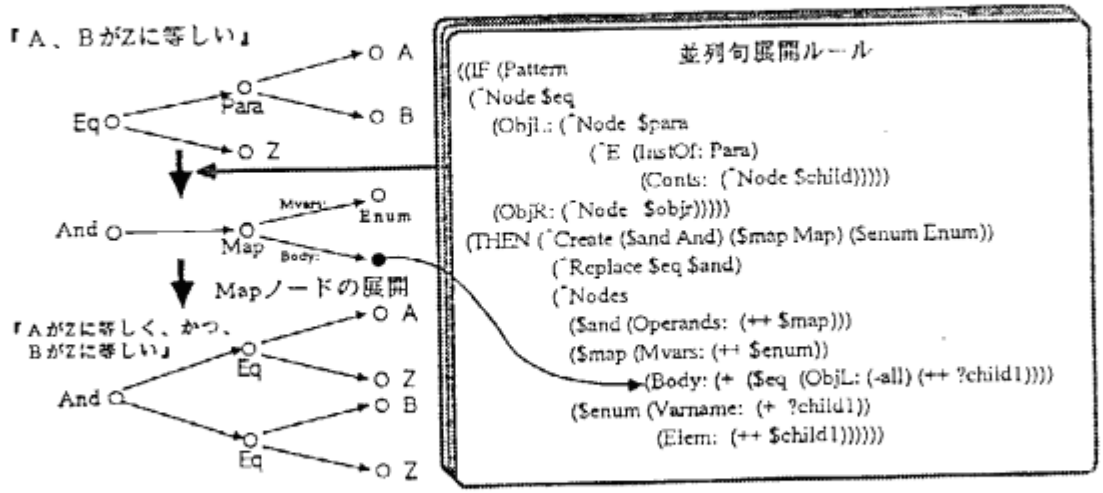

図5.1 Ｍａｐ表現

6．知識処理システムへの応用

6.1 プログラム自動生成システムへの応用

プログラム自動生成システム［岩元他87］［和田他88］は、自然言語の仕様を構文解析して初期の意味表現ネットワークのNネットを生成する。自然言語の仕様からプログラムへ詳細化する際、プログラミングの知識をルール化し、本意味解析方法を導入する。現在、COBOLプログラムの出力を目的としており、自然言語とプログラムの意味を表現する基本要素をMモデルとしてあらかじめ登録する。自然言語の仕様をそのまま表現しているNネットから、変換ルールによりまず初期のMネットへ変換（図6.1参照）し、プログラミング知識を表現した変換ルールによって逐次変換し最終的なMネットをえる。開発は、この変換過程は大きく分けて以下の通りである。

（1）Mネットの生成
（2）並列句の展開
（3）繰り返しループの生成
（4）省略語の補完
（5）不用な入れ子ブロックの消去

ここでは、Mネットの生成と省略語の補完について例を示す。

6.1.1 Mネットの生成

自然言語の仕様を構文解析してえられたNネットから、プログラム変換のための意味解析によって操作されるMネットへ変換する。図6.1で示すようにNネットの各ノードのNtom:スロットにMノードが生成される。Nネットのすべての子ノードにNtom:スロットが生成されていなければ親ノードのMノードは生成できない。そこで4.3節のルールの連鎖適用法により、Mモデルの Ntom:スロットのIfNeeded:に自分自身のMノードを生成する関数を記述することによって、Nネットのルートノードの Ntom:スロットを参照するだけでMネットがえられる。

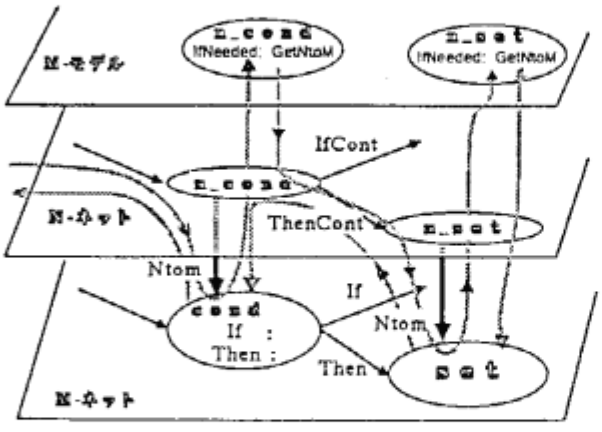

図6.1 NネットからMネットへの変換

6.1.2 文脈による省略語の補完

プログラムを自動生成する過程として重要な問題に省略語の補完がある。ここで取り扱う補完には主として文脈による補完と文型による補完がある。まず、文脈による補完について説明する。

「AとBを加え、Zに代入する。」

は、何を「Zに代入する」のかが、明示的に記述されていない。しかし、局所的な文脈を理解することによって「AとBを加えた結果」であることがわかる。これをシステムが実現するためには、まず図6.2で示すように文脈の局所有効範囲をMネットのProcBノード、省略語を$$ノードで示す。 Setノードの Mモデルには、「代入できるもの」はExprクラスに属するノードでなくてはならないことがアークスロットに記述されている。図6.3で示す補完ルールは、ProcBノードの兄弟ノードの中でセットノードより前に存在しExprクラスのノードを探し、条件に満足するAddノード以下をSetノードの$$ノードの位置に置き換える。補完の結果を図6.4に示す。

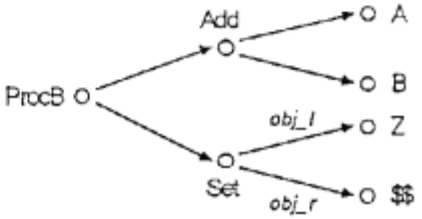

図6.2 省略語を含んだMネット

```
((IF (PATTARN
 (^Node $set
   (^E (InstOf: Set)
       (ObjR:
        (^Node $objr
          (^E (InstOf: $$))))
       (Reverse:
        (^Node $proc
          (^E (InstOf: ProcB)
              (Conts:
               (^Node $expr
                 (^E (Akos: (^Elem $akos Expr))))
               (^Node $set)))))))))
 (THEN (^Nodes
         ($procb (Conts: (-- $expr)))
         ($set   (ObjR:  (++ $expr)))))
```

図6.3 文脈による補完ルール

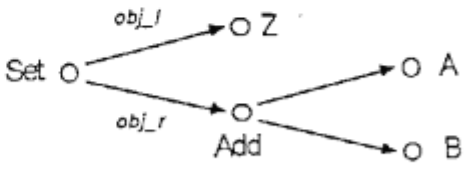

図6.4 文脈による補完の結果

6.1.3 文型による補完

文型による補完は、省略を表す$$ノードを含むMネット中のサブネットと相似なサブネットをMネット中に探す。探す範囲は、問題にしている省略を含んだ日本文より前に書かれている日本文を表現したMネットの部分である。相似なサブネットが見つかれば、$$ノードを見つかったサブネット中で$$に対応する位置にあるノードで置き換える。例えば、

「Aが1に等しいとき、Bをする。
2に等しいとき、Dをする。」

このとき、「何が2に等しい」かがわからないが、相似なサブネットが「Aが1に等しい」であることから、Aを補完する。これを補完ルールで記述するときは、はなれた位置を指定するために複数のノードをマッチできるFlexibleノードをもちいる。インタプリタがそのノードにあたる部分を必要なだけネットワークをたどりふさわしい構造を見つける。

6.2 質疑応答システムへの応用

前節の省略語の補完を応用することによって、質疑応答システムのメカニズムに利用できる。概観を図6.5に示す。質疑応答システムは、あらかじめ知識ベースをMネットによって構築しておき、その知識ベースに対する問い合わせに応答する。問い合わせ文は省略を表す$$ノードをともなったMネットに変換され、知識ベース中のMネットとパターンマッチされる。パターンマッチされた質問の答えは、変換ルール記述方式と同等な記述によって対応するサブMネットから自然言語を生成する。[杉山他88]

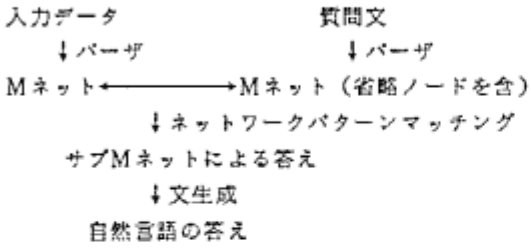

図6.5 質問応答システムの概観

7. まとめ

変換ルール記述言語はプログラム言語によってネットワークを操作するより、意味表現を探索したり処理内容を抽象化したレベルで操作ができることを確認した。変換ルールの可読性が高いためルール定義者以外の人でも保守や拡張が可能である。ルール適用においてなるべく無駄なパターンマッチングを行なわずに効率よく変換が行なわれるようなルール制御が行えた。さらに、Mモデルのデーモン機能と組み合わせることによってランダムでない連鎖的なルール適用も可能となった。

今後の課題としては、どの分野に本意味解析システムを用いてもルールが不自由なくパターンマッチ及び変換操作を遂行できる記述能力に高めること、変換ルールが複雑になり、一つのノードから複数のルールが適用可能になるとき、それら競合集合(conflict set)から最適なルールを一つ選ぶ機構を上位に設けなくてはならない。そして、マルチプロセッサのデータフローモデルでの並列型ルール適用制御へ拡張することがあげられる。

謝辞

本研究の一部は、ＩＣＯＴの再委託研究の一環として行なわれた。ＩＣＯＴの関係各位に深謝する。また、有益なご意見をいただいた日本電気の川越恭二課長、和田孝氏に感謝する。

参考文献

[Barwise&Perry83] Barwise,J. & Perry,J. Situations and Attitudes, The MIT Press. 1983.

[Lehnert&Wilks79] Lehnert,W. & Wilks,Y. A critical perspective on KRL. Cognitive Science,3, PP.1-28. 1979.

[Minsky75] Minsky,M. A framework for representing knowledge. In P.Winston(Ed.), The psychology of computer vision. McGraw-Hill. 1975.

[Schank75] Schank,R.C. Conceptual information processing. North-Holland. 1975.

[和田他88] 和田孝、杉山高弘、阪田全弘他 「プログラム自動生成のための日本語仕様記述言語」 情報処理学会プログラミング言語研究会、16-5、1988

[岩元他87] 岩元完二、西谷泰昭、和田孝 「プログラム自動生成システム」 ＮＥＣ技報、Vol.40、No.1、pp.35-38、1987

[杉山他88] 杉山高弘、宮下洋一、岩元完二 「プログラム自動生成システムＰＧＥＮにおける生成パターンを用いた多言語生成機能」 情報処理学会第３６回全国大会、1T-4 1988

[原本他89] 原本純一、秋口忠三、杉山高弘他 「Ｘウィンドウ上のマルチメディアユーザインタフェース構築環境：揃」 情報処理学会第３０回プログラミング・シンポジウム 1989