TREAT: A BETTER MATCH ALGORITHM
FOR AI PRODUCTION SYSTEMS;
LONG VERSION

Daniel P. Miranker

Technical Report AI TR87-58
July 1987

This paper contains a more detailed presentation of the TREAT match algorithm for AI production systems than appears in the proceedings of AAAI-87 as well as an additional section sketching the compilation of the algorithm for sequential machines. The TREAT algorithm introduces a new method of state saving in production system interpreters called conflict-set support. Also presented are the results of an empirical study comparing the performance of the TREAT match with the commonly assumed best algorithm for this problem, the RETE match. On five different OPS5 production system programs TREAT outperformed RETE, often by more than fifty percent. This supports an unsubstantiated conjecture made by McDermott, Newell and Moore, that the state saving mechanism employed in the RETE match, condition-element support, may not be worthwhile.
### Table of Contents

1. Introduction  
2. The Development of Matching algorithms  
   2.1. Analogy with Relational Database systems  
   2.2. Temporal Redundancy  
   2.3. Methods for Introducing State into Match Algorithms  
   2.4. McDermott, Newell and Moore’s Conjecture  
3. The RETE Algorithm  
   3.1. The Details of the RETE Match  
   3.2. Tradeoff’s in RETE  
4. TREAT: A New Production System Matching Algorithm  
   4.1. Exploiting Conflict Set Support  
   4.2. Handling Negated Condition Elements  
   4.3. Detailed TREAT Algorithm  
   4.4. Join Optimization  
   4.5. An Example using TREAT  
5. Comparative Performance of TREAT and RETE on a Sequential Machine  
   5.1. Synopsis of the OPS5 Programs Used as Benchmarks  
   5.2. Counting Comparisons for Variable Bindings  
6. Compiling the TREAT Algorithm for a Sequential Machine  
7. Future Work  
8. Conclusion  
9. Acknowledgements
1. Introduction

Production systems are the basis of many expert systems [McDermott 81, Stolfo and Vesonder 82, Griesmer 84]. The growing use of expert systems is well known as is their large computational requirements. Thus it is important to search for more efficient ways to execute production system programs.

In general, a production system [Newell 73, Rychener 76] is defined by a set of rules, or productions, that form the production memory together with a database of current assertions, called the working memory (WM). Each production has two parts, the left-hand side (LHS) and the right-hand side, (RHS). The LHS contains a conjunction of pattern elements that are matched against the working memory. The RHS contains directives updating the working memory by adding or removing facts, and directives affecting external side effects, such as reading or writing an I/O channel.

In operation, a production system interpreter repeatedly executes the following cycle of operations:

1. Match. For each rule, compare the LHS against the current WM. Each subset of WM elements satisfying a rule’s LHS is called an instantiation. All instantiations are enumerated to form the conflict set.

2. Select. From the conflict set, choose a subset of instantiations according to some predefined criteria. In practice a single instantiation is selected from the conflict set on the basis of the recency of the matched data in the WM.

3. Act. Execute the actions in the RHS of the rules indicated by the selected instantiations.

In general much of the working memory of a production system remains unchanged across production system cycles. Therefore it is worthwhile for the production system interpreter to incrementally compute the contents of the conflict set. The RETE match [Forgy 82], briefly outlined in section, has been commonly assumed to be the best algorithm for this problem. However, the literature contains no comparative analysis of the RETE match with any other algorithm and a conjecture made by McDermott, Newell and Moore [McDermottetal 78], suggests that the state saving mechanism employed in the RETE match, condition-element support, may not be worthwhile. Section describes several methods for introducing state into a production system interpreter, including a new method incorporated into the TREAT algorithm called conflict-set support. Section describes the TREAT algorithm. Section presents the results of an empirical study comparing the performance of RETE and TREAT for the execution of five different OPS5 programs. For all five programs TREAT required fewer comparisons to do variable binding than RETE. In two instances TREAT required fewer than half.
2. The Development of Matching algorithms

We will use the OPS5 style production system rule and initial working memory in Figure 2-1 for several examples in the text. In the LHS of the rule the capital letters represent constants, the characters in brackets pattern variables. Though not illustrated in the example, condition elements may be negated and attributes identified by name rather than position.

**Rule:**

(P example-rule
 (A <x>)
 (B <x> <y>)
 (C <y>)
 -->
 ; no RHS actions)

**Initial Working Memory:**

(A 1)

(B 1 2)

(B 2 3)

(B 2 4)

(C 3)

(C 2)

Figure 2-1: Example Rule System

2.1. Analogy with Relational Database systems

A convenient way to describe the primitive operations in a production system algorithm is to make an analogy to more familiar relational database terminology. If the working memory elements of a production system are considered to be tuples of some universal relationship in a relational database, then it becomes apparent that the LHS of a rule in a production system is analogous to a query in a relational database language. Figure 2-2 shows the rule in Figure 2-1 represented as a database query.

Join (Join (Select(A), Select(B)), Select(C))

Figure 2-2: Production System Rule as Database Query

The constants in a single-condition element may be viewed as a relational selection over a database of working memory. We say a working memory element *partially matches* a condition element if it satisfies the select operators or the intra-condition element pattern contraints. Consistent bindings of pattern variables between distinct condition elements may be regarded as a database join operation on the relations formed by the selections. A collection of rules may be viewed as a collection of concurrent database queries. The conflict set as the union of the query results of each of the rules in the system.

2.2. Temporal Redundancy

A production system is considered *temporally redundant* if on each cycle proportionally few changes to working memory are made, hence most of the working memory remains the same from cycle to cycle. R1, which incrementally builds a solution to the VAX configuration problem [McDermott 82], is typical
of temporally redundant production systems. Systems that change large portions of their working memory on each cycle are not temporally redundant. Examples of temporally non-redundant production systems often have the form of searching through large databases, such as ACE [Stolfo and Vesonder 82], or are sensor-based systems as would be found in a robot task, where the current perception of the world is stored in working memory and is constantly changing as the robot moves.

The OPS5 production-system language contains no explicit operators for the manipulation of sets nor does it have a built-in database interface. Thus, OPS5 programs tend to display a high degree of temporal redundancy. It is therefore considered worthwhile to save state between OPS5 production-system cycles and to formulate the low-level matching problem in the following way: Given a set of productions, a set of working memory elements, a conflict set of rule instantiations and a set of changes to the working memory, what is the fastest way to determine the changes to the conflict set?

![Diagram](image)

**Figure 2-3**: Black Box View of a Production System Algorithm

We may also view the matching algorithms themselves as displaying varying degrees of temporal redundancy depending on how much state is stored within the black box and to what extent that state is exploited by the algorithm. Algorithms based on this formulation may be viewed as black boxes into which changes to the working memory enter and changes to the conflict set exit. (See Figure 2-3.)

### 2.3. Methods for Introducing State into Match Algorithms

McDermott, Newell and Moore [McDermott et al. 78] have identified three types of knowledge or state information that may be incorporated into a production-system algorithm to gain efficiency. A fourth method, conflict-set support is at the heart of the TREAT algorithm. In detail the methods are:

- **Condition Membership**: Provides knowledge about the possible satisfaction of each condition
element. Associated with each condition element in the production system is a running count indicating the number of working memory elements partially matching the condition element. A match algorithm that uses condition membership may ignore those rules that are not active. A rule is active when each of its positive condition elements is partially satisfied. Using the database analogy the count represents the size of the relation resulting from the select operations.

- **Memory Support**: Provides knowledge about which working memory elements partially satisfy each condition element. An indexing scheme indicates precisely which subset of working memory partially matches each condition element. Using the database analogy, memory support implies explicitly maintaining a representation of the relation resulting from the select operations. Later this representation will be called an *alpha-memory*.

- **Condition Relationship**: Provides knowledge about the interaction of condition elements within a rule, and the partial satisfaction of rules. Using the database analogy this corresponds to explicitly maintaining the intermediate results of a multiway join.

- **Conflict Set Support**: The conflict set is explicitly retained across production system cycles. By doing so, it is possible to limit the search for new instantiations to those instantiations that contain newly asserted working memory elements.

2.4. McDermott, Newell and Moore's Conjecture

McDermott, Newell and Moore conjectured that the cost of maintaining the state required for condition relationship exceeds the cost of the comparisons that otherwise would have to be recomputed.

"It seems highly likely that for many production systems, the retesting cost will be less than the cost of maintaining the network of sufficient tests." [McDermottetal 78]

3. The RETE Algorithm

The RETE match algorithm, developed by Charles Forgy [Forgy 82], is the most commonly used algorithm for the implementation of production systems. The RETE match incorporates memory support and condition relationship. Until now, no work has been done to repudiate or confirm McDermott et. al.'s conjecture. Despite that conjecture, and a lack of any comparative studies of the RETE match with any other production system algorithm, the RETE match is commonly assumed to be the best algorithm for production system matching.

3.1. The Details of the RETE Match

The RETE algorithm compiles the left-hand sides of the production rules into a discrimination network. The network is in the form of an augmented dataflow network. Changes to the working memory serve as the input to the network. The network, in turn, reports changes to the conflict set. Figure 3-1 shows the RETE network for the rule in Figure 2-1 after the initial working memory is entered and the activity that
transpires when a new working memory element (A 2) is added. The network contains two categories of nodes: test nodes and memory nodes. When a working memory change enters the network, a "plus" or a "minus" sign is appended to the working memory element indicating whether the element is to be added or deleted from the working memory. A pointer to the change, called a token, is then replicated and passed to a number of entry points into the network.

![Diagram of RETE Match Network](image)

**Figure 3-1:** RETE Match Network for the Rule in Figure 2-1.

A primary optimization used when executing database queries is to perform the selects before the joins. The top portion of the RETE network thus contains chains of tests that perform the select operations. Associated with the output arc of such a chain is a memory node called an *alpha-memory*. Those elements that satisfy a chain of tests at the top of the network *partially match* a particular condition element. That satisfaction is represented by a token stored in an alpha-memory, thus forming the memory support part of the algorithm. Minus tokens that have reached an alpha-memory node have a corresponding plus token already present in the alpha-memory; that plus token is removed. Once a token
updates an alpha-memory, it continues to propagate through the network.

Following the alpha-memories are test nodes called the two input test nodes. The two-input test nodes test for consistent variable bindings between two condition elements. In essence the two input nodes incrementally compute the join of memories on their input arcs. When a token enters a two-input node, it is compared against the tokens in the memory on the opposite arc. Any tokens that have consistent variable bindings are paired with the first token to form new tokens that are stored in a token memory, following the test node, called a beta-memory. The beta-memories embody condition relationship information. A beta-memory contains the result of joining the two relations stored on the input arcs to the test node.

Tokens that propagate from the last beta-memory in the network reflect changes to the conflict set.

3.2. Tradeoffs in RETE

The advantages of RETE are; the large amount of stored state minimizes the number of times two WMEs may be compared.1

The disadvantages of RETE are;

1. When a working memory element is removed the state must be unwound, often requiring the repetition of the sequence of operations that were performed upon its addition. Consider removing the element (A 2) that was added in Figure 3-1.

2. The size of the beta-memories may be combinatorially explosive.

3. To maintain the beta-memories the join operations must be performed in fixed order. This order must be determined statically at compile time.

4. Does not use condition support. To maintain consistent state in the network RETE often performs extensive computation for rules that are not active.

The incentive to develop TREAT was created by the difficulties associated with using RETE on DADO, a distributed memory parallel computer [Stolfo and Miranker 84, Gupta 84a]. In a sequential computer, or in a shared-memory parallel computer, RETE tokens may be manipulated by simple writes to memory. In a distributed-memory parallel computer manipulating tokens may involve costly communication steps. Further the size of RETE beta-memories can be combinatorially explosive. The RETE match

---

1Previous presentations of this material indicated that the ability to share work among similar rules, a feature of RETE, was not possible in the TREAT algorithm. This was an error. Sharing is possible in TREAT. The ability to share is a property associated with processing changes through the network depth first. The depth first processing of changes is desirable in sequential implementations. Sharing is generally not desirable in parallel implementations of either algorithm. The development of TREAT was initially motivated by issue in parallel processing. Thus, the ability to share in TREAT was overlooked.
unnecessarily risks exhausting the memory capacity of computers without virtual memory. In a shared-memory parallel computers the manipulation of token memories requires resolving a large consistency and contention problem [Gupta 86].

4. TREAT: A New Production System Matching Algorithm

4.1. Exploiting Conflict Set Support
To exploit conflict set support two observations must be made. Assume for the moment that there are no negated condition elements in the production system. This assumption will be removed in the next section. If the only action of a fired rule is to add a new working memory element, then the conflict set remains the same, except for the addition of new instantiations that contain the new working memory element. That is, the only rules that will become newly instantiated are those with a condition element that will match the new WME. In the example above, adding (A 2) resulted only in instantiations containing (A 2). The second observation is that if the only action of a fired rule is to delete a working memory element, then no new rules will be instantiated, and the only instantiations that will be invalidated and removed from the conflict set will contain the removed working memory element.

Another way to look at conflict set support is to observe that the RETE network stores certain state information redundantly. In Figure 3-1, illustrating the RETE network, tokens representing the pairing of WMEs (A 1) and (B 1 2) are stored in two different beta-memories and, though not illustrated, this pairing is represented third time in conflict set. Thus the conflict set by itself contains much of the information stored in the beta-memories.

The essence of the TREAT algorithm is to exploit these observations. By retaining the conflict set across cycles, additions to working memory may be used as seeds to initiate a constrained search for new instantiations. Deletions are processed by examining the conflict set directly. Any instantiation containing the deleted working memory element is simply removed. There is no matching required to maintain internal state when processing a deletion.

4.2. Handling Negated Condition Elements
The presence of negated condition elements complicates the algorithm slightly. The TREAT algorithm must consider four cases, the addition or deletion of working memory elements that partially match both positive condition elements or negated condition elements. The cases concerning the addition and the deletion of working memory elements partially matching positive condition elements remain unchanged
from the previous section.

The third case arises if the action of a rule firing adds a working memory element that partially matches a negated condition element. In that case there may be some rule instantiations that are invalidated and will have to be removed from the conflict set. To determine which instantiations must be removed from the conflict set, the partially matched, negated condition element is temporarily considered to be positive and the new working memory element is used as a seed to build instantiations of this new rule. These new instantiations are then compared with the conflict set. Any instantiation appearing in the conflict set is removed. Though this case appears expensive, the same instantiations are computed by the RETE network and removed from the conflict set.

The fourth case is when a working memory element is removed and it partially matches a negated condition element. In this case, removing the element may permit rule instantiations to enter the conflict set. These instantiations are precisely those that would enter had the condition element been positive and the working memory element had just been added. There may, however, be another working memory element similar to the removed element that continues to prevent these new instantiations from entering the conflict set. Such an element would necessarily satisfy the negated condition element precisely the same way as the removed element, i.e. of the pattern matching attributes tested by the condition element all the constant values and variable bindings must be the same as the removed element. Before searching for new instantiations of rule, working memory is may be scanned for such an element. In the implementation reported here it was simpler to scan for these at the end of the search. In the five measured systems such an element was never found.

4.3. Detailed TREAT Algorithm

The TREAT algorithm makes use of condition membership, memory support and conflict set support. All the condition elements in a production system are numbered. The number associated with a condition element is called the condition element number (CE-num). Information relevant to condition elements is stored in arrays indexed by CE-num. The alpha-memories of the RETE match form the memory support part of the RETE match. These memories are formed and used by TREAT as well, but rather than existing amorphously in a network they are formed explicitly as a vector, each entry containing an alpha-memory. The alpha-memories are broken into three partitions: old, new-delete and new-add.\(^2\) The old

\(^2\)In the implementations reported here, these are formed by three separate vectors. However, a vector of structures would probably have resulted in better paging characteristics.
partition, or old-mem, contains the known partially matched elements that were processed during the previous cycles. During the act phase, elements are not added to the old-mem but to the memories in the add and delete partitions, new-add-mem and new-del-mem. The calculation of the contents of the alpha-memory could be done by building the top portion of a RETE network. The implementation reported here used a hash function whose argument is the value of the first attribute in an OPS5 working memory element.

In addition to memory support, TREAT takes advantage of condition support. Associated with each rule is a property called *rule-active*. A rule is active if each of its positive condition elements is partially matched by some working memory element. The TREAT algorithm explicitly maintains the set of active rules. Whenever an old-mem is updated a test is made to see if its size has become zero or nonzero. If the critical change is detected, the size of each of the old-mems for the rule is examined and the set of active rules is updated accordingly.

If an alpha-memory of an active rule is altered and the change corresponds to one of the three cases where a search for instantiations is required, then the search takes place among the changed alpha-memory, the one in a "new" partition, and the old-memories that correspond to the remaining condition elements in the rule. Figure 4-1 contains an abstract program for the TREAT algorithm.

### 4.4. Join Optimization

The join operation is commutative and associative. Thus when searching for consistent variable bindings the alpha-memories may be considered in any order. The query-by-example system [Zloof 77] (QBE), suggests the relations should be considered in increasing cardinality. There are many other methods of optimization [Ullman 82]. The QBE method was considered for the TREAT implementation, however in OPS5 the small size of the alpha-memories and the very small number of WM changes per cycle, (an average of 2.5), dictated that for an optimization to be useful it must be simple to compute and result in a deterministic ordering of the alpha-memories. Three possible orderings were studied. Static-ordering, where the alpha-memories where considered in the lexical order of condition elements. Seed-ordering, where the changed alpha-memory is considered first, since in OPS5 these changes are almost always small, considering them first will greatly constrain the search. The third method, based on semi-join reductions, was not successful and will not be detailed. Note that the use of join optimizations allows TREAT to be used effectively for other production system languages. If a system is temporally-nonredundant the search for instantiations may still be performed in a different but still optimal order.
1. Act: Set CHANGES to the WM updates required of the RHS.

2. For each WM change in CHANGES do;
   a. For each condition element, CE_i do;
      • If partially match of the element against CE_i is successful
         and if addition to working memory
         then add WM-element to new-add-mem[CE_i].
         else add WM-element to new-del-mem[CE_i].
      end do;
   end do;


4. For each nonempty del-mem do;
   a. Set cur-ce = CE-num of the selected memory.
   c. If size of old-mem[cur-ce] = 0 then update-rule-active.
   d. Case: If CE corresponding to the new-del-mem is positive or negated.
      i. Positive: Search conflict set for instantiations containing the deleted wm-elements. If
         found remove them.
      ii. Negative: If the affected rule is active, then perform search for new instantiations by
         searching new-del-mem[cur-ce] and the old-mems that correspond to the remaining
         condition elements that are part of the affected rule. Check that the new instantiations
         are not invalidated by elements in old-mem[cur-ce].
   end do;

5. end do;


7. For each nonempty add-mem do;
   a. Set cur-ce = CE-num of the selected memory.
   b. Set old-size = the size of old-mem[cur-ce].
   d. If size of old-mem[cur-ce] = 0 then update-rule-active.
   e. If the rule is active, then perform search for new instantiations by searching new-add-mem[cur-
      ce] and the old-mems that correspond to the CEs of the remaining CEs that are part of the
      affected rule.
   f. Case: If CE corresponding to the del-mem is positive or negated.
      i. Positive: Add these new instantiations to the conflict set
      ii. Negative: Search the conflict set for each of the new instantiations and remove them if
         found.
   end do;

8. end do;

Figure 2-1: Abstract Program Illustrating TREAT
4.5. An Example using TREAT

Figure 4-2 shows the initial state created by the TREAT algorithm as well as the activity during the addition and deletion of a WME (A 2). Notice in the add cycle that the activity of TREAT and RETE are identical except for the maintenance of the beta-memories and that the beta-memories did not contribute constructively to the computation of the new instantiation. During the delete cycle the alpha-memory and the conflict-set are updated directly. For a delete, the RETE match must recompute the token found in the beta-memories and then delete them. To be fair note that had the WME partially matched the "C" branch of the network only the beta-memory in the RETE network would have to be searched, while TREAT would have to search both remaining alpha-memories. The key issue is does the number extra comparisons performed by TREAT while searching for instantiations exceed the number of comparisons performed by RETE while processing deletions? The results of an empirical study of this question are presented in the next section.

5. Comparative Performance of TREAT and RETE on a Sequential Machine

This section presents quantitative measurements of identical runs of OPS5 programs on several different OPS5 interpreters. The RETE-based OPS5 interpreter is the familiar one distributed by Forgy from Carnegie Mellon University. The TREAT-based OPS5 interpreters were written as part of this research.

5.1. Synopsis of the OPS5 Programs Used as Benchmarks

Five OPS5 programs representing a wide variety of characteristics were obtained from diverse sources. Some characteristics of these systems are summarized in Figure 5-1.

- MAB: Monkeys and Bananas is included due to the familiarity brought on by its use as the test program that accompanies the standard release of OPS5 as well as being the primary example in a text book on OPS5 programming [Brownston et al. 85].

- Waltz: A set of rules that add three dimensional labeling information to a two dimensional picture of a blocks world image using Waltz’s method of constraint propagation [Winston 72].

- Mapper: The Manhattan Mapper is the product of a student project at Columbia University. Given starting and destination points in Manhattan and a set of constraints such as the time, weather and budget the Mapper will suggest what it considers to be the optimal way of travel. The Mapper contains a very large working working memory, much of which is used to store long-term knowledge. Nearly the entire Manhattan bus and subway routes are entered into the system at initialization as 1124 working memory elements.

- Mud: A system that was imported from Carnegie Mellon University. The version studied here is the basis for a commercial system to analyze the castings from the holes drilled for oil. It should also be noted that this is precisely the same system used by Gupta [Gupta 86] in his study of parallelism in OPS5.

- Mesgen: A natural-language program written by Karen Kukich at the University of
Figure 4-2: TREAT Illustration

Initial State as Sensed by TREAT

Conflict Set

Select

Activity of TREAT During an Addition

Join y

Activity of TREAT During a Deletion
Pennsylvania that takes Dow Jones figures and converts them into the text heard on the evening news describing the course of trading during the day.

<table>
<thead>
<tr>
<th>Number of Rules</th>
<th>Number of Condition elements</th>
<th>Average Number of Working Memory elements per cycle</th>
<th>Cycles for test run</th>
<th>Average CS size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAB</td>
<td>13</td>
<td>34</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>Mud</td>
<td>884</td>
<td>2134</td>
<td>241</td>
<td>972</td>
</tr>
<tr>
<td>Waltz</td>
<td>33</td>
<td>130</td>
<td>42</td>
<td>71</td>
</tr>
<tr>
<td>Mesgen</td>
<td>155</td>
<td>442</td>
<td>34</td>
<td>138</td>
</tr>
<tr>
<td>Mapper</td>
<td>237</td>
<td>771</td>
<td>1153</td>
<td>84</td>
</tr>
</tbody>
</table>

Figure 5-1: Summary of the Gross Characteristics of the Studied Systems

5.2. Counting Comparisons for Variable Bindings

It has been reported that 90% of the execution time of a production system is spent in the match phase. Evidence indicates that in the RETE-OPS5 implementeration the majority of the match time is spent in performing variable binding and maintaining the beta-memory nodes [Gupta 84b]. The critical difference between the algorithms is the method used to handle variable binding.

The graphs in Figure 5-2 show the number of comparisons required to do variable binding for each of the OPS5 programs for two variations of each algorithm. The bars are normalized to the number of comparisons required by normal execution of the RETE match with sharing. The dark portion of the bars indicates the number of comparisons required during the add cycles, the light portion, the number for the delete cycles. The axis on the right indicates in absolute terms the number of comparisons to do variable bindings.

The first bar in each graph, (RS), represents the performance of the standard release of RETE-based OPS5. In sequential environments, where changes to the working memory are processed depth first, parts of the match for similar rules may share some of the computation. This type of sharing is not an advantage in parallel computers. To see how these algorithms might compare in a parallel environment the sharing feature was turned off. The second bar, (RN), indicates the performance of RETE without sharing. We see that sharing does not contribute significantly if at all to the variable binding phase of the RETE match.

The third bar in each graph, (TL), represents the performance of TREAT without any optimizations and without sharing. Search is performed in lexical order. Depending on the system this version of algorithm

³Not Available
Figure 5-2: The Number of Comparisons Required to Compute Variable Bindings for each OPS5 Implementation.
may perform better or worse than the RETE match. Thus, some run-time optimization is necessary.

The fourth bar, (TS), represents the performance of TREAT using the seed-ordering heuristic, and no sharing. Inspection of the graphs shows that TREAT with seed-ordering always performed better than RETE even on a sequential computer. Except for the Mapper with its very large working memory the algorithm requires roughly half of comparisons required of the RETE match.\footnote{The large WM of the Mapper suggests that a hash method be used to access the memory nodes. Further examination of the data has revealed that if hashing were added to both RETE and TREAT, the TREAT based execution of the Mapper would exhibit a performance improvement closer to that of the other systems.} Note for each successful comparison performed by the RETE match there is the additional expense of maintaining a beta-memory.

6. Compiling the TREAT Algorithm for a Sequential Machine

We have recently been considering compiling rule systems using TREAT and targeting sequential machines. All of the work on TREAT until now has concerned the parallel implementation of rule systems. In a sequential environment things become much simpler. In the parallel environment the alpha-memories were partitioned, (old, new-add and new-del,) in order to partially match all the changes to WM on each cycle and then perform variable binding for different rules in parallel. The sequential implementation processes each change, depth first to completion, before processing the next. Figure 6-1 contains an abstract program for this version of the algorithm.

By considering the sequential version of the TREAT algorithm the similarities with RETE become clearer and it is apparent that we can adopt Forgy's compilation methods and incorporate the sharing feature of RETE into TREAT. The reader is assumed to be familiar with the contents of Forgy's seminal paper on the RETE match [Forgy 82]. In that paper Forgy describes an assembly language notation as the object of a rule compiler. The handling of condition membership support, the single input tests, is the same for both algorithms. It is the arcs that propagate out of the alpha-memories that must be handled differently. In RETE these arcs are merged by the two input test node. For TREAT we must generate an independent chain of tests for each alpha-memory.

Given a rule:

\[
(p \text{ rule-example}
\begin{align*}
(C1 & \ldots) \\
(C2 & \ldots) \\
(C3 & \ldots) \\
\rightarrow & \\
\end{align*}
)\]
1. Act: Set CHANGES to the WM updates required of the RHS.

2. For each WM change in CHANGES do;
   - For each condition element, CE_i do;
     - If partially match of the element against CE_i is successful
     - If WM change is an addition to working memory then do;
       a. add new-element to alpha-mem[i].
       c. If alpha-mem-size[i] = 1 then update-rule-active.
       d. If the rule is active, then perform search for new instantiations starting with the
          new-element and searching the alpha-mems that correspond to the remaining CEs
          that are part of the affected rule.
       e. Case: If CE that partially matched the new element is positive or negated.
          i. Positive: Add these new instantiations to the conflict set.
             ii. Negative: Search the conflict set for each of the new instantiations and
                 remove them if found.
     end do;
   - If WM change is a deletion from working memory then do;
     a. Remove the deleted element from the alpha-memory[i].
     c. If alpha-memory-size[i] = 0 then update-rule-active.
     d. Case: If CE partially matching the deleted element is positive or negated.
        i. Positive: Search conflict set for instantiations containing the deleted wmem-
           element. If found remove them.
        ii. Negative: If the affected rule is active, then perform search for new
            instantiations by searching new-del-mem[cur-ce] and the old-mems that
            correspond to the remaining condition elements that are part of the affected
            rule. Check that the new instantiations are not invalidated by elements in
            old-mem[cur-ce].
   end do;
end do;

**Figure 4-1:** Abstract Program Illustrating Sequential TREAT
Consider that a token propagating from the alpha-memory matching the first condition element, C1, will only be matched against other alpha-memories, no beta-memories. If we generate the instruction sequence that Forgy describes, except leaving out the instructions that add tokens to the beta-memories, then this is almost precisely the sequence of instructions we need to compile for TREAT. Now we must independently generate a list of instructions for the tokens that will propagate from the alpha-memories corresponding to the second and third condition elements. For the second consider the permutation of the rule-example:

\[
\begin{align*}
(p \text{ rule-example} \\
(C2 \ldots) & \quad; C2 \text{ is moved to the front as} \\
(C1 \ldots) & \quad; \text{per the seed ordering heuristic} \\
(C3 \ldots) & \\
\rightarrow & \\
)
\end{align*}
\]

Compiling this rule, the sequence of instructions that RETE requires following the alpha-memory node for condition element C2 is again almost precisely the instructions required by TREAT. However, this sequence of instructions must be linked into the network after the output of the second alpha-memory. For C3 consider the permutation of the rule-example:

\[
\begin{align*}
(p \text{ rule-example} \\
(C3 \ldots) & \quad; C3 \text{ is moved to the front} \\
(C1 \ldots) & \\
(C2 \ldots) & \\
\rightarrow & \\
)
\end{align*}
\]

Clearly the code generated for TREAT will be larger than for RETE. However, there is some opportunity to use the "fork" instructions to incorporate sharing among different parts of different rules as well as to merge parts of the instruction sequences with in a rule. With respect to the system as a whole the extra code space should not be significant and the extra speed afforded by TREAT due to fewer comparisons and the lack of beta-memory maintenance should make this a worthwhile space/time tradeoff.

Here is an explicit example using a rule similar to the one in Forgy's paper:

\[
\begin{align*}
(p \text{ Plus0x} \\
(goal \ ^{\text{type}} \ ^{\text{simplify}} \ ^{\text{object}} \ <n>) \\
(expression \ ^{\text{name}} \ <n> \ ^{\text{arg1}} \ ^{0} \ ^{\text{op}} \ ^{+} \ ^{\text{arg2}} \ <x>) \\
(c3 \ ^{\text{arg2}} \ <x>) \\
\rightarrow & \\
)
\end{align*}
\]
For this the following code would be generated:

```
Root
    fork 1003
    fork 1004
    teqa 0, goal
    teqa 1, simplify
    and (ce:1, offset:2) = (ce:2, offset:1) ; bind <n>
    and (ce:2, offset:5) = (ce:3, offset:2) ; bind <x>
    term plus 0x

1003
    teqa 0, expression
    teln 2, 0
    teqa 3, +
    and (ce:2, offset:1) = (ce:1, offset:2) ; bind <n>
    and (ce:2, offset:5) = (ce:3, offset:2) ; bind <x>
    term plus 0x

1004
    teqa 0, c3
    and (ce:3, offset 2) = (ce:2, offset:5) ; bind <x>
    and (ce:1, offset:2) = (ce:2, offset:1) ; bind <n>
    term plus 0x
```

Forgy's definition of "and" will have to be slightly modified by removing beta-memory maintenance, and adding condition membership support. Variations on this theme are necessary for the other three cases of TREAT.

Also note that an unconditional jump, jump to lshare, may be added at l$ causing some sharing of code for a single rule. Additional use of the "fork" instruction permits sharing of among rules.

7. Future Work

The seed-ordering optimization was developed to meet requirements to have an optimization that was easily computable at run-time and provided some determinism in the ordering of the bindings and tests. The new work on compiling TREAT indicates any join-ordering optimization is feasible without any run-time overhead provided it is determined statically at compile time. There is then opportunity to experiment with a variety of optimization techniques that are not reasonable as dynamic run time optimizations. Additional improvements are likely with the incorporation of optimizations that consider the dependencies of condition elements with each other [Ullman 82].
8. Conclusion

In many cases TREAT without any optimization outperforms the RETE match. With seed-ordering optimization, TREAT always outperforms RETE. In two instances TREAT required less than half of the comparisons to perform variable bindings than RETE. This does not consider the additional cost of maintaining the beta-memories. Since the algorithms are nearly identical in all other respects, it may be concluded that TREAT is a better production system algorithm in both time and space. Further this study supports the conjecture made by McDermott, Newell and Moore.

9. Acknowledgements

The ideas concerning the compilation of TREAT came from discussions with my students Arun Chandra and Bernie Lofaso. Most of the remainder of this work was done while the author was at Columbia University developing the DADO parallel computer. I would like to explicitly thank Sal Stolfo for instigating this research and Richard Reed for his great contributions in implementing the software systems described herein.
References

[Brownston et al. 85]
Brownston L., R. Farrell, E. Kant, and N. Martin.
Addison Wesley, Reading, Massachusetts, 1985.

[Forgy 82] Forgy C. L.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem.

YES/MVS: a continuous real time expert system.

Implementing OPS5 Production Systems on DADO.

Parallelism in Production Systems: The Sources and the Expected Speed-up.

[Gupta 86] Gupta, A.
Parallelism in Production Systems.

[McDermott 81] McDermott J.
R1: The Formative Years.

[McDermott 82] McDermott J.
R1: A Rule Based Configurer of Computer Systems.

The Efficiency of Certain Production System Implementations.
Pattern-directed Inference Systems.

Visual Information Processing.

[Rychener 76] Rychener, M.
Production Systems as a Programming Language for Artificial Intelligence.
Ph.D. Thesis.

[Stolfo and Miranker 84]
Stolfo S. J., and D. P. Miranker.
[Stolfo and Vesonder 82]  
Stolfo S. J., and G. Vesonder.  

[Ullman 82]  
Ullman, J.D.  
Principles of Database Systems.  

[Winston 72]  
Winston, P. H.  
Learning structural descriptions from examples.  
The Psychology of Computer Vision.  

[Zloof 77]  
Zloof, M. M.  
Query-by-Example: A Data Base Language.  