The Do-loop Considered Harmful in Production System Programming

Michael van Biema
Daniel P. Miranker
and
Salvatore J. Stolfo

CUCS-228-86

In this paper we focus on some aspects of Expert System programming. In particular we consider some of the language constructs which form part of a new production system language known as Herbal that we are developing at Columbia University. These language constructs greatly increase the expressiveness of a typical production system language and can be efficiently executed on a parallel machine. We briefly describe the DADO machine under development at Columbia University and a basic algorithm for production system execution for that machine. We conclude with a discussion of some performance statistics recently calculated from an analysis of production systems simulations and describe the expected effects of our added language constructs on these statistics.

This research has been supported by the Defense Advanced Research Projects Agency through contract N00039-84-0165, as well as grants from Intel, Digital Equipment, Hewlett-Packard, Valid Logic Systems, and IBM Corporations and the New York State Foundation for Advanced Technology. We gratefully acknowledge their support.

1. Introduction

Due to the dramatic increase in computing power and the concomitant decrease in computing cost occurring over the last decade, many researchers are attempting to design computer systems to solve complicated problems or execute tasks which have in the past been performed by human experts. The focus of *Knowledge Engineering* is the construction of such complex, knowledge-based expert computing systems.

In general, knowledge-based expert systems are Artificial Intelligence (AI) problem-solving programs designed to operate in narrow "real-world" domains, performing tasks with the same competence as a skilled human expert. Elucidation of unknown chemical compounds [Buchanan and Feigenbaum 1978], medical diagnosis [Davis 1976], mineral exploration [Duda et al. 1979] and telephone cable maintenance [Stolfo and Vesonder 1982] are just a few examples.

The heart of these systems is a *knowledge base*, a large collection of facts, definitions, procedures and heuristic "rules of thumb", acquired directly from a human expert. The knowledge engineer is an intermediary between the expert and the system who extracts, formalizes, represents, and tests the relevant knowledge within a computer program.

Just as robotics and CAD/CAM technologies offer the potential for higher productivity in the "blue-collar" work force, it appears that AI expert systems will offer the same productivity increase in the "white-collar" work force. As a result, Knowledge Engineering has attracted considerable attention from government and industry for research and development of this emerging technology. However, as knowledge-based systems continue to grow in size and scope, they will begin to push conventional computing systems to their limits of operation. Even for experimental systems, many researchers reportedly experience frustration based on the length of time required for their operation. Much of the research in AI has focused on the problem of representing and organizing knowledge, but little attention has been paid to parallel machine architectures and languages supporting problem-solving programs.

In this paper we first present a brief overview of the current state of expert system technology and then outline the basic production system formalism. After this background material, we then present some the limitations of current production system programming languages and suggest a set of additions that greatly increase the expressiveness of current languages and which can efficiently be executed in a parallel environment. We then describe some of the hardware and software work that is going on at Columbia which we hope will supply empirical support for our claims. We conclude with a presentation of performance statistics recently calculated from simulations of production systems and describe the expected effect of our language constructs on these statistics.

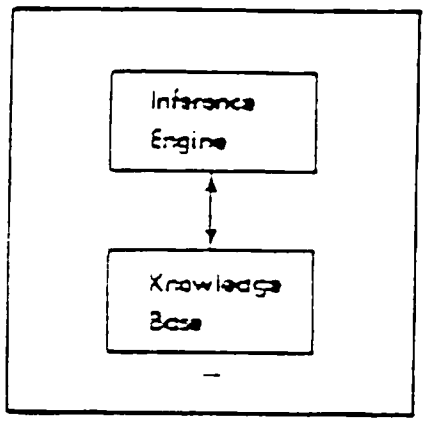
2. Expert Systems

2.1. Current Technology

Knowledge-based expert systems have been constructed, typically, from two loosely coupled modules, collectively forming the *problem-solving engine* (see Figure 2-1). The *knowledge base* contains all of the relevant domain-specific information permitting the program to behave as a specialized, intelligent problem-solver. Expert systems contrast greatly with the earlier general-purpose AI problem-solvers which were typically implemented without a specific application in mind. One of the key differences is the large amounts of problem-specific knowledge encoded within present-day systems.

Much of the research in AI has concentrated on effective methods for representing and operationalizing human experiential domain knowledge. The representations that have been proposed have taken a variety of forms including purely declarative-based logical formalisms, "highly-stylized" rules or productions, and structured generalization hierarchies commonly referred to as semantic nets and frames. Many knowledge bases have been implemented in rule form, to be detailed shortly.

Figure 2-1: Organization of a Problem-Solving Engine.



The *inference engine* is that component of the system which *controls* the deductive process: it implements the most appropriate strategy, or *reasoning* process for the problem at hand.

The earliest AI problem-solvers were implemented with an iterative branching technique searching a large combinatorial space of problem states. Heuristic knowledge, applied within a static control structure, was introduced to limit the search process while attempting to guarantee the successful formation of solutions. In contrast, current expert systems encode the control strategy and deposit it in the knowledge base along with the rest of the domain-specific knowledge. Thus, the problem-solving strategy becomes domain-dependent, and contributes to the good performance exhibited by today's systems. However, a great deal of this kind of knowledge is necessary to achieve highly competent performance.

Within a number of existing expert system programs, the corpus of knowledge about the problem domain is modified by a *Production System* program. As has been reported by several researchers [Rychener 76], production system representation schemes appear well suited to the organization and implementation of knowledge-based software. Rule-based systems provide a convenient means for human experts to explicate their knowledge, and are easily implemented and readily modified and extended. It is the ease with which rules can be acquired and explained that makes production systems so attractive.

2 Production Systems

In general, a *Production System* [Newell 1973, Rychener 1976, McDermott and Forgy 1978] is defined as a set of rules called *productions*, which form the *Production Memory*(PM), together with a database of assertions, called the *Working Memory*(WM). Each production consists of a conjunction of *pattern elements*, called the *left-hand side* (LHS) of the rule, along with a set of actions called the *right-hand side* (RHS). The RHS specifies information that is to be added to (asserted) or removed from WM when the LHS successfully matches against the contents of WM. An example production, borrowed from the blocks world, is illustrated in figure 2-2. In this paper we have chosen to give our examples using OPS5 syntax [Forgy 81]. For no other reason than that it is probably the most widely used.

Figure 2-2: An Example Production.

```
(p Blockhead
(Goal ^value Clear-top-of-Block)
(Object ^id <x> ^type Block)
(On-top-of ^object1 <y> ^object2 <x>)
(Object ^id <y> ^type Block)  -->
            (remove 3)
            (make On-top-of ^object1 <y> ^object2 Table)

If the goal is to clear the top of a block,
  and there is a block (x)
  covered by something (y)
  which is also a block,
        then
           remove the fact that y is on x from WM
           and assert that y is on top of the table.
```

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

1. *Match*: For each rule, determine whether the LHS matches the current environment of WM. All matching instances of the rules are collected in the *conflict set of rules*.
2. *Select*: Choose exactly one of the matching rules according to some predefined criterion.
3. *Act*: Add to or delete from WM all assertions specified in the RHS of the selected rule or perform some operation.

During the selection phase of production system execution, a typical interpreter provides *conflict resolution strategies* based on the *recency* of matched data in WM, as well as syntactic discrimination.

Rules matching data elements that were more recently inserted in WM are preferred, with ties decided in favor of rules that are more specific (i.e., have more constants) than others.

On conventional von Neuman machines the rules of a typical production system interpreter are often compiled into a data-flow network through which WM elements flow. State of the previously computed partial matches is normally maintained in this network in order to speed the matching process of newly inserted data. See [Forgy 1980] and [Forgy 1982] for details of his Rete match algorithm.

3. What is wrong with Do-Loops

As noted above the LHS of production system rules can be characterized as the conjunction of a series of existentially quantified terms. This causes certain difficulties when for example we wish to express such well defined semantics as: For all objects of type X do function Y. For example, suppose we were writing a farming expert system and we wanted to turn all of our rotten melons into melon balls. The standard OPS type rule would look something like:

```
(p make-melon-balls
    (current-task ^taskname melon-balls)
    (produce ^type melon ^used no ^status rotten)
-->
    (modify 2 ^type melon-ball ^used yes)
    (modify 1))
```

Here the last RHS action serves only to make the current task the most recently added working memory element (note no modification is made to the element, but rather it is just reasserted in order to be chosen by the next round of conflict resolution). In other words we must force the rule interpreter to iterate over the set of rotten melons. Another perhaps more common way to write this would be in terms of the following three rules:

```
(p make-task-melon-balls                    ;Initialization --
    (produce ^type melon ^used no ^status rotten)
-->
    (make current-task ^taskname melon-balls))

(p make-melon-balls                         ;Body
    (current-task ^taskname melon-balls)
    (produce ^type melon ^used no ^status rotten)
-->
    (modify 2 ^type melon-ball ^used yes))

(p
finish-task-make-melon-balls                ;Termination
    (current-task ^taskname melon-balls)
    - (produce ^type melon ^used no ^status rotten)
-->
    (remove 1))
```

These rules, of course, being the expression of a standard do-loop statement. What of course we really wish to write is:

```
(p make-melon-balls
    (current-task ^taskname melon-balls)
    FOR ALL (produce ^type melon ^used no ^status rotten)
->
    (modify 2 ^type melon-ball ^used yes))
```

Here we have added universal quantification to our language. The second term no longer represents the single instance of a working memory element satisfying the term, but rather the universal set of all working memory elements satisfying the term. Therefore our RHS modify action also refers to this universal set rather than a single element of it.

Let us examine what we have gained here. We have certainly made it easier to express what we really wanted to do. We have also avoided having to update the conflict set by modifying over and over the working memory element that describes the task we currently wish to do (modify 1 in the first production). On a sequential machine, however, we have a problem. Assuming that the productions have been compiled into a Rete-Match network we have no efficient way of implementing the semantics of this rule. Which is in fact probably the reason this construct is missing from the OPS class of languages.

Now assume we have at our disposal some form of associative memory. It is clear that the semantics presents no problem in this situation. Furthermore, if our associative memory has some processing power attached to it we can execute this global change to working memory in one cycle time.

Now that we have universal quantification in our language another additional construct that follows naturally is to allow predicates on the sets formed by our universal quantifier. For example, returning to our farm, suppose we know that if we have more than 10 pregnant cows we had better put the bull out to pasture. A set of OPS type rules for this would look something like: .

```
(p count-cows
    (current-task ^name count-cows)
    (counter ^value <n>)
    (^animal cow ^counted no)
->
    (modify 2 ^value (plus1 <n>))
    (modify 3 ^counted yes)
    (modify 1))

(p bull-out-to-pasture
    (counter ^value >10)
    (^animal bull ^location barn)
->
    (modify 2 ^location pasture))
```

What we really wanted to write was the following:

```
(p bull-out-to-pasture
    (cardinality (FOR ALL (^animal cow ^counted no)) > 10)
    (^animal bull ^location barn)
->
    (modify 2 ^location pasture))
```

Here we have once again increased the ease of expression in our language. In addition, this increased

expressiveness has allowed us to reduce the number of rules as well as the number of rule firings. What is needed in order to achieve this? We allow user written predicates to operate over sets. Once again there is no simple way to implement this on a sequential machine using the Rete-Match type of algorithm. On a parallel machine capable of mimicking an associative memory with some local processing power it is quite easy to visualize how these predicates might be implemented. Later we will describe how they can be implemented with performance O(log n) where n is the size of the set. The issue of side effects of these predicates is an important one, but not within the scope of this paper. Note also that we use the term predicate here in a weak sense in that other than boolean values may be returned. To see why we want this, consider the production:

```
(p apple-sauce
    (more-than-ten (FOR ALL (fruit ^type apple)))
-->
    (remove 1)
    (make (confinement ^type applesauce ^amount (cardinality-of 1))))
```

This production says that if we have more than ten apples we want to make applesauce and the final amount of apple sauce made is the cardinality of the set of apples.

We note that it is now very easy to express semantics corresponding to both set union and set intersection in our language and that this was not in general possible before our additions.

By the addition of universal quantification to our production system language we have shown that we can greatly increase the expressibility of our language and we claim that we also increase the efficiency of language on a parallel machine. We will have more to say on the issue of efficiency later, but first we describe the actual machine on which we plan to implement the language.

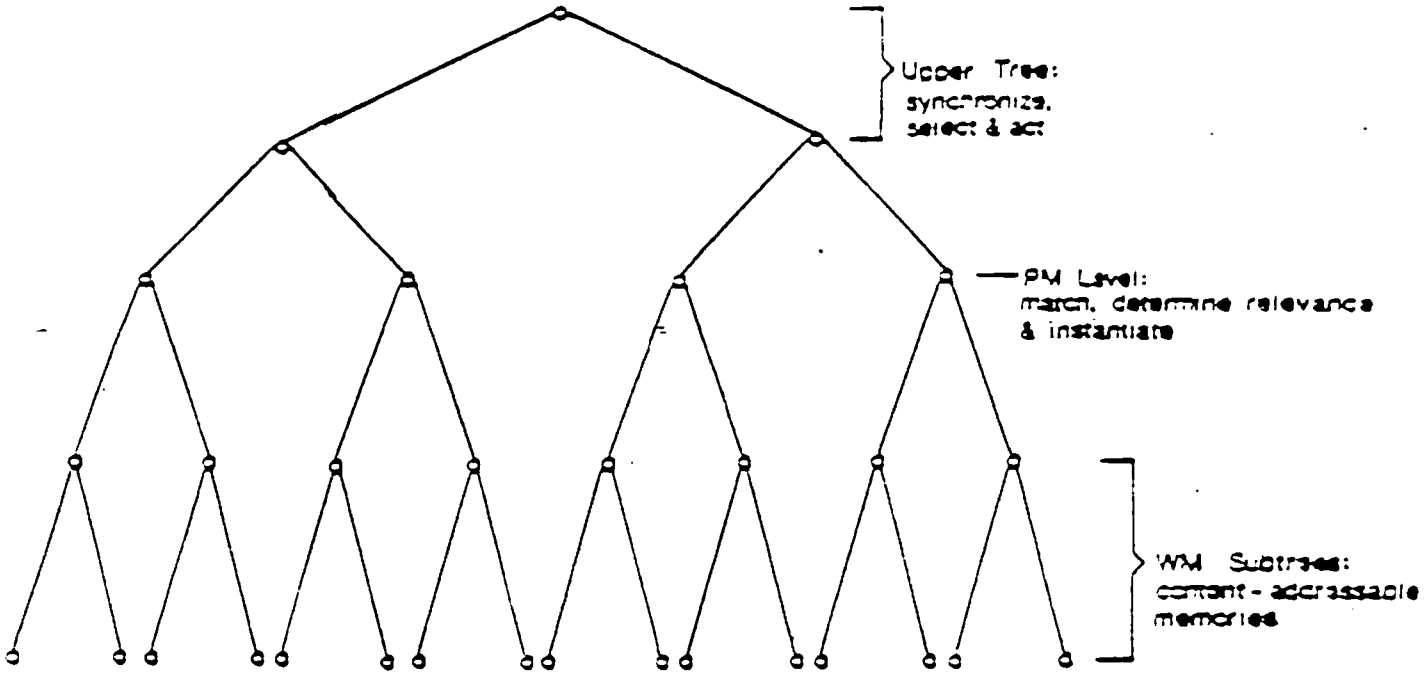
4. The DADO Machine

DADO is a medium-grain, parallel machine where processing and memory are extensively intermingled. A full-scale production version of the DADO machine would comprise a very large set of *processing elements* (PEs) (on the order of thousands), each containing its own processor, a small amount (16K bytes, in the current design of the prototype version) of local random access memory (RAM), and a specialized I/O switch. The PEs are interconnected to form a *complete binary tree* (see figure 4-1).

Within the DADO machine, each PE is capable of executing in either of two modes under the control of run-time software. In the first, which we will call *SIMD mode* (for Single Instruction Stream, Multiple Data stream [Flynn 1972]), the PE executes instructions broadcast by some ancestor PE within the tree.

In the second, which will be referred to as *MIMD* mode (for Multiple Instruction Stream, Multiple Data stream), each PE executes instructions stored in its own local RAM, independently of the other PEs. A single conventional co-processor, adjacent to the root of the DADO tree, controls the operation of the entire ensemble of PEs.

Figure 4-1: Functional Division of the DADO Tree.



When a DADO PE enters MIMD mode, its logical state is changed in such a way as to effectively "disconnect" it and its descendants from all higher-level PEs in the tree. In particular, a PE in MIMD mode does not receive any instructions that might be placed on the tree-structured communication bus by one of its ancestors. Such a PE may, however, broadcast instructions to be executed by its own descendants, providing all of these descendants have themselves been switched to SIMD mode. The DADO machine can thus be configured in such a way that an arbitrary internal node in the tree acts as the root of a tree-structured SIMD device in which all PEs execute a single instruction (on different data) at a given point in time. This flexible architectural design supports *multiple-SIMD* execution (MSIMD) as for example [Siegel et al. 1981] but on a much larger scale. Thus, the machine may be logically divided into distinct partitions, each executing a distinct task. This is the primary source of DADO's speed in executing a large number of primitive pattern matching operations concurrently.

The DADO I/O switch, has been implemented in semi-custom gate array technology and incorporated within the 1023 processing element version of the machine, has been designed to support rapid global communication. In addition, a specialized combinational circuit incorporated within the I/O switch allows very rapid selection of a single distinguished PE from a set of candidate PEs in the tree, a process

called *resolving*.

The many advantages of the binary tree architectures such as scalablity have been pointed out elsewhere[Stolfo, 1983] and we will not reiterate them here. What is important from the language point of view is that the tree architecture allows the implementation of O(log n) tree associative operations.

4.1. Production System execution on DADO

In this section we outline an abstract algorithm for production system execution on DADO. Although we have actually developed 6 different algorithms which cater to different classes of production systems we present only the simplest one here as it is sufficient for a discussion of the language issues in which we are interested. As one might well imagine the distribution of productions and working memory to the tree has very important effects on performance [Ishida 1984]. A detailed treatment of these algorithms has appeared elsewhere [Stolfo 1984] [Miranker 1984b].

4.2. Original DADO Algorithm

The original DADO algorithm detailed in [Stolfo 1983] makes direct use of the machine's ability to execute in both MIMD and SIMD modes of operation at the same point in time. The machine is logically divided into three conceptually distinct components: a *PM-level*, an *upper tree* and a number of *WM-subtrees* (see figure 4-1). The PM-level consists of MIMD-mode PEs executing the match phase at one appropriately chosen level of the tree. A number of distinct rules are stored in each PM-level PE. The WM-subtrees rooted by the PM-level PEs consist of a number of SIMD mode PEs collectively operating as a content-addressable memory. WM elements relevant to the rules stored at the PM-level root PE are fully distributed throughout the WM-subtree. The upper tree consists of SIMD mode PEs lying above the PM-level, which implement synchronization and selection operations.

It is probably best to view WM as a distributed *relation*. Each WM-subtree PE thus stores relational tuples. The PM-level PEs match the LHS's of rules in a manner similar to processing relational queries. In terms of the Rete match, *intracondition* tests of pattern elements in the LHS of a rule are executed as relational *selection*, while *intercondition* tests correspond to *equi-join* operations. Each PM-level PE thus stores a set of relational tests compiled from the LHS of a rule set assigned to it. Concurrency is achieved between PM-level PEs as well as in accessing PEs of the WM-subtrees. The algorithm is illustrated in figure 4-1.

It is quite easy to see how to map the language constructs we described in Section 2 on top of this algorithm. The FOR ALL constructs merely enables all PEs with WM elements satisfying the term the FOR ALL modifies and disables any PEs not containing such elements. This is basically just using the tree as an associative memory. The set predicates can be mapped into tree associative operations on the enabled set of PEs. As we have already stated these operations can be performed in O(log n) time, assuming that WM is fully distributed and that the size of the set is large.

Figure 4-2: Original DADO Algorithm.

1. Initialize: Distribute a match routine and a partitioned subset of rules to each PM-level PE. Set CHANGES to the initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do:
    a. Broadcast WM-change to the PM-level PEs and an instruction to match.
    b. The match phase is initiated in each PM-level PE:
        i. Each PM-level PE determines if WM-change is relevant to its local set of rules by a partial match routine. If so, its WM-subtree is updated accordingly. [If this is a deletion, an associative probe is performed on the element (relational selection) and any matching instances are deleted. If this is an addition, a free WM-subtree PE is identified, and the element is added.]
        ii. Each pattern element of the rules stored at a PM-level PE is broadcast to the WM-subtree below for matching. Any variable bindings that occur are reported sequentially to the PM-level PE for matching of subsequent pattern elements (relational equi-join).
        iii. A local conflict set of rules is formed and stored along with a priority rating in a distributed manner within the WM-subtree.
    c. end do;
4. Upon termination of the match operation, the PM-level PEs synchronize with the upper tree.
5. Select: The max-RESOLVE circuit is used to identify the maximally rated conflict set instance.
6. Report the instantiated RHS of the winning instance to the root of DADO.
7. Set CHANGES to the reported action specifications.
8. end Repeat;

5. Parallelism in Production Systems

A nice study of parallelism in OPS style production systems has recently been completed [Gupta 1984]. The somewhat surprising result of this study is that potential parallelism in OPS style production systems is very low. Although surprising at first, on closer examination this finding is not in fact so surprising, The OPS languages have been specifically designed to enable their efficient implementation on sequential machines. These languages therefore encourage users to serialize their algorithms. The most blatant example of this is, in fact, the absence of universal quantification from OPS style languages. The result of this is that the programmer is forced to write rules to explicitly iterate over sets of working memory elements.

There are three possible major sources of production system parallelism. They are not surprisingly: production parallelism, action parallelism and conflict parallelism. Most current estimates place 80-90% of production system execution time in the match phase. Hence a significant speedup must be obtained by matching in parallel if the production system is to be efficiently executed. The affect-set, the number of productions affected by a single WM change, and therefore the number of productions for which matching maybe done in parallel is therefore critical. Gupta has found the average size of the affect-set to be quite low (around 32). We, however, hypothesize that this may be in part due to the sequential enumeration of various WM sets. Since the affect-set size for such an enumeration is 1 and these enumerations may constitute a significant proportion of the rule firings this may well account for the small average affect-set size. Unfortunately Gupta does not report the standard deviations of his averages which would aid in the evaluation of the validity of this hypothesis.

We note that our constructs increase not only the potential production parallelism, but also the action parallelism and conflict parallelism. These do account for a much smaller percentage of the cycle time and we therefore expect their overall effect to be less significant. Finally the total number of production cycles may be significantly reduced by the addition of these constructs since we replace iteration over a set of WM elements by a single parallel operation on the set. In the ACE system it is estimated that a large percentage of its time is spent executing precisely such rules[1]. Our own studies at Columbia on a simple expert system that does Waltz labeling has shown that we can reduce the static number of rules by a factor of 4 and the number of execution cycles by a factor of 10. We are not claiming that such good results may be obtained for all classes of expert systems, but that there does exist a large group of expert systems where such results may be easily obtained. Guta has also pointed out that for a version of XSEL system [McDermott 1981] that directly accesses an external database such behavior is observed as well as a much larger average affect-set size.

---

[1] Private communication with Greg T. Vesonder of AT&T Bell Labs.

6. Conclusions and the Future

We have described the addition of several constructs involving universal quantification to OPS style production systems. These constructs have been shown to add significantly to the expressiveness of the language and, unlike most such constructs, have also been shown to increase the efficiency of execution in a parallel environment. What remains is to provide the further empirical support for our conclusions by analyzing existing production systems and possibly recoding them using these new constructs.

The constructs we have suggested are in some sense the easy ones. They immediately came to mind in the context of thinking about production systems and parallelism. What remains to be done is to search for other, less obvious constructs that will increase either the expressiveness or the parallelism of production system languages, or better yet which increase both. Finally, a model should be developed so that new constructs can be evaluated to the degree of parallelism they provide and how they interact. This search forms a major part of the current research being conducted by the DADO parallel computer project.

REFERENCES

Buchanan, B. G. and Feigenbaum, E. A.
"DENDRAL and Meta-DENDRAL: Their applications dimension",
Artificial Intelligence, 11:5-24, 1978.

Davis, R. "Applications of meta-level knowledge to the
construction, maintenance and use of large knowledge bases",
Computer Science Department, Stanford University,
Rep. No. STAN-CS-76-552, 1976.

Duda, R., Gashnig, J. and Hart, P.E.
"Model design in the PROSPECTOR consultant system for mineral exploration",
In D. Michie (Ed.), Expert systems in the micro-electronic age,
Edinburgh University Press, 153-167, 1979.

McDermott, J. and C. Forgy, "Pattern-directed Inference Systems"
Academic Press, 1978.

Forgy, C. L., "A Note on Production Systems and *ILLIAC IV*",
Technical Report 130, Department of Computer Science,
Carnegie-Mellon University, 1980.

Forgy, C. L., "OPS5 User's Manual", Technical Report,
Carnegie-Mellon University, Order Number CMU-CS-81-135, 1981.

Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/ Many Object
Pattern Match Problem", Artificial Intelligence 19, 1982.

Gupta, A., "Parallelism in Production Systems:
The Sources and the Expected Speed-up", Technical Report,
Carnegie-Mellon University, Order Number CMU-CS-84-169, 1984.

Ishida T., and S. J. Stolfo,"Simultaneous Firing of Production Rules on
Tree-structured Machines", Technical Report, Department of Computer
Science, Columbia University, 1984.

McDermott, J., "*R1*: The Formative Years", AI Magazine
2:21-29, 1981.

Newell, A., "Production Systems: Models of Control Structures",
In W. Chase (editor), Visual Information Processing,
Academic Press, 1973.

Rychener, M., "Production Systems as a Programming Language for
Artificial Intelligence Research", Ph.D. thesis, Department of Computer
Science, Carnegie-Mellon University, 1976.

Siegel, H. J., L. J. Siegel, F. C. Kemmerer, P. T. Mueller, H. E.
Smolky and D. S. Smith, "PASM: A Parititionable SIMD/MIMD System for
Image Processing and Pattern Recognition",
IEEE Tran. on Computers, 1981.

Stolfo, S. J., "The DADO Parallel Computer", Technical Report,

Department of Computer Science, Columbia University, 1983.

Stolfo, S. J., and G. T. Vesonder, "ACE: An Expert System Supporting Analysis and Management Decision Making", Bell System Technical Journal, 1982.

Table of Contents

List of Figures