Dynamic Augmentation of Generalized Rete Networks for Demand-Driven Matching and Rule Updating

Ho Soo Lee and Marshall I. Schor

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

Abstract

This paper describes algorithms for dynamically augmenting an existing Rete network having non-empty partial match memories, and describe two uses of this operation: (1) adding a new pattern to the Rete where it both shares match nodes with the existing Rete, and matches existing data, and (2) implementing an efficient method of *demand-driven* (as opposed to data-driven) pattern matching, which makes it possible to match a pattern against existing data when demanded while keeping that pattern out of the normal Rete pattern matching.

The algorithms described are applicable to generalized rete networks in which arbitrary pattern association is allowed. The trade-offs involved in the design are discussed, including considerations for compilation.

AI topic: Production systems, pattern matching
Domain area: Rete algorithm, Rete pattern matching
Language: Common Lisp
Status: Implemented in IBM Enhanced Common Lisp Production System
Effort: Approximately 1 person-year
Impact: Enables faster development and closer fit of language to problems for pattern matching systems; demand-driven matching can be orders of magnitude more efficient than traditional pattern matching.

1. Introduction

The Rete algorithm supports efficient matching between a large number of patterns, and a large amount of slowly changing data [1]. It was originally developed for OPS5 by Charles Forgy and has been described extensively in the literature [2, 3, 4]. In production systems using the OPS5 paradigm, the patterns arise from the left-hand-sides of rules. The match activity in a Rete network occurs when data (represented using Working Memory Elements or WMEs, an object-attribute-value data model) are created, modified, or removed; changes are propagated through the Rete network. The output of the Rete is a set of new matched patterns, and a set of previously matched patterns now no longer matching.

The data matched with the patterns is organized into instances of classes. Each class has a predefined structure, with named attributes. For example, the class `inventory` may have attributes `item-name` and `on-hand-quantity`, etc. Instances of this class might represent a table of items in an inventory.

Two kinds of tests occur in patterns. One kind matches individual instances of classes. For example, a pattern may select `inventory` instances whose `on-hand-quantity` is below a certain number. This kind of pattern has one condition element, and can be written:[1]

```
(inventory  on-hand-quantity: < 100)
```

Other kinds of patterns are built from joins of multiple condition elements. For example, a pattern finding people having the same parents might look like:

```
<c1> (person mother: <m> father: <f> name: <name>)
<c2> (person mother: <m> father: <f> name: ne <name>)
```

This pattern has two condition elements, labeled <c1> and <c2>. The first one matches all people, and the second one matches all people. The join of these two sets is subject to the specified inter-element tests: the mothers and fathers must be the same, while the names must be different. The meaning of join here is the same as its meaning in relational data base technology. The tests associated with each condition element individually are performed in the *alpha* part of the Rete; the join operations are done in the *beta* part of the Rete. When more than two join operations are done, they may be done in many orders. OPS5 restricts this order to be left associative; that is, in a pattern of 4 condition elements, `(c1) (c2) (c3) (c4)`, first `(c1) join (c2)` is computed, then that result is joined with `(c3)`, and then that result is joined with `(c4)`.

In contrast, the *generalized Rete network* allows arbitrary association of patterns; extra parentheses are used to denote arbitrary join association or ordering. For example, the pattern

```
((c1) (c2)) ((c3) (c4))
```

represents joining `(c1)` and `(c2)`, then `(c3)` and `(c4)`, and finally joining the two results together. For a complete description of the Rete, see reference [3]. Match algorithm extensions that support correct operations for generalized Retes are described in [5].

Rete nodes are classified into alpha and beta nodes in the literature. The beta nodes follow the alpha nodes, and are characterized by having partial match memory (also called result memory) preceding them. The alpha nodes in contrast have no partial match memories preceding them. In between the two kinds of nodes are the first partial match memory nodes; beta nodes (with some exceptions) record the successful match data in following result memory structures.

Rete algorithms achieve their efficiencies in two ways: they collect pattern tests that occur in multiple patterns into a discrimination network, sharing (where feasible) tests across multiple patterns, and they preserve successful partial match results as sets of data-tuples matching up to a node in the network. In subsequent join operations, the saved partial match results are used to avoid recomputing partial match sets for non-changing data.

This paper presents algorithms for dynamically adding new patterns, which are not necessarily rules, to existing Rete networks in a way that shares tests and exploits previous match results where feasible. These algorithms are specialized to provide two basic functions: adding/changing rules and a variation of pattern matching, called *demand-driven*, where no match work is done for a particular pattern until it is requested. This latter function corresponds to a traditional data base query operation.

[1] When writing patterns, we use the syntax
```
(class-name attribute-name: optional-test value
            attribute-name: optional-test value ...).
```
Attribute names end with a colon mark(:) and pattern variables are names like <x>, beginning with "<" and ending with ">", as in OPS5.

Although algorithms for adding new patterns to existing Retes are well known, the algorithms presented here differ in that the added patterns take advantage of existing partial match memories in the existing Rete, and they work correctly when the Rete network structure is generalized to allow the join nodes in the beta portion of the Rete to be grouped arbitrarily. Match algorithms supporting generalized Rete networks are described in detail elsewhere [5].

This paper first discusses the idea of compilation applied to Retes. This enables a discussion of design trade-offs that shift computation toward compile-time, achieving a more efficient run-time execution. The concept of the mini-Rete is introduced. We describe the notion of compilation as applied to Rete networks, and show the mini-Rete as a unit of compilation. Section 2 describes what dynamic augmentation of a Rete is, in terms of mini-Retes, and compares it to the OPS5 implementation. The uses of dynamic augmentation are next described. Section 3 presents the algorithms for implementing dynamic augmentation and their variations and design trade-offs. Section 4 addresses how non-shared portion of mini-Retes are updated.

1.1. Rete Compilation

When Retes are built from user written patterns, the pattern is syntactically processed to create Rete structure. This structure may then be further compiled. We use compilation as an abstraction of any operation that moves significant parts of the Rete computation out of the run-time environment and into an one-time operation, done when the pattern is first "processed". This can include moving the "interpreting" of data structures into machine language.

In many "fully compiled" versions of the Rete network, all of the steps involved in processing a change through a Rete are converted into corresponding machine language. Such systems usually cannot merge new patterns compiled separately with existing patterns, since this would involve merging fragments of machine language. The trade-offs we consider are based on a compilation model that keeps Rete nodes as data structures, although the test fragments are indeed pieces of compiled code. This hybrid approach achieves almost all of the efficiency of fully compiled methods, while maintaining the ability to dynamically augment existing Rete structure with new patterns.

1.2. Mini-Retes

Mini-Retes were first described in [4]. We define a *mini-Rete* as the Rete corresponding to one pattern. Patterns in turn consist of one or more condition elements, each of which can match a particular WME. A typical pattern may look like this:

```
(defrule reorder-item-rule
 when
 (reorder   item-name: <c>   valid: yes  threshold: <q>)
 (inventory item-name: <c>
            on-hand-quantity: <a> & < <q>)
 -(hold     item-name: <c>  status: active)
 then
 (issue-reorder <c> <q> <a>)) ;function call with 3 args
```

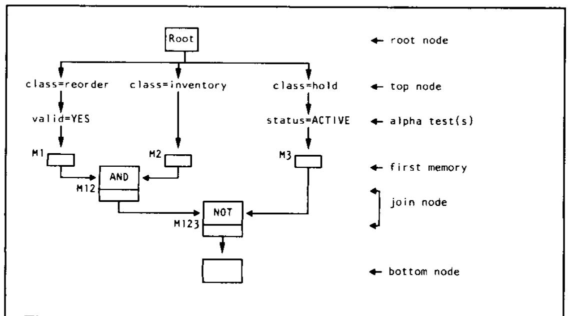


Figure 1. Mini-Rete compiled from the match pattern of a sample rule.

The left-hand-side pattern of the rule `reorder-item-rule` has three condition elements. The last one is negated, meaning the pattern is satisfied only when no WMEs can be found matching the last pattern. Each pattern starts with the name of the class (for example, `reorder`) and can have several attribute tests. This pattern says when there is a valid reorder for an item and the number on-hand of that item is below a threshold, and there is not an active hold on that item, then issue a reorder for the item. The mini-Rete corresponding to this pattern is shown in Figure 1.

All Retes have a root node where data change tokens enter for match processing. The top nodes discriminate initially on the class of the WME (usually using a hash table keyed on the class name). Bottom nodes process the result of completing the pattern matching. bottom nodes are production nodes for rules, as in OPS5, or special MATCH nodes, for demand-driven matching. There is one bottom node per mini-Rete.

Following the alpha tests, the first partial result memory appears and stored in the *first memories* (it is also called the alpha memories in other literatures). This paper presumes the existence of first memory nodes at the juncture between the alpha and beta portions of the Rete. In Figure 1, the nodes, M1, M2 and M3, represent the first memories; the nodes, M12 and M123, are join nodes that include result memories storing partial match results. It also presumes that all Rete nodes have pointers to their predecessor (or for join nodes, their left and right predecessor) nodes. These predecessor links are used in traversing the network as described later.

In this example, the first AND join node includes the test for equality between the `item-name` attribute of `reorder` and the `item-name` attribute of `item`, and also a test for `on-hand-quantity` being below the threshold point.

Mini-Retes are formed as in OPS5. The condition elements may be arbitrarily grouped in other than left associative manner, giving rise to a generalization of the Rete where both the left and right inputs to a join node may themselves come from join nodes. See references [4,5] for a detailed discussion of generalized Rete networks. Where possible, we merge common tests used in the patterns, to both save space when the pattern is compiled, and also to simplify the merge algorithm.

For example, the following pattern has class A self-merged (that is, the test for the class = A is shared for two paths) as shown in Figure 2.

```
(a x: 1)
(b ... )
(a y: 2)
 ...
```

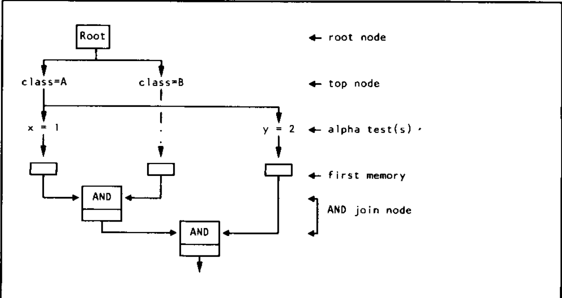


Figure 2. Example of self-merged mini-Rete.

The mini-Rete is the smallest unit of compilation; sets of mini-Retes, merged together, can also be a unit of compilation. When a mini-Rete is compiled, it is converted to a fast-loadable data structure where some parts of the nodes are compiled to machine language representing particular tests being done at the node. The general code that implements the various node types can be compiled separately, and shared for all nodes of that type.

With this model of Rete compilation, we can proceed to a description of dynamic augmentation.

2. Dynamic Augmentation

In OPS5, as new rules are processed, the Rete corresponding to each rule pattern is incrementally built and merged together with other patterns. The Rete representing already merged patterns we designate as the existing *main Rete*. As each pattern is processed, it is merged with this main Rete. We separate this process into its two steps:

1. building the mini-Rete corresponding to a new pattern, and
2. merging that mini-Rete with an existing main Rete.

Dynamic augmentation is the step of taking a mini-Rete and attaching it to the existing main Rete. It consists of two parts:

Step 1 Merging the mini-Rete with the main Rete so as to achieve sharing of common pattern matching tests where feasible.

Step 2 Initializing the part of the mini-Rete that is not sharable with the main Rete so that it matches any existing WME data.

This process is similar to the OPS5 implementation, with two significant differences:

1. OPS5 does not initialize the non-shared part of the newly built Rete with existing partially matched data.
2. The OPS5 Rete structures are restricted to left associative Rete structures, not generalized ones.

We next motivate dynamic augmentation by describing several uses.

2.1. Functional Uses of Dynamic Augmentation

An earlier paper describes the functional uses of dynamic augmentation [4]. We review the uses here. The obvious use is to allow addition of new rules while running a system, where the new rules match existing data. This allows a productive incremental development approach, where the developer may fix bugs in rules in the middle of a run. In addition, systems can be constructed which use programming constructs to add or modify rules while running.

Dynamic augmentation can also form the basic implementation of a demand-driven pattern match function. By demand-driven, we mean pattern matching where no work is done for the particular pattern until it is "demanded". Note that this is a very common function in data base systems; it corresponds to a relational data base query operation. It differs significantly from normal Rete operations in that the Rete normally does all pattern matching only when data changes, on behalf of all patterns present in the Rete. We show a variation of dynamic augmentation that efficiently computes pattern matching on demand, making use of preexisting partial match results that may be already present in the existing main Rete. This kind of demand-driven pattern matching was observed in one application to reduce the computational time for a problem by over an order of magnitude [4].

An advantage of implementing demand-driven matching in this manner, besides potential efficiency gains, is uniformity. The programmer need learn and write only one kind of pattern. If written as the pattern of a rule, it may be used to trigger the rule when data changes cause the pattern to match (data-driven matching). If written in an on-demand context, it may be used to extract the current set of matching data when requested (demand-driven matching).

In the Enhanced Common Lisp Production System product [6], several forms are provided for expressing demand-driven matches. They each take patterns which are identical to the patterns that may be written for rule triggering. The principal ones are:

MATCH This takes a pattern and a set of forms to evaluate. The pattern is used to extract a list of matching WME sets (each set has one WME per non-negated condition element) and the forms are evaluated with variables bound to lists of matched data.

FOR-ALL-MATCHES-OF This takes the same arguments as MATCH. It then iteratively executes its set of forms, rebinding the variables at each iteration to the next set of WME data, and iterates for each set of matched data.

Additional forms such as `remove-match` also use patterns but are expanded in terms of the basic match operations. The following example does a demand-driven pattern match, iterating over all matches of items needed reordering.

```
(defrule reorder-item-demand-driven-rule
 when
  <g> (goal type: list-reorder-items)
 then
  (for-all-matches-of
   (reorder   item-name: <c>  valid: yes threshold: <q>)
   (inventory item-name: <c>
              on-hand-quantity: <a> &  <  <q>)
  -(hold      item-name: <c>  status: active)
   do
   (say "Reorder item" <c>)) ;print items
  (remove <g>)) ;when this rule fires, goal is removed
```

This rule is triggered when there is the single instantiation of WME of the class `goal`. Neither WMEs of the class `reorder` nor those of the class `inventory` are used as triggers for this rule; they are only referenced and matched on demand, when this rule fires. Notice that the mini-Rete compiled by the on-demand match form is reused when this rule fires again later. By virtue of this demand-driven matching we can save much of the unnecessary pattern matching work, which often results in large performance improvements in production systems.

2.2. Outline of Dynamic Augmentation Algorithms

The dynamic augmentation algorithms take as input a mini-Rete representing a new pattern, and an existing main Rete, having perhaps non-empty partial match memories.

The first step merges the mini-Rete with the main Rete, while recording where the mini-Rete starts to differ from the main Rete. This information is recorded in data structures called *synapses*.

The second step matches existing WMEs and partially matched results from the main Rete against the portion of the pattern in the mini-Rete that differed from the main Rete, using the synapses to specify the connection paths between the existing main Rete and the non-shared parts of the mini-Rete.

2.3. Synapse Connections

There are two kinds of synapses: those representing connections from an existing main Rete node in the alpha portion of the Rete, and those representing connections from a beta node in the main Rete. These are called *alpha* and *beta synapses*, respectively.

2.4. Variations in the Augmentation Algorithms

When a new mini-Rete is attached via the augmentation algorithms, it is attached permanently or temporarily. By permanent, we mean attachment such that future WME data changes propagated through the Rete are sent through the new mini-Rete portion as well. This is the kind of augmentation adding a new rule would entail, for example.

Temporary attachment means that the non-shared part of the mini-Rete is not attached as successor nodes to main Rete nodes. Future WME data changes, in this case, are not propagated into the new mini-Rete portion. This kind of augmentation is used to support demand-driven pattern matching activity; it has the property that unless the match is requested, no matching work is done for the unique part of the match pattern.

Both temporary attachment and permanent attachment use the same merge algorithm and create the same synapses. Normally, the synapses are discarded after they are used to initialize the non-shared part of the mini-Rete. In one case, however, they are preserved. This is the case of a temporary attachment, where it is presumed that the on-demand pattern may likely be requested again (that is, the same query operation may be executed multiple times, each time perhaps returning a different answer, based on the current WME data). In this case, the synapses created by the merge of the mini-Rete with the main existing Rete are preserved, and on subsequent match operations for this pattern, the merge step can be skipped.

When the synapses are preserved, algorithms that might delete main Rete nodes as unused (perhaps in response to a delete-rule function) must be kept aware of what Rete nodes have synapse connections to on-demand matches that are being kept for possible reuse.

2.5. *Delaying the Merge Operation for Demand-Driven Matches*

When an on-demand match form is compiled, it generates a mini-Rete, in the same manner as a rule pattern.

When a rule is loaded, the mini-Rete representing the rule's triggering pattern is merged with any existing main Rete, so the rule can become available to be triggered. In contrast, when an on-demand form is loaded, there is a choice of when to merge the mini-Rete representing the pattern with the existing main Rete. We choose to delay the merge step until the match is first called for. This allows a maximal amount of normal Rete structure to be accumulated in the main Rete prior to merging the on-demand mini-Rete. This increases the chance that some other pattern in the main Rete may in fact be already computing parts of the demand-driven match.

3. Merging Mini-Retes with Generalized Rete Structures

We now describe the merge algorithm that merges a self-joined mini-Rete with an existing main Rete, where multiple match condition elements may be left and/or right associated (the generalized Rete structure).

Before presenting the merge algorithm in detail, a simple example is provided to see briefly how mini-Retes are merged into the existing main Rete. Consider the four patterns each of which is compiled into the mini-Rete shown in Figure 3.

```
Match pattern P1: (a) ((b) (c))  ;right associative
Match pattern P2: (b) (c)
Match pattern P3: (b) (a)
Match pattern P4: (c) (d)
```

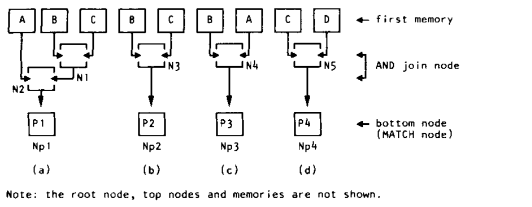


Figure 3. Four mini-Rete networks constructed from match patterns.

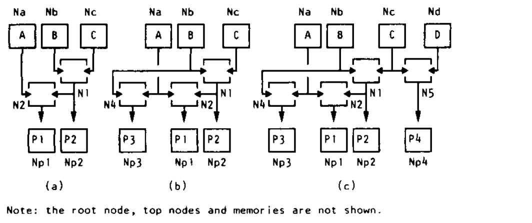


Figure 4. Resulting main Rete networks after adding match patterns in an incremental manner.

When we have multiple mini-Retes to merge, we merge them with the existing main Rete in an incremental manner, i.e., one after another, resulting in a new main Rete for each mini-Rete merge. As soon as a mini-Rete node is determined to be merged, it is coupled to the main Rete. Hence, a main Rete may be dynamically modified during the merge process.

Examples of the dynamic augmentation of Retes are depicted in Figure 4. It is assumed that the initial main Rete is being built from the pattern P1 only as shown in Figure 3(a). Each Rete in (a), (b) and (c) depicts the resulting main Retes after merging each mini-Rete, P2, P3 and P4, to the existing main Rete in an incremental manner. Merging the pattern, P2, augments the main Rete in Figure 4(a), where the node N1 merges the node N3 in the mini-Rete in Figure 3(b). Only the bottom node Np2 corresponds to the non-shared Rete portion. When the pattern P3 is added no join node is shared. Only the first memory nodes, Na and Nb, are shared and the join node N4 in the mini-Rete shown in Figure 3(c) is augmented to the main Rete. The non-shared Rete in this case consists of N4 and the bottom node Np3. Unlike the above two merge cases, adding the pattern, P4. no top node of class D appears in the main Rete. The new top node of the class D is therefore augmented to the main Rete. The join node N5 in Figure 3(d) is also augmented to the main Rete. The non-shared Rete includes N5, Np4, and the top node of D.

A mini-Rete is merged with the existing main Rete such that maximal node sharing is achieved. The merge step attempts to find nodes in the main Rete that duplicate the match node functions in the mini-Rete, in the same sequence from top entry node(s). We describe an algorithm that looks for mergable nodes under the simplifying constraint of not reordering commutative tests. For example, the two patterns

```
(class-a  attribute1: 3  attribute2: 4)
```

and

```
(class-a  attribute2: 4  attribute1: 3)
```

could be merged provided the Retes associated with the chain of alpha tests could be reordered. An extension of this algorithm could be done that would include this aspect, but we exclude this from the present paper.

In describing the following algorithm we denote the final node of a mini-rete that token propagation can reach as the bottom node. Top nodes are nodes where WME changes of particular classes enter the Rete.

We consider a Rete structure having multiple top nodes, one per class, perhaps including one for an indeterminate class. (The indeterminate class is used for patterns having no class specified. For example, the pattern (status: active) means all WMEs of any class having a status attribute whose value is active.). The merge algorithm walks the mini-Rete. The output of this walk is twofold: a set of synapses are created that record the point(s) at which the mini-Rete starts to differ from sharable portions in the main Rete, and the mini-Rete nodes (and under some conditions, the main Rete nodes) are updated to reflect the merged result.

3.1. *Three Degrees of Main Rete Connection to a Mini-Rete*

If a new rule triggering pattern is being added to the main Rete, the main Rete is updated at the synapse points to add the non-shared mini-Rete nodes as successors of the last mergable nodes in the main Rete. This insures that future data changes are propagated through the unique non-shared part of the new mini-Rete.

If the mini-Rete is a demand-driven pattern, the main Rete is not modified to attach the non-shared mini-Rete nodes as successors of the last mergable main Rete node. This prevents future data changes from doing match work on behalf of the demand-driven pattern when the data changes.

If the mini-Rete represents a demand-driven pattern that may be reused, the main Rete is modified to record which nodes could be requested (when a reuse occurs) to serve as inputs to the non-mergable part of the mini-Rete. This marking is used only to prevent deletion of the main Rete nodes, should all other uses of the nodes vanish (due perhaps to rules being deleted).

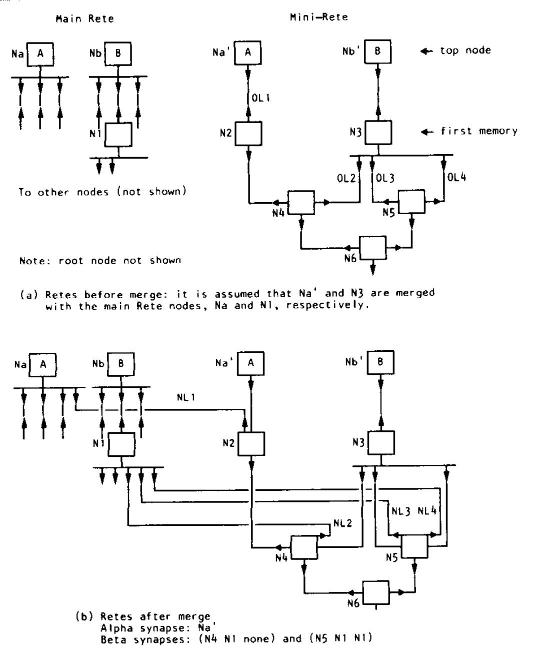
3.2. *Determining Mergability of Individual Mini-Rete Nodes*

Rete nodes can be merged if they perform the same function and have all their predecessors merged. The same function means, more precisely:

- The nodes are the same type (e.g., alpha-test, AND-join, NOT-join, etc.).
- If the nodes contain tests, the tests are the same.
- If the nodes are top nodes, the classes they represent are the same.

In the following discussion, we use the term *non-mergable node* to denote the first non-mergable node in a mini-Rete along some path from a top node. Of course, any successors of this node are also non-mergable, but we are interested in particular in the first one along a path.

The Retes depicted in Figure 5, together with Figure 6 showing detailed merge process, will be used to show several aspects of the merging process discussed in the remainder of this paper. In Figure 5(a), assuming that the first memory node, N2, and beta nodes, N4 and N5, be the first non-mergable nodes leads to node linkings and creation of synapses depicted in (b). In this figure the bidirectional arrows indicate linking from predecessor to successor and vice versa (In the previous figures, only forward links were drawn for simplicity).



(a) Retes before merge: it is assumed that Na' and N3 are merged with the main Rete nodes, Na and N1, respectively.

(b) Retes after merge
Alpha synapse: Na'
Beta synapses: (N4 N1 none) and (N5 N1 N1)

Figure 5. Merge of a mini-Rete with main Rete, showing node linking and synapses.

3.3. *Walking the Mini-Rete*

The mini-Rete is walked from the bottom node to the various top nodes, in a depth-first, left-to-right manner. During this walk, some nodes may be reached multiple times, due to the mini-Rete being self-merged. The algorithm first walks up to the top nodes; the merge processing is done as the walker returns back down. For instance, the mini-Rete nodes in Figure 5(a) are visited in the order they appear in the first column of of Figure 6. When a top node is reached, it is merged with a main Rete top node of the same class (if one doesn't exist, it is created). This merge is always successful.

Sequence	Node Visited	Returning Successful-Merge Flag	Creation of Synapses		Node Linking
			Alpha	Beta	
1	N6				
2	N4				
3	N2				
4	Na'	TRUE			
5	N2 ▼	FALSE	Na'		NL1
6	N4 ▼				
7	N3				
8	Nb'	TRUE			
9	N3 ▼	TRUE			
10	N4 ▼	FALSE		(N4 N1 none)	NL2
11	N6 ▼				
12	N5				
13	N3				
14	Nb'	TRUE			
15	N3 ▼	TRUE			
16	N5 ▼				
17	N3				
18	Nb'	TRUE			
19	N3 ▼	TRUE			
20	N5 ▼	FALSE		(N5 N1 N1)	NL3, NL4
21	N6 ▼	FALSE			

↳ nodes marked with "▼" denotes that they are revisited while walking down from top nodes.

Figure 6. A table showing the merge process of a mini-Rete.

Node merging is defined recursively, using the function MERGE which takes one argument: a mini-Rete node. Initially this is the bottom node of the mini-Rete. MERGE returns two values, a flag indicating successful merge of the argument node, and a pointer to the merged node (if merged) or a pointer to the original argument node.

As MERGE does its work, it has three kinds of side effects:

1. it records the connection points between the main Rete and the first non-mergable nodes in the mini-Rete, using synapses,
2. it updates the pointers in the first non-mergable mini-Rete beta nodes that refer to partial result memories to point to the corresponding main Rete nodes having the partial result memories, and
3. (optionally) it updates the main Rete to include the first non-mergable nodes as successors, when attaching a pattern permanently.

A mini-Rete can have many connection points with a main Rete. Each can occur along a path from the bottom node to the potentially many top nodes. We discuss each connection point individually, while realizing there may be many of them in any particular mini-Rete merge.

3.4. *Constructing Synapses*

The first non-mergable node in a mini-Rete path can occur prior to any partial result memory. This happens, for example, when a new alpha test not already existing in the main Rete is present in the mini-Rete. In this case, an alpha synapse is created which points to the top node of the mini-Rete reachable from the non-mergable node. Because the alpha portion of a Rete has no join nodes, there is a unique top node that leads to this non-mergable node. In the subsequent update step, this alpha synapse specifies which class of WMEs are to be sent through the mini-Rete's top node. WMEs are then sent through the mini-Rete's top node to initialize just the mini-Rete, without affecting the main Rete. Alpha synapses record two pieces of information:

1. the class of the top node, and
2. a pointer to the top node in the mini-Rete

For example, in Figure 5(b), the top node Na' in the mini-Rete becomes an alpha synapse.

When the first non-mergable node in a mini-Rete path occurs after a point where a partial match memory exists in the main Rete, a beta synapse is constructed to record this connection point. A beta synapse consists of the following three pieces of information:

1. a pointer to a mini-Rete node (called *drain node*) that is the first non-mergable node in a chain of nodes from a top node;
2. a pointer to the last successfully merged node (called *right source node*) in the main Rete which is the logical right-input predecessor of the above mini-Rete node; and
3. a pointer to the last successfully merged node (called *left source node*) in the main Rete which is the logical left-input predecessor of the above mini-Rete node.

A beta synapse will be denoted by an ordered list of three items such as:

```
(drain node, right source node, left source node)
```

When a beta join node is the first non-mergable node, it may have one or both of its predecessors be merged. If only its left predecessor is merged, then the beta synapse records `none` for its right source node, and vice versa. As an example, again see Figure 5. If N4 be the first non-mergable node a beta synapse `(N4 N1 none)` is created. Similarly, another beta synapse `(N5 N1 N1)` is created if N5 is also the first non-mergable one.

3.5. Determining Mergable Nodes

When MERGE is called, it first calls itself recursively for the predecessor (or predecessors, in the case of join nodes). These calls each return a successful-merge flag, and a pointer to the main Rete node if a merge occurred. The successful-merge flag is computed based on the diagram in Figure 7.

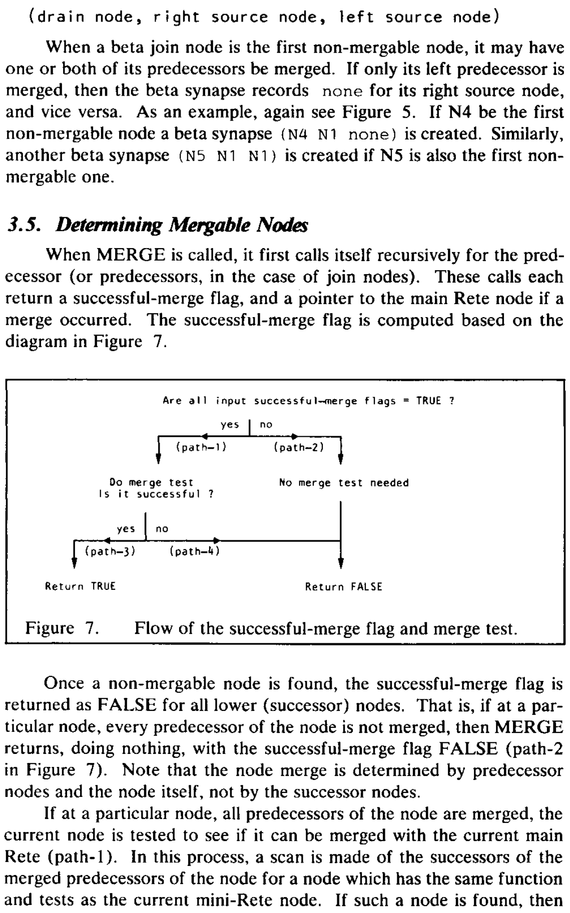


Figure 7. Flow of the successful-merge flag and merge test.

Once a non-mergable node is found, the successful-merge flag is returned as FALSE for all lower (successor) nodes. That is, if at a particular node, every predecessor of the node is not merged, then MERGE returns, doing nothing, with the successful-merge flag FALSE (path-2 in Figure 7). Note that the node merge is determined by predecessor nodes and the node itself, not by the successor nodes.

If at a particular node, all predecessors of the node are merged, the current node is tested to see if it can be merged with the current main Rete (path-1). In this process, a scan is made of the successors of the merged predecessors of the node for a node which has the same function and tests as the current mini-Rete node. If such a node is found, then MERGE returns a pointer to it, and also sets the successful-merge flag TRUE (path-3). Otherwise, it treats the current mini-Rete node as a non-mergable node of this path with the successful-merge flag FALSE (path-4).

If at a particular two-input node, only one of the predecessors is merged, then the current node cannot be merged; it is treated as a non-mergable node with the successful-merge flag FALSE (path-2).

There is one final non-obvious case to consider. This case occurs when the walk of the mini-Rete revisits nodes already processed for merging. This occurs when the mini-Rete is self merged, and has several paths that converge together as the mini-Rete is walked toward the top node(s). This case is detectable by examining the predecessor pointer(s) of the mini-Rete node and seeing if they point to predecessor nodes returned by the recursive call to MERGE. This means that the node was already determined to be a non-mergable node along a previous path from the mini-Rete bottom node. In this case, no further work should be done for this path, so the merge routine returns the current-node and a false successful-merge flag.

As an example of determining mergable nodes in terms of the successful-merge flag, see the mini-Rete nodes in Figure 5(a), and the successful-merge flags returned while walking down the mini-Rete shown in Figure 6.

3.6. Processing Non-Mergable Nodes

When the non-mergable node is discovered, two things happen:

1. A synapse is built that records the connection point
2. Node linking occurs

The kind of node linking varies according to what function is being implemented. In all cases, however, the mini-Rete node predecessor pointer is *updated* to point to the main Rete node; in consequence, the predecessor node in the mini-Rete can no longer be accessed from the first non-mergable node. For example, consider the predecessor pointers in the original node links, OL1, OL2, OL3, and OL4, shown in Figure 5(a). When new synapses are created they are updated so as to point the main Rete merge nodes, Na, N1, N1, and N1, respectively, as depicted in Figure 5(b) while creating new node links, NL1, NL2, NL3 and NL4.

When a new rule pattern is being added, the main Rete node successor list is augmented with the new mini-Rete node. If an on-demand pattern is being processed, this is not done, so that no match work is done for the on-demand pattern, when future data changes are pushed through the Rete. For beta nodes, if an on-demand pattern may be reused, the main Rete beta node is modified to note that this synapse is a potential future user of its partial result memory.

Duplicate synapses may occur when there are multiple paths from a bottom node to the same non-mergable node. In the case of alpha synapses, since the alpha synapse points to a top node in the mini-Rete, different alpha merge paths could also create identical alpha synapses. These duplicates are detected by the merge algorithm earlier so the final list of synapses has no duplicates.

4. Update the Non-Shared Mini-Rete Portion

Having a set of synapses for an attached mini-Rete, we now describe how they are used to update the non-merged part of that Rete. For this purpose we use RULE-Match (Right-Update-Left-Extended) algorithm described in [5]. A proof of the correctness of this algorithm is also provided in the reference.

For beta synapses (connections occurring after a main Rete match memory), match information held in the source nodes is used to initialize non-merged portions of the mini-Rete. For alpha synapses (connections occurring before any main Rete match memories), all WMEs corresponding to a particular top node are pushed through the mini-Rete.

For alpha synapses, since no memory is kept of the results of pattern tests, the tests must be repeated even though some of the alpha nodes may be mergeable. The unique mini-Rete structure from the top node down through the first non-merged node is used to redo the alpha pattern testing, while insuring that results are sent only to the mini-Rete. After this use, if no other on-demand pattern use is anticipated, this portion of the mini-Rete is no longer needed, and is discarded.

4.1. Clearing the Mini-Rete Non-Shared Memories

When the mini-Rete is first attached to the main Rete, its non-shared result memories in the mini-Rete are empty. The update operation pushes tokens through the non-shared part, using an ADD operation, to initialize the non-shared partial match memories.

If the mini-Rete represents a reusable demand-driven pattern match, a subsequent match request needs to update the non-shared result memories. We implement the method of clearing the memories and redoing the update, using the same algorithm as used initially. Alternatively, one could accumulate at the interface synapses, lists of new, modified, and changed partial match tokens, and use these to update the mini-Rete; we viewed this as a poor space/time trade-off.

Given that the mini-Rete is cleared before each update, we chose to do the clear operation following each use, in order to release the memory used to hold the matches sooner.

4.2. Updating a Mini-Rete

RULE-Match algorithm updates non-shared part of a mini-Rete using the existing partial match data, which are stored in source nodes of beta synapses, or in the working memory (in case of alpha synapses) [5]. The update is first done by using the beta synapses, then alpha synapses.

This algorithm requires a specific ordering of the drain nodes of the beta synapses to avoid duplicate or missing join results. The ordering of the drain nodes is derived as a side-effect of the depth-first walk of the mini-Rete from the bottom node. Conceptually a drain node of a beta synapse corresponds to a resume node in the RULE-Match algorithm, and the right (left) source node identified by the beta synapse corresponds to a right (left) stop node. The drain nodes are ordered from deepest (nearest the bottom) to the shallowest (nearest the top nodes). For each ordered beta synapse designating a left input into a drain node, tokens are sent from the result memory of the source node designated in the synapse to the drain node's *left* input, using an ADD operation. Beta synapses designating only a *right* input into a drain node are discarded.

Tokens at the drain nodes and below are processed using the RUL-Match (Right-Update-Left) algorithm for generalized Retes. RUL-Match is a Rete match algorithm used for processing a token when it arrives at a node in a generalized Rete network. For completeness of this paper, RUL-Match is briefly described below. For detailed description of RUL-Match algorithm, refer to [5].

> Form new tokens by joining the token with the opposite predecessor's result memory elements. For each new token created, do the next three steps:
>
> 1. Right distribution: for each right successor nodes, push the new token to its successor nodes, from the shallowest to the deepest order among the successor nodes (ties are arbitrarily assigned). At each node pushed to, apply the RUL-Match algorithm, recursively.
> 2. Update the node's result memory according to the token's operation.
> 3. Left distribution: for each of the left successor nodes, push the new token to successor nodes, from the deepest to shallowest order (again, ties are arbitrarily assigned). At each node pushed to, apply the RUL-Match algorithm recursively.

After the beta synapses are used to update the mini-Rete, the alpha synapses are used in arbitrary order. For each alpha synapse, the WMEs associated with the top node are sent through the mini-Rete alpha pattern match structures and on into the remaining part of the mini-Rete using the RUL-Match algorithm for generalized Retes.

This particular updating algorithm depends on the fact that all beta mini-Rete nodes that could be stop nodes are two input join nodes. This is because partial matches in the main Rete associated with beta synapses are only sent to left-side inputs of join nodes. For common Rete algorithms, such as used in OPS5 and the Enhanced Common LISP Production System [6], this is indeed the case. The approach could be extended to cover other cases, as well.

Summary

We have described several concepts and functions that work together to provide efficient pattern matching operations for adding new rule patterns and doing demand-driven Rete pattern matching in the context of Rete algorithms, where the match pattern is not restricted to be left associated but is arbitrary. Having demand-driven Rete pattern matching extends the existing Rete algorithm in that it accommodates pattern matching when demanded, as well as existing data-driven pattern matching.

We first compile a match pattern into a mini-Rete. A mini-Rete is merged with an existing main Rete by using the merge algorithm. As the result of merge process, a new main Rete and a set of synapses are created. Then, non-shared part of mini-Retes are updated using the synapse nodes.

Among the advantages, some remarkable ones are as follows.

- Although we modify a ruleset we do not need to compile all rules and run the production system again from the beginning. Instead we reflect only the modifications to the existing Rete and update the newly augmented part.
- Demand-driven matching does not activate patterns which are irrelevant to the demand-driven pattern match, which otherwise may trigger a set of unwanted rules.
- Pattern once compiled and used could be reused later when requested.

Several trade-offs have been described involving compilation issues and space/time issues. The algorithms described are essentially those used in the Enhanced Common Lisp Production System product [6], and have proven themselves in practical experience.

References

[1] C. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem", *Artificial Intelligence* **19**, pp. 17-37, 1982.

[2] C. Forgy, OPS5 user's manual, *Technical Report*, Department of Computer Science, Carnegie-Mellon University, 1981.

[3] L. Brownston, R. Farrell, E. Kant and N. Martin, *Programming expert systems in OPS5: an introduction to rule-based programming*, Addison-Wesley, 1985.

[4] M. Schor, T. Daly, H.S. Lee and B. Tibbitts, "Advances in Rete pattern matching", *Proceedings AAAI'86*, Philadelphia, PA, pp. 226-232, 1986.

[5] H.S. Lee and M. Schor, "Match Algorithms for Generalized Rete Networks", submitted for publication, 1989. It is also available as an IBM Research Report RC-14709, 1989.

[6] *Enhanced Common LISP Production System - User's Guide and Reference*, IBM publication number SC38-7016, 1988.