連想 Rete ネットワークの逐次コンパイル法†

荒屋 真二††

プロダクションシステムの応用分野の拡大に伴い，その推論効率の向上は依然として強く望まれている．それに対する一つのアプローチとして，Rete ネットに代表されるように，ルール集合をあらかじめコンパイルする方法がある．この立場からの改良方式がこれまでいくつか提案され，推論効率は確かに改善されてきた．しかし，コンパイル処理の複雑化に伴う前処理時間の増大は，ルールベース開発効率の悪化という新たな問題を提起した．本論文では，ルールコンパイル方式の中で最も推論効率の良い連想 Rete ネット方式を対象とし，ルールベースの変化から連想 Rete ネットの変化だけを求める逐次コンパイル法を提案する．連想 Rete ネットは Rete ネットをサブネットとして完全に包含し，ルール相互の支持関係を表す連想リンクがさらに付加されたものである．提案手法では，まず Rete ネットの部分を筆者等が以前に提案した逐次コンパイル法によって修正する．次に，連想リンクの変化に影響を及ぼす可能性のある動作パタンだけを Rete ネットにトークンとして流すことにより実際の変化を求める．また本論文では，提案手法の性能解析のための数式モデルを提案し，ルール数や1ルールあたりの平均条件パタン数などがコンパイル効率に及ぼす影響を解明する．

1. まえがき

プロダクションシステム (PS) は人工知能研究やエキスパートシステム開発の基本的アーキテクチャとして高い評価を得ており多用されている．応用分野の拡大と共に，その推論効率の向上は依然として強く望まれている．推論実行前にルールをコンパイルする方法は有力なアプローチであり，よく知られているものに Rete アルゴリズムがある[1]．その改良方式では，知識の排他性[2]，条件のきつさ[3]，パタンマッチ成功確率[2)~4)]，ルール間の連鎖性[5]など，一種のメタ知識を用いて Rete ネットをさらに構造化し，高速化に成功している．これらルールコンパイル方式は確かに推論効率を改善するが，前処理としてのコンパイル時間の増大を引き起こす．ゆえに，ルールの追加／削除を頻繁に行う状況では，ルールベースの開発効率を悪化させ，コンパイル自体の効率化が強く望まれている[2),7)]．そのため，PS 記述言語 OPS5 の LISP 版では，逐次コンパイル機能が提供されており[6]，また，コンパイル自体に Rete ネットを利用した方式も提案されている[7]．しかし，Rete ネットをさらに複雑化した上記改良方式に対しては，コンパイル時間が Rete ネットよりも増加するにもかかわらず，コンパイルの効率化に関する研究は等閑視されている．

本論文では，現在のところ推論効率が最も良い反面，コンパイルに最も時間がかかる連想 Rete ネット[5]を対象とし，その効率的コンパイル法を提案する．本方式の特徴は，既存の連想 Rete ネットを利用することにより，ルールベースの変化から連想 Rete ネットの変化だけを求める点にある．本論文では，さらに，提案手法の性能解析のための数式モデルを提案し，ルール数や1ルールあたりの平均条件パタン数などがコンパイル効率に及ぼす影響を明らかにする．

次章で連想 Rete ネットの概要を説明し，3章においてその逐次コンパイルの1実現法を提案する．4章では解析的評価により提案方式の有効性を示す．5章では本論文をまとめると共に，今後の課題についてふれる．なお，本論文では PS として OPS5 に限定して議論を展開し，その文法[6]と Rete アルゴリズム[1]に関する予備知識を前提とする．

2. 連想 Rete ネットの概要

Rete ネットは，ルール条件部の類似性に関する知識を用いて，ルールベースをネットワーク状にコンパイルしたものである[1]．推論実行時には，作業記憶 (WM) に追加／削除された作業記憶要素 (WME) が，トークンとして Rete ネットのルートノードから流される．トークンは各ノードでテストを受け，それに成功すれば後続ノードへ，失敗すればバックトラックして流れる．一方，連想 Rete ネットは，WME の追加／削除を引き起こす動作項から，連想ノードへ至る連想リンクを Rete ネットに追加したものである[5]．ここで，連想ノードとは，ある動作項によって追加／削除される WME が Rete ネットを流れるときに，

† Incremental Compilation of the Associable Rete Network by SHINJI ARAYA (Department of Communication and Computer Engineering, Faculty of Engineering, Fukuoka Institute of Technology).

†† 福岡工業大学工学部通信工学科

必ず到達する最も深いノードで，かつその下流に位置する端末ノードのうち，少なくとも一つにトークンが到達する可能性があるようなノードである．また，連想リンクとは，ある動作項から，その動作項の連想ノードへ至るリンクである．ゆえに，連想 Rete ネット方式では，推論実行時にルートノードからではなく，動作項の連想ノードからトークンを直接流すため，明らかに成功あるいは失敗するテストを回避できる．ただし，動作パタンのインスタンシエーションを連想ノードから流しても，成功する条件パタンが必ず存在するとは限らない．

例えば，次の二つのルール R1，R2 に対する連想 Rete ネットは**図1**のようになる．

```
(P R1 (C1 ^A11 1 ^A12 <X>)
      (C2 ^A21 2 ^A22 3 ^A23 <X>)
  --> (REMOVE 1)
      (MAKE C3 ^A31 5 ^A32 <X>))
(P R2 (C2 ^A21 2 ^A22 4 ^A23 <X>)
      (C3 ^A31 5 ^A32 > <X>)
  --> (MAKE C4 ^A41 6 ^A42 <X>))
```

ルール R1 の第1動作項 (REMOVE 1) の動作パタンは (C1 ^A11 1 ^A12 <X>) である．これは，変数 <X> の束縛値が何であっても，必ず2入力ノード A12=A23 の左メモリに到達する．ゆえに，その連想ノードは2入力ノード A12=A23 の左メモリである．そのため，(REMOVE 1) から A12=A23 の左メモリへ至る連想リンクが付加されている．同様に，R1 の第2動作項の連想ノードは A23<A32 の右メモリである．R2 の動作項は連想ノードをもっておらず，行き止まり動作項と呼ばれる[8]．ルール R1 が発火した場合，第1動作項の実行に際して4回，第2動作項の実行に際して4回，計8回のノードテストが Rete ネット方式より少なくて済む．R2 が発火した場合には，Rete ネット方式ではトークンがルートノードから流され，ノードテストに3回失敗したあと終了する．一方，連想 Rete ネット方式では，R2 の動作項に連想ノードがないのでトークンを全く流さずに済む．

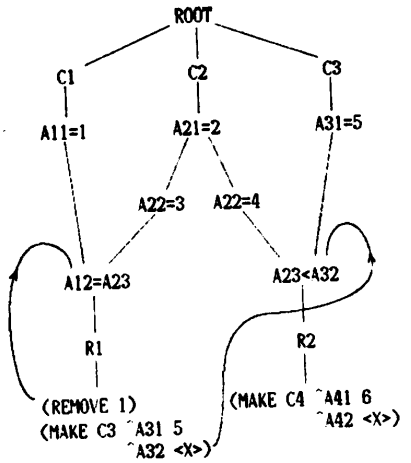

図 1 ルール R1 と R2 に対する連想 Rete ネット
Fig. 1 Associable Rete net for rules R1 and R2.

以上の単純な例からも推察されるように，ルールベースを連想 Rete ネットの形であらかじめコンパイルしておくことは，推論効率化に貢献する．しかし，Rete ネット方式に比べ，すべての動作項に対する連想ノードを求めるという処理が余分に必要となるので，コンパイル時の計算量は増加する．ルールベースが大規模になればなるほど，コンパイルのための計算量は増大する．

3. 連想 Rete ネットの逐次コンパイル

本章では，ルールベースの修正を頻繁に行う状況において，既存連想 Rete ネットを効率的に修正する逐次コンパイル法を提案する．

3.1 具 体 例

ここでは，ルールの追加／削除時に，既存連想 Rete ネットに対してどのような修正が必要になるかを具体例を用いて示す．前章の二つのルール R1，R2 および次のルール R3 を考える．ただし，これらのルールは紙面を節約するための仮想的なものであり，現実的な意味はない．

```
(P R3 (C3 ^A31 5)
      (C4 ^A41 6 ^A42 >7)
  --> (MAKE C1 ^A11 1 ^A12 8))
```

(1) R1，R2 に R3 を追加する場合

ルール R1 および R2 に対する連想 Rete ネットは図1であったが，これにルール R3 を追加すると連想 Rete ネットは**図2**のように変化する．図2の破線は新たに追加された部分を示しており，既存ネットは新しいネットの部分集合となっている．図2をもう少し詳しく見ると，ルール追加にあたって必要となる既存ネットの変更は次の3種類に分けられる．

① Rete ネットの部分（ルートノードから端末ノード R1～R2 までの部分）に，追加ルールの条件をテストするためのノードと端末ノードを付加する．

② 追加ルールの動作項の連想ノードが Rete ネッ

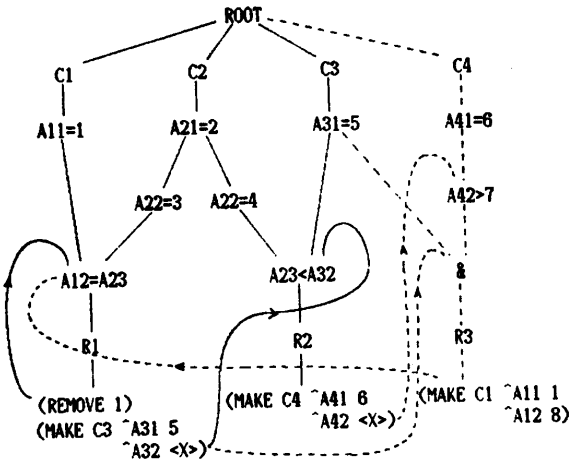

図 2 ルール R1, R2 に R3 が追加されたときの連想 Rete ネットの変化
Fig. 2 Changes of the associable Rete net when rule R3 is added to R1 and R2.

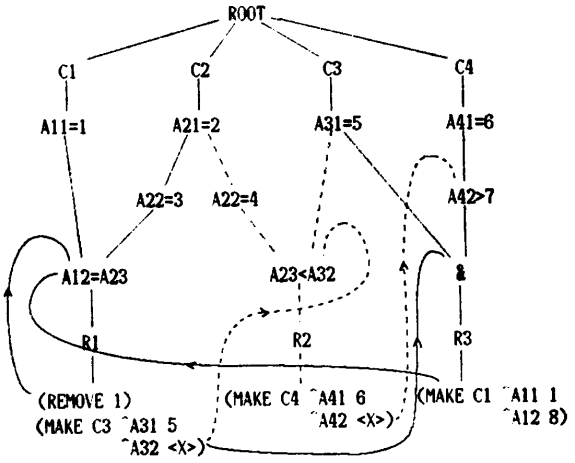

図 3 ルール R1, R2, R3 から R2 が削除されたときの連想 Rete ネットの変化
Fig. 3 Changes of the associable Rete net when rule R2 is deleted from R1, R2 and R3.

ト中に存在するならば，その連想リンクを付加する．

③ 既存ルールの動作項の連想ノードが上記①で付加されたノードの中に存在するならば，その連想リンクを付加する．

明らかに，ルールの追加によって既存の連想リンクが消滅することはない．

（2） R1, R2, R3 から R2 を削除する場合

ルール R1, R2, R3 に対する連想 Rete ネットは図2となったが，これからルール R2 を削除すると連想 Rete ネットは**図3**のように変化する．図3の破線部分は，R2 の削除により既存連想 Rete ネットから削除される部分を示している．この例からもわかるように，ルール削除時に必要となる既存連想 Rete ネットの変更は，次の3種類に分けられる．

① 削除ルールに特有の条件をテストするノードと端末ノードを Rete ネットの部分から削除する．

② 上記①に伴い，削除ルールの動作項から出ていた連想リンクを除去する．

③ 上記①によって削除されたノードの中に連想ノードがあるならば，既存ルールの動作項からそれへ至る連想リンクを除去する．

明らかに，ルールの削除によって新たな連想ノードが発生することはない．

3.2 逐次コンパイル処理

前節の例からもわかるように，ルールの追加／削除にあたって必要となる連想 Rete ネットの変更はごく一部であるため，変更後のルール集合を一括処理して新しい連想 Rete ネットを作り直す方法はかなり多くの無駄を含む．その無駄は，ルールベースが大規模になればなるほど増大する．この無駄を回避するためには，ルールベースの変化から既存ネットの変化だけを求めてやればよい．この考え方に基づく連想 Rete ネットの逐次コンパイル処理の概要を以下に示す．

[1] 連想 Rete ネットの一部を構成する Rete ネットの部分だけをまず修正する．ルールの削除ならば [4] にとぶ．

[2] 追加ルールの動作項の連想ノードを求め，連想リンクを付加する．

[3] 上記 [1] によって追加されたノードの中に，既存ルールの動作項の連想ノードが存在するならば，それに対応する連想リンクを付加し，処理を終了する．

[4] 削除ルールの動作項から出ていた連想リンクを削除する．

[5] 上記 [1] によって削除されたノードに入っていた連想リンクを除去し，処理を終了する．

上記[1]は Rete ネットの逐次コンパイルであり，OPS5 (Lisp 版) で既に実現されている．ただし，OPS5 ではルール削除時に単にそのルールの発火が禁止されるだけであり，実際に Rete ネットは修正されない[6]．そのため，推論時には削除したルールに対応したノードにもトークンが流されるという無駄が行われている．この推論時の無駄をなくすと共に，逐次コンパイルに必要なパタン照合を高速化した改良アルゴリズムも提案されている[7]．ゆえに，上記 [1] の処

理はこのアルゴリズムをそのまま利用することができる．

[2]の処理は連想ノードを一括して求める方法で使用されている基本アルゴリズム[6]を流用できる．すなわち，全ルールの動作パタンではなく，追加ルールの動作パタンだけを Rete ネットのルートノードからトークンとして流せばよい．[4]の処理は単純であり説明するまでもない．また，[5]の処理は連想リンクを逆向きにたどれるようにしておけば，簡単に実現できる．ゆえに，以下では，上記[3]の具体的実現方法を述べる．

再び，ルール R1，R2 が既に存在しているとき，ルール R3 が追加された場合（図2）を考えよう．ここでは，既存ルール R1 の第2動作項および R2 の動作項に新たな連想ノードが発生している．R1 の第1動作項の動作パタンのクラス名は C1 であるが，追加ルールはクラス名が C1 の条件パタンを含んでいない．ゆえに，その動作項の連想ノードが新たに発生することはないので，その動作パタンをトークンとしてルートノードから流すことは無駄である．一方，R1 の第2動作項の動作パタンおよび R2 の動作項の動作パタンのクラス名はそれぞれ C3 および C4 であり，これらは追加ルールの条件パタンに含まれている．ゆえに，これらの動作項に対しては新しい連想ノードが発生する可能性があり，それを検査するために動作パタンを Rete ネットに流す必要がある．

この例から，ルール追加時における既存動作項の連想ノードの変化は，追加ルールの条件パタンを支持する可能性のある既存動作パタンだけを Rete ネットに流せば求められることがわかる．そして，既存動作パタンが追加ルールの条件パタンを支持するための必要条件は，その動作パタンのクラス名と同じクラス名を持つ条件パタンが追加ルールの中に存在することである．なぜならば，もしクラス名が異なっているならば，支持する可能性は絶対にないからである．ゆえに，ルールが追加されたときに，その条件パタンのクラス名から，それと同一のクラス名をもつ動作パタンをもつ既存ルールを取り出せるようにしておけばよい．

そこで，クラス名をキーとし，そのクラス名を動作パタンに含むルールの名前のリストを値とする属性リストを導入する．今，このような属性リストをもつ記号を CLASS-TO-RULE とする．そして，ルールを追加／削除する度に，この属性リストを変更する．例えば，図1の時点における CLASS-TO-RULE の属性リストは (C1 (R1) C3 (R1) C4 (R2)) である．ルール R3 の追加により，それは (C1 (R1 R3) C3 (R1) C4 (R2)) に変化し，ルール R2 の削除により (C1 (R1 R3) C3 (R1)) となる．

このような属性リストを導入することにより，以下の手順で既存動作項の連想ノードを効率的に変更することができる．

(1) 追加ルールの条件パタンに含まれるクラス名をすべて抽出する．

(2) クラス名をキーとして，そのクラス名を動作パタンに含むすべてのルールの名前を CLASS-TO-RULE から取り出し，その各々に対して以下①～③の処理を行う．

① ルール名からルール本体を取り出し，そのクラス名を含む動作パタンを切り出す．

② その動作パタンを Rete ネットのルートノードから流すことにより連想ノードを求める．

③ その連想ノードへ至る連想リンクをその動作項に付加する．

(3) もし，未処理のクラス名が残っているならば (2)に戻る．

(4) 追加ルールの動作パタンに含まれるクラス名をすべて抽出し，CLASS-TO-RULE の属性リストを変更する．

ここで，(2)①においてルール名からルール本体を取り出す機能は既に OPS5 に組み込まれている．また，(2)②③の処理は一括手法の場合と同じである．なお，ルール削除時の処理[4]においても，この CLASS-TO-RULE の属性リストを変更しておく必要がある．

Rete ネットの逐次コンパイルでは，追加／削除ルールの条件パタンだけを Rete ネットに流せばよかった[7]．それに対し，連想 Rete ネットのルール追加時の逐次コンパイルでは，条件パタンだけではなく，それら条件パタンを支持する可能性のあるすべての動作パタンも流す必要がある．

4. 提案方式の評価

連想ノードを逐次的に求める提案手法が，従来の一括手法と比較してどの程度有効であるかを解析的に調べる．Rete ネットの部分の修正に必要な計算量は一括手法と同じなので，ここでは連想ノードを求める計算量だけを比較する．今，次のように記号を定義す

る．

n：既存ルールの総数

L：1ルールあたりの平均条件パタン数

R：1ルールあたりの平均動作パタン数

K：1クラスあたりの平均動作パタン数

ここで，K に関してはもう少し説明が必要であろう．あるクラス名をもつ WME を追加／削除する動作項は，ルール全体を考えれば一般に複数個存在し，その数はクラスによって異なる．クラス名 C_i を含む動作パタンの数を M_i とし，ルール全体で使用されるクラス名の数を S とすれば，

$$K=(M_1+M_2+\cdots+M_S)/S$$

である．例えば，図2の場合には $M_1=2$, $M_2=0$, $M_3=M_4=1$, $S=4$ であるので，上式より $K=1$ となる．

さて，連想ノードを求めるために Rete ネットに流さなければならない動作パタンの数を，一括手法と逐次手法のそれぞれに対して求める．まず一括手法では，ルール追加時には平均 $R(n+1)$ 個の動作パタンをルートノードから流す必要がある．ルール削除時には，平均 $R(n-1)$ 個の動作パタンをルートノードから流す必要がある．

一方，逐次手法では，ルール追加時には，追加ルールの動作パタン R 個をまずトークンとして流す必要がある．追加ルールの条件部に含まれるクラス名の数の平均値は，1ルールあたりの平均条件パタン数 L にほぼ等しい．ゆえに，追加ルールの条件パタンを支持する可能性のある動作パタンの数の平均値は LK 個であり，これも Rete ネットに流す必要がある．したがって，ルール追加時に流す必要がある動作パタンの数の平均値は，$R+LK$ 個となる．前章で述べたように，逐次手法ではルール削除時に動作パタンを Rete ネットに流す必要は全くない．

ルールベースの開発にあたっては，ルールの追加と削除が頻繁に行われる（ルールの修正は，既存ルールの削除と新ルールの追加からなると考える）．平均して考えた場合，ルール削除1回につき，α 回の割合でルール追加が行われるものとする（一般には削除より追加の方が多いので，$\alpha>1$ となる）．このとき，ルール追加あるいは削除1回あたりに流される動作パタンの数の平均値は，一括手法の場合を J_1 とすると，

$$J_1=\{\alpha R(n+1)+R(n-1)\}/(\alpha+1)$$
$$=R\{n+(\alpha-1)/(\alpha+1)\}$$

となり，逐次手法の場合を J_2 とすれば，

$$J_2=\alpha(R+LK)/(\alpha+1)$$

となる．J_1 はルール数に比例して増加するのに対し，J_2 はルール数に依存しない．

$K=\alpha=5$ の時の n に対する J_1/J_2 の値を，L と R の比率をパラメータとしてプロットすると図4となる．例えば，$n=1000$, $L/R=1.0$ のとき，J_1/J_2 の値は約200となる．これは，連想ノードを求めるために一括手法は提案手法に比較して約200倍のトークン処理を必要とすることを意味する．J_1/J_2 の値は n に比例するので，ルール数が多くなるほど提案方式は威力を発揮する．また，図5は $n=1000$, $L/R=0.5$ のときの α に対する J_1/J_2 の値を，K をパラメータとし

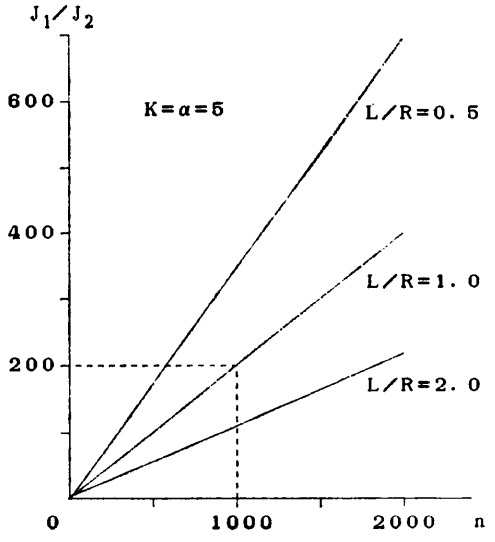

図4 n に対する J_1/J_2 の変化
Fig. 4 J_1/J_2 versus n.

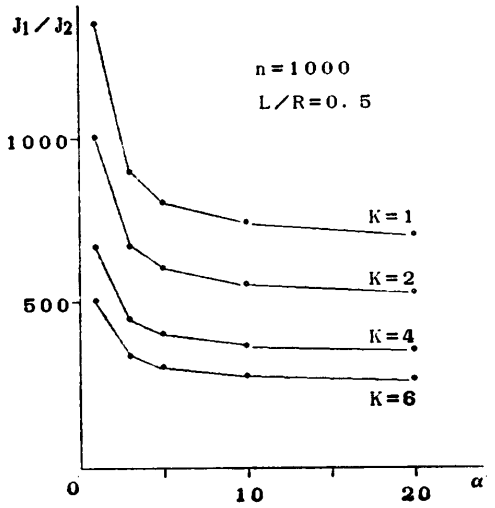

図5 α に対する J_1/J_2 の変化
Fig. 5 J_1/J_2 versus α.

てプロットしたものである．J_1/J_2 の値は α に反比例し，$\alpha>10$ ではあまり変化しなくなることがわかる．また，α および K が小さいときには提案方式は極めて有効である．実際のルールベースでは，n, L, R, K の値は様々であり，α の値も対象により異なる．ゆえに，提案手法の有効性も対象によりかなり変化する．

5. まとめ

本論文では，連想 Rete ネットを逐次的にコンパイルする1手法を提案した．また，本手法の計算量を評価するための解析モデルを提案し，ルールベースの特徴を表す種々のパラメータ（ルール数や条件パタン数の平均値など）が，コンパイル効率に与える影響を明らかにした．その結果，提案した逐次手法は，従来の一括手法のコンパイル効率を大きく改善できることが判明した．これまで，推論効率は優れているがコンパイル時間の点で問題のあった連想 Rete ネット方式を，より実用的なものにできる見通しを得た．本機能を持つ PS のインプリメントと本格的な実験的評価は今後の課題である．

謝辞 本研究の一部は福岡工業大学附属エレクトロニクス研究所より助成を受けた．ここに心より謝意を表する．

参考文献

1) Forgy, C. L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artif. Intell.*, Vol. 19, pp. 17-37 (1982).
2) 荒屋ほか：プロダクションシステムのための高速パターン照合アルゴリズム，情報処理学会論文誌，Vol. 28, No. 7, pp. 768-775 (1987).
3) 田野ほか：知識ベースシステム構築用ツール EUREKA における高速処理方式，同上，Vol. 28, No. 12, pp. 1255-1262 (1987).
4) 田野ほか：推論高速化のための弁別ネットワークの動的変形法，第 33 回情報処理学会全国大会論文集 (1988).
5) 荒屋ほか：プロダクションシステムにおける効率的パタン照合のための連想 Rete ネットワーク表現，情報処理学会論文誌，Vol. 29, No. 8, pp. 741-748 (1988).
6) Brownston, L. et al.: Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming, Addison-Wesley (1985).
7) 荒屋ほか：知識ベースの逐次構造化—Rete ネットワークの逐次構築法，電子情報通信学会論文誌，Vol. J71-D, No. 6, pp. 1100-1108 (1988).
8) Nguyen, T. A. et al.: Checking an Expert Systems Knowledge Base for Consistency and Completeness, *IJCAI-85*, pp. 375-378 (1985).

(昭和 63 年 4 月 6 日受付)
(昭和 63 年 9 月 5 日採録)

荒屋 真二（正会員）

昭和 24 年生．昭和 47 年東北大学工学部通信工学科卒業．同年三菱電機(株)入社．昭和 60 年福岡工業大学通信工学科助教授．この間，昭和 58 年 4 月～9 月神戸大学工学部非常勤講師．工学博士（東京大学）．人工知能，知識工学，コンピュータミュージックの研究に従事．昭和 57 年電気学会論文賞受賞．電気学会，電子情報通信学会，人工知能学会，IEEE 各会員．