TR-AC-0015　　　　002

プロダクションシステム高速化の検討

原田 良雄　篠原 直嗣（実習生）

1998. 3. 2

ATR環境適応通信研究所

目次

1 プロダクションシステム **1**
1.1 Production Memory 1
1.2 Working Memory 1
1.3 Interpreter 1

2 従来の高速化アルゴリズム **2**
2.1 従来の高速化アルゴリズムの分類 2
2.2 RETE アルゴリズム 3
2.3 TREAT アルゴリズム 3
2.4 排他 RETE アルゴリズム 4
2.5 ELITE アルゴリズム 5

3 高速化の検討 **5**
3.1 検討の対象 5
3.2 PM のネットワーク表現 5

4 まとめ **6**

1 プロダクションシステム

プロダクションシステムはエキスパートシステムにおける基本的アーキテクチャとして多用されている。
構成として Production Memory(PM)、 Working Memory(WM)、 Interpreter の 3 つからなる。

1.1 Production Memory

PM には「IF → THEN」で記述されるルールが格納される。各ルールはルール名、条件部、行動部からなり、さらに条件部は条件要素から、動作部は動作項から構成される。条件要素はクラス名のあとに属性名と属性値の対が 0 個以上続く。属性値には定数と変数がある。属性値の前には >(より大きい) や <=(以下) などの範囲を指定する述語を使用できる。

(例) (rule1 (C1 (a2 > 70) (a3 < X))
(C1 (a2 < 10) (a3 X)
(C1 (a2 20) (a3 600))
-> (...) ...)

C1 がクラス名、 a1、 a2、 a3 が属性名、 70、 10、 20、 600 が属性値を表す。X は変数をあらわし、同一変数は同値をとる。それぞれの条件要素は

1. 1 つの WM 要素で条件成立判定ができる条件
2. 複数の WM 要素間で条件成立判定ができる条件

に分類することができ、前者をイントラ条件、後者をインター条件と呼ぶ。上記の例の条件部は 2 つのインター条件と 1 つのイントラ条件からなる。

1.2 Working Memory

WM は WM 要素の集合である。 WM 要素にはルールによって参照・更新の対象とされるデータがそれぞれ格納される。WM 要素の形式は条件要素とほぼ同様である。ただし、属性値として変数、述語を使用できない。

(例) (C1 (a1 12)
(a2 85)
(a3 930))

1.3 Interpreter

インタプリタは次の認知・実行サイクルを繰り返しながら推論を行なう。

1. 照合 (パターンマッチ): WM と PM の間で照合を行ない、条件を満たす全てのルール (競合集合) を求める。
2. 競合解消: 競合集合の中から競合解消戦略にもとづいて、特定のルールを選択する。
3. 実行: 選択されたルールの行動部の動作を実行し、 WM の内容を更新する。

プロダクションシステムの構造を図 1 に示す。

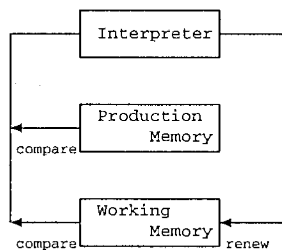

図 1: プロダクションシステムの構造

パターンマッチとは、各ルールのイントラ条件、インター条件を満たすWM要素の組を捜すことである。インター条件はWM要素の組で条件判定を行なう必要があるため、効率上「まず、イントラ条件を満たすWM要素を求め、次に、イントラ条件を満たしたWM要素間でインター条件の判定を行なう」の順で処理する。

例えば、3つの条件要素A、B、Cがあり、AとB、BとC、AとC間にインター条件がある場合、このパターンの満たすべき条件の構造は、次のようなツリーで表現することができ、これは同時にパターンマッチの処理構造を示している。

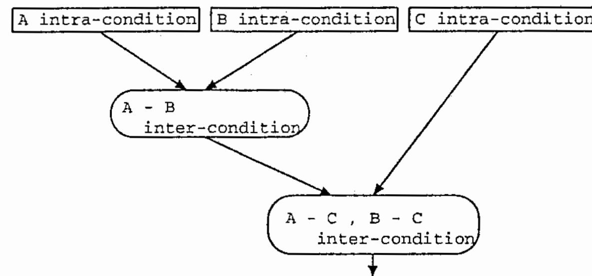

図2: 条件判定の処理構造

図2は、まずA、B、Cの条件要素のイントラ条件を満たすかどうかをそれぞれ判定する。そしてイントラ条件を満たしたWM要素間でインター条件の成立を判定すること、およびインター条件の成立判定順序が、「AとB間」、「AとC間、BとC間」の順であることを示している。

2 従来の高速化アルゴリズム

完全な競合集合を求める最も単純な方法は、それぞれの認知・実行サイクルにおいて、すべての条件要素とすべてのWM要素との間でパターンマッチを行なうことである。しかしこの方法はパターンマッチ処理だけのために全実行時間の大部分を費やすため、推論効率は極めて悪い。この問題を解決するために、これまでに多くの研究が行なわれてきた。それらの研究は大きく3つに分類できる。

1. 無駄なパターンマッチの削減による高速化
2. コンパイラによる高速化
3. 並列処理による高速化

本報告では上記1のアプローチによる高速化について調査、検討を行なう。

2.1 従来の高速化アルゴリズムの分類

無駄なパターンマッチを削減するために提案されている具体的な方法は次の2点に集約できる。

- 現在のWMに対するマッチングの結果を保存し、次の認知・実行サイクルでマッチングに活用する。これにより、変化しなかったWM要素とのマッチングを再度やり直すという無駄を排除できる。
- 条件要素間の部分的共通性をあらかじめ抽出しておき、同一の照合をまとめて行なうようにする。これにより、同一の照合を何度も繰り返すという無駄を回避できる。

ここで、マッチングの結果の保存という観点で、従来方式を分類すると、

1. 過去のマッチングの結果を利用しない。
2. 最終結果だけを利用する。
3. 最終結果と途中結果 (イントラ条件成立のみ) を利用する。
4. 最終結果と途中結果 (イントラ条件、インター条件成立) を利用する。

の4つに分類される。

方式4は最も有名なRETEアルゴリズムであり、その改良アルゴリズムは多くのシステムで用いられている。

方式3はRETEアルゴリズムより高速であると主張しているTREATアルゴリズムである。

方式1と2は以前は高速化をやっていないシステムに採用されていただけだっが、この方式が効率がよいとして、この方式を採用するシステムも現れている。

次にそのRETEアルゴリズム、TREATアルゴリズムとそれらの改良アルゴリズムを紹介する。

2.2 RETEアルゴリズム

RETEアルゴリズムはルール間で同じマッチングを行なっている部分をマージしてRETE木と呼ばれるデータフローネットワークを作る。そこにWMの変化をその追加、削除にともないプラストークン、マイナストークンとして流すことによりマッチングを行なう。

トークンに対してまずイントラ条件の判定を行ない、成功したものはαメモリと呼ばれるネットワーク内メモリに保存される。次にそのトークンに対しインター条件の判定を行なう。これに成功したものはβメモリに保存される。各マッチングではこのαメモリとβメモリを活用して前回の認知・実行サイクルと重複するマッチングを省略することができる。

例として、次に示すようなルールが与えられているとする。

```
(r1 (C1 (a1 X)(a2 4))
    (C2 (a1 X)(a2 6))
 -> (... (C4 (a1 8)))

(r2 (C1 (a1 X)(a2 4))
    (C2 (a1 X)(a2 7))
    (C3 (a1 6))
 -> (... (C4 (a1 5)))
```

この2つのルールr1、r2をRETE木にで表現すると次のようになる。

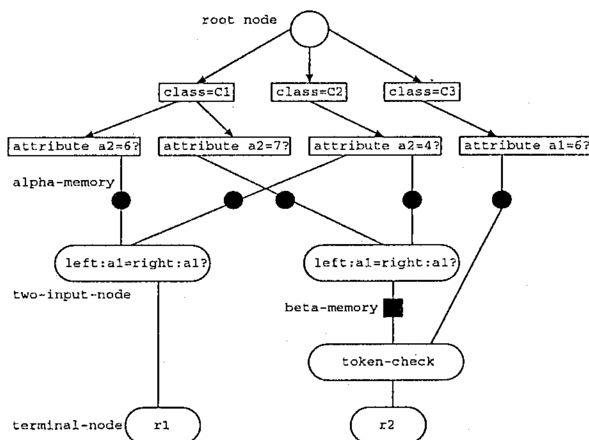

図 3: RETE ネットワーク

[問題点]

WMの更新が頻繁におこるような場合には、2入力ノードにおけるメモリ変更の手間が増加して、速度の低下につながる。

2.3 TREATアルゴリズム

Reteアルゴリズムからβメモリとルール間のマージを取り去ったものである。

βメモリを使用しない理由としてそのメンテナンスにかなりの手間が必要なことがあげられる。βメモリの役目はインター条件のマッチングの結果を保存しておくことに有るのだが、実際にはその結果はあまり使われず、かえって無駄な情報を保存していることの方が多い。特にWM要素の削除の場合、いちいちマイナストークンとしてβメモリをメンテナンスしなければならないのが大きな無駄となっている。

もう一つの問題点として、動的に必要なメモリが非常に多量となる場合があることがあげられる。β メモリは α メモリの組み合わせを記憶するものであるから、α メモリの平均的な量が n で抑えられるときでも条件要素間の照合条件が厳しくない場合にはルール中の条件要素数を m として n^m に比例する量の β メモリが必要となる可能性がある。

次に、ルール間のマージについてであるが、この手法は手間が掛かる割にあまり効果をあげないことが報告されている。実際ある種のエキスパートシステムではマージを行なった場合と行なわない場合とではほとんど差がでない。このルール間のマージはルールに冗長性があるときに効果があるものである。

またインター条件のマッチングの順序がコンパイル時に静的に決定されてしまうことも欠点の一つである。

TERAT アルゴリズムの特徴としては次の 2 つがあげられる。

- ルールのアクティブ性アクティブとはルールのすべての条件要素の α メモリが空でない時をいう。もしルールがアクティブでなかったらそれ以後のマッチングは行なわずに済ますことが出来る。RETE アルゴリズムではたとえばルールの最後の条件要素にマッチする WM がない時でも、途中の β メモリを更新する必要があるためマッチングをする必要がある。

- マッチング順序の動的な変更マッチング順序の動的な変更とはインター条件のマッチングの順序を、例えば α メモリの少ない順にする等、実行中に最適化することをいう。これは、TREAT アルゴリズムにおいてはルール間のマージを行なわないため可能なことで、RETE アルゴリズムでは不可能である。

[問題点]
ルール間のマージを行なわないことによる冗長な α メモリの存在がある。β メモリの節約分で補えるとはいえ、やはり動的なメモリを多く使うことになるし、メンテナンスの手間もそれだけかかることになる。

2.4 排他 RETE アルゴリズム

ある WM 要素に対して、ある条件要素のクラス名のマッチングが成功したならば、異なるクラス名をもつ他の条件要素とのマッチングは必ず失敗する。また、ある属性値のマッチングが成功したときには、クラス名が同じでも対応する属性値が異なっている条件要素とのマッチングは必ず失敗する。このような知識の排他性をあらかじめ抽出しておき、実行時に利用すれば明らかに失敗するマッチングを回避することができる。

また、排他関係にあるクラス名や属性値が複数存在する場合、なるべく早い時点でマッチングが成功するようにマッチングする順序を工夫してやれば、より多くのマッチングの失敗を避けられる。

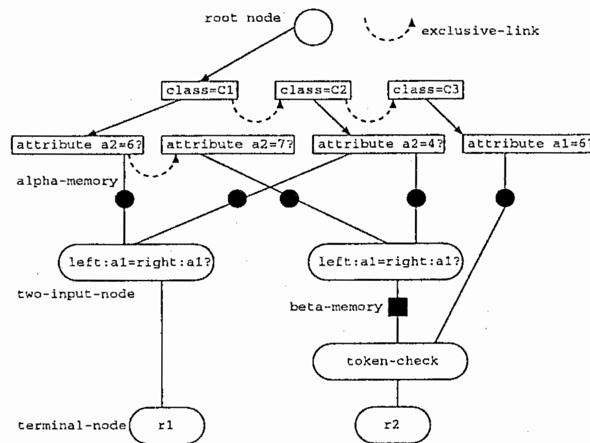

図 4: 排他 RETE ネットワーク

図 4 のように排他関係にある 1 入力ノードが排他リンクで連結され、他のリンクと区別される。排他リンクが出ているノードでは、そこのマッチングが成功したときには排他リンクはないものとみなされ、その先でバックトラックによりトークンが排他リンクの方へ流れることはない。マッチングが失敗したときにのみ、排他リンクへトークンが流される。これ以外の点は RETE アルゴリズムと同様である。

2.5 ELITE アルゴリズム

ELITE アルゴリズムは、「パターンマッチ処理の履歴情報の記憶・利用レベルは、対象とする問題の性質によりルール条件部の条件パターンごとに決めるべきである」という理由から、最適な履歴情報の記憶・利用レベルを導くことで高速化を行なうものである。

最適な履歴情報の記憶・利用レベルの問題は、ルール条件の判定順序に大きく左右されるため、両者を組合わせて解析し、最適な条件成立判定の処理構造を導き出す。具体的には

1. 実問題での推論処理過程をモニタし、そのログデータを解析し対象とする問題の性質を導く。
2. すべての条件判定順序とすべての履歴情報の記憶・利用レベルの組合わせから最適なパターンマッチ処理構造を導く。

3 高速化の検討

前節までに紹介した従来の高速化方式をふまえ、電話の基本サービスを対象としたプロダクションシステムにおいて、その高速化を検討する。

3.1 検討の対象

PM は次のようなものである。

```
 rule1 idle(a)              In-event:offhook(a)     -> dial-tone(a)
 rule2 dial-tone(a)         In-event:dialing(a,b)   -> wait-after-dial(a) Out-event:dial(a,b)
 rule3 wait-after-dial(a)   In-event:nack-sig1(b,a) -> busy(a)
 rule4 wait-after-dial(a)   In-event:ack-sig1(b,a)  -> ringback(a,b)
 rule5 ringback(a,b)        In-event:sig2(b,a)      -> talk(a,b)
 rule6 talk(a,b)            In-event:onhook(a)      -> idle(a)            Out-event sig3(a,b)
 rule7 talk(a,b)            In-event:sig3(b,a)      -> busy(a)
 rule8 busy(a)              Inevent:onhook(a)       -> idle(a)
 rule9 not-idle(a)          In-event:dial(b,a)      ->                    Out-event:nack-sig1(a,b)
rule10 idle(a)              In-event:dial(b,a)      -> ringing(a,b)       Out-event:ack-sig1(a,b)
rule11 ringing(a,b)         In-event:offhook(a)     -> talk(a,b)          Out-event:sig2(a,b)
rule12 dial-tone(a)         In-event:onhook(a)      -> idle(a)
rule13 ringback(a,b)        In-event:onhook(a)      -> idle(a) idle(b)
```

この PM の特徴として次の 2 つが挙げられる。

- ルールの条件部はターミナルの状態 (idle,dial-tone, …) とイベント (offhook,onhook, …) の対である。
- 条件部の条件要素はすべて 2 つである。

3.2 PM のネットワーク表現

排他 RETE アルゴリズムを適用するためルールの条件要素に state と event というクラスを付加し、ルールを次のように書き換える。

```
(rule1  (state (^name idle)(^terminal <a>))
        (event (^name offhook)(^terminal <a>))
->      -(state (^name idle)(^terminal <a>))
        -(event (^name offhook)(^terminal <a>))
        +(state (^name dial-tone)(^terminal <a>)))
```

そして排他 RETE アルゴリズムの考えか方を導入し

- ルールの共通部分をマージする。
- 両クラスの属性値はそれぞれ排他関係にあるので排他リンクで結ぶ。
- マッチングの途中結果と最終結果を保存する。

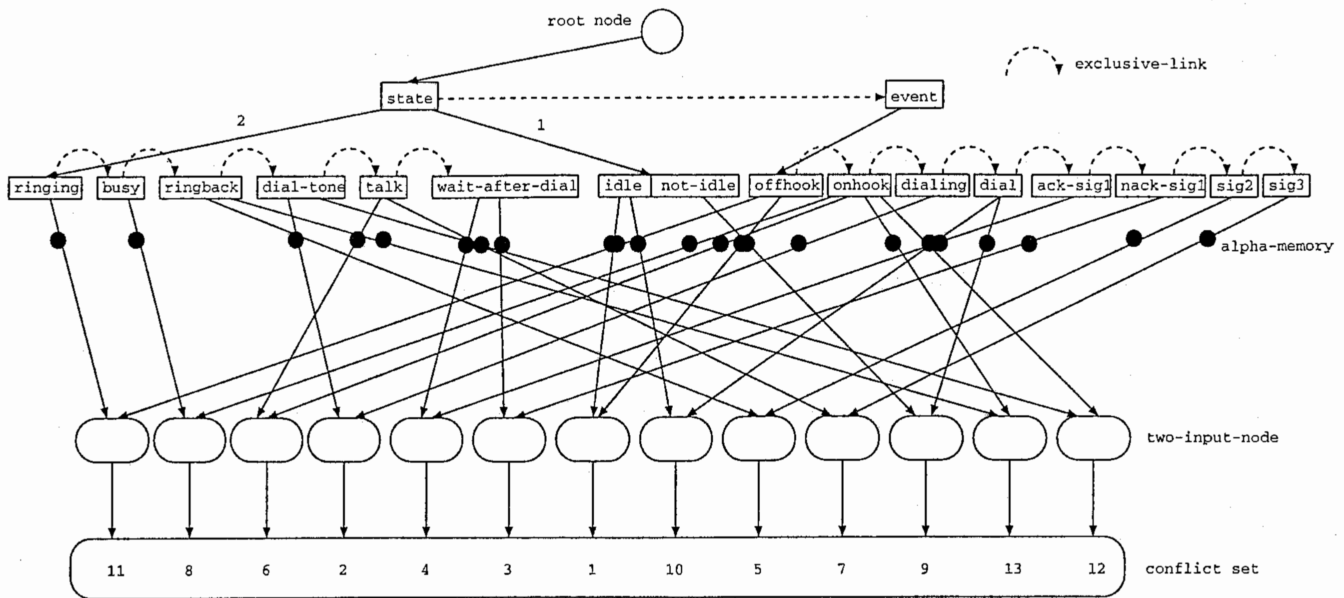

図 5: プロダクションネットワーク

ということを行ない、図5のプロダクションネットワークを作成した。なお、対象としたPMは条件部の条件要素がすべて2つであることから、βメモリは使用する必要がない。また、対象としたプロダクションシステムの性質上、offhook、onhook、dialingの3つのイベントは外部からプラストークンとして流される。それに加えて、ルールの行動部によるWMの更新に伴いプラストークン、マイナストークンを流す。

処理は次の手順で行なわれる。

1. (イベント発生、offhook、onhook、dialing)
2. rootノードから流れきたトークンのクラスのマッチング
3. 属性名(状態名、イベント名)のマッチングとその結果の保存状態の更新
4. 2入力ノードでのマッチング
5. 競合集合の更新
6. 競合解消戦略による発火ルールの選択
7. ルールの行動部によるWMの更新

4 まとめ

従来の高速化アルゴリズムを調査し、対象としたプロダクションシステムに排他RETEアルゴリズムを適用することで、変化しなかったWM要素とのマッチングの回避、ルールのマージによる同一のマッチングの繰り返しの回避を行ない、その高速化が可能であることを示した。

また、ルールの共通性を利用することで成功と失敗の両方のマッチングの回数を減少させることができ、ルールの排他性を利用することで失敗のマッチングの回数を減少させる効果があることがわかる。

対象としたプロダクションシステムはルールの条件要素がすべて2つであるためマッチングの途中結果の保存にはαメモリのみを使用した。しかし、条件要素が3つ以上の場合にもネットワークの階層化、およびβメモリを使用することで高速化がはかれることがRETEアルゴリズムによって明らかである。

今後の課題としてELITEアルゴリズムで提案されているように、さまざまな事例から得たログデータを解析して、マッチングの順序を最適化すること、また、TREATアルゴリズムなどの他のアルゴリズムの適用による高速化の可能性について検討することが挙げられる。