5D−8　競合集合の拡張による プロダクションシステムのデバッグ

平尾卓也　深澤良彰　門倉敏夫
早稲田大学　理工学部

１．はじめに

プロダクションシステム型エキスパートシステムにおける従来のデバッギング手法は、トレーサによるものが主体である。このトレーサの出力により、競合集合、その時点での作業記憶の変更等に関する情報を得ることができる。その他にも、ブレークポイントを設定したり、指定された要素パターンに一致する作業記憶要素を検索する方法などがある。

しかし、これらの機能は、以下のような難点がある。

・知識ベースが巨大になると、トレース等による情報量が非常に多くなり、開発者の負担が増す。
・バグを含むプロダクションルールが、競合集合に入らない時、上の機能では発見しにくい。

本システムは、以上の難点を克服し、開発者の負担を最小限にし、デバッグを支援することを目的としている。以下では、プロダクションシステムとして、ＯＰＳ５を例にとって説明する。

２．前提

本システムが対象とするユーザは、プロダクションシステム型エキスパートシステム（以下ＥＳと略す）の開発者である。この開発者は、ルールの実行順序は、把握していないが、構築したシステムを実行させたときの出力の順序、及び、出力文中の変数の値に関しては理解しているものとする。従って、ユーザは、出力によってのみ、ＥＳ内にバグがあると認識するものとする。以下、出力を含むルールを出力ルールと呼ぶ。また、ＥＳは、文法エラーを含まず、動作可能なものとする。

３．拡張競合集合の導入

従来の競合集合は、条件部の要素のすべてがその時点の作業記憶要素にマッチしたルールのみを含む。トレーサ等の利用によりユーザは、その競合集合内のルール群は知り得るが、条件部の要素のいくつかがマッチしなかったルールの存在を知ることはできない。一般的なバグは、このようなルールに存在する場合が多く、そのルールを検出する必要がある。そこで、本システムでは、これらのルールを含めた競合集合を導入し、これを拡張競合集合と呼ぶ事にする。即ち、（競合集合）＋（条件部の要素のＮ個がマッチしないルール群）を拡張競合集合と定義する。但し、Ｎは任意に変更できるものとする。

図１のプログラム例は、RULE_TRUE が選択されねばならないところ、その時点での作業記憶が、図２の状態のため、RULE_FALSEが選択され実行されてしまう例である。

この時、以下のようにバグを含むルール RULE_TRUEが拡張競合集合に含まれている。ただし、Ｎ＝１と仮定している。

```
従来の競合集合：RULE_FALSE 1 2
拡張競合集合　：RULE_FALSE 1 2
                RULE_TRUE  1 NIL
```

```
        ：
(P RULE_TRUE
  (CLASS ^A a)
  (CLASS ^B b)
-->
  (WRITE |RULE_TRUE WAS SELECTED.|))
(P RULE_FALSE
  (CLASS ^A a)
  (CLASS ^B c)
-->
  (WRITE |RULE_FALSE WAS SELECTED.|))
        ：
```

図１：ＥＳのプログラム例

1:CLASS ^A a　　2:CLASS ^B c

図２：ある時点での作業記憶

４．システム概要

本システムは、ＯＰＳ５のインタプリタを内蔵しており、ＥＳを出力毎に停止させながら実行できる。以下に、本システムを用いてプロダクションシステム型エキスパートシステムのデバッグを行う標準的な手続きを示す。

A Debugging Tool For Production System Using Extended Conflict Set
Takuya HIRAO ,Yoshiaki FUKAZAWA ,Toshio KADOKURA
School of Science and Engineering,WASEDA University

4．1　ユーザによる実行の中断と情報入力

ユーザは、本システムのインタプリタを用いてＥＳの実行を行う。実行は、出力毎に停止するため、ユーザは、出力結果中に誤りを発見すると、その時点で実行を中断する。この中断後、本システムは、誤りに関して、以下のような情報を入力するようユーザに求める。

Ａ）ユーザの意図通りに実行された時に選択されるべき出力ルール

Ｂ）出力文中の変数の正しい値

本システムは、Ａ）のルールが、Ｂ）の値で出力されることを４．２の繰り返し実行の終了条件とする。

4．2　繰り返し実行

本システムは、その後、中断直前のルールが選択された際の作業記憶に対する拡張競合集合を求める。次に、優先度の高い順にルールを選択する。このルールが選択されるように作業記憶を変更した後、実行を再開する。このプロセスを、上記の終了条件が満たされるまで繰り返す。

ここで、拡張競合集合の優先度を、以下のように定義している。

1．従来の競合集合に存在するルール群に対しては、従来の戦略をそのまま適用する。

2．条件部の要素がマッチしていない個数が少ない順に、優先度を高くする。

3．条件部の要素がマッチしていない個数が同じルール群に対しては、従来の戦略に従う。

作業記憶を変更する際には、従来の競合集合の中で優先度が最も高くなるように、変更する。すなわち、

1．従来の競合集合に存在するルール群中の１つを実行させる場合には、そのルールの条件要素にマッチした作業記憶要素のタイムタグを最大にする。

2．条件部の要素のいくつかがマッチしていないルールを実行させる場合には、マッチしなかった条件部の要素とクラス名、属性名が同じで、属性値のみが異なる作業記憶要素を作業記憶中から検索する。そのうちタイムタグが最大のものの属性値をマッチしなかった条件部の要素にマッチするように変更する。

したがって、図１の例では、RULE_FALSEの代わりに、RULE_TRUE が実行されるように図３のように作業記憶を変更する。

```
1:CLASS ^A a      2:CLASS ^B b
```

図３：変更後の作業記憶

拡張競合集合内のルール群からの実行をすべて行っても終了条件が満たされない場合には、中断時のルールの直前に実行されたルールが選択された時の作業記憶の状態に戻り、同様のプロセスを繰り返す。

4．3　デバッギング環境

最初の実行の中断時の作業記憶の状態と、終了条件を満たした場合の作業記憶の状態の違いとバグの原因には、密接な関係がある。したがって、この違いから、以下のような、デバッグに有効な情報を提示する。

・拡張競合集合のルール群から、要素パターンの違いによって条件部がマッチせず競合集合から漏れたルール

・実行トレース中のルール群から、要素パターンの違いの原因となった動作部をもつルール

具体的には、本システムは、これらのルールの条件部、動作部を提示し、ユーザにチェックしてもらう。

図１の例では、本システムは、上にあてはまるルールを検索し、RULE_FALSEとRULE_TRUE の条件部の要素を提示する。また、このルールの前に実行されたルール群の中から、RULE_FALSE を選択させる原因となった作業記憶要素を変更しているルールとその動作部を検索し提示する。

5．おわりに

図４は、３０程度のルールからなるデモシステムにバグを人為的に加えたものを、本システムでデバッグしたときの実行ルール数と、そのルールが選択された時の拡張競合集合に含まれるルールの数との関係を示したものである。11回目の実行時の出力文に誤りが発見され、本システムの繰り返し実行を行い、５回目のルール中でバグが発見された。

Ｎの値を増やせば、バグの拾い出しに成功する確率も増加するが、Ｎ＝２以上の場合に拡張競合集合に含まれるルール群の数は、Ｎ＝１の場合と比較して圧倒的に多い（図４）。従って、繰り返し実行の回数も増加し、効率があがらない。

今後、出力文の少ないプロダクションシステムの対策を考えていく必要がある。

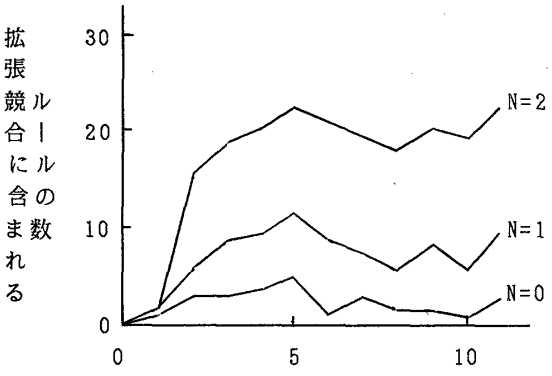

図４：実行ルール数と拡張競合集合に含まれるルールの数