CMU-CS-83-167

# Measurements on Production Systems

Anoop Gupta

Charles L. Forgy

December 1983

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-83-167

# Measurements on Production Systems

December 1983

**Anoop Gupta and Charles L. Forgy**
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

## Abstract

This paper presents measurements on six OPS5 production system programs running at CMU. The measurements contribute to the study of static and run-time characteristics of production system programs. In particular, the measurements will enable us to explore the possibility of using parallelism in executing production system programs. The complete set of measurements is divided into three parts. The first part consists of measurements on the textual structure of the production system programs. The second part consists of measurements on the compiled form of the productions, and the third part consists of run-time measurements on the production system programs. Along with the data, a number of examples are given to show how the data may be used to evaluate parallel implementations of production systems.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

# Table of Contents

1. Introduction
2. Background
    2.1. OPS5
        2.1.1. Working Memory Elements
        2.1.2. The Left Hand Side of a Production
        2.1.3. The Right Hand Side of a Production
    2.2. The Rete Algorithm
3. Surface Measurements on Production Systems
    3.1. Condition Elements per Production
    3.2. Actions per Production
    3.3. Negative Condition Elements per Production
    3.4. Attributes per Condition Element
    3.5. Tests per Two Input Node
    3.6. Variables per Production
    3.7. Variables Bound and Referenced
    3.8. Variables Bound and Not Referenced
    3.9. Variable Occurrences in Left Hand Side
    3.10. Variables per Condition Element
    3.11. Condition Element Classes
    3.12. Action Type Table
    3.13. Summary of Surface Measurements
4. Measurements on the Rete Network
    4.1. Number of Nodes in the Network
    4.2. Network Sharing
    4.3. Network Branching
    4.4. Summary of Rete Network Measurements
5. Measurements on the Run-time Behavior of the Rete Algorithm
    5.1. T-const Nodes
    5.2. Alpha-mem Nodes
    5.3. Beta-mem Nodes
    5.4. And-Nodes
    5.5. Not-Nodes
    5.6. P-Nodes
    5.7. Summary of Run-time Measurements
6. Conclusions
7. Acknowledgments
References

# 1. Introduction

*Production Systems* (or rule-based systems) are widely used in Artificial Intelligence for modeling intelligent behavior [10, 8], and building *expert systems* [9, 11, 7, 6, 1]. Although several large production system programs are now in existence, there has been little publication of data concerning the structure and execution characteristics of these systems. Charles Forgy presented some measurements in his Ph.D. thesis [2] in early 1979, but the spectrum of production system programs (in terms of their size and the applications addressed) available for measurement has significantly changed since then. Furthermore, the emphasis of measurements at that time was to aid in the design of algorithms for uniprocessors, while our current measurements are aimed at exploring the possibility of exploiting parallelism in executing production systems.

This paper presents data on the *static* and *dynamic* characteristics of production systems implemented using the OPS5 language [3]. By static characteristics we refer to those features of production systems that can be measured without executing the production system. The static measurements include measurements on the text form of the production system programs, and measurements on the static data structures constructed by the OPS5 interpreter to execute the programs. By dynamic measurements we refer to the run-time statistics gathered on the data structures and the operations performed by the OPS5 interpreter.

The following are the six production system programs[1] for which data are presented:

- R1 [9], a program for configuring VAX computer systems. It consists of 1932 productions.
- XSEL, a program which acts as a sales assistant for VAX computer systems. It consists of 1443 productions.
- PTRANS [5], a program for factory management. It consists of 1016 productions.
- HAUNT, an adventure game program. It consists of 834 productions.
- DAA [7], a program for VLSI design. It consists of 131 productions.
- SOAR [8], an experimental problem solving architecture implemented as a production system. It consists of 103 productions.

These programs were chosen because they were easily available (all of them were developed at CMU), and also because they represent a wide spectrum of applications.

[1] In some cases only a subset of the complete production system program was measured because of problems with the LISP garbage collector. The number of productions given for each program corresponds to the number in the program actually measured.

## 2. Background

Before we present the measurements, it is necessary to understand the computational model underlying production systems and the algorithms used to implement the required computation. This knowledge is necessary to understand the results of the measurements (most of which are algorithm specific), and also to abstract implications concerning the execution of production systems. In the following paragraphs we first describe the syntax and semantics of OPS5, and then describe the Rete algorithm that implements the OPS5 language.

### 2.1. OPS5

A production system is composed of a set of *if-then* rules called *productions* that make up the *production memory*, and a database of assertions called the *working memory*. The assertions in the working memory are called *working memory elements*. Each production consists of a conjunction of *condition elements* corresponding to the *if* part of the rule (also called the *left hand side* of the production), and a set of *actions* corresponding to the *then* part of the rule (also called the *right hand side* of the production). The actions associated with a production can add, remove or modify working memory elements, or perform input-output. Figure 2-1 shows an OPS5 production named p1, which has three condition elements in its left hand side, and one action in its right hand side.

```
(p  p1  (C1  ↑attr1 <x>    ↑attr2 12)
        (C2  ↑attr1 15     ↑attr2 <x>)
      - (C3  ↑attr1 <x>)
   -->
        (remove 2))
```

**Figure 2-1:** A Sample Production

The OPS5 interpreter executes a production system program by performing the following operations:

- MATCH: In this first phase, the left hand sides of all productions are matched against the contents of working memory. As a result we obtain a *conflict set*, which consists of *instantiations* of all satisfied productions. An instantiation of a production is an ordered pair. Its first element identifies the production, and its second element is an ordered list of working memory elements that satisfies the left hand side of the production. At any given time, the conflict set may contain zero, one, or more instantiations of a given production.

- CONFLICT RESOLUTION: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.

- ACT: In this third phase, the actions of the production selected in the conflict resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

### 2.1.1. Working Memory Elements

A working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by the operator ↑. The values are symbolic or numeric constants. For example, the following is a very small working memory element of class **C1**, having the value **12** for attribute **attr1** and the value **15** for attribute **attr2.**

```
(C1     ↑attr1  12     ↑attr2  15)
```

### 2.1.2. The Left Hand Side of a Production

The condition elements in the left hand side of a production are parenthesized lists similar to working memory elements. They may optionally be preceded by the symbol −. Such condition elements are called *negated condition elements.* For example, the production in Figure 2-1 contains three condition elements, with the third one being negated. Condition elements are interpreted as partial descriptions of working memory elements. When a condition element describes a working memory element, the working memory element is said to *match* the condition element. A production is said to be *satisfied* when:

- For every non-negated condition element in the left hand side of the production, there exists a working memory element that matches it.

- For every negated condition element in the left hand side of the production, there does not exist a working memory element that matches it.

Like a working memory element, a condition element contains a class name and a sequence of attribute-value pairs. However, the condition element is less restricted than the working memory element; while the working memory element can contain only constant symbols and numbers, the condition element can contain variables, predicate symbols, and a variety of other operators as well as constants. Only variables and predicates will be described here since they are the most important of the operators. A variable is an identifier that begins with the character "<" and ends with ">"—for example, **<x>** and **<status>** are variables. The predicate symbols in OPS5 are:

```
<     =     >     <=     >=     <>     <=>
```

The predicates have their usual meanings for numerical and symbolic values. For example, the first predicate in the list, "<", denotes the *less-than* relationship, the second predicate, "=", denotes *equality*, and the last predicate, "<=>", denotes *of the same type* relationship. The following condition element contains one constant value (the value of **attr1**), one variable value (**attr2**), and one constant value that is modified by the predicate symbol **<> (attr3).**

```
(C1     ↑attr1 nil     ↑attr2 <x>     ↑attr3 <> nil)
```

A working memory element matches a condition element if the object types of the two match and if the value of every attribute in the condition element matches the value of the corresponding attribute in the working memory element. The rules for determining whether a working memory element value matches a condition element value are:

- If the condition element value is a constant, it matches only an identical constant.
- If the condition element value is a variable, it wil! match any value. However, if a variable occurs more than once in a left hand side, all occurrences of the variable must match identical values.
- If the condition element value is preceded by a predicate symbol, the working memory element value must be related to the condition element value in the indicated way.

Thus the working memory element

```
        (C1       ↑attr1   12       ↑attr2   15)
```

will match the following four condition elements

```
        (C1       ↑attr1   12       ↑attr2   <x>)
        (C1       ↑attr2   15)
        (C1       ↑attr2 > 0)
        (C1       ↑attr1   <x>      ↑attr2   <> <x>)
```

but it will not match the condition element

```
        (C1       ↑attr1   <x>       ↑attr2   <x>).
```

### 2.1.3. The Right Hand Side of a Production

The right hand side of a production consists of an unconditional sequence of actions which can cause input-output, and which are responsible for changes to the working memory. Three kinds of actions are provided to effect working memory changes. *Make* creates a new working memory element and adds it to working memory. *Modify* changes one or more values of an existing working memory element. *Remove* deletes an element from the working memory. As an example of a make, the action

```
      (make   C1     ↑attr1   12)
```

will add the working memory element

```
      (C1     ↑attr1   12)
```

to working memory when it is executed. As an example of a modify, the action

```
      (modify   1     ↑status satisfied)
```

will change the working memory element matching the rule's first condition element, substituting "satisfied" for the prior value of the status attribute. As an example of remove, the action

```
      (remove   2)
```

will cause the working memory element matching the second condition element in the instantiation of the production to be deleted from working memory.

### 2.2. The Rete Algorithm

The most time consuming phase in the execution of production systems is the match, where the left hand sides of all productions are matched against the complete working memory. For example, consider a production system with 1000 working memory elements and 1000 productions, where each production has three condition elements. In a naive implementation each production will have to be matched against all permutations of size three from the working memory, leading to over a trillion match operations for each execution cycle. More complex algorithms, however, can achieve the same result with only a small fraction of the work above. The Rete match algorithm [4] is one such algorithm.

The Rete algorithm uses an augmented discrimination network compiled from the left hand sides of the productions to perform the match. To generate the network for a production, the network compiler proceeds first with the individual condition elements in the left hand side. For each condition element it chains together test nodes that check:

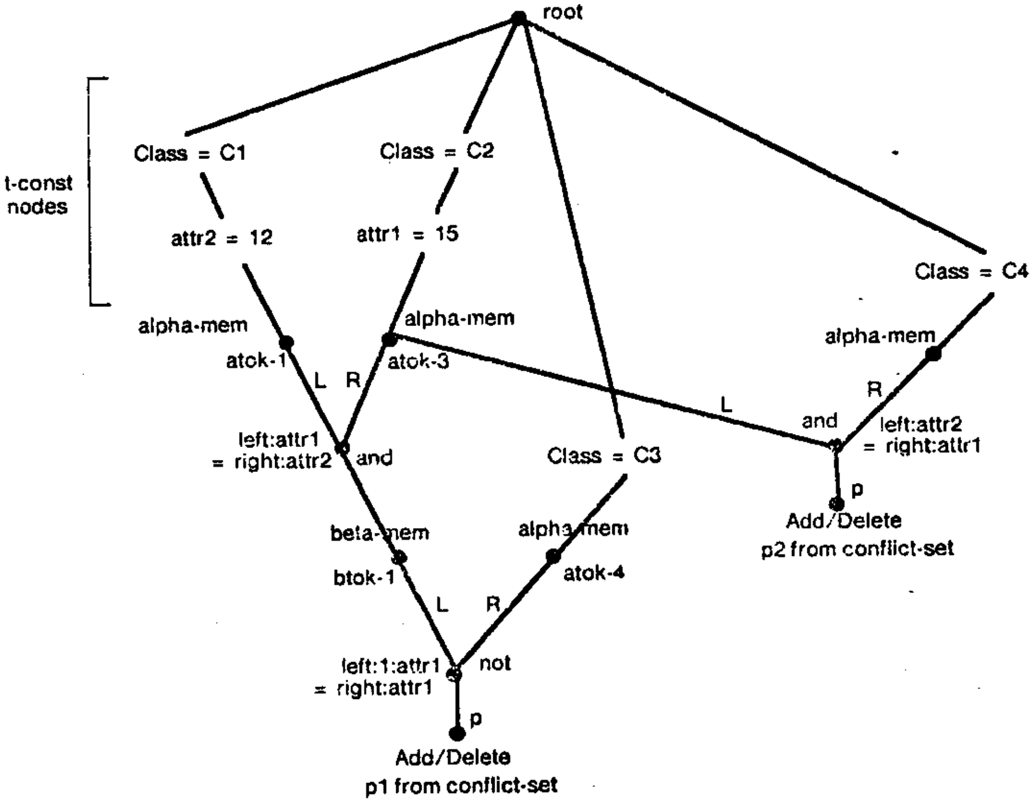
- If the attributes in the condition element that have a constant as their value are satisfied.
- If the attributes in the condition element that are related to a constant by a predicate are satisfied.
- If two occurrences of the same variable within the condition element are consistently bound.

Each node in the chain performs one such test. (The three kinds of tests above are called *intra-condition* tests, because they correspond to individual condition elements.) Once the network compiler has finished with the individual condition elements, it adds nodes that check for consistency of variable bindings across the multiple condition elements in the left hand side. (These tests are called *inter-condition* tests, because they refer to multiple condition elements.) Finally the compiler adds a special terminal node to represent the production to which this part of the network corresponds.

Figure 2-2 shows such a network for productions p1 and p2 which appear in the top part of the figure. In this figure, lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top node down along these paths. The nodes with a single predecessor (near the top of the network) are the ones that are concerned with individual condition elements. The nodes with two predecessors are the ones that check for consistency of variable bindings between condition elements. The terminal nodes are at the bottom of the figure. Note that when two left hand sides require identical nodes, the compiler shares part of the network rather than building duplicate nodes.

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as *state* within the network. This way, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network, and where relevant, the

```
(p p1   (C1 ↑attr1 <x>  ↑attr2 12)                    (p p2   (C2 ↑attr1 15  ↑attr2 <y>)
        (C2 ↑attr1 15  ↑attr2 <x>)                            (C4 ↑attr1 <y>)
       - (C3 ↑attr1 <x>)                                   -->
  -->                                                         (modify 1 ↑attr1 12))
        (remove 2))
```



Add to Working Memory

1. (C1 ↑attr1 12 ↑attr2 12)
2. (C2 ↑attr1 12 ↑attr2 15)
3. (C2 ↑attr1 15 ↑attr2 12)
4. (C3 ↑attr1 12)

**Figure 2-2:** The Rete Network

state stored in the network is updated. The output of the network consists of a specification of changes to the conflict set.

The objects that are passed between nodes in the network are called *tokens*, which consist of a *tag* and a *list of working memory elements*. The tag can be either a +, indicating that something has been added to the working memory, or a $-$, indicating that something has been removed from it. (No special tag for working memory element modification is needed because a modify is treated as a delete followed by an add.) The list of working memory elements associated with a token corresponds to the permutation of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the left hand side.

The discrimination net produced by the Rete network compiler consists of a number of different types of nodes:

- **Root Node:** This node forms the root of the discrimination net. It broadcasts tokens corresponding to any change in the working memory to all its successor nodes. In Figure 2-2, the root node is shown at the top.

- **T-Const Nodes:** These nodes are used in the network to perform intra-condition tests, for example, to check if condition element attributes that have constant symbols or numbers as their values are satisfied. Each t-const node checks for one feature. Whenever the token arriving at the input of a t-const node satisfies the associated test, it is passed on to the successors of the t-const node. If the token does not satisfy the test, it is not passed on to the successors. In Figure 2-2, the nodes towards the top of the network are t-const nodes. Since the second condition element of production p1 is similar to the first condition element of production p2, t-const nodes "Class =C2" and "attr1 = 15" are shared in the network for p1 and p2.

- **$\alpha$-mem Nodes:** If a working memory element satisfies all intra-condition tests for a condition element (note it may not, as yet, satisfy all the inter-condition tests), the working memory element is said to *partially match* the condition element. Tokens corresponding to working memory elements that partially match a condition element are stored in the $\alpha$-mem node for that condition element. When a token arrives at an $\alpha$-mem node with a $+$ tag, the token is stored in the $\alpha$-mem node and a copy of the token is passed to the node's successors. If the tag is $-$, a corresponding token with a $+$ tag must already exist in the $\alpha$-mem. The corresponding token is deleted from the $\alpha$-mem node, and the incoming token is passed down to the successors of the $\alpha$-mem node. If two condition elements, in the same or different productions, have exactly the same tests for a successful partial match, the network compiler generates a shared $\alpha$-mem node for the two. This sharing of an $\alpha$-mem node can be seen in Figure 2-2.

- **$\beta$-mem Nodes:** Just as $\alpha$-mem nodes store tokens that partially match individual condition elements, so $\beta$-mem nodes store tokens that partially match two or more condition elements in the left hand side of a production. The list of working memory elements in $\beta$-mem tokens has length two or more. The response of $\beta$-mem nodes to arrival of tokens at their inputs is exactly the same as that of $\alpha$-mem nodes. The $\beta$-mem nodes form the left input of and-nodes and not-nodes.

- **And-Nodes:** The and-nodes are the first of the two-input node types. The primary function of an and-node is to check for consistency of variable bindings between the partially matched tokens it receives on its left and right inputs. The right input of an and-node always comes from an $\alpha$-mem node, while its left input can come from an $\alpha$-mem or a $\beta$-mem node. Whenever a token arrives at the left input of an and-node, the and-node compares the incoming token to each token stored in the mem-node connected to its right input, to check if they are consistent. For every right-token which is consistent with the left-token, a new token is constructed and sent down to the successor nodes. The new token has the same tag as that of left-token, and the list of working memory elements is the concatenation of the working memory element lists for the left and right tokens. The case when a token arrives at the right input of an and-node is processed exactly as above, with left and right interchanged.

- **Not-Nodes:** The not-nodes are the second of the two-input node types. They also have a left and a right input. The not-nodes are used by the network to implement the semantics of negated condition elements. Their functionality differs from that of and-nodes only in minor ways—they additionally keep reference counts with tokens in left memory to find when there are no tokens in the right memory that are consistent with them.

- **P-Nodes:** These are the terminal nodes in the network, and there is one such node associated with each production. Whenever a token with a + tag flows into a p-node, it adds an instantiation (corresponding to the token) of the associated production into the conflict set. The arrival of a token with a − tag leads to the deletion of the corresponding production instantiation from the conflict set.

- **Other Nodes:** Other than the node types mentioned above, the network uses two more node types. These are the *two-nodes* and the *any-nodes.* The two-node is used as a place filler in some circumstances, and the any-node is used when the value of an attribute is to be one of a number of alternatives. Although measurements for these two node types are presented in the later sections for the sake of completeness, they have largely been omitted from the analysis.

To conclude this section we present an example of performing match using the Rete algorithm. The example corresponds to the two productions and the network given in Figure 2-2. We now go through the match process. as the four working memory elements shown in the bottom left corner of Figure 2-2 are sequentially added to the working memory.

When the first working memory element is added, token $\alpha$tok-1,

```
< + , (C1 ↑attr1 12 ↑attr2 12) >
```

is constructed and sent to the root node. The root node broadcasts the token to all its successors. The associated tests fail at all successors except at one which is checking for "Class = C1". This t-const node passes the token down to its single successor, another t-const node, checking if "attr2 = 12". Since this is so, the token is passed on to the $\alpha$-mem node, which stores the token and passes a copy of the token to the and-node below it. The and-node compares the incoming token on its left input to tokens in its right memory (which at this point is empty), but no pairs can be formed. At this point, the network has stabilized: in other words no further activity occurs, so we go on to the second working memory element.

The token for the second working memory element, $\alpha$tok-2,

```
< + , (C2 ↑attr1 12 ↑attr2 15) >
```

is constructed and sent to the root node, which broadcasts the token to its successors. The token passes the "Class = C2" test but fails the "attr1 = 15" test, so no further processing takes place.

The token for the third working memory element, $\alpha$tok-3,

```
< + , (C2 ↑attr1 15 ↑attr2 12) >
```

passes through the tests "Class = C2" and "attr1 = 15", and is stored in the $\alpha$-mem node below them. The $\alpha$-mem node passes a copy of the token to the two successor and-nodes below it. The and-node on the right finds no tokens in its right memory, so no further processing is done there. The and-node on the left checks the token for consistency against token, $\alpha$tok-1, stored in its left memory. The consistency check is satisfied as the variable <x> is bound consistently. The and-node creates a new token, $\beta$tok-1,

```
< + , ((C1 ↑attr1 12 ↑attr2 12), (C2 ↑attr1 15 ↑attr2 12)) >
```

and passes it down to the $\beta$-mem node below, which stores it. The $\beta$-mem node now passes a copy of the token to the not-node below it. The not-node finds that its right memory is empty, which implies that there are no working memory elements which consistently bind the left token $\beta$tok-1. (Recall that this corresponds to the semantics of satisfaction of production p1.) As there are no matching tokens in the right memory, a reference count of zero is stored with token $\beta$tok-1 in the not-node. The not-node passes a copy of token $\beta$tok-1 to the p-node below it, which then inserts an instantiation of production p1 in the conflict set.

On addition of the fourth working memory element, token $\alpha$tok-4,

```
< + , (C3 ↑attr1 12) >
```

is sent to the root node, which broadcasts it. The token passes the test "Class = C3" and passes on to the $\alpha$-mem below it. The $\alpha$-mem node stores the token and passes a copy of the token to the not-node below it. The not-node checks for consistent bindings in the left memory and finds that the newly arrived token, $\alpha$tok-4, is consistent with token $\beta$tok-1 stored in its left memory. The not-node increments the reference count stored with $\beta$tok-1 by 1. Since the change in the reference count is from 0 to 1 (recall that a reference count of zero implies that a token was sent down below the not-node with a + tag), a token exactly the same as $\beta$tok-1, but with the tag set to −, is sent by the not-node to its successors. On receiving this token, the p-node for production p1 deletes the instantiation of production p1 from the conflict-set. This is the final state of the network and conflict set following the addition of the four working memory elements.

## 3. Surface Measurements on Production Systems

Surface measurements refer to measurements on the textual features of production system programs. Examples of such features are—the number of condition elements in the left hand sides of productions, the percentage of productions which have one or more negated condition elements, the number of attributes per condition element.

In the following subsections we present the set of measured features, with a brief description of how the measurements were made. Data about the same features of different production systems are presented together, and have been normalized to permit comparison[2]. Along with each data graph the *average*, the *standard deviation*, and the *coefficient of variance*[3] for the data points are given.

## 3.1. Condition Elements per Production

Figure 3-1 shows the number of condition elements per production for the six production system programs. The number of condition elements per production includes both positive elements and negative ones. The curves for the programs are normalized by plotting *percent* of productions, instead of *number* of productions, along the y-axis. To help compute absolute numbers from the normalized numbers, the total number of productions for each of the programs is provided in Table 3-8.

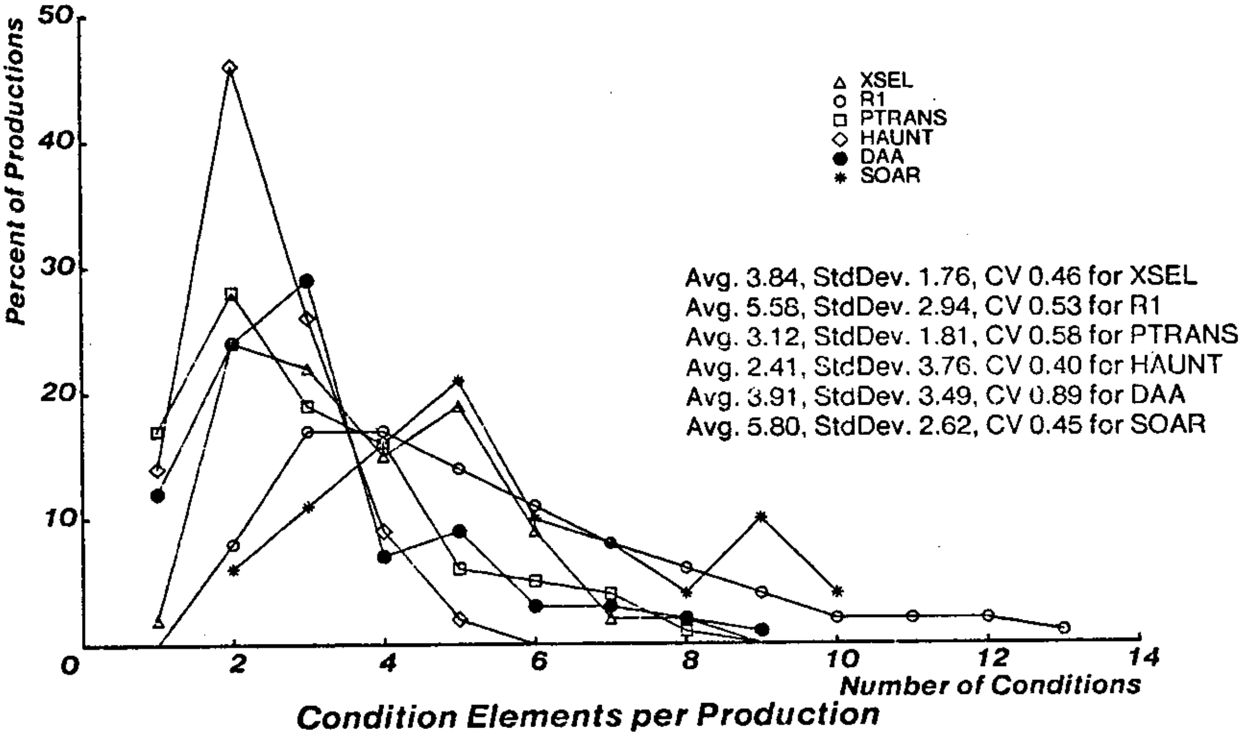


**Figure 3-1:**

[2] The limits of the axes of the graphs are adjusted to show the main portion of the graph clearly. In doing this, however, in some cases a few extreme points could not be put on the graph. For this reason, the reader should not draw conclusions about the maximum values of the parameters in question from the graph.

[3] Coefficient of Variance = Standard Deviation / Average

### 3.2. Actions per Production

Figure 3-2 shows the number of actions per production. The number of actions reflects the complexity of executing the right hand side of a production. A large number of actions per production also implies a greater potential for parallelism, because then a large number of changes to the working memory can be processed in parallel, before the next conflict resolution phase is executed.

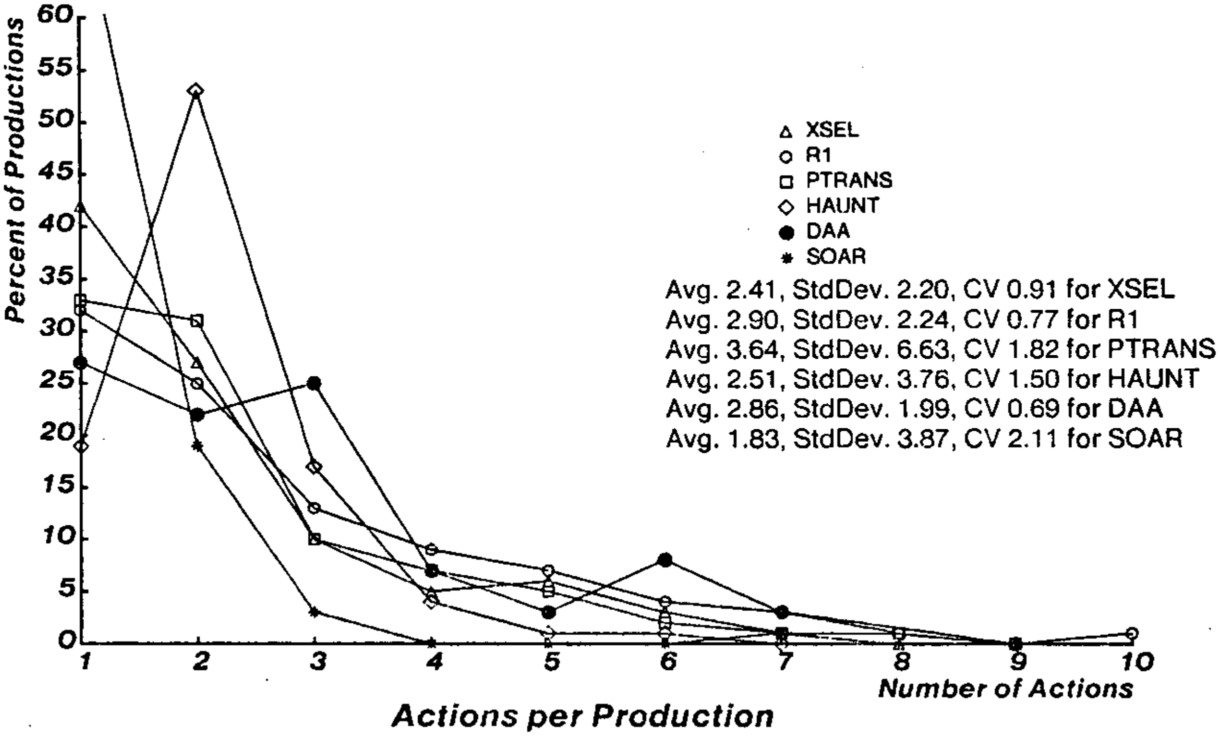


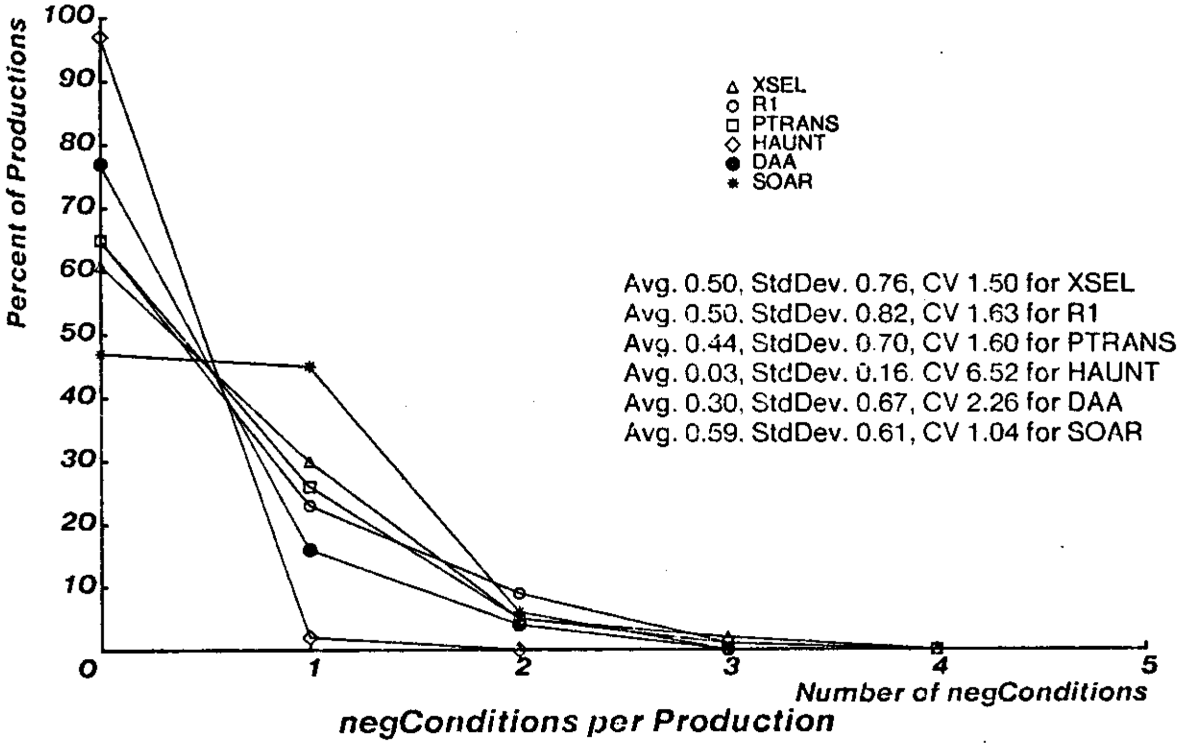
Figure 3-2:

### 3.3. Negative Condition Elements per Production

The graph in Figure 3-3 shows the number of negated condition elements in the left hand side of a production versus the percent of productions having them. It shows that almost 30 percent of productions have one or more negated condition elements. Since negated condition elements denote universal quantification over the working memory, the percentage of productions having them is an important characteristic of production system programs. The measurements are also useful in calculating the number of not-nodes in the unshared Rete network.

### 3.4. Attributes per Condition Element

Figure 3-4 shows the distribution for the number of attributes per condition element. The *class* of a condition element, which is an implicit attribute, is counted explicitly in the measurements. The number of attributes in a condition element reflects the number of tests that are required to detect a matching working



Avg. 0.50, StdDev. 0.76, CV 1.50 for XSEL
Avg. 0.50, StdDev. 0.82, CV 1.63 for R1
Avg. 0.44, StdDev. 0.70, CV 1.60 for PTRANS
Avg. 0.03, StdDev. 0.16. CV 6.52 for HAUNT
Avg. 0.30, StdDev. 0.67, CV 2.26 for DAA
Avg. 0.59. StdDev. 0.61, CV 1.04 for SOAR

negConditions per Production

Figure 3-3:

memory element.

### 3.5. Tests per Two Input Node

This feature is specific to the Rete match algorithm and refers to the number of variable bindings that are checked for consistency at each two-input node (and-node or not-node). A value of zero indicates that no variables are checked for consistent binding, while a large value indicates that a large number of variables are checked. For example, if the number of tests is zero, for every token that arrives at the input of an and-node, as many tokens as there are in the opposite memory are sent to its successors. This usually implies a large amount of work. Alternatively, if the number of tests is large, then the number of tokens sent to the successors is small, but doing the pairwise comparison for consistent binding now takes more time. The graph for the number of tests per two-input node is shown in Figure 3-5.

### 3.6. Variables per Production

The distribution for the number of variable occurrences (not the number of distinct variables) in the left hand side of a production is shown in Figure 3-6. The values indicate the number of variables that are either bound or tested for consistent binding in the left hand side of a production.

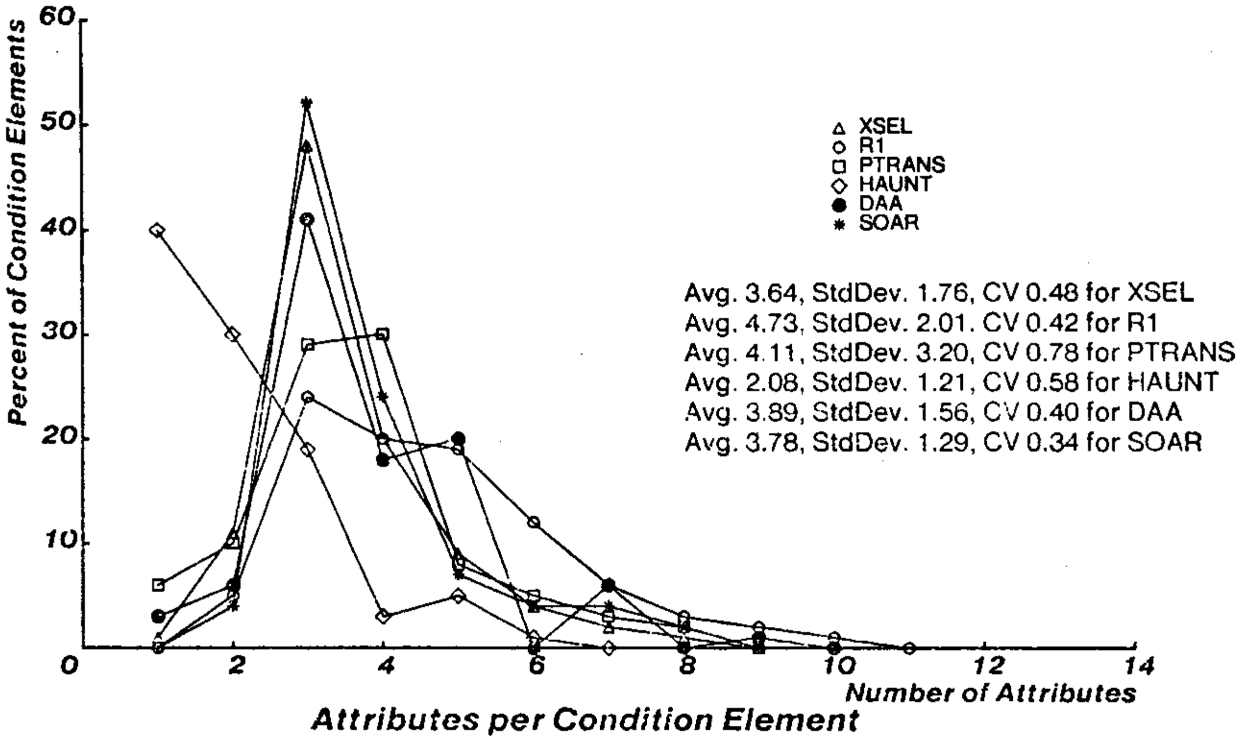


Figure 3-4:

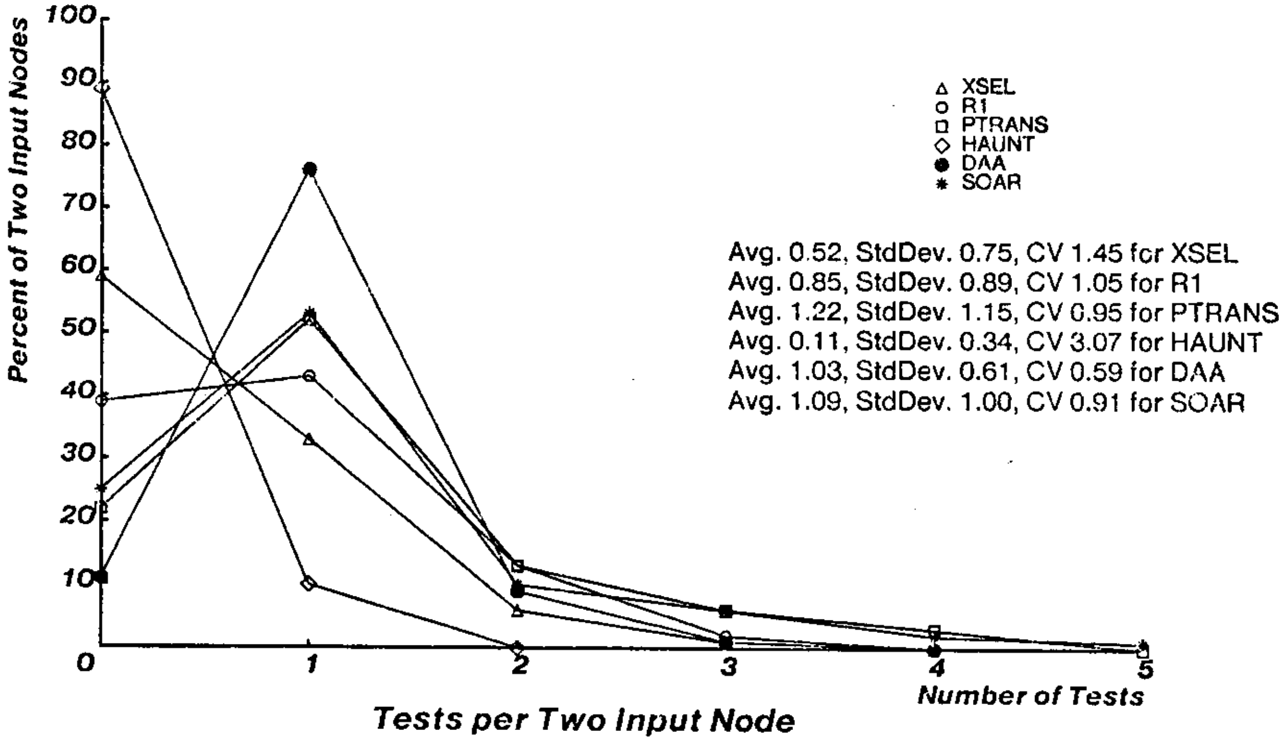


Figure 3-5:

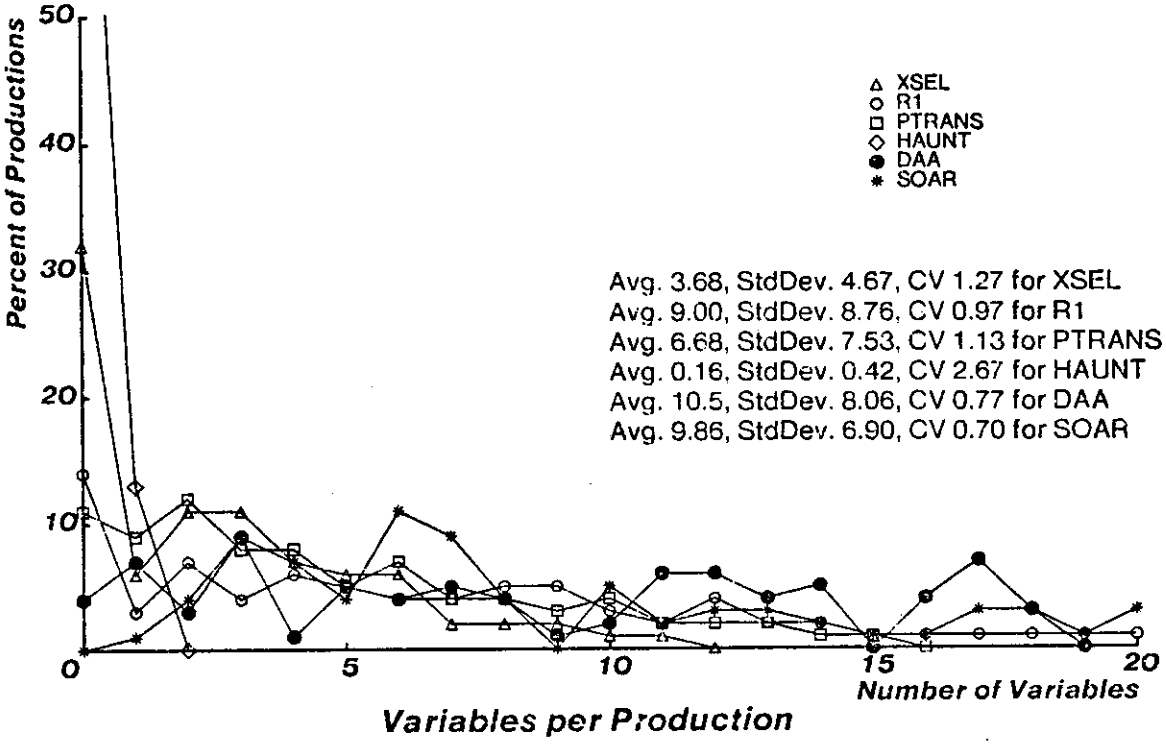


Figure 3-6:

### 3.7. Variables Bound and Referenced

Figure 3-7 shows the number of distinct variables which are both bound and referenced in the left hand side of a production. Consistency tests are necessary only for these variables. Beyond the $\alpha$-mem nodes, all processing done by the two-input nodes requires access to the values of only these variables; values of no other variables or attributes are required. This implies that the tokens in the network may only store the values of these variables instead of storing complete copies of working memory elements. For parallel architectures, this can lead to significant improvements in the storage requirements and in the communication overhead associated with tokens.

### 3.8. Variables Bound and Not Referenced

Figure 3-8 shows the number of distinct variables which are bound but not referenced in the left hand side of a production. (The bindings are probably used in the right hand side of the production.) This indicates the number of variables for which no consistency checks have to be performed.

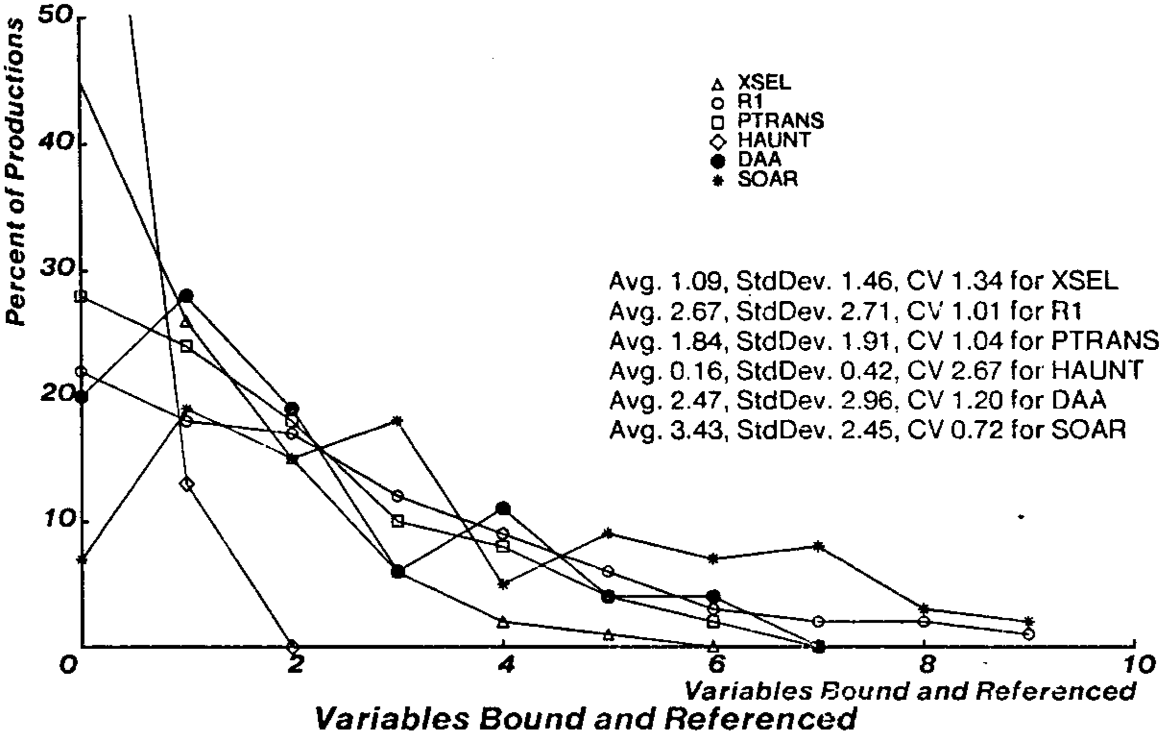


Figure 3-7:

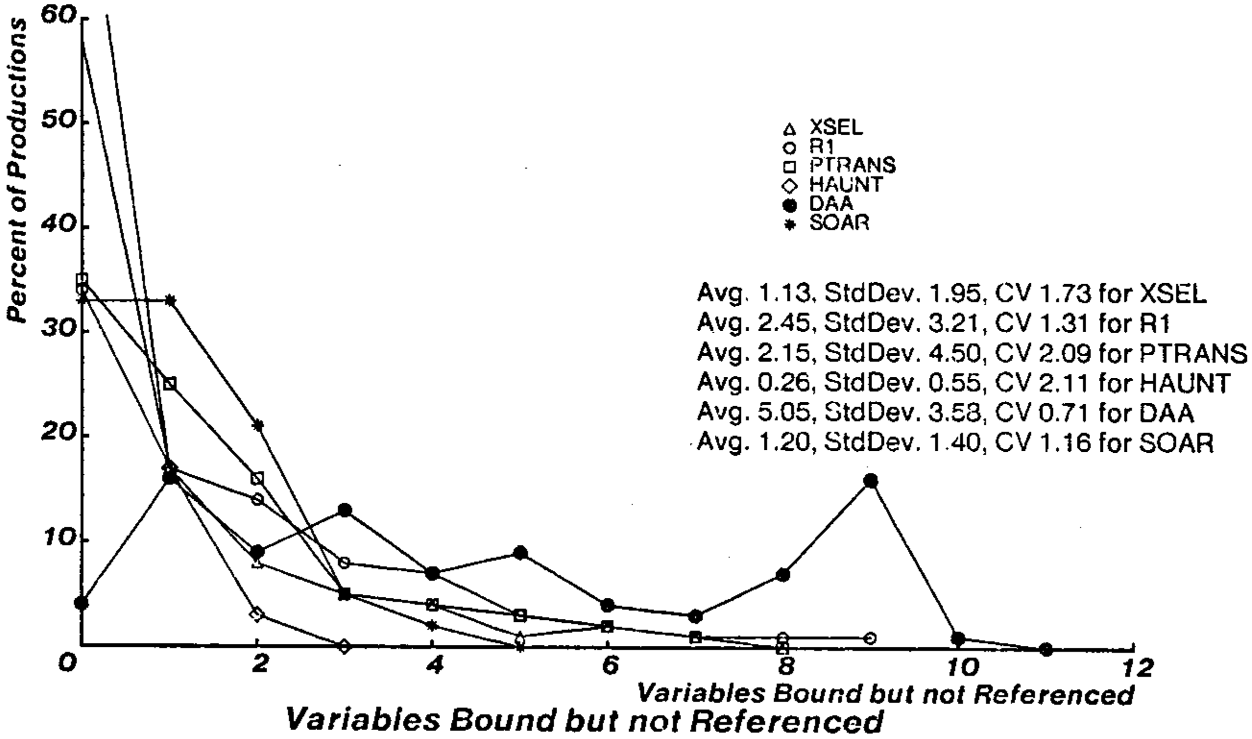


Figure 3-8:

### 3.9. Variable Occurrences in Left Hand Side

Figure 3-9 shows the number of times each variable occurs in the left hand side of a production. Both positive and negative condition elements are considered in counting the variables. Our measurements also show that variables almost never occur multiple times within the same condition element (average of 97.7% over all systems). Under this assumption, the number of occurrences of a variable also represents the number of condition elements within a production in which the variable occurs.

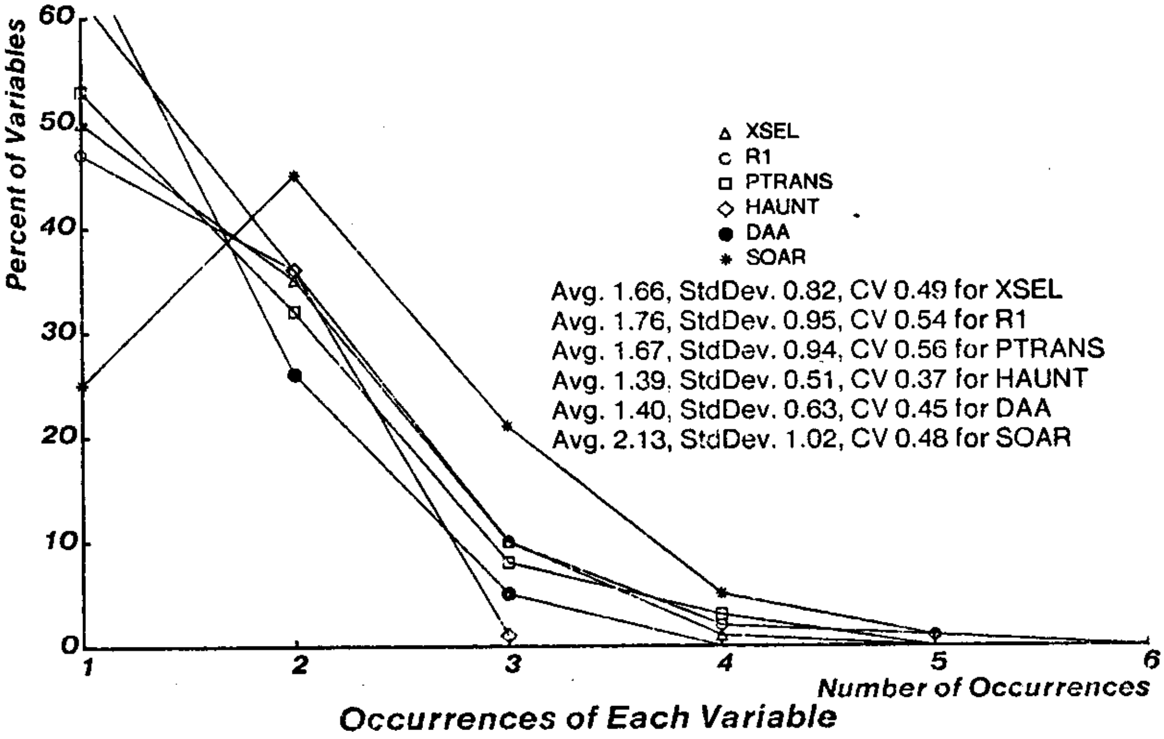


Figure 3-9:

### 3.10. Variables per Condition Element

Figure 3-10 shows the number of variable occurrences (not necessarily distinct) within a condition element. If this number is large it usually implies that the selectivity of the condition element is small: in other words a large number of working memory elements will match that condition element.

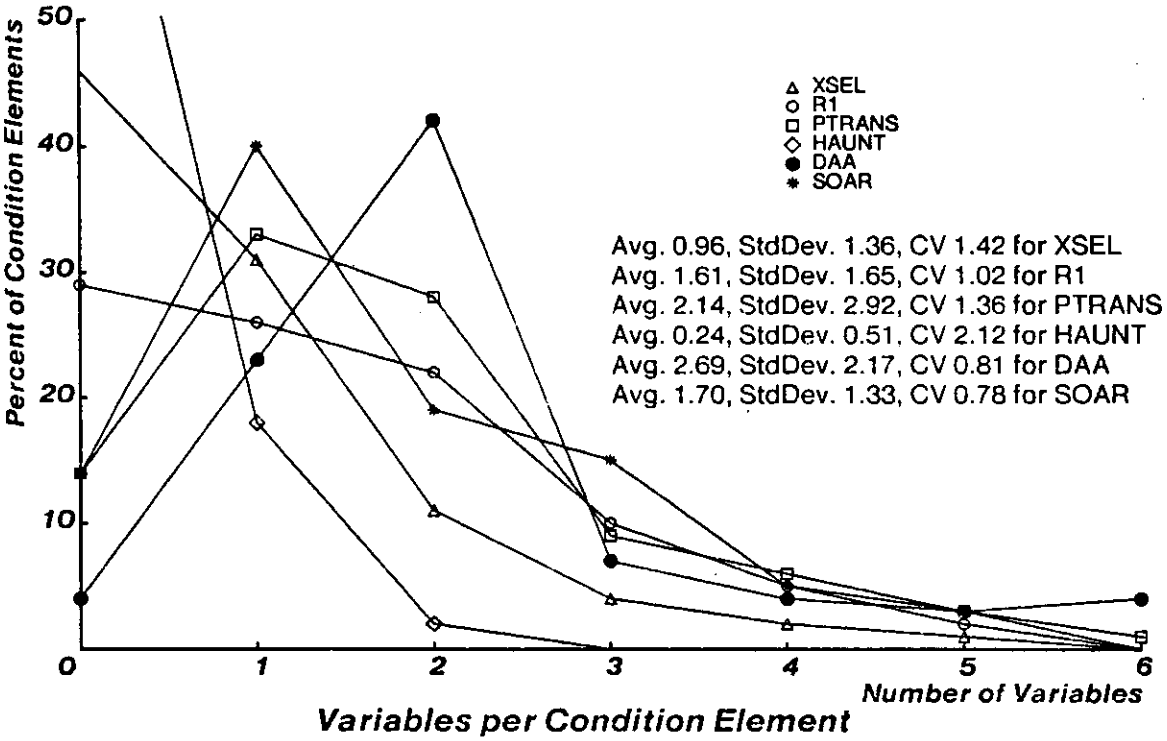


Figure 3-10:

### 3.11. Condition Element Classes

Tables 3-1, 3-2, 3-3, 3-4, 3-5, and 3-6 list the seven condition element classes occuring most frequently for the production system programs. The tables also list the total number of attributes, the average number of attributes and its standard deviation, and the average number of variable occurrences in condition elements of each class.[4] The total number of attributes for a condition element class gives an estimate of the size of the working memory element. This information is important because it indicates the communication overhead in transporting working memory elements amongst multiple memories in a parallel architecture. It also has implications for space requirements for storing the working memory elements. If we subtract the average number of variables from the average number of attributes for a condition element class, we obtain the average number of attributes which have a constant value for that class. This number in turn has implications for the selectivity of condition elements of that class.

[4] In Table 3-4, for a few entries, the average number of variables is more than the average number of attributes. This apparent disagreement is due to the use of vector-valued attributes in DAA.

Table 3-1: R1: Condition Element Classes

| Class Name | # of CEs(%) | Tot-Attr | Avg-Attr | SD-Attr | Avg-Vars |
|---|---|---|---|---|---|
| component | 2597 (24%) | 152 | 5.04 | 2.03 | 2.24 |
| context | 1996 (18%) | 8 | 2.23 | 0.53 | 0.24 |
| pcon | 1337 (12%) | 100 | 4.76 | 1.98 | 2.19 |
| datum | 1262 (11%) | 53 | 2.96 | 1.07 | 1.26 |
| computation | 875 (8%) | 110 | 3.90 | 1.38 | 1.63 |
| set-attribute | 757 (7%) | 59 | 4.89 | 2.01 | 2.64 |
| template | 533 (4%) | 27 | 3.61 | 1.09 | 1.97 |

Total number of condition element classes is 31

Table 3-2: XSEL: Condition Element Classes

| Class Name | # of CEs(%) | Tot-Attr | Avg-Attr | SD-Attr | Avg-Vars |
|---|---|---|---|---|---|
| context | 1382 (24%) | 3 | 2.11 | 0.36 | 0.07 |
| interaction | 920 (16%) | 21 | 2.66 | 0.98 | 0.66 |
| line-item | 917 (16%) | 27 | 3.43 | 1.82 | 1.70 |
| component | 752 (13%) | 106 | 3.29 | 2.20 | 1.71 |
| local | 354 (6%) | 17 | 2.14 | 0.96 | 0.64 |
| discr-list | 174 (3%) | 5 | 1.90 | 0.94 | 0.41 |
| datum | 147 (2%) | 15 | 2.71 | 0.93 | 0.52 |

Total number of condition element classes is 36

Table 3-3: PTRANS: Condition Element Classes

| Class Name | # of CEs(%) | Tot-Attr | Avg-Attr | SD-Attr | Avg-Vars |
|---|---|---|---|---|---|
| task | 587 (18%) | 4 | 1.86 | 0.71 | 0.79 |
| arg | 521 (16%) | 4 | 2.94 | 0.27 | 1.85 |
| line-item | 170 (5%) | 47 | 4.55 | 3.49 | 3.44 |
| call | 125 (3%) | 16 | 3.68 | 1.52 | 1.90 |
| order | 123 (3%) | 64 | 3.60 | 6.60 | 3.03 |
| period | 120 (3%) | 13 | 3.44 | 1.42 | 2.61 |
| wip | 119 (3%) | 32 | 3.82 | 4.39 | 2.98 |

Total number of condition element classes is 81

Table 3-4: HAUNT: Condition Element Classes

| Class Name | # of CEs(%) | Tot-Attr | Avg-Attr | SD-Attr | Avg-Vars |
|---|---|---|---|---|---|
| location | 499 (24%) | 12 | 1.85 | 1.34 | 0.22 |
| input | 497 (24%) | 0 | 0.00 | 0.00 | 0.33 |
| object | 327 (16%) | 15 | 2.25 | 0.80 | 0.44 |
| x | 237 (11%) | 1 | 1.00 | 0.00 | 0.00 |
| status | 197 (9%) | 13 | 0.85 | 0.44 | 0.10 |
| place | 76 (3%) | 4 | 1.45 | 0.50 | 0.04 |
| time | 43 (2%) | 9 | 1.67 | 0.86 | 0.98 |

Total number of condition element classes is 23

Table 3-5: DAA: Condition Element Classes

| Class Name | # of CEs(%) | Tot-Attr | Avg-Attr | SD-Attr | Avg-Vars |
|---|---|---|---|---|---|
| context | 132 (25%) | 4 | 3.58 | 0.72 | 3.64 |
| port | 121 (23%) | 5 | 2.10 | 0.54 | 1.95 |
| module | 64 (12%) | 6 | 3.36 | 1.53 | 2.19 |
| link | 57 (11%) | 6 | 3.44 | 1.82 | 3.44 |
| lists | 38 (7%) | 3 | 1.82 | 0.39 | 2.16 |
| outnode | 27 (5%) | 6 | 2.63 | 0.67 | 2.33 |
| operator | 25 (4%) | 10 | 5.20 | 2.77 | 5.00 |

Total number of condition element classes is 20

Table 3-6: SOAR: Condition Element Classes

| Class Name | # of CEs(%) | Tot-Attr | Avg-Attr | SD-Attr | Avg-Vars |
|---|---|---|---|---|---|
| current | 273 (45%) | 2 | 1.96 | 0.20 | 0.71 |
| att-val | 98 (16%) | 3 | 2.83 | 0.47 | 1.84 |
| context | 54 (9%) | 8 | 3.48 | 1.83 | 3.07 |
| choice | 51 (8%) | 8 | 5.43 | 1.35 | 3.37 |
| operator | 34 (5%) | 6 | 3.65 | 0.94 | 2.79 |
| applied | 25 (4%) | 5 | 3.68 | 0.84 | 3.16 |
| state-op | 20 (3%) | 5 | 2.80 | 0.60 | 1.90 |

Total number of condition element classes is 12

## 3.12. Action Type Table

Table 3-7 gives the distribution of actions in the right hand side into classes *make, remove, modify, write, and other* for the production system programs. The only actions that affect the working memory are of type make, remove, or modify. While each make and remove action causes only one change to the working memory, a modify actions causes two changes to the working memory. This data then gives an estimate of the percentage of right hand side actions that change the working memory.

Table 3-7: Action Type Distribution

| Action Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| MAKE | 34% | 31% | 22% | 10% | 34% | 71% |
| MODIFY | 25% | 35% | 15% | 19% | 7% | 12% |
| REMOVE | 13% | 8% | 7% | 25% | 26% | 0% |
| WRITE | 9% | 3% | 10% | 44% | 17% | 17% |
| OTHERS | 17% | 20% | 44% | 2% | 13% | 7% |

### 3.13. Summary of Surface Measurements

Table 3-8 gives a summary of the surface measurements for the production system programs. It brings together the average value of the various features for all six programs. The features listed in the table are condition elements per production, actions per production, negated condition elements per production, attributes per condition element, variables per condition element, and tests per two-input node.

Table 3-8: Summary of Surface Measurements

| Feature | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| Productions | 1932 | 1443 | 1016 | 834 | 131 | 103 |
| CEs/Prod | 5.58 | 3.84 | 3.12 | 2.41 | 3.91 | 5.80 |
| Actions/Prod | 2.90 | 2.41 | 3.64 | 2.51 | 2.86 | 1.83 |
| nCEs/Prod | 0.50 | 0.50 | 0.44 | 0.03 | 0.30 | 0.59 |
| Attr/CE | 4.73 | 3.64 | 4.11 | 2.08 | 3.89 | 3.78 |
| Vars/CE | 1.61 | 0.96 | 2.14 | 0.24 | 2.69 | 1.70 |
| Tests/2inp | 0.85 | 0.52 | 1.22 | 0.11 | 1.03 | 1.09 |

## 4. Measurements on the Rete Network

In this section we present measurements made on the Rete network constructed by the OPS5 network compiler. The measured features include—the number of nodes of each type in the network, the amount of sharing that is present in the network, and the average branching factor for the various node types.

### 4.1. Number of Nodes in the Network

In Table 4-1, we present data on the number of nodes of each type in the network for the various production system programs. These numbers reflect the complexity of the network that is constructed for the programs. Table 4-2 gives the normalized number of nodes: that is, the number of nodes per production. The normalized numbers are useful for comparing the average complexity of the productions for the various production system programs.[5]

In Table 4-3, we present the number of nodes per condition element for the production system programs. As we can see from the table, the numbers are all very close to 1.9 nodes per condition element. This number can thus be used to predict the number of nodes in other production system programs, for which no measurements have been made. The number 1.9 nodes per condition element is small because there are a lot of nodes shared between condition elements. In case no sharing is allowed, this number jumps up two to three fold, as is shown in Table 4-4.

[5] All the numbers listed in Tables 4-1 and 4-2, are for the case where the network compiler is allowed to share nodes.

Table 4-1: Number of Nodes

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| T-const | 3193 | 1916 | 1616 | 874 | 118 | 154 |
| $\alpha$-mem | 2366 | 1432 | 920 | 623 | 91 | 120 |
| $\beta$-mem | 3866 | 1824 | 738 | 367 | 206 | 295 |
| and | 5282 | 2762 | 1353 | 1012 | 285 | 377 |
| not | 1760 | 608 | 403 | 21 | 34 | 60 |
| two | 1700 | 576 | 386 | 20 | 27 | 60 |
| any | 204 | 322 | 69 | 53 | 13 | 7 |
| p | 1931 | 1443 | 1016 | 834 | 131 | 103 |
| Total | 20302 | 10883 | 6501 | 3805 | 915 | 1176 |

Table 4-2: Nodes per Production

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| T-const | 1.65 | 1.32 | 1.59 | 1.04 | 0.90 | 1.50 |
| $\alpha$-mem | 1.22 | 0.99 | 0.90 | 0.74 | 0.69 | 1.16 |
| $\beta$-mem | 2.00 | 1.26 | 0.72 | 0.44 | 1.57 | 2.86 |
| and | 2.73 | 1.91 | 1.33 | 1.21 | 2.17 | 3.66 |
| not | 0.91 | 0.42 | 0.40 | 0.03 | 0.26 | 0.58 |
| two | 0.88 | 0.40 | 0.38 | 0.02 | 0.20 | 0.58 |
| any | 0.11 | 0.22 | 0.07 | 0.06 | 0.10 | 0.07 |
| p | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Total | 10.50 | 7.52 | 6.39 | 4.54 | 6.89 | 11.41 |

Table 4-3: Nodes per Condition Element (with sharing)

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| Total CEs | 10780 | 5541 | 3169 | 2009 | 512 | 597 |
| Total Nodes | 20302 | 10883 | 6501 | 3805 | 915 | 1176 |
| Nodes/CE | 1.88 | 1.96 | 2.05 | 1.89 | 1.79 | 1.97 |

Table 4-4: Nodes per Condition Element (without sharing)

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| Total CEs | 10780 | 5541 | 3169 | 2009 | 512 | 597 |
| Total Nodes | 55990 | 29272 | 13523 | 9178 | 1978 | 2545 |
| Nodes/CE | 5.19 | 5.28 | 4.27 | 4.57 | 3.86 | 4.26 |
| Avg. Sharing | 2.76 | 2.69 | 2.08 | 2.41 | 2.15 | 2.16 |

## 4.2. Network Sharing

The OPS5 network compiler exploits similarity in the condition elements of productions to share nodes in the Rete network. Such sharing will not be possible if a parallel implementation of the production system is desired, where each production is placed on a separate processor. To estimate the extra computation required due to loss of sharing, we present the ratios of the number of nodes in the unshared Rete network to the number of nodes in the shared Rete network in Table 4-5. The ratios do not directly give the extra computational requirements, because they are only a static measure, and the actual numbers will depend on the dynamic flow of information (tokens) through the network. Note that in the unshared Rete network, nodes are not shared between the networks for different productions, but they may be shared within the network for the same production. Also note that the reported ratios correspond to the amount of sharing or similarity exploited by the OPS5 network compiler, which may not be the same as the maximum exploitable similarity available in the production system program.

Table 4-5: Network Sharing

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| T-const | 8.76 | 7.00 | 3.80 | 5.45 | 6.69 | 6.32 |
| $\alpha$-mem | 4.05 | 3.50 | 2.84 | 2.16 | 3.92 | 4.60 |
| $\beta$-mem | 1.39 | 1.36 | 1.42 | 1.29 | 1.27 | 1.17 |
| and | 1.28 | 1.22 | 1.26 | 1.14 | 1.20 | 1.15 |
| not | 1.17 | 1.20 | 1.10 | 1.00 | 1.15 | 1.01 |
| two | 1.20 | 1.26 | 1.15 | 1.05 | 1.44 | 1.01 |
| any | 3.08 | 1.98 | 1.35 | 1.15 | 1.30 | 2.00 |

## 4.3. Network Branching

In constructing the Rete network for two similar condition elements, the network compiler shares nodes as long as the test nodes required by the two elements are same. When the compiler gets to a point where tests can no longer be shared, the network branches, and distinct paths having distinct nodes are constructed. Table 4-6 shows data about the average branching factor for the various node types. The branching factor for nodes in the network is necessary to develop models for flow of information through the network. The branching factor for the root and t-const nodes can also be used to determine when hashing is a good technique for visiting the successors of those nodes selectively.

Table 4-6: Network Branching

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| root | 31 | 36 | 81 | 23 | 20 | 12 |
| T-const | 1.78 | 1.9 | 1.7 | 1.9 | 1.9 | 1.8 |
| $\alpha$-mem | 3.2 | 2.8 | 2.4 | 2.6 | 4.1 | 4.1 |
| $\beta$-mem | 1.20 | 1.20 | 1.25 | 1.23 | 1.09 | 1.07 |
| and | 1.03 | 1.05 | 1.04 | 1.03 | 1.01 | 1.02 |
| not | 1.02 | 1.02 | 1.02 | 1.00 | 1.12 | 1.01 |
| two | 1.03 | 1.05 | 1.04 | 1.05 | 1.26 | 1.00 |
| any | 1.31 | 1.27 | 1.08 | 1.11 | 1.15 | 1.28 |

### 4.4. Summary of Rete Network Measurements

In Subsection 4.1, we presented the number of nodes found in the Rete networks for the different programs. We showed that the total number of nodes in the network correlates very well with the number of condition elements in the production system program, and on an average there are 1.9 nodes per condition element in the shared network. The number of nodes per condition element is more than doubled for an unshared Rete network.

The data in Subsection 4.2 characterizes the sharing of nodes in Rete networks. The main conclusion is that sharing is large only for t-const and $\alpha$-mem nodes, and small for all other node types. This information is useful in determining the processing requirements for unshared Rete networks. Examples of such use are presented in the next section.

In Subsection 4.3, we present data about branching in Rete networks. We hope to use this information to construct probabilistic models of flow of information in Rete networks, and also to determine when indexing will win over linear search algorithms in evaluating the successors of root and t-const nodes.

## 5. Measurements on the Run-time Behavior of the Rete Algorithm

In this section, we present data on the run-time characteristics of production system programs. The measurements are useful to identify operations frequently performed by the interpreter and to explore available parallelism in its implementation[6]. Although the reported measurements are only for the Rete algorithm, a number of general conclusions can be drawn from the measurements.

### 5.1. T-const Nodes

In Table 5-1, we present run-time statistics for t-const nodes. The first line of the table, labeled "visits/action", refers to the number of t-const node visits (activations) for each change made to the working memory. Note that "action" in this context refers to the single insertion or deletion of an element from the working memory, and not to "actions" as present in the right hand side of a production.

In the second line of the table, we present the number of t-const activations as a fraction of the total number of node activations. Although t-const node activations constitute a large fraction (57% on average) of the total node activations, a relatively small fraction of the total match time is spent in processing them. This is because the processing associated with a t-const node is very simple compared with other nodes like $\alpha$-mem nodes, or and-nodes.

[6] Note: The measurements correspond to run-time behavior of shared Rete networks, i.e., where sharing of nodes in the Rete network is allowed.

In the third line of the table, labeled "success", we report the percent of t-const node activations that have their associated test satisfied. As the numbers show, this is a very small percentage. This suggests that by using hashing, many t-const node activations that do not result in satisfaction of the associated tests can be avoided. This is especially true for the t-const nodes immediately below the root node. The tests made by the t-const nodes immediately below the root-node check for the class of the working memory element (see Figure 2-2), and since a working memory element has only one class, all but one of these t-const nodes fail their test. Our calculations show that by using hashing at this top-level, we can reduce the total number of t-const node activations by 42% (average over all systems).

Table 5-1: T-const Nodes

| Feature | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| visits/action | 122.29 | 94.04 | 119.43 | 80.96 | 35.39 | 25.97 |
| % of total | 62% | 63% | 70% | 58% | 54% | 32% |
| success (%) | 14% | 15% | 10% | 6% | 10% | 15% |

### 5.2. Alpha-mem Nodes

An $\alpha$-mem node associated with a condition element stores tokens corresponding to working memory elements that partially match the condition element: that is, tokens that satisfy all intra-condition tests for the condition element. These nodes are the first significant nodes that get affected when the working memory changes. It is only later that changes filter through $\alpha$-mem nodes to and-nodes, not-nodes, $\beta$-mem nodes, and p-nodes.

The first line of Table 5-2, labeled "visits/action", gives the average number of $\alpha$-mem nodes that are activated for each change made to the working memory. This number is significant in a number of ways: First, the number of node activations (along with the average cost associated with each activation) indicates the amount of emphasis that should be placed on optimizing the execution of $\alpha$-mem nodes. Second, the number of activations multiplied by the dynamic sharing factor,[7] gives the number of condition elements that partially match a working memory element. Under the assumption that each of the affected condition elements belongs to a different production, the number above gives the number of productions affected by a working memory element. The number of productions affected by a working memory element, in turn, has implications for the amount of parallelism that can be successfully exploited when each production is allocated a separate processor.

In the second line of Table 5-2, we report the average number of tokens present in an $\alpha$-mem node, when it

[7] Note that the numbers in Table 4-5 give the static sharing factor and not the dynamic sharing factor. They can, however, be used as rough estimates of the dynamic sharing factor.

Table 5-2: α-mem Nodes

| Feature | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| visits/action | 9.29 | 6.26 | 7.20 | 3.20 | 2.00 | 2.58 |
| avg. tokens | 56 | 13 | 17 | 2.0 | 51 | 15 |
| SD.[8] tokens | 61 | 10 | 2.4 | 0.77 | 32 | 11 |
| max. tokens | 500 | 82 | 51 | 32 | 346 | 105 |

is activated. This number indicates the complexity of the processing performed by an $\alpha$-mem node. When an $\alpha$-mem node is activated by an incoming token with a $-$ tag, the node must find a corresponding token in its stored set of tokens, and then delete that token. If a linear search is done to find the corresponding token, on average, half of the stored tokens will be looked up. Thus the complexity of deleting a token from an $\alpha$-mem node is proportional to the average number of tokens. On arrival of a token with a $+$ tag, the $\alpha$-mem node simply stores the token. This involves allocating memory and linking the token, and takes a constant amount of time. In case hashing is used to locate the token to be deleted, the delete operation can be done in constant time. However, then we have to pay the overhead associated with maintaining a hash table. Hash tables become more economical as the number of tokens stored in the $\alpha$-mem increases. The numbers presented in the second line are useful for deciding when hash tables (or other indexing techniques) are appropriate.

In the third and fourth lines of Table 5-2, we report the approximate standard deviation and the maximum number of tokens found in an $\alpha$-mem node for the various programs. The data shows that there is a large variance in the number of tokens found in an $\alpha$-mem node. This in turn implies a large variance in the processing time for $\alpha$-mem nodes (if the linear search scheme explained in the previous paragraph is used). In the model where each active node is allocated to a separate processor, a large variance in the processing time of nodes, implies less speed-up in parallel processing of nodes. The data above are also essential in the design of hardware associative memories to hold the tokens.

### 5.3. Beta-mem Nodes

The data for $\beta$-mem nodes is given in Table 5-3. $\beta$-mem nodes are very similar to $\alpha$-mem nodes, and data for them can be interpreted in the same way as that for $\alpha$-mem nodes. There are, however, a few exceptions. First, the number of $\beta$-mem node activations does not correlate in the same way to the number of productions that are affected by a working memory element. Second, while all activations of $\alpha$-mem nodes can be processed in parallel, it is not so for $\beta$-mem nodes. The reason is that, while all activations of $\alpha$-mem nodes are independent of each other, the activations of $\beta$-mem nodes are not. The activation of one $\beta$-mem node can cause the activation of another $\beta$-mem node via an intermediate and-node activation. This

[8] The numbers given for standard deviation are not exact. They are calculated from a lumped distribution of the data points, and give a lower bound of true standard deviation.

dependence reduces the amount of parallelism that can be exploited in evaluating $\beta$-mem nodes.

Table 5-3: $\beta$-mem Nodes

| Feature | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| visits/action | 3.03 | 2.41 | 3.51 | 9.28 | 1.99 | 8.49 |
| avg. tokens | 12.4 | 1.4 | 54.5 | 9.8 | 37.0 | 6.7 |
| SD. tokens | 7.6 | 0.7 | 39.4 | 1.6 | 54.7 | 4.7 |
| max. tokens | 92 | 14 | 281 | 32 | 870 | 132 |

### 5.4. And-Nodes

The run-time measurements for and-nodes are given in Table 5-4. Each line in the table consists of a pair of numbers for each of the production system programs. The numbers on the left are data for activations of and-nodes from the left, and the numbers on the right are data for activations from the right. The distinction between left and right activations of and-nodes is important. The right activations of and-nodes, which are caused by activations of the $\alpha$-mem nodes, can always be processed in parallel. The left activations of and-nodes, which are primarily caused by the activations of $\beta$-mem nodes, cannot be processed in parallel. The reason is that all $\alpha$-mem node activations can be processed in parallel, but all $\beta$-mem nodes activations can not be processed in parallel (explained in the previous subsection).

The first line of Table 5-4 gives the number of and-node activations for a single change to the working memory. The data shows that on average 77% of the activations are from the right. This indicates substantial potential for parallel execution of and-node activations.

In the second line of Table 5-4, the number on the left is the percentage of left activations of and-nodes for which no tokens were found in the associated right memory node. The number on the right gives the percentage of right activations with an empty left memory. For example, for the R1 program, the first line in the table shows that there are 34.3 activations from the right. Of the 34.3 right activations, 33.6 (98% of 34.3) have an empty left memory. Recall that an and-node activation for which there are no tokens in the opposite memory requires very little processing. Thus, evaluating the majority of the and-node activations is very cheap, and most of the processing effort is going into evaluating the small fraction of activations which have non-empty opposite memories. This also means that if all and-node activations are evaluated on different processors, then the majority of the processors will finish very early compared to the remaining few. This large variance in the processing requirements of and-nodes reduces the effective speed-up that can be obtained by evaluating each and-node activation on a different processor.

The third line shows the average number of tokens found in the opposite memory, on activation of an

Table 5-4: And Nodes

| Feature (L, R) | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| visits/action | 2.58, 34.3 | 4.8, 19.2 | 4.73, 20.16 | 17.08, 18.52 | 3.65, 16.64 | 10.38, 25.42 |
| null-mem | 53%, 98% | 75%, 95% | 43%, 90% | 61%, 77% | 5%, 91% | 21%, 72% |
| tokens | 3.0, 1.6 | 3.0, 1.3 | 18.3, 5.7 | 2.7, 3.9 | 106, 6.8 | 3.4, 5.6 |
| tests | 2.7, 1.2 | 2.4, 0.5 | 18.5, 5.7 | 2.6, 3.0 | 111, 7 | 3.4, 5.4 |
| pairs | 1.4, 1.1 | 1.4, 1.1 | 1.0, 0.67 | 1.02, 0.86 | 0.71, 0.6 | 0.64, 0.76 |

and-node, when the opposite memory is not empty. (Data about the percentage of times when the opposite memory is empty is given in the second line of the table.) This number represents the average number of tokens against which the incoming token is matched to determine consistent pairs of tokens. The magnitude of this number can be used to determine if hashing or other indexing techniques ought to be used to limit this search.

The numbers in the fourth line of the table indicate the average number of tests performed by an and-node when a token arrives on its left or right input, and its opposite memory is not empty. The number of tests performed is equal to the product of the average number of tokens found in the opposite memory (given in the third line) times the number of consistency tests that have to be made to check if the left and right tokens of the and-node are consistent.

The numbers in the fifth line of the table show the average number of consistent token-pairs found after matching the incoming token to all tokens in the opposite memory. For example, for the DAA program, on the left activation of an and-node, an average of 106 tokens are found in the right memory. On average, however, only 0.71 tokens are found to be consistent with the left token. This indicates that the right memory contains a lot of information, of which only a very small portion is relevant to the current context. The numbers in the fifth line also give a measure of token regeneration taking place within the network. We expect to use this data to construct probabilistic models of information flow within the Rete network.

### 5.5. Not-Nodes

Not-nodes are very similar to and-nodes, and the data for them are interpreted in exactly the same way as that for and-nodes. The data are presented in Table 5-5.

Table 5-5: Not Nodes

| Feature (L, R) | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| visits/action | 1.62, 8.55 | 1.34, 7.05 | 2.12, 7.99 | 0.70, 0.74 | 0.20, 1.65 | 1.27, 2.37 |
| null-mem | 32%, 93% | 30%, 90% | 18%, 90% | 59%, 58% | 7%, 45% | 7%, 46% |
| tokens | 5.9, 4.2 | 10.3, 1.2 | 19.1, 18.2 | 2.8, 1.0 | 78, 1.2 | 14.7, 8.9 |
| tests | 8.0, 6.1 | 12.3, 0.4 | 29.9, 28.8 | 2.2, 0.08 | 82, 0.25 | 14.9, 8.6 |
| pairs | 0.25, 0.39 | 0.3, 0.46 | 0.26, 0.17 | 0.05, 0.96 | 0.56, 0.05 | 0.49, 0.88 |

### 5.6. P-Nodes

Activations of p-nodes correspond to insertion and deletion of production instantiations from the conflict set. The first line of Table 5-6 gives the number of changes to the conflict set for each change made to the working memory. The second line gives the number of changes made to the working memory for every production firing, and the third line, the product of the first two lines, gives the average number of changes made to the conflict set per production firing. The data in the third line gives the number of changes that will be transmitted to a central conflict resolution processor, in an architecture using centralized conflict resolution.

Table 5-6: P Nodes

| Feature | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| visits/action | 0.96 | 1.74 | 1.72 | 1.51 | 1.98 | 3.98 |
| actions/cycle | 3.82 | 1.88 | 2.07 | 2.74 | 2.51 | 3.15 |
| mods./cycle | 3.6 | 3.2 | 3.4 | 4.0 | 5.0 | 12.6 |

### 5.7. Summary of Run-time Measurements

Table 5-7 summarizes data for the number of node activations, when a working memory element is inserted or deleted from the working memory. The data shows that a large percentage (56.5% on average) of the activations are of t-const node type. T-const activations, however, require very little processing compared to other node types, and furthermore, a large number of t-const activations can be eliminated by suitable indexing techniques (see Subsection 5.1). To eliminate the effect of this large number of relatively cheap t-const activations, we subtracted the number of t-const activations from all other node activations. These numbers are shown in the bottom line of Table 5-7, labeled "Tot – T-const". The numbers show that the total number of node activations per action is relatively independent of the number of productions. An important implication of this is that, the way production system programs are currently written, actions of productions do not have global effects, but only affect a small number of productions. Furthermore, the number of productions affected is independent of the total number of productions present in the system. It also follows that allocating one processor to each production is probably not a good idea.

Table 5-8 gives general information about the runs of the production system programs from which data is presented in this paper[9]. The first two lines of the table, give the average and maximum sizes of the working memory. The third and the fourth lines give the average and maximum values for the sizes of the conflict set. The fifth and the sixth lines give the average and maximum sizes of the token memory. (The size of the token memory at any instant is the total number of tokens stored in all memory nodes at that instant.) The last line

[9] The word NA in Table 5-8 means the data for that entry is not available

Table 5-7: Node Visits per Action

| Node Type | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| T-const | 122.79 | 94.04 | 119.43 | 80.96 | 35.39 | 25.97 |
| $\alpha$-mem | 9.29 | 6.26 | 7.20 | 3.20 | 2.00 | 2.58 |
| $\beta$-mem | 3.03 | 2.41 | 3.51 | 9.28 | 1.99 | 8.49 |
| and | 36.93 | 24.03 | 24.91 | 35.66 | 20.31 | 35.83 |
| not | 10.18 | 8.40 | 10.12 | 1.14 | 1.85 | 3.65 |
| any | 13.49 | 11.24 | 2.67 | 7.51 | 0.47 | 0.57 |
| two | 1.53 | 1.22 | 2.11 | 0.55 | 0.16 | 1.29 |
| p | 0.96 | 1.74 | 1.72 | 1.51 | 1.98 | 3.98 |
| Total | 198.2 | 149.34 | 171.67 | 139.81 | 65.15 | 82.36 |
| Tot − T-const | 75.41 | 55.30 | 55.24 | 58.85 | 29.76 | 56.39 |

in the table gives the total number of changes made to the working memory in the production system run, from which the statistics for this paper are gathered.

Table 5-8: General Run-Time Data

| Feature | R1 | XSEL | PTRANS | HAUNT | DAA | SOAR |
|---|---|---|---|---|---|---|
| Avg. WM | NA | 62 | NA | 60 | 708 | 353 |
| Max. WM | NA | 89 | NA | 63 | 1191 | NA |
| Avg. CSet | NA | 10 | NA | 2 | 43 | 9 |
| Max. CSet | NA | 22 | NA | 6 | 421 | 26 |
| Avg. TM | NA | 368 | NA | 473 | 1904 | 2413 |
| Max. TM | NA | 559 | NA | 487 | 4616 | NA |
| WM Changes | 1247 | 756 | 984 | 559 | 16839 | 631 |

## 6. Conclusions

In this paper, we have presented measurements on the static structure and the run-time behavior of production systems. Along with the measurements, we have given interpretations for some of the data. For example, we show that actions of productions do not have global effects, but only affect a small number of productions. Furthermore, the number of productions affected is independent of, and does not increase with, the total number of productions present in the system. The main purpose of giving the interpretations, however, was to serve as illustrations of how the data may be used. They should not be viewed as the only interpretations that may be given to the data, or the only interpretations that may be derived from the data.

The reported measurements form only a subset of all useful measurements that could be made on the production system programs. We think, however, that the reported measurements are comprehensive enough to form a good starting point for the design of specialized architectures for production systems. We also expect to use the measurements to develop probabilistic models of production system programs. These models will help us to predict the behavior of production system programs other than the ones we have measured.

## 7. Acknowledgments

We wish to thank Allen Newell and John Laird for valuable comments and the careful reading of earlier drafts of this paper.

## References

[1] B.G. Buchanan and E.A. Feigenbaum.
DENDRAL and Meta-DENDRAL: their applications dimensions.
*Artificial Intelligence* 11, 1978.

[2] Charles L. Forgy.
*On the Efficient Implementations of Production Systems.*
PhD thesis, Carnegie-Mellon University, 1979.

[3] Charles L. Forgy.
*OPS5 User's Manual.*
Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[4] Charles L. Forgy.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
*Artificial Intelligence*, September 1982.

[5] P. Haley, J. Kowalski, J. McDermott, and R. McWhorter.
*PTRANS: A Rule-Based Management Assistant.*
In preparation, Carnegie-Mellon University, 1983.

[6] Jin Kim, John McDermott, and Daniel Siewiorek.
TALIB: A Knowledge-Based System for IC Layout Design.
*AAAI*, 1983.

[7] Ted Kowalski and Don Thomas.
The VLSI Design Automation Assistant: Prototype System.
In *Proceedings of the 20th Design Automation Conference.* ACM and IEEE, June, 1983.

[8] John Laird and Allen Newell.
A Universal Weak Method: Summary of Results.
In *IJCAI.* 1983.

[9] John McDermott.
*R1: A Rule-based Configurer of Computer Systems.*
Technical Report CMU-CS-80-119, Carnegie-Mellon University, April, 1980.

[10] Paul S. Rosenbloom and Allen Newell.
Learning By Chunking, Summary of a Task and Model.
In *AAAI.* 82.

[11] E. H. Shortliffe.
*Computer-Based Medical Consultations: MYCIN.*
North-Holland, 1976.