From: AAAI-86 Proceedings. Copyright ©1986, AAAI (www.aaai.org). All rights reserved.

ADVANCES IN RETE PATTERN MATCHING

Marshall I. Schor, Timothy P. Daly, Ho Soo Lee, Beth R. Tibbitts

IBM T. J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598 USA

Abstract

A central algorithm in production systems is the pattern match among rule predicates and current data. Systems like OPS5 and its various derivatives use the RETE algorithm for this function. This paper describes and analyses several augmentations of the basic RETE algorithm that are incorporated into an experimental production system, YES/OPS, which achieve significant improvement in efficiency and rule clarity.

Introduction

Rule based systems often spend a large fraction of their execution time matching rule patterns with data. The production system OPS5 [FOR1] and many other systems (e.g. [ART1] [YAP1] [FOR3]), each use the OPS5 pattern match algorithm known as RETE. This paper describes four augmentations of the basic RETE algorithm that achieve much improved performance and rule clarity. As we describe each augmentation, we give an analysis of its effects, and some examples of its use. These ideas are implemented in an experimental production system language, YES/OPS, running on LISP/VM in IBM Yorktown Research.

We presume some familiarity with production systems, and the RETE algorithm. The reader is referred to the book, *Programming Expert Systems in OPS5* [BRO1], and the AI Journal article on the RETE algorithm [FOR2] for background information.

The first augmentation involves handling changes to existing data. In OPS5, three operations affect the data being matched with the rule patterns: `make`, which adds new data, `remove`, which removes data previously added, and `modify`, which modifies data previously added. However, `modify` is implemented in OPS5 as a `remove` of the previously existent data, followed by the creation of new data that is a copy of the previous data, except for the attributes that were changed. This new data is then added, which causes a new match cycle to occur. We change this to support `modify` as an update-in-place operation, and change how the rules are (re-)triggered, for greater clarity.

The second augmentation allows the user to group rule patterns (called condition elements) together, in an arbitrary fashion. This enables specifying negated joins of patterns, not just individual condition elements, and plays an important role in specifying when to do maximize and minimize operations (which follows). The grouping can also be used to increase pattern match result sharing among the rules, for efficiency.

The third augmentation supports the specification of sorted orderings among sets of data, in a much more efficient and syntactically clear manner.

The final augmentation is the ability to do the pattern matching on demand, incrementally. This supports both the incremental addition of new rules, such that the new rule does match the existing data (not possible in OPS5), and the matching of particular patterns as part of an action done when a rule fires, not when the data changes. This aspect eliminates the (OPS5) requirement that data to be manipulated in the action part of a rule must be matched by a condition element pattern in the rule's tests (its Left Hand Side). This allows many practical rule sets to achieve orders of magnitude performance improvement, by reducing the pattern matching part of the rules to just that part which needs to be data-change sensitive.

All examples of rules are written using the YES/OPS syntax. This is similar to OPS5 syntax, except: 1) attributes are not preceded by an "↑" character, but are followed instead by a colon ":"; 2) the rule form is:

```
(P rule-name
  WHEN
    pattern matching specifications
  THEN
    actions to be done)
```

MODIFY as update-in-place, new triggering conditions

OPS5's implementation of `modify` as a `remove` of the old value, and a re-`make` of it with the modified attributes causes excessive re-triggering of rules. Two commonly occurring instances of unwanted re-triggering are modification of attributes not tested in a rule and modification of an attribute to a value that still passes the same rule patterns as before.

Example: Don't-care slots re-triggering

Suppose the user structures his working memory elements for a problem involving genealogy research, as follows:

```
Classname: PERSON
Attributes: Name:  Father:  Mother:  Gender:
            Native-language:  Native-country:
            Language:  Marital-status:  Spouse:
```

Now suppose some rules infer about ancestry, and other rules infer about languages spoken. If the ancestry rules have fired, and now, some new information about language causes the person's `language:` attribute to be changed, in OPS5, the ancestry rules would fire again, even though they had taken all the actions appropriate for their matches to the existing data, and that data had not changed in the attributes of interest.

The solution to this behavior in OPS5 is to separate attributes whose change should not re-trigger other rules, into different working memory elements. This is often not the natural partition of the knowledge, and is less efficient, because the RETE must now do run-time joins of the split-apart attributes.

Example: Tests true once, true again after modifying, re-triggering

In OPS5, whenever a rule's action part modifies a working memory element such that it still satisfies the rule's tests, that rule loops. Users are told to "get around" this problem by coding extra control information in the working memory element and set flags that prevent looping. An example from the book *Programming Expert Systems in OPS5* [BRO1] is the problem of adding one to a set of items. The natural formulation (the one inexperienced users tend to write) looks like:

```
(p add-1-to-items
 when
    (goal name: add-1-to-items)
           ;the goal to do it
<i> (item value: <v>)
           ;an item, whose value is <v>
 then
    (modify <i>             ;modify the item
      value: (<v> + 1) ))  ;setting the value
                            ;to <v> + 1
```

This works in YES/OPS, when `modify` is update-in-place, but loops in OPS5. The suggested rule formulation to get around this problem in OPS5 is to add an extra attribute to `item`, called `status`, and set it from `nil` to `marked` when doing the adding. After adding to all the items, the `goal` is advanced to `unmark`, and another rule fires repeatedly, once per item, to change the `status` attribute back to `nil`. This clearly is more rule firings, and also, more testing (the value of the `status` attribute must be tested). The example also is now cluttered up with control information, unrelated to the task of adding 1 to a set of items, which makes these OPS5-style rules less readable:

```
(p add-1-to-items
 when
    (goal name: add-1-to-items)
                              ;the goal to do it
<i> (item value: <v> status: nil)
 then
    (modify <i> value: (compute <v> + 1)
                status: MARKED))
-------------------------------------------
(p change-task          ;this rule fires after
 when                   ;prev. rule because it
                        ;tests fewer things
<g> (goal name: add-1-to-items)
 then
    (modify <g> name: UNMARK))
-------------------------------------------
(p unmark
 when
    (goal name: UNMARK)
<i> (item status: MARKED)
 then
    (modify <i> status: NIL))
```

Having to code this kind of status information makes the rules less clear. Without implementing the new `modify` definition, the natural rule an expert often writes would need to be "fixed" to eliminate the unwanted triggering. The efficiency also suffers, in that the fixes require more pattern matching tests.

New modify definition removes re-triggering problems

We define `modify` as an atomic update-in-place operation, rather than as a `remove` followed by a `make` of the modified working memory element. The triggering rules are changed so that an existing instantiation that continues to exist after the `modify`, does NOT cause re-triggering.

In addition to improving performance by eliminating extra control flags and their testing and maintenance, `modify` done as update-in-place reuses existing working memory data structure and RETE memory nodes. This improves the performance by reducing the activity involved with maintaining these structures.

Triggering on any change

The new `modify` semantics normally trigger a rule when a rule instantiation that was not previously present gets created. This means that a `modify` operation does not re-trigger a rule, if it does not result in a new instantiation.

Sometimes, however, triggering on *any* change is desirable. An example might be a rule that counted how many times a person's marital status changed. Here, we want the rule to re-trigger, no matter what the status changed to. To provide for this case, we extend the syntax to allow specifying re-triggering on any change of one or more selected attributes, by preceding the attribute name by an *exclamation point* (!). In addition, to specify re-triggering on the change of any attribute in the class, an exclamation point may be placed in front of the class name. This gives behavior like OPS5. For example:

```
(p count-marital-status-changes
 when
    (person ! marital-stat:)
                        ;! retriggers on change
<c> (counter type: marital-stat-chg value: <v>)
 then
    (modify <c> value: (<v> + 1)))
```

New algorithm for Modify in RETE Beta Join nodes

Tokens passed down the RETE have the operation ADD, REMOVE, or MODIFY associated with them (ADD corresponds to `make`). For `modify` operations, if at some point in the processing, the test result of the *previous* value of the modified working memory element differs from that of the current value, the `modify` operation is converted to a remove or add operation:

	CASE 1	CASE 2
Previous value:	tests fail	tests OK
Current value:	tests OK	tests fail
New operation:	ADD	REMOVE

When a token arrives at the bottom of the RETE, if the operation is add or remove, then the rule instantiation in the production node is either inserted to or removed from the conflict set, according to the operation; if the operation is `modify`, then nothing is done. This prevents re-triggering.

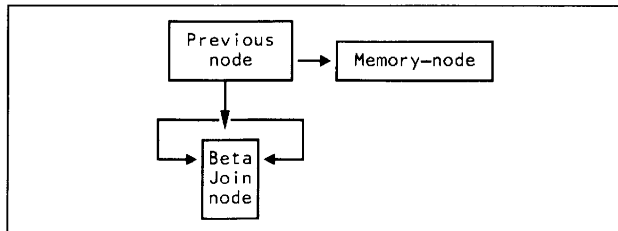
For `modify` operations, specification of re-triggering attributes causes an exception. If one or more of the attributes was preceded by an "!" to indicate that re-triggering is wanted on any change of that attribute, the attributes so designated are compared with those that were modified; if one or more match, then the rule is reinserted into the conflict set even if it has already fired.

Join nodes where left and right predecessors are identical

Special case handling is required where the left and right inputs to a join node are identical. This arises in rules like:

```
(p find-skilled-persons
 when
  (person name: <s> skill: <s1>)
  (person name: <n> needs-skilled-service: <s1>)
 then
  (say <s> can help <n> with service <s1>))
```

This yields the RETE structure:



A problem can occur when a new working memory element is added which matches with itself; in this example, this could happen if the person needs the skilled-service which he himself has. The problem happens because the RETE algorithm sends the result of any changes in a node to all of its successors. In particular, a change token arriving at the *previous node* would be sent down both the right and left legs to the same Beta Join node. If no special consideration is taken, what can happen is that the change token on each path causes an instantiation to be added to the conflict set, resulting in double instantiations.

OPS5 handles this case by first sending the token to all successors having left inputs, before updating the *memory node* by adding or removing (depending on the operation being done) the token to/from the memory. Thus, a change only sees itself on one leg (the right leg for add, the left leg for remove).

With `modify` implemented as an update-in-place, the token in question is *already* in the memory node. The new RETE code removes the particular element temporarily, before sending the `modify` operation to the left-successors. This prevents it from seeing itself during this phase. Then it puts it back into the list before sending it to the right successors.

Running backwards

Normal forward running of production systems repeats a cycle of matching rules with data, picking a rule to fire, and executing the picked rule's actions, which may change the data being matched. A very useful debugging tool is the ability to run backwards, that is, restore the state of the system to that which existed in previous cycles. OPS5 implements the `back` function for this; we have extended this function to handle `modify` as update-in-place.

The utility of `back` requires that the user be able to make top-level changes as well; otherwise, when forward running resumes, the system would merely repeat what it had already done. Two kinds of changes are possible: changing data, and changing rules.

In OPS5, incrementally added (or changed) rules do not match the existing data, which means that adding or changing rules dynamically is not practical. Extensions we have implemented for procedural matching support matching new or changed rules with existing data, making incremental rule editing a powerful debugging technique, usable with `back`.

`Back` requires that a history of changes to working memory and rule refractions (the firing of a rule instantiation) be kept during forward running. This history record is used to incrementally undo rule firing effects and restore the system to a previous state. `Modify` operations record the previous (unmodified) value, together with a pointer to the current working memory element in this history, so that the previous values can be restored when backing up.

Generalization of OPS5 validity test for reinserting refracted rules

When a rule fires, a record is made; when backing up, that rule is reinserted into the conflict set, re-enabling it to fire, *unless* something was done (at top level) that prevents it from being true anymore. In OPS5, the test done was to verify that all the working memory elements, which matched *positive* condition elements of a rule being backed up, were still present. This test is inadequate in the general case. Consider the following example:

```
(p back-bug  ;This rule depends on
  when       ;(b) not being present,
    (a)      ;in order to fire.
   -(b)
  then
   (....))
```

Now suppose we do the following top level actions:

1. `(make a)`; this will insert the "back-bug" rule into the conflict set.
2. `(run 1)`; fires the rule, running forward
3. `(make b)`; add `(b)` to the working memory
4. `(back 1)`; backs up 1 rule

If step 3 had not been done at top level, we would expect to see the "back-bug" rule reinserted into the conflict set. However, because `(b)` now exists, that instantiation is no longer valid.

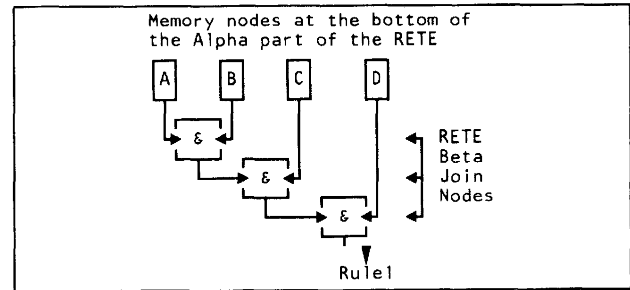
YES/OPS verifies that a reinserted instantiation actually exists, before reinserting it into the conflict set. To do this, we keep a RETE memory with each rule representing its current instantiations, given the current data in working memory. Before reinserting a rule when backing up, the instantiation is looked up in this memory. If it is present, then the rule instantiation is reinserted into the conflict set. If it is not present, then some top-level action changed working memory in such a manner to preclude this instantiation being true. In this case, the instantiation is not reinserted.

Arbitrary grouping of pattern condition elements

Rule condition element patterns of rules in OPS5 are grouped in a left-associative manner. For example, the joining of condition elements of the rule

```
(p rule1 when (a) (b) (c) (d) then ...)
```

results in a RETE join tree:



We have augmented the basic RETE to allow arbitrary groupings, in addition to the default left-to-right linear associative grouping.

Sharing pattern matching work among several rules

Part of the RETE algorithm efficiency comes from sharing pattern matching tests which are identical among all the rules that have the tests. However, the OPS5 RETE shares results of join tests only if the patterns are the same starting from the first one. For example, consider the three rules:

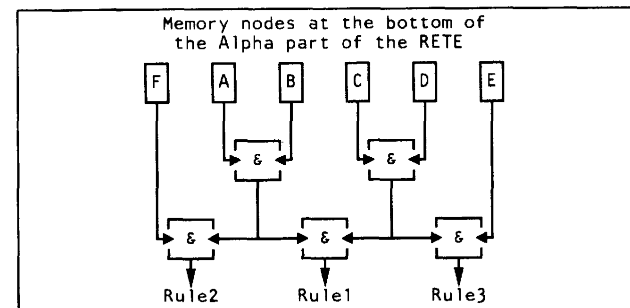
```
(p rule1    (p rule2    (p rule3
  when        when        when
   (a)         (a)         (c)
   (b)         (b)         (d)
   (c)         (f)         (e)
   (d)
  then ..     then ..     then..
```

The join for `(a)` and `(b)` are shared between rule1 and rule2, but the join of `(c)` and `(d)` in rule1 and rule3 are not shared, because of the top-to-bottom associativity of the joins.

By grouping as follows, one can get the benefits of shared tests:

```
(p rule1    (p rule2    (p rule3
  when        when        when
   (a)         (a)         (c)
   (b)         (b)         (d)
   ((c)        (f)         (e)
    (d))
  then ..     then ..     then..
```

The join part of the RETE would look like this:



One of the major factors in the run-time performance in OPS5 is the number of beta nodes (two-input join nodes). That is due to the fact

that testing beta nodes involves time-consuming tasks proportional to the size of the memory nodes, e.g., checking bound variables for possible join, evaluation of predicates, subsequent update of beta memories, etc. Reducing the number of beta nodes, by sharing RETE structures, increases the run-time performance.

Negating joined groups

One of the constructs supported by RETE is the negated condition element. Our grouping extension allows the negation of arbitrary combinations of condition elements. For example, a rule that verifies that no men and women pairs in a group share the same birthday:

```
(p no-same-birthday
 when
  (goal type: check-shared-birthdays)

 -((person gender: male   birthday: <bd>)
   (person gender: female birthday: <bd>))

 then
  (say No man and woman share the same birthday))
```

The above rule could not have been expressed in OPS5 without creating new working memory elements containing all the attributes to be negated, because the negated conditions have joins among themselves, and the test is for whether or not the join result is empty.

In OPS5, because only single condition elements could be negated, the knowledge programmer would have to rearrange the working memory data structures such that any test for non-existence would involve only single condition elements, never joins of multiple ones. Grouping gives the knowledge programmer the freedom to design working memory elements in a way that best suits the problem, without having to be concerned with support for negated conditions.

Maximize/Minimize

Many problems require sorting and selection of "best" or perhaps, "top two," for example, finding the maximum, finding the best two financial alternatives, etc. The OPS5 technique for specifying these patterns is somewhat obscure:

```
(p top-student
 when
  (student grade:   <top>
           name: <name>)
 -(student grade: gt <top>)
 then ...)
```

This rule logically means "find a student having a grade `<top>` such that no other student has a grade which is greater than `<top>`". This is semantically equivalent to finding the student (or students in case of a tie) who have the best grades.

We have augmented the syntax and RETE algorithm to support a clearer and more efficient expression of this kind. The same rule in the new syntax is:

```
(p top-student
  when
   (student grade: maximize  name: <name>)
  then ...)
```

The implementation is done by keeping the normal partial match memory nodes maintained during the RETE algorithm in sorted order, and adding a new kind of RETE node to do the selection of the maximum, or top two or minimum, etc.

Analysis of sorting efficiency

A simple binary tree search to insert a new element into a sorted list takes O(log n) comparisons, where n is the number of elements in the list. The average complexity to create a sorted list of n elements using the binary tree search, and pick the maximum is O(n log n).

When n elements are added to the working memory in the OPS5 formulation, the RETE does $O(n^2)$ comparisons. The situation gets worse if the top two students are requested: The OPS5 formulation is:

```
(p select-best-two
 when
  (student grade: <top1>   name: <n1>)
  (student grade: <top2> & le <top1>
           name: <n2> & ne <n1>)
 -(student grade: gt <top2> name: ne <n1>)
 then ...)
```

The first two condition elements cause a join involving $O(n^2)$ comparisons, and this is joined with the third (negated) condition element, yielding a complexity of $O(n^3)$. When the top k values are wanted, O(n**k+1) complexity ensues.

Such shortcomings can be avoided by keeping memory nodes sorted, if rule patterns include sorting operators. Once memory nodes are sorted, selection of the top, or the top 2 or 3, etc., elements is fast.

Selection operators

The syntax supports selection of both maximum and minimum sorting sequences, and the selection of the top "n" elements, assuming there are that many. For example:

```
(person age: minimize select 2 to 4)
```

selects persons whose ages, when ranked in ascending order, are the second, third, and fourth in the ranking. This selection ignores the fact that some of the items may have the same sort value. Alternatively, one may instead pick all items having the second thru fourth unique values, using the following variation:

```
(person age: minimize select-values 2 to 4)
```

Sorting over arbitrary expressions

The sorts described so far sort on the value of one attribute of one working memory element. In general, the sort can be done on an expression involving multiple attributes from multiple working memory elements. Consider the following example where `prodigy-score` is a Lisp function:

```
(person age: <a>
  piano-skill-level: <p>
  & maximize (prodigy-score <a> <p>))
```

This would pick the top person by some combination of skill and early age.

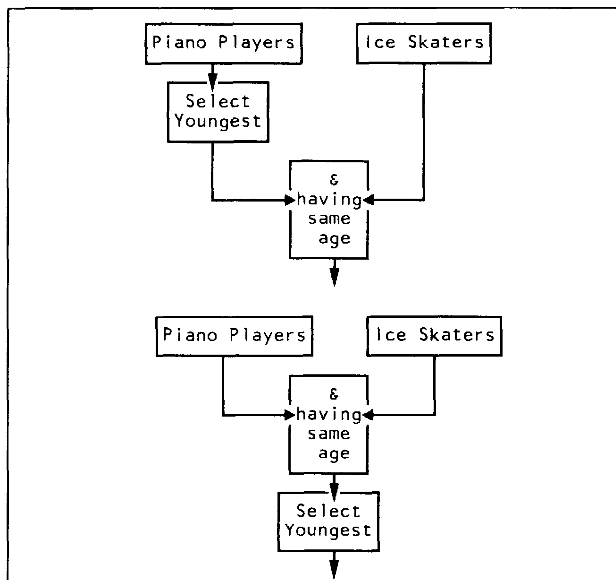
Placement of selection in the RETE

The following examples illustrate the importance of placing the sorting and selection operators at the proper point in the RETE. Grouping of condition elements is required to achieve correct placement. Consider the following two rules:

```
(p same-age-wonder-kids1
  when
   (person skill: piano-player age: <x>)
                           minimize <x>
   (person skill: ice-skater   age: <x>)
  then ...)

(p same-age-wonder-kids2
  when
  ((person skill: piano-player age: <x>)
   (person skill: ice-skater   age: <x>))
                           minimize <x>
  then ...)
```

These build the following RETE fragments:



The first case picks the youngest piano-player, who, let us suppose, is 4 years old. If there are no ice-skaters who are 4 years old, then *the join in the first case is empty*, because the ages do not match. The second case first forms pairs of same-aged piano-players and ice-skaters, and then, from that set, picks the youngest. The grouping construct described earlier is required to give the correct meaning to the sorting constructs.

Sorting over subsets of a memory node

In many cases of picking the maximum, we want to find the maximum over subsets of a memory node. For example, suppose we wanted to know the oldest speaker of each language:

OPS5 method:

```
(p oldest-speaker
 when
  (person language: <l> age: <a>)
  -(person language: <l> age: gt <a>)
 then  ...)
```

YES/OPS method:

```
(p oldest-speaker
 when
  (person language: <l>
      age: FOR-UNIQUE <l> maximize)
 then ...)
```

Without the FOR-UNIQUE clause, the `maximize` would merely find the oldest person. Relational database query languages, for example, SQL [DAT1], support this same notion of determining subsets over which to apply group operations, like maximum. The subset classification is done on the basis of unique values for attributes, or for some expressions involving one or more attributes.

Sorting extensions being considered

The `select` operation for sorted memory nodes can be extended to select the top half, etc. The goal is to eventually specify a fixed interface for selection to enable the user to use his own particular notion.

Sorting is only one of many operations that can be done on subsets of a memory node. Other examples we are investigating are the common operations available from relational database, such as counting the number in the subset, computing the average, selecting the item closest to the mean, etc. The eventual goal is to provide the tools to allow the user to write his own group operations as needed to augment the ones supplied by the system.

Procedural match augments data-driven match

In OPS5, in order to reference any working memory attribute value, the working memory has to be matched by a condition element in the rule's pattern. This invokes all the same RETE machinery that make the rule sensitive to changes in data matching that pattern. Often, this causes unwanted triggering, and is not the way the rule writer initially conceives of the knowledge. Consider the following example rule to print lists of language translators:

```
(p translators1
 when
   (goal type: print-translators)
   (language from: <from-lang> to: <to-lang>)
   (person translate-from: <from-lang>
             translate-to: <to-lang>
                     name: <n>)
 then
   (say <n> can translate <from-language>
                       to <to-language>))
```

Some of the characteristics of this knowledge representation for printing translators are: The `goal` working memory element can't be removed by this rule when the task is completed; a "cleanup" rule must also be written that fires when all the instantiations of the `translators1` rule have fired, presumably by being less specific than this rule. To print "headings" for the list, another rule must be written that will fire *before* this one to print the headers.

Allowing matching in the action part of a rule alleviates these problems. The rule writer can choose whether to make a match be a triggering condition or not. The following example does a procedural match, iterating over all matches of the language-persons combination.

```
(p translators2
 when
<g> (goal type: print-translators)
      ; This is the trigger condition
 then
      ; print heading once
  (say Source-language Target-language Person)
  (for-all-matches-of
    (language from: <from-lang> to: <to-lang>)
    (person translate-from: <from-lang>
              translate-to: <to-lang>
                      name: <n>)
   do
    (say <from-lang> <to-lang> <n>)
  )
  (remove <g>)) ;one rule fires, goal removed
```

The pattern matching work to find all languages and persons and compute their join is not done *until the rule has fired.*

Implementation of procedural matching

A mini-RETE is created for the match expression. For efficiency reasons, the compilation of the mini-RETE is delayed until the first time the match is called for. This mini-RETE is then built in such a way as to reuse, wherever possible, partial matches already present in the main RETE. It is *temporarily* grafted onto the main RETE, and the partial matches present at the graft points form the starting point for computing the match. This section of the added RETE is "turned off" after the procedural match execution takes place, and only "turned on" again when the rule fires again. In this manner the procedural matching isn't done again until (and unless) the rule fires again.

New rules matching existing working memory data

In OPS5, if one compiles a large set of rules, then does many `makes`, then starts to run the production system and discovers a bug in one

of the rules, one is prohibited from simply fixing the rule and recompiling it, since it would not match against existing working memory. The same problem pertains when writing a "debugging" rule in the middle of a run to try and determine the cause of some bug. The debugging rule *doesn't* match existing working memory and is therefore not of much help at finding problems with existing data.

The procedural matching ability in YES/OPS allows rules to be added *after* working memory has been defined, and *these added rules match the existing working memory elements.* For example, the following rule could be added after the production system had started running, to "catch" the rule that changes one spouse to be divorced but not the other one, assuming that it wasn't obvious by inspection.

```
(p catch-unfinished-divorces
 when
  (person name: <s1> marital-stat: divorced  )
  (person name: <s2> marital-stat: ne divorced
       spouse: <s1>)
 then priority 100    ;a high rule priority
  (say the culprit has been found!)
  (back 1)     ;run back to the previous state
  (halt))      ;and stop
```

Without having the rule match existing data, the knowledge of the spouse-spouse join would be missing, if it existed before the rule was added. The `priority` specification causes this rule to fire earlier than other rules in the conflict set, assuming we want to be notified of the condition as soon as it appears.

Building new rules as a rule action

An interesting consequence of this feature is that rules can be added, or existing rules changed, while running, by the action part of some other rule, and they will match existing data. This feature can be used in constructing self-modifying rule systems (a form of learning), although we have not yet experimented with this.

Implementation of incremental rule addition

The incremental rule addition handles its matching in a similar way to the procedural match discussed above. A mini-RETE is created for the new rule, sharing existing RETE structures if previously compiled rules contain matching patterns that can be reused by the new rule. This new RETE is then grafted onto the existing RETE; in this case, the new addition to the RETE is permanent; the new nodes are not "turned off" when the match is complete. Existing memory nodes at the points where the new mini-RETE is added are pushed down through the new part of RETE, thus matching the new rule's patterns with existing working memory elements.

Performance of YES/OPS

Using YES/OPS, small projects done so far have exhibited orders of magnitude improvement in certain cases, even when the new extensions are minimally used. A subset of the rules of a large OPS5 system was converted to YES/OPS, without being rewritten to take advantage of the new `modify` technology. It ran approximately 20 CPU seconds in OPS5, but only 2 CPU seconds in YES/OPS. Furthermore, a slight expansion of the problem (more working memory elements) increased the OPS5 time by 30%, while the YES/OPS time increased only about 5%. The performance comparison can be made arbitrarily good by increasing the size of the problem.

The performance improvements come from five factors: The `modify` as update-in-place substantially reduces the flags that must be set and tested to control rule re-triggering. The grouping construct allows more sharing of pattern tests in the RETE. The sorted memory nodes trade algorithms of complexity $O(n \log n)$ for $O(n ** k+1)$, for the operations of selecting the best k elements from a set of alternatives, an often used function. The procedural matching, done on demand instead of included in the RETE match and updated at every change of the data, reduces the number of patterns that are active to just those that are required to trigger the actions. And, finally, the internal structure of the RETE representation and the algorithms were timed and tuned carefully.

Summary

These ideas have been implemented in an experimental production system language, YES/OPS [SCH1], built using LISP/VM [IBM1]. The guiding principles in the design of YES/OPS include

- the development of clean semantics, designed for data-driven production system applications,
- full integration with the underlying procedural language(s) (e.g., LISP/VM), including communication with other languages and environments (for example, GDDM (Graphical Data Display Manager) and the XEDIT editor),
- generality in rule expression, and
- efficiency of space and time, especially for large production systems.

Other features of YES/OPS include

- `When-no-longer-true`, which triggers actions when an instantiation, having once matched working memory, *later ceases to match*. This is useful for catching conditions that have no other explicit means to determine when they happen.
- Rule priorities, which allow ordering of rules to fire, in addition to conflict resolution. Rule priorities can be numeric, or expressions involving working memory attribute values in the instantiation being considered in conflict resolution.

Some of these ideas have also been incorporated into another experimental production system language extension on top of PL/1, YES/L1 [MIL1].

Many people at the IBM Yorktown Research Center participated in the discussions that evolved into these extensions. The ideas, support, and encouragement of Dr. Se June Hong are gratefully acknowledged.

References

ART1
Bruce Clayton
ART Programming Tutorial
Inference Corporation, March 15, 1985

BRO1
Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin
Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming
Addison-Wesley, 1985

DAT1
C. J. Date
An Introduction to Database Systems
Second Edition, Addison-Wesley, 1977

FOR1
Charles Forgy
OPS5 User's Manual
Department of Computer Science, Carnegie-Mellon University, 1981

FOR2
Charles Forgy
"RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem"
Artificial Intelligence, Volume 19, pp. 17-37, 1982

FOR3
Charles Forgy
"The OPS83 Report"
Technical Report CMU-CS-84-133, Department of Computer Science, Carnegie-Mellon University
May 1984

IBM1

Cyril Alberga, Martin Mikelson and Mark Wegman
LISP/VM User's Guide
IBM SH20-6477, October 1985

MIL1

K.R. Milliken, A.V. Cruise, R.L. Ennis, J.L. Hellerstein, M.J. Masullo, M. Rosenbloom, and H.M. Van Woerkom,
YES/L1: A Language for Implementing Real-Time Expert Systems,
Technical Report RC-11500, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, 1986

SCH1

Marshall I. Schor, Timothy P. Daly, Ho Soo Lee, and Beth R. Tibbitts
"YES/OPS Extensions to OPS5: Language and Environment"
Technical Report RC-11900, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, 1986

YAP1

L. Allen
"YAPS: Yet Another Production System"
Technical Report TR-1146, Department of Computer Science, University of Maryland, Feb. 1982