NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A PRODUCTION SYSTEM MONITOR FOR PARALLEL COMPUTERS

Charles L. Forgy
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

April, 1977

Abstract: Production systems cannot compete on an equal basis with conventional programming languages until the efficiency of their monitors is improved. The process that finds the true productions on every cycle is most in need of improvement; even in today's small systems this process is often expensive and it is likely to become more expensive if productions systems become larger. There are a number of possible ways to achieve the greater efficiency, including taking advantage of the slow rate of change of the data base, taking advantage of similarities in structure of the antecedent conditions of productions, expending a minimum of effort on false antecedent conditions, avoiding whenever possible the operations that are likely to be most time consuming, and making efficient use of hardware. Since computer power is achieved most economically today through parallelism, no algorithm can be truly efficient in its use of hardware unless it can be executed in parallel. A production system monitor has been implemented that responds in a reasonable manner to both the peculiar nature of the task and the realities of current hardware technology.

This work was supported by the Defense Advanced Research Projects Agency (F44620-73-C-0074) and monitored by the Air Force Office of Scientific Research.

I. Introduction

Production systems have generated much interest in the Artificial Intelligence community in recent years. In just over a decade they have evolved from little more than an interesting mathematical formalism into tools that have been used to good effect in both practical [2; 10] and theoretical [3; 9] areas. The successes of these and like endeavors argue that the use of production systems will continue to spread, but there are problems that will hinder the spread if they cannot be overcome. One of these problems, the comparatively high cost of using production systems is examined in this paper. The concern here is less with the rather small systems in use today than with the much larger systems that may soon appear.

Although the four systems cited above are like enough to all be called production systems, they differ enough to make it unlikely that efficiency can be achieved in the same manner for all. This paper treats only one kind of production system. It assumes that production systems are programs whose active part comprises a large number of imperative statements called *productions* and whose passive part (i.e., data part) comprises a smaller number of assertions called *data elements*. Data elements are discrete, usually simple symbol structures (list structures, associative structures, etc.). They are held in *working memory*, the single global data base of the system. A production is an ordered pair *(LHS, RHS)*. LHS is a conditional expression; RHS, an unconditional sequence of actions. The condition expressed by each LHS in a production system can be interpreted as the description of some state that working memory may attain. The existence of a production is an assertion that whenever working memory is such that LHS is true, then the actions in RHS are appropriate. Each step in the execution of a production system involves choosing one production that is appropriate to the current state and executing the actions in its RHS. The actions performed in executing a production cause the state of working memory to change, and a new set of LHSs will generally become true after each step. Processing terminates when an unrecognizable state is reached.

Production systems lack much of the structure found in conventional programs. Conventional languages provide statements of various kinds, means to group statements into

larger units (compound statements, etc.), and means to group these into still larger units (programs and subprograms). In most languages means are provided for making the information in one subprogram either completely unavailable or available only with difficulty to the other subprograms. In contrast, if the programmer wishes to incorporate such structures into a production system, he must create the means himself.[1] In a production system all productions bear the same relation to the system as a whole and all have access to the whole of working memory.

An interpreter for production systems is called a *monitor*. The name *recognize-act cycle* has been given to the control cycle of production system monitors. The details of the recognize-act cycle are most easily discussed if it is assumed that there are three distinct phases to the cycle, *match*, *conflict resolution*, and *action*. In match the monitor computes the *conflict set*, the set of all productions with true LHSs. In conflict resolution it applies one or more *conflict resolution rules* to eliminate all but one from the set. Finally in action it executes the actions in the remaining production's RHS. Within the monitor there is often only a single process for the two phases of match and conflict resolution. When the conflict resolution rules are simple, a single process that attends simultaneously to the needs of both phases can usually select the production to execute without computing the entire conflict set. Unfortunately, the two phases cannot always be performed simultaneously; some of the more useful conflict resolution rules cannot reasonably be applied unless the entire conflict set is available to be examined [6]. Thus there are two types of monitors in use, those with three distinct processes for the three phases of match, conflict resolution and action, and those with one process for action and another process that combines match and conflict resolution.

All four of these processes (action is the same in both types of monitors) must be considered in the design of algorithms for monitors, but they do not merit like amounts of effort. The match is expensive to perform even with today's small production systems and is likely to become more expensive if production systems grow larger. Conflict resolution and match together are also expensive now and likely to become more expensive. Conflict resolution alone may be expensive if the rules used are complex or if the size of the conflict set is large, but it is hard to justify a study of conflict resolution. Conflict resolution involves little more than the already extensively studied process of sorting (of the conflict set). Action will probably never be expensive to perform because production systems encourage a style of programming in which only a few simple actions are performed on any one cycle. Thus, most of the effort of a study should be directed toward the processes for match alone and for match and conflict resolution together.

These considerations shape the discussion in this paper. The primary focus of the paper is on increased sophistication in the algorithms for performing the match, with the intent of making the algorithms more responsive to the peculiar nature of the task. The

1 This is not a weakness of production systems. Many of their advantages, including extensibility and modifiability, arise because so much of the structure that is implicit in conventional programs is made explicit and brought out into the open in production systems.

secondary focus is on efficient use of hardware. One of the realities of current hardware technology is that a problem may be solved more economically on many small computers than on one larger computer of comparable power -- provided the power of the multiple small computers can be applied efficiently.

I.2. Introduction to OPS

The examples in this paper use OPS [4], a production system implemented using the techniques to be presented here. Since OPS is new and not widely known, an overview of the language is in order.

OPS is embedded in L*, a list processing system, and all structures within OPS, including both data elements and productions, are list structures. An OPS production has the form

PNAME (CE1 CE2 ... CEm --> AE1 AE2 ... AEn),

where PNAME is the name of the production, the CEi, which constitute the LHS, are *condition elements*, and the AEi, which constitute the RHS, are *action elements*. There are many kinds of actions, but only two, those that add new elements to and those that delete old elements from working memory, are of interest here.[2] Condition elements are patterns to be instantiated by data elements from working memory. A data element may instantiate a condition element if the two are EQUAL (the Lisp function) constants, if the condition element is a variable that can be bound to the data element, or if the corresponding subelements of each match according to this rule. A variable, which is distinguished from a constant by being preceded by "=", may be bound to any single condition subelement. If a variable occurs multiple times within an LHS, all occurrences must receive EQUAL bindings. Thus, the condition element,

(Q 19 =X =X),

which contains two constants, Q and 19, and two occurrences of the variable X, would match the data elements

(Q 19 A A),
(Q 19 (A) (A)), and
(Q 19 19 19),

but not

2 It is sufficient to restrict attention in this discussion of monitors to just these two actions. Any action that transforms working memory can be constructed from these two. An existing data element may be modified, for example, by deleting the old form of the element and adding the new.

(Q 19 A A A),
(Q 19 A (A)),
(Q A A A), or
(19 Q A A).

A condition element may be negated by preceding it with "¬". An LHS is satisfied when all its non-negated condition elements and none of its negated condition elements are instantiated. If negated and non-negated condition elements have variables in common, whether the negated condition elements can be instantiated will often depend on the bindings of the shared variables. For example, the production

P0 ((A =X) ¬ (B =X) ¬ (C =X) --> ...)

has only one legal instantiation (the one binding X to 2) when working memory contains

(A 1),
(A 2),
(A 3),
(B 1), and
(C 3).

Lest this brief introduction give a false impression of the power of OPS and, by association, the power of the methods used in its implementation, it should be noted that the introduction has omitted most features of OPS entirely and presented the rest in their simplest forms. OPS is actually a powerful and flexible production system.

II. The Match: A Model

Figure 1 shows a sample production, P1, a small working memory, WM1, and the data from WM1 matched by P1 (the symbol "::" is read "matches"). In this section the question of how the monitor might find this instantiation of P1 is considered. One answer to the question is given. In the next subsection a broad class of match routines, which we call the *discriminating matches*, is described. Then in subsection II.3 a simple model of the discriminating matches is developed. This model will be of use in the discussion in the rest of the paper.

```
P1 ( (=X) (A =X) (=X B =X) --> ... )

WM1 ( (A 1) (A 2) (A 3) (A 4) (1) (3) (P) (1 2 3) (1 B 2) (1 B 1) )

P1 :: ( (1) (A 1) (1 B 1) )
```

Figure 1. Instantiation of P1.

II.2. Discriminating Matches

Perhaps the most widely used match routine is the one that we will call the *basic match*. The basic match instantiates LHSs one at a time. Beginning with the leftmost condition element of an LHS it tests data elements one at a time until it finds a legal instantiation for the condition element. Then one by one, the remaining condition elements are instantiated in a similar fashion. Since variables are bound as the match proceeds, the condition elements become progressively more difficult to instantiate. When unable to instantiate some condition element, the basic match backs up and attempts to re-instantiate the previous condition element. If it succeeds in finding an instantiation for the entire LHS, it reports that fact and then, since it must find all instantiations, attempts to re-instantiate the last condition element. Exhausting the possible instantiations for the first condition element terminates processing of the LHS. In short, the basic match searches depth first through a space of possible partial instantiations for each LHS.

After making a move in the space, the basic match performs a number of tests and will abandon the point just reached if any one of the tests fails. This, the importance it ascribes to the failure of a test, marks it as belonging to the class of discriminating matches. The discriminating matches include all match routines that:

1. Determine from an LHS the characteristics required of any data matching the LHS.
2. Propose trial instantiations for the LHS.
3. Apply tests based on the characteristics for the purpose of showing the proposed instantiations illegal.
4. Find a proposed instantiation to be legal by virtue of its not having failed any test.

Precisely what forms these characteristics can take depends, of course, on the language implemented. Typical characteristics are shown in Figure 2, which again shows production P1 and the data from WM1 matched by P1. This time the data are labeled with the characteristics that a monitor for OPS would have to check to show the legality of the instantiation.

P1 ((=X) (A =X) (=X B =X) --> ...)

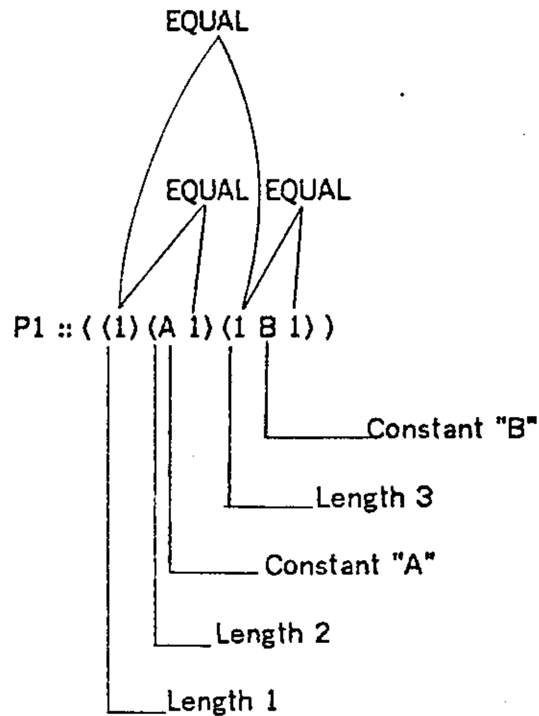

Figure 2. Characteristics of instantiations of P1.

The importance ascribed to the failure of tests by discriminating matches makes them inappropriate for showing the legality of LHSs that contain negated condition elements. In the simple discriminating match, the failure of a single test is taken as proof that a trial instantiation is illegal. If an LHS contains negated condition elements, however, failure of some tests is not only acceptable, but required. Such LHSs are generally dealt with not by extending a discriminating match in some way, but by combining several discriminating matches. As an example, consider what the basic match would do in finding an instantiation for the production

P2 ((A) (B =X) ¬ (A =X) --> ...)

when working memory contains

(A),
(B 1), and
(A A).

It would find the instantiations for the two non-negated condition elements as before. It would then use the same techniques in an attempt to instantiate the negated condition element; in this it would fail, and by this local failure, succeed in its main task of instantiating the LHS. If there were other negated condition elements, the techniques would again be used. In general, an LHS with N negated condition elements requires N+1 discriminating matches, and legality of an instantiation is proven by success of the first and failure of the remaining N.

Even if the special needs of negated condition elements are ignored, the class of discriminating matches fails to be inclusive. Match routines falling properly outside this class include unification algorithms, which cannot make the distinction between patterns and data required by discriminating matches, and the abstracting match in Newell, Shaw, and Simon [8], whose basic operations remove rather than test for differences between patterns and data. Nevertheless, the class is large enough to be interesting, and the discussion in this paper is limited to the discriminating matches.

II.3. The Model

Discriminating matches differ from one another in the order in which they test characteristics, the number of LHSs instantiated at one time, the criteria by which they decide when to give up, the forms taken by their intermediate results, etc. Some of these will be discussed later in the paper, and others will be ignored. The model presented here will aid in focusing attention on the things of interest.

The primary difference to be hidden is the representation chosen for the intermediate results of the match process. While the objects processed by one match routine may differ greatly from those processed by another, they will have one important similarity: they will both contain some information found in the attempt to instantiate one or more LHSs. Any intermediate result can be equated with the set of fully specified instantiations to which it can be extended. The object that represented the intermediate result then, in a sense, describes a set. This object will be called a *set description*. (This will sometimes be shortened to just set or description.) The operations performed by the match then become operations on sets.

Match predicates allow a time independent statement of the processing of sets needed to perform the match. A match predicate is a truth valued function on (descriptions of) sets. Predicates are in a one-one correspondence with the characteristics used by discriminating matches, and a predicate has value true when applied to a set if all elements of the set possess the predicate's characteristic. The responsibility of the match is to compute for each LHS the subset of instantiations that make the LHS's predicates true.

Filters allow expressing the order in which (particular) match routines evaluate match predicates. A filter is a function from descriptions of sets to descriptions of sets. Filters are in a one-one correspondence with predicates, and the result of a filter applied to a set is the

subset that makes the filter's predicate true. Sequential execution of filters is indicated by composition of the filters. If the symbol ⊗ is used to denote composition of functions, and the usual convention of right to left execution is followed, the expression

$$F2 \otimes F1(S)$$

indicates that filter F1 is applied to the set S and then F2 applied to the value produced by F1. To model purely sequential match routines requires only one more piece of machinery, a function to generate the initial set descriptions. This function, π, maps from working memory to the sets of all possible matches. For example, one match for an LHS with five characteristics to be checked, and thus five filters, say, Fa, Fb, Fc, Fd, and Fe, to be processed, would be

$$Fe \otimes Fd \otimes Fc \otimes Fb \otimes Fa \otimes \pi.$$

Parallel[3] execution of filters is modeled by intersection of composed sequences. One possible parallel match for the same LHS is

$$(Fe \otimes Fb \otimes \pi) \cap (Fd \otimes Fc \otimes Fa \otimes \pi),$$

which indicates the sequence Fb-Fe is to be executed in parallel with the sequence Fa-Fc-Fd, and the results of the two sequences intersected to provide the final result.

Sets, match predicates, and sequences of filters constitute the model. Of course, this model hides features in addition to the already mentioned representation of intermediate results. Depending on the representation chosen, the number of LHSs in the system, the number of elements in working memory, the complexity or simplicity of π, etc., the match may have to process many sets. The match may choose to order this processing in any of a number of ways, from processing all sets in parallel to processing each to completion individually. The model does not represent this choice. Neither does it try to represent the modifications to set descriptions that occur as the match progresses. A more general model would include functions that map from set descriptions to more fully specified set descriptions. These functions would occur along with filters in the compositions of functions that define match routines. For the purposes of this paper it will be sufficient to assume that set descriptions are modified by filters. When a filter processes a set whose description it finds in some way inadequate, it creates a new description for the set.

In summary, the model is concerned with time. Match predicates make possible a time independent statement of the responsibility of routines: to compute the conflict set, a match routine must find the instantiations that make each LHS's match predicates true. Time

---

3 Parallel is used in a weak sense to indicate only that there is no logical dependency between the filters and that they could therefore execute in parallel. Conceivably, a monitor could be such that parallel execution is not used, and yet the model of its match for some LHSs would show parallel execution of filters.

dependence of one kind, the order in which particular match routines perform each necessary test, is modeled by compositions of functions. Time dependence of another kind, the relative order in which sets are processed, is ignored by the model.

III. Issues

Some characteristics of production systems that affect the efficiency of monitors are discussed in this section. Suggestions are given as to how monitors should respond to the characteristics. No attempt is made to evaluate the relative importance of the characteristics. Because the relative importance varies with factors such as the number of productions in the system, the complexity of the LHSs, and the number of elements in working memory, such an attempt would be futile. Many of the characteristics have received attention elsewhere; see, for example, McDermott, Newell, and Moore [7] and Hayes-Roth and Mostow [5].

III.2. Selectivity of Filters

In a composition of filters like

$$F5 \otimes F4 \otimes F3 \otimes F2 \otimes F1 \otimes \pi$$

the first filter will process every set in existence, the second will process only those passed by the first, the third only those passed by both the first and second, and so on. If there are N sets to be processed and the probability that filter Fj will pass any given set is Pj, the expected number of filter evaluations is

$$N(1 + P1 + P1P2 + P1P2P3 + P1P2P3P4)$$
$$= N(1 + P1(1 + P2(1 + P3(1 + P4)))).$$

Other compositions will result in expressions with similar form but with the positions of the Pi's interchanged. The above will have the minimum value of all the expressions if

$$P1 \leq P2 \leq P3 \leq P4 \leq P5.$$

Hence, if the filters are ordered by their selectivity, most selective first, the number of required filter evaluations is minimized.

Unfortunately, expected total cost of filter evaluation, which is a more realistic measure than expected number of filter evaluations, is not so easily minimized. An expression similar to the above, but incorporating factors for cost of filter evaluation is easy to construct. Without some knowledge of the relative costs and of the Pi's, however, little information can be extracted from the expression.

III.3. Structural Similarity

Because a production can perform only a limited amount of processing, most tasks will require more than one production. The productions for each task will certainly have much commonality of structure. There will be LHSs containing identical condition elements, condition elements containing identical subelements, condition elements with identical lengths, and LHSs containing similar patterns of variable occurrences. Whichever of these forms it takes, commonality of structure has only one effect on the match: it causes multiple occurrences of match predicates. If there exist two filters based on the same match predicate and always processing the same input sets, the two will always produce the same results, and the two filters will be completely equivalent. The match would do well to avoid the evaluation of more than one of a group of equivalent filters.

A match routine needs two abilities to take full advantage of multiply occurring match predicates. The ability to recognize equivalence of filters is basic, but not in itself sufficient. Also needed is the ability to bring about equivalence. Since equivalence of filters depends on the filters' inputs as well as on their predicates, a rearrangement of the filters for the LHSs of two productions will sometimes cause nearly equivalent filters to become fully equivalent. For example, even though there are three pairs of filters performing identical tests in the following two sequences:

$$Fa \otimes Fb \otimes Fc \otimes Fd \otimes \pi$$
$$Fa \otimes Fb \otimes Fc \otimes \pi$$

none of the filters are equivalent. Rearranging the first sequence to

$$Fa \otimes Fb \otimes Fd \otimes Fc \otimes \pi$$

makes the filters of one pair equivalent. Again rearranging to

$$Fd \otimes Fa \otimes Fb \otimes Fc \otimes \pi$$

makes the filters of all three pairs equivalent.

III.4. Temporal Redundancy

One philosophy of production system use holds that working memories should be small. Even when this philosophy is followed and working memory is kept small in an absolute sense, it will almost certainly be large relative to the number of actions in a typical RHS. When the philosophy is not followed, working memory may be orders of magnitude larger. Thus the number of elements changed on any cycle will be small compared to the number left unchanged, and the match routine will find it advantageous to process only the changed elements. If it has retained the information (i.e., the set descriptions) from the previous cycle, this is possible. To revise the information from the previous cycle it need

only delete from the sets all references to the elements that were just deleted from working memory and add, where appropriate, references to the elements that were just added.

III.5. Match Predicate Semantics

The three occurrences of the variable X in the production

P3 (=X (=X =X) --> ...)

make possible six match predicates concerned with equality of variable bindings: those requiring equal bindings for the first and second, the first and third, the second and third, the third and second, the third and first, and the second and first occurrences of X. The match need not use all six. Because EQUAL is a commutative, transitive relation, any group of predicates that together mention all three occurrences of X captures the semantics of all six predicates. This is only one example of the fact that most LHSs admit of more than one formulation for their match. Certainly, some formulations will be preferable to others; some will include fewer predicates; some, predicates that are less expensive to evaluate; and some, predicates that are more likely to occur in other LHSs. Strangely, many match routines ignore these differences and choose formulations at random. For example, in processing P3 the basic match would use the predicates requiring equal bindings for the first and second and the first and third occurrences of X. Section III.7 will explain why this is not the best choice.

III.6. Independence of Predicates

The last section pointed out that the match can take advantage of redundant predicates. It seems reasonable also to try to take advantage of nearly redundant predicates, that is, of those predicates whose truth is usually, but not always, implied by that of others. Such predicates typically arise from the use of non-minimal encodings in the condition elements. For example, if the constant "ISA" appears as the first subelement of a condition element, the probability is high that the condition element will have three subelements. An appropriate response to the existence of nearly redundant predicates is to process the filters based on those predicates first. This order is justified by the principle of section III.2; if one predicate is seldom true unless all of another group of predicates is true, the filter for that predicate will be almost as selective by itself as the collection of all filters for the other predicates. It will therefore probably be the most selective of the filters.

III.7. Entropy of Set Descriptions

When a set is tested by a filter or intersected with another set, energy is expended and information is created. If this information is not to be lost, it must be encoded in the descriptions of any sets that result from the processing. The descriptions of the resulting

sets, containing as they do both the information created during the processing and the information encoded in the descriptions of the original sets, represent the elements of the sets more precisely than did the descriptions of the original sets. Sometimes innocuous, this increase in specificity can at other times make more work for the later stages of the match. When the result of the increase is to make a set's description so precise that it is incapable of covering all elements of the set, the set must be split into several smaller sets whose descriptions are of like specificity, but whose contents can be covered by a single description. The new sets, though smaller than the original set, take individually as much effort to process on the succeeding steps as the original set would have taken if it could have been processed as a whole.

Filters decrease something besides the ability of descriptions to cover many elements. Before the first step, sets are large and descriptions imprecise. As the match proceeds, elements are eliminated, set descriptions become more precise, and the number of elements to be covered decreases at the same time that the covering power of descriptions decreases. The two effects often counteract one another. It becomes necessary to split sets into smaller sets only when the ability of set descriptions to cover elements decreases faster than the number of elements to be covered.

It can be shown that only one type of filter can make necessary the splitting of sets, but to do so directly without appealing to a particular representation is difficult. To argue about the effect of the splitting on the total effort of the match is less difficult, and that path is taken here. Because their condition elements can be processed individually, LHSs with no variables common to multiple condition elements require effort directly proportional to the number of condition elements that they contain. Some LHSs with variables common to multiple condition elements require effort that is an exponential function of the number of condition elements. Examples of these are LHSs that solve NP-complete problems such as the K-clique problem [1] by recognition. Thus, the efficiency of the match is greatly affected by (some) filters that test for equality of bindings to variables.

The monitor can eliminate some of these filters and can take steps to lessen the impact on efficiency of the rest. When there exist alternative formulations of the match (see section III.5), the monitor can chose the formulation that results in the fewest such filters. Those it cannot eliminate can be delayed until late in the match; the reduction in number of candidate instantiations brought about by the earlier filters will then minimize the number of highly specified sets these filters must produce.

The mistake made by the simple match routine in section III.5 can now be explained. It chose to use the predicates requiring equal bindings for the first and second and first and third occurrences of X. Since the second and third occurrences are within the same condition element, a better choice would have been to use the predicates requiring equal bindings for the first and second and second and third occurrences.

III.8. Context of Filters

The final observation to be made here is that filters do not work in isolation. The set of filters that work together to perform the match for an LHS (i.e., the set that belong to one composed sequence) will be called a *context*. In the parallel match example used earlier,

$$(Fe \otimes Fb \otimes \pi) \cap (Fd \otimes Fc \otimes Fa \otimes \pi),$$

all five filters belong to the same context. Filters that are shared in response to structural similarity may belong to many contexts. Certainly, if the filters of a context are to work together, they must communicate some information among themselves. The minimum to communicate is the descriptions of the sets passing the tests at each filter. But to communicate only the minimum makes the match less efficient than it could be. For example, if one filter shows an LHS to be unsatisfiable and if it communicates that fact to the other filters in the same context, those filters can cease to process their sets. Any results they might produce while the LHS is unsatisfiable would be of no use. When the first filter determines that changes to working memory have made the LHS possibly satisfiable again and communicates that fact to the other filters, they can resume work. Nothing will have been lost by this, but some unnecessary filter evaluations will have been avoided.

There is an interaction between this use of context and temporal redundancy. If the monitor takes advantage of temporal redundancy and saves the results of all computations, there is a likelihood that there will eventually be a need for the results of any computation. The importance of context is thus lessened. But because data elements eventually leave working memory, it is not completely eliminated. If a filter is inactive for a long period, an element can enter working memory, remain for a time, and then leave, all within the period of inactivity of the filter.

IV. The Rete Match

The match routine to be described here, the rete match, makes use of a process that can reasonably be called compilation of productions. Before beginning interpretation of a production system, the monitor compiles the LHSs of the system into a program to perform the match for that one production system. Since the compiler builds into the program all information from the LHSs needed to perform the match, the LHSs need never be examined again.

The bulk of this section is devoted to a description of the programs built by the LHS compiler. Particular attention is paid to the response of the rete match to the issues of the last section. Since ordinary compiling techniques are quite sufficient, there will be no discussion of the LHS compiler.

IV.2. Basic Organization

The programs constructed by the LHS compiler are based on the data flow model. A data flow program takes the form of a graph structure in which the nodes are procedures and the edges communication paths. Entities called tokens flow through the graph communicating both data and control between nodes. A node, normally inactive, becomes active upon the arrival of a token on one of its inputs, performs some computation, and when the computation is completed, becomes inactive again. If the node has results to communicate to other nodes it creates tokens to hold the results and sends the tokens out along the appropriate edges. If an attempt is made to send a token to a node that is unable to accept the token immediately, the token will be queued by the edge carrying it until the node is ready. A data flow program interacts with its environment by sending tokens to and receiving tokens from the environment.

This description of data flow programs in general is a correct, if superficial, description of the programs constructed by the LHS compiler. As will become apparent when the rete match is described in more detail, the nature of the task has made it worthwhile to deviate from the data flow model in a few particulars.

Even at this level of detail, the program is somewhat constrained by the nature of the task. The nodes, or at least some of the nodes, will contain filters -- perhaps one per node, perhaps several per node. The tokens passed between nodes will contain set descriptions. The nodes receiving input from the environment will be those containing the first level of filters, and will receive tokens whose set descriptions are of sets of all candidate instantiations. The nodes giving outputs to the environment will be those containing the last level of filters, and will output tokens whose set descriptions are of sets of fully verified instantiations.

The program that the OPS LHS compiler would construct for production P1 is shown in Figure 3. If the reader assumes for the moment that the tokens are simply working memory elements and lists of working memory elements, he can see how the match would proceed. Assume the elements in WM1 are deposited into an initially empty working memory. Tokens representing all ten elements will be sent to each of the three nodes at the bottom of the graph. Each node will produce one output token for each input passed by the filter at that node. The output tokens will be identical to the input tokens. The output of the node testing "Length 3" will be tokens representing

(1 2 3),
(1 B 2), and
(1 B 1).

These are sent to the node testing "Constant 'B'-Position 2", which examines the second subelement (this is the meaning of "Position 2") of the datum of the tokens and produces one output for each that has a "B" in this position. The outputs of this node will be tokens representing

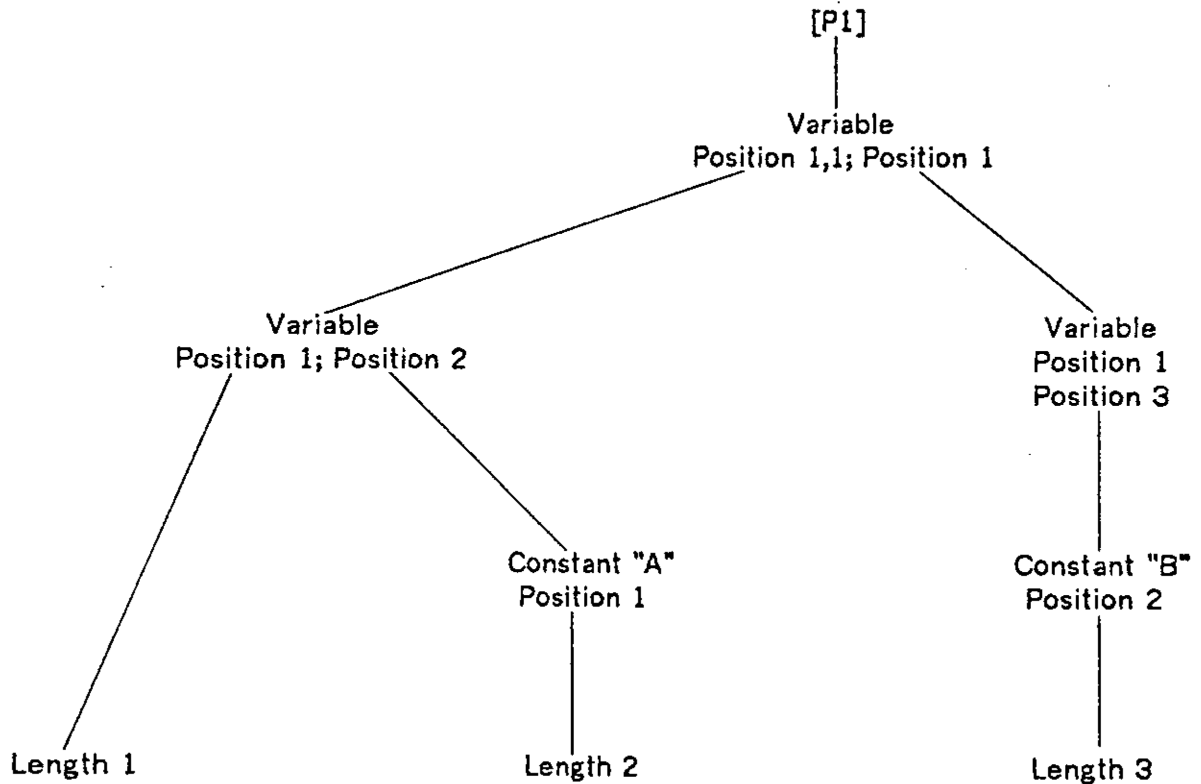

Figure 3. Match routine for P1.

(1 B 2) and
(1 B 1).

These are sent to the node testing "Variable-Position 1-Position 3". "Position 1" and "Position 3" indicate that the first and third subelements are to be tested. This node tests for equality of the subelements in these two positions. The node produces only one output, a token representing

(1 B 1).

The node to which this token is sent, "Variable-Position 1,1;Position 1", differs from the previous three nodes in that there are two edges leading into it. At the right input it receives the token whose processing has just been described,

(1 B 1).

At the other input it will receive two tokens,

((1) (A 1)) and
((3) (A 3)).

The process by which these two tokens arrive at the left input is identical to that in the part of the graph being described. The node compares the first subelement of the first data element of each token on the left (i.e., "Position 1,1") with the first subelement of the data elements of each token on the right ("Position 1"). There are two pairs of tokens compared in this example. Only one pair has equal subelements and passes the test,

((1) (A 1)) and (1 B 1).

By appending the data of the left token to the data of the right token, the node builds a single token to represent the pair. The data of the new token is

((1) (A 1) (1 B 1)).

There are no nodes to which the token can be sent.

Because there are no more filters to evaluate, any token output by this node is a legal instantiation for production P1. The association between P1 and the node is shown in Figure 3 by placing P1 in brackets above the node. (In the programs actually constructed by the OPS LHS compiler, the information that a given node is the last node for some production is stored in the node itself.) Whenever this node creates a token to output, it informs the processes for conflict resolution that the data of the token is a legal instantiation for P1. The processes for conflict resolution must then modify the conflict set accordingly.

IV.3. Parallelism

There is much potential for parallel execution within the rete match. Every token in the graph represents a potential locus of control that can be processed independently of (in parallel with) the other loci. When a node is activated by the arrival of a token, the processing it undertakes involves the data from that token and possibly other data already stored in the node, but nothing from elsewhere in the graph. Any activity at the other nodes in the net, including the creation or destruction of other tokens, is of no concern to this node. The only limit to the parallelism, other than the physical limit imposed by the machine on which the program is run, is that a node can process only one token at a time.

There is potential for parallel execution even in the processing of a single LHS. As can be seen in Figure 3, the processing for each condition element is kept independent of that for the other condition elements as long as possible. In OPS, the only dependencies between condition elements are those arising because the condition elements have variables in common. The filters concerned with the bindings to these variables are processed last. Logically, since the nodes containing these filters combine the results of other filters processed in parallel, these nodes must also perform set intersections. In fact, the

intersections, and thus the parallelism, are free, a side effect of the evaluation of this kind of filter on the set representations used.

The rete match will support even greater degrees of parallelism, though at some cost. Any sequence of nodes in a graph may be split into two or more parallel sequences if other nodes are provided to intersect the results of the several paths. The cost of the additional parallelism is the cost of the intersection nodes, the space they occupy and the time required for their execution.

IV.4. Number of Inputs

Nodes with any number of inputs are possible, but OPS makes use only of nodes with one input and with two inputs. The subgraph that performs the match for any one LHS takes the form of a tree rooted at the last node. Although it is impossible to build non-degenerate trees using nodes of one input only, it is possible to build the equivalent (in some sense) of any tree using both nodes of one input and nodes of two inputs. Nodes with higher numbers of inputs can be simulated by grouping several nodes with two inputs. For example, a node with three inputs can be simulated by two nodes with two inputs; a node with four inputs by three nodes with two inputs. Thus, OPS is not restricted in having only two kinds of nodes. Nodes with one input are called *one-input nodes*; nodes with two inputs, *two-input nodes*.

The number of inputs that a node should have depends on its function. Nodes that perform set intersection must have two inputs (or more) to get the sets to intersect. All other nodes need only one input. As noted in the previous subsection, some nodes perform implicit set intersections. These must have two inputs just as the nodes that perform explicit intersections must.

IV.5. Ordering Filters

Ordering of filters, which Section III showed essential to efficiency in the match, is almost completely unconstrained in the rete match. Since compilation is performed only once, the compiler can spend much time choosing a reasonable ordering. It has complete freedom in effecting the chosen ordering; it can put filters in different nodes and order by linking the nodes appropriately, or it can put filters in the same node and order by specifying the node appropriately. Generally, it will use a combination of the two methods.

IV.6. Temporal Redundancy

The rete match occupies an extreme point on the store/recompute spectrum, choosing to store everything and recompute nothing. At the beginning of each cycle, the processes that maintain working memory inform the process for the match of all changes made to working memory on the previous cycle (by inputting tokens representing the changed

elements to the first level of nodes). The match processes map from these changes to changes in the set descriptions retained from the previous cycle and from there to changes in the conflict set. They then inform conflict resolution of the changes in the conflict set and rely on conflict resolution to make the changes.

The space required to store the sets from cycle to cycle is not excessive; only two-input nodes store information. A one-input node need save no state from one cycle to the next because the tests it performs are independent of the history of the node. Whether a given token will pass the tests at a one-input node depends only on the contents of the token. Whether a token will pass the tests at a two-input node depends on both the contents of that token and the contents of the tokens that arrived earlier on the node's other input. Consider again the example program in Figure 3. Recall that the final node received two inputs on the left,

((1) (A 1)) and (1)
((3) (A 3)) (2)

and one input on the right,

(1 B 1). (3)

Suppose these had arrived in the order, (1) -- (3) -- (2). When (1) arrived there would be nothing from the other side with which it could be compared. The token would have to be stored until something arrived. When (3) arrived it could be compared with (1). The tests would succeed and an output token would be produced. When (2) arrived it could be compared with (3). Since the tests would fail, no output would be produced. This token would then be stored away to await the arrival of other tokens on the right input. The processing of a two-input node can be described more formally. When processing a newly arrived token, a two-input node intersects the set of the new token with the sets of all tokens that arrived earlier on the other input. It produces one output for each resulting set that passes all the filters at the node. It then stores away the new token so that the set of the token may be intersected with the sets of tokens that arrive later on the other input. The token must be stored as long as it remains valid.

When an element leaves working memory, all sets depending on that element for their validity must be deleted from the node memories. The match finds and deletes these sets using methods that differ only slightly from the methods used in generating the sets initially. When an element is deleted from working memory, tokens are created and input to the bottom of the net as before. This time, however, the tokens are tagged as representing an outgoing working memory element. These tokens pass through the graph as before, moving along the same arcs, activating the same nodes, and causing the nodes to produce outputs that are, with one exception, identical to earlier outputs. The exception is that all outputs are tagged, as the initial tokens were, as resulting from the processing of an outgoing working memory element. When a node with an input memory processes a token with such a tag, instead of storing the token in its memory, it deletes from the memory a token that is

identical except for the tag (i.e., the token stored when the element entered working memory). When conflict resolution receives a token with such a tag, it understands the token to mean that an LHS has just become false and modifies the conflict set accordingly. The final result of processing the tagged tokens is to delete from the node memories all sets made invalid (and from the conflict set all productions whose LHSs became false) without examining any node memories unnecessarily.

Adds and deletes are processed simultaneously in the rete match. The processing of one does not interfere with the processing of the other.

IV.7. Structural Similarity

The response of discriminating matches in general to structural similarity is to share filters; the response of the rete match in particular is to share nodes. Sharing of nodes is effected through the use of multiple output edges. When a node has two or more edges outgoing, the work of both that node and all its predecessor nodes is shared. Figure 4 shows how the graph of Figure 3 would be modified to allow processing of the production

P4 ((=Y) (B =Y =Y) --> ...).

This production, with condition elements nearly identical to those of P1, shares six of its seven nodes with P1. One of the nodes shared is a variable node -- even though the names of the variables in P1 and P4 differ. As it should be, the OPS LHS compiler is insensitive to the renaming of bound variables.

IV.8. Context

The rete match can be quite responsive to context. Since there are explicit links (the edges in the graph) between nodes in the same context, communication between the nodes presents no difficulties. This section examines the questions of what types of information should be communicated and how a node is to determine that it has some information of value to the other nodes. Only the common use of context is considered: allowing one node to inform the others that it has shown the LHSs to be unsatisfiable and that they may therefore suspend processing.

An agreement about directions within the graph will be of use in the discussion of context. The following discussion will assume the conventions that the direction in which tokens travel is up and that the inputs of a two-input node are left and right.

The use of tokens, which communicate control as well as data between nodes, gives the match already described some sensitivity to context. If a token moving upward reaches a node that has shown the LHSs above to be unsatisfiable, the node will produce no output, and processing of the set represented by the token will be suspended. For example,

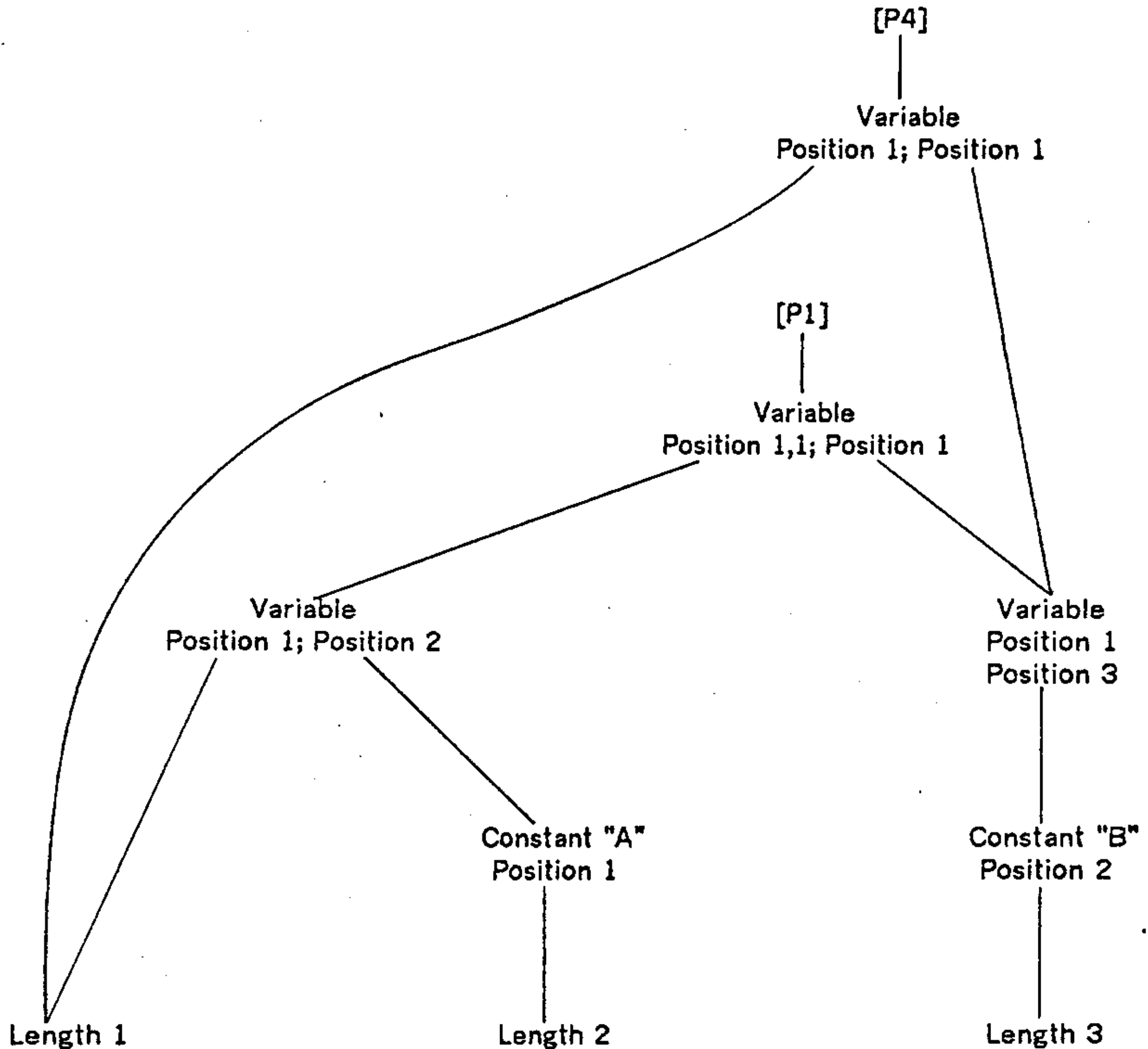

Figure 4. Match routine for P1 and P4.

suppose the token reaches a two-input node that has nothing in the memory of the other side. For most types of two-input nodes this guarantees that the tests will fail if performed. If this node is one of those, it will produce no output at that time. Later, if a set arrives on the other input and if the tests at the node are all successful, a token will be output and processing will resume. In effect, each node helps to determine the context of its successor nodes.

The sensitivity to context provided by the use of tokens is limited by their ability to pass control upward only. Certainly, if nodes can pass control upward only, each node can control only those nodes lying directly above itself. If nodes could pass control both up and down, a control signal originating at any node in a context could propagate to every other

node in the context. Thus a full sensitivity to context requires extensions to the already described mechanisms.

One typical extension requires that the communication paths (the edges in the graph) be made bidirectional and two bits of state be added to each node. The two bits of state allow four states for each node, *passive*, *full active*, *left active*, and *right active*. A passive node accepts no inputs; it relies on the edges leading into the node to queue all arriving tokens. A full active node accepts and processes all inputs. A left active node accepts and processes only the tokens arriving on its left input; a right active, only those arriving on its right. Nodes with associated productions are always in one of the active states. All other nodes are passive until made active by one or more of their immediate successors. A full active node signals both its predecessor nodes to become active, a left active signals its left predecessor, and a right active signals its right predecessor. A node signaled to become active chooses for itself which of the three active states to assume. A one-input node has no choice; all active states are equivalent for one-input nodes. A two-input node makes its choice based on the current state of its input memories. A node with at least one token in each of its memories becomes full active. A node with tokens only in the left memory becomes right active. A node with tokens only in the right memory becomes left active. A node with no tokens in either memory can be either left or right active -- it does not matter which. Nodes are sensitive to the arrival of new control signals just as they are sensitive to the arrival of tokens. When a control signal arrives, the nodes awakens, examines the signal and the contents of its memories, and if necessary, changes state.

A change in state may result in the node's initiating further activity. If it makes one (or both) of its inputs active, it will process any tokens that have been queued on the input. In addition, it will send a signal to the node below that input telling it to become active. If it makes one of its inputs passive, it will send a signal to the node below telling it to become passive.

Using a very conservative definition of unsatisfiability, these rules try to maintain a single active path from each node associated with an unsatisfiable production to the first level of nodes. The rules take advantage of the fact that a two-input node will produce no outputs while either of its memories is empty.[4] Their criterion of unsatisfiability of a production is the existence of one node in the production's subgraph that has an empty memory. If such a node exists, the rules suspend all activity in the subgraph below the node except that which can result in something being deposited into the empty memory. To see how this works, consider the graph in Figure 3. When processing first begins, before anything has been deposited into working memory, only the top node is active. It is a two-input node with two empty input memories so it may become either left or right active. Assume right active is chosen. It will activate the one-input node below its right input. This node will activate the node below itself, and that node will activate the bottom node on the path. No changes in state will occur until the token for

(1 B 1)

4 This will be qualified somewhat in Subsection IV.10.

reaches the top node. This node will then have one token in its right memory and none in its left; it will become left active. Passing signals down both sides, it will deactivate the nodes below it on the right and activate those below it on the left. Because there is a two-input node below it on the left that has two empty input memories, not all the nodes become active. As processing proceeds, however, the lower two-input node will get tokens in first one and then the other of its two memories and become full active. Similarly, the top node will finally receive a token on its left input and become full active. Finally, the entire subgraph for P1 will be active. P1 will then be shown satisfied. The entire subgraph for any production will become active before the production is shown satisfied.

Sometimes nodes do not become passive when signaled to do so. If the graph is processed in parallel, a node may process many tokens before the signal arrives. Because some nodes are shared, a node may not be able to become passive after the signal arrives; if even one of its successor nodes requires it to be active, it must remain active. Since a node stops accepting tokens immediately after passivating an input, the edge below that input must be ready to queue any arriving tokens.

The edge should do more than just queue the tokens. If the tokens arriving at an inactive input were simply queued, when the input was reactivated the node would process exactly what it would have processed if the input had been continuously active. Much of this processing might have been avoided if the edge leading into the node had the power to delete tokens. Deletion under certain circumstances is warranted. It can be expected that if an input remains inactive for any significant period of time, there will be queued up some pairs of tokens, one token resulting from the entry of an element into working memory and the other resulting from the departure of the same element. There is little reason for processing such pairs. If the edges in the graph are given enough processing power to locate and delete these pairs, the match becomes quite effective at taking advantage of context.

Both simpler and more complex implementations of context sensitivity are possible. A simpler implementation might use fewer states or work for only certain kinds of nodes (perhaps only for the two-input nodes). A more complex implementation might try to take advantage of sharing of nodes. For productions that share no nodes the implementation described will maintain whenever possible a single active path from the node associated with the production to one node on the lowest level of the graph. Since it does not take into account the existence of other active paths when setting up a new path, however, it will sometimes make several active paths for a production sharing nodes with other productions. For the same reason, it will sometimes set up separate active paths for two productions that share nodes and could therefore share one active path.

IV.9. Computing Partial Conflict Sets

The preceding sections have described a version of the rete match that computes the entire conflict set. That is the most useful form of the match; most conflict resolution rules

demand that the match work this way. Some, however, will allow the match to compute only part of the conflict set. The rule that chooses on the basis of an ordering on the productions is a common example. This rule assumes there to be a total ordering on the productions. It chooses the first production in the order that has an instantiation. When this rule is used, it is advantageous to look first for instantiations of the first production, then for instantiations of the second, and so on. The match can then terminate after the first instantiation is found.

A simple modification to the context mechanisms will allow the rete match to compute the conflict set in this incremental manner. In the earlier version of the context mechanism, all nodes that had associated productions were made permanently active. In the modified mechanism, they are activated one at a time -- or several at a time if the conflict resolution rule is less selective than the production order rule described above -- as the match proceeds. At the beginning of the match, the nodes with associated productions are all made passive except the node associated with the dominant production. Then the match is performed as before. When the program quiesces, if there is an instantiation for the production, the match phase is ended. If not, another node associated with a production is activated. Because there are now tokens queued at the lower level nodes, there is no need for the processes that maintain working memory to again input tokens. Processing begins again as soon as the activation signals have filtered down a few levels. If there is no instantiation found after the program again quiesces, another node is activated. This continues until either an instantiation is found or there are no more nodes to activate.

IV.10. Negated Condition Elements

Since the rete match will support languages that, like OPS, allow negated condition elements, it is not a pure discriminating match.

In the rete match, the multiple discriminating matches required by LHSs containing negated condition elements coexist within the single graph structure (and sometimes share nodes). For an LHS with one negated condition element, there will be one subgraph for the non-negated condition elements, another for the negated condition element, and one node to combine the results of the two subgraphs. Except at this last node, the negated condition element is treated by the match as if it were a non-negated condition element. That is, the match tries to find legal instantiations for the negated condition element. If any are found, the filters at the last node check for consistency of variable bindings between these and any instantiations found for the non-negated part of the LHS. An instantiation for the non-negated part is a legal instantiation for the entire LHS only if there are no instantiations for the negated condition element with consistent variable bindings.

The nodes for combining discriminating matches are similar to the nodes responsible for checking for consistency of variable bindings between non-negated condition elements. Nodes of both kinds have two inputs, receive inputs from similar subgraphs, and test only variable bindings. The criterion for success of one node is the criterion for failure of the other. One outputs tokens with data parts identical to those of the input tokens; the other,

tokens with augmented data parts. That these two node types should be so similar is not unreasonable when one considers the similarities in productions like

P5 ((A =X =X) (B =X =Z) --> ...) and
P6 ((A =X =X) ¬ (B =X =Z) --> ...).

These nodes differ in some ways from the other nodes; it is necessary to make changes in the mechanisms for context and temporal redundancy if they are to operate at these nodes. Since neither the presence nor the absence of a possible instantiation for a negated condition element can make an LHS unsatisfiable, the context mechanisms must ignore these in deciding whether to passivate other nodes. Since satisfiability is an absolute, to take advantage of temporal redundancy at these nodes requires that all changes be noted, but only some be acted upon. Only the changes that make an instantiation change from legal to illegal or illegal to legal should be acted upon. For example, if working memory contained

(A 1 1),
(B 1 1), and
(B 1 2),

P6 would have no legal instantiations. If (B 1 1) were deleted from working memory, it would still have no legal instantiations. But if (B 1 2) were then also deleted, it would finally have a legal instantiation. The common way to take advantage of temporal redundancy at these nodes is to keep a count of the number of times an instantiation is made illegal. The count is adjusted as necessary in response to changes in working memory. When the count changes from zero to one, an instantiation becomes illegal. When the count changes from one to zero, the instantiation becomes legal. No other changes are of immediate consequence. In the example above, the original count for the instantiation was two. After the first deletion it became one. After the second deletion it became zero and the instantiation was thereby shown legal.

V. Correctness of the Rete Match

The correctness of the rete match is less than obvious; there are the parallel execution of the program, the simultaneous processing of adds to and deletes from working memory, the changing of state of nodes in response to changes in context, the occasional deletion of tokens by the communication paths, etc. Certainly anyone who intends to use the rete match would be well advised to convince himself of the correctness -- and thereby to determine the details to which he must attend to preserve the correctness. The following paragraphs sketch out proofs by which this correctness can be established. (Full proofs cannot be given because these involve case analysis, and the cases depend on the language implemented.)

Inductive arguments of correctness are appropriate with these graph structured programs. In such an argument, the first step is to show that the individual components of the graph, the nodes and edges, always produce the correct output when given the correct input. Then it must be shown that the match will not terminate before all tokens have been processed. Assuming the bottom nodes receive the correct inputs from the processes maintaining working memory, the correctness of the match as a whole then follows by an inductive argument.

The first programs to consider are those in which the edges are nothing more than queues. In these programs each node receives exactly the tokens produced by its predecessor node, in exactly the order produced. Given that its inputs arrive in order, all that must be shown to establish the correctness of a one-input node is that it correctly implements its filters. A two-input node must in addition be shown to be independent of the relative order of arrival of tokens on its two inputs. For while the edges maintain the correct order at each individual input, the parallel execution of the program as a whole means there can be no guarantee of relative order on different inputs. Since this independence of order is achieved through the use of input memories, a proof of independence is primarily a proof of correct use of the input memories.

Because it executes in parallel, the graph may pass through incorrect states, but it need never terminate in an incorrect state. For example, suppose a node receives two tokens on some cycle, the first of which adds something to the memory of one side, and the second of which deletes something from the memory of the other side. It is possible that the arrival of the first will make it appear that an output token should be produced and the arrival of the second show that it was incorrect to produce the output. If the token had already been output when the second token arrived, the part of the graph above this node would be in an incorrect state. To correct the mistake, the node could create another token like the first except tagged as resulting from an outgoing working memory element. To the nodes above, it would appear as if a working memory element had been added and then immediately deleted. If these nodes reacted correctly to the two tokens, the graph would be in a correct state when the match process terminated. Fortunately, it is not necessary that the nodes have the ability to recognize when they or their predecessors have made mistakes. It is sufficient that they treat the arrival of each token as an individual event. If every node processes each token to completion, modifying its internal memories and producing all outputs that appear necessary before turning its attention to the next token, mistakes will be corrected without ever being recognized as mistakes.[5]

The correctness arguments become more complicated when edges are given the power to delete tokens. Because this moves some of the processing out of the nodes and into the edges leading into the nodes, it becomes difficult to treat nodes and edges

5 Since even the highest nodes can receive incorrect inputs and produce incorrect outputs, conflict resolution must cooperate with the match and be ready to accept information to correct earlier erroneous information. Here also it is sufficient to treat the arrival of each token individually.

individually in the proofs. The number of cases that must be examined in proving the correctness of the larger entity of a node and the edges leading into it is somewhat greater than the number otherwise required.

VI. The Complete Monitor

Even though the time spent in action and conflict resolution is small compared to the time spent in match, it is unwise to execute the match in parallel and then fail to take advantage of parallelism in the other two phases.

One form of parallel execution, overlapping execution of conflict resolution and action with execution of the match, is not always possible. First, of course, the machine on which the monitor is to be implemented must support this kind of parallelism (array processors will not). In addition, match and conflict resolution must support incremental computation of the following kind. The Recognize-Act cycle begins at action; match and conflict resolution are idle. Action completes one RHS action and informs match of the change to working memory. Match then begins computing changes to the conflict set. When match finds the first change, it informs conflict resolution, and conflict resolution begins making the change. While match executes, action continues to report changes to working memory; while conflict resolution executes, match continues to report changes to the conflict set. Finally, all changes will have been reported and responded to, all phases will have completed their work, and it will be time for another action phase to begin. The rete match will support this kind of incremental execution (incremental execution is its normal mode of operation). Conflict resolution might not. It is difficult to perform conflict resolution incrementally when the relations of dominance induced by the conflict resolution rules are lacking either transitivity or trichotomy.

An alternative form of parallel execution, parallel execution within each phase, may be easier. Parallel execution within conflict resolution is nothing more than the familiar problem of sorting in parallel. Parallel execution within action is trivially easy when the RHS actions have no side effects; it can be quite difficult when the actions have side effects.

Since both forms of parallelism present certain difficulties, use of both forms may be indicated for some production systems. To prevent the problems that can result from side effects in the RHS actions, action might be performed serially, but overlapped with the match. To prevent the problems that can result from complex conflict resolution rules, conflict resolution might be performed in parallel, but with no overlap with the match.

VII. Conclusion

This paper has described a new architecture for production system monitors, an architecture for efficient execution of large production systems, probably on parallel computers. Detailed analyses of performance have not yet been made, but preliminary studies with serial computers, small production systems (never more than a few hundreds of productions), and small working memories (again, never more than a few hundreds of elements) show the performance of this architecture to compare favorably with that of alternative architectures. And arguments can be made that this architecture will be more attractive with large production systems, large working memories, or both. Its responses to the issues of context (which takes advantage of differences in productions) and structural similarity (which takes advantage of similarities in productions) together should make the rete match relatively insensitive to the number of productions in the system. Its response to temporal redundancy should make it relatively insensitive to the number of elements in working memory.

Balanced against the advantage of efficiency is the disadvantage of difficulty in implementation. Monitors making use of the rete match are no larger than other monitors, but they are somewhat more difficult to implement. As section V attempted to show, one must attend to many details to insure a correct implementation.

References

1. Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2. Anderson, R. H., and Gillogly, J. J. Rand intelligent terminal agent (RITA): design philosophy. The Rand Corporation, 1976.

3. Erman, L. D. and Lesser, V. L. A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 483-490.

4. Forgy, C. and McDermott, J. The OPS reference manual. Technical Report, Computer Science Department, Carnegie-Mellon University, 1976.

5. Hayes-Roth, F. and Mostow, J. An automatically compilable recognition network for structured patterns. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 246-252.

6. McDermott, J. and Forgy, C. Production system conflict resolution strategies. Technical Report, Computer Science Department, Carnegie-Mellon University, 1976.

7. McDermott, J., Newell, A., and Moore, J. The efficiency of certain production system implementations. Technical Report, Computer Science Department, Carnegie-Mellon University, 1976.

8. Newell, A., Shaw, J. C., and Simon, H. A. A variety of intelligent learning in a general problem solver. Yovits and Cameron (eds), *Self-Organizing Systems.* Pergamon, 1960, pp. 153-180.

9. Rychener, M. D. Production systems as a programming language for Artificial Intelligence applications. Technical Report, Computer Science Department, Carnegie-Mellon University, 1976 (forthcoming).

10. Shortliffe, E. H. *Computer-Based Medical Consultations: Mycin.* American Elsevier, 1976.