Control in Parallel Production Systems
A Research Prospectus

Daniel E. Neiman

COINS Technical Report 91-2

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
CSNET: DANN@CS.UMASS.EDU

Abstract

This report describes a prospectus for research on control issues raised by incorporating rule parallelism into production systems. The proposed research develops the thesis that successful development of parallel production systems requires a careful analysis of the nature of the algorithms being implemented, a re-evaluation of existing conflict resolution techniques, construction of algorithms with a tolerance for temporary inconsistencies in working memory, development of a vocabulary for describing temporal interdependancies between rules and data, and language mechanisms for supporting parallel rule execution.

[0]This work was supported in part by the Office of Naval Research under a University Research Initiative grant, ONR N00014-86-K-0764, and NSF-CER contract DCR-8500332

Abstract

Rule-based programming requires matching productions against a state memory in order to determine which rules are able to fire. It is commonly held that matching is the most time-consuming phase of computation in these systems. Previous studies have shown that parallelism can improve the performance of a rule-based system by increasing the speed of the matching process. In general, the speed up is limited by the number of productions affected by a change to memory, the number of changes to memory taking place, the size of the memory elements, the number of elements currently in memory, and the cost of scheduling parallel processes. In existing OPS5 systems, the average values of these parameters [Gupta89a] indicate that the potential speedup due to match-level parallelism is about one order of magnitude. Most of the systems which were analyzed use control structures which rely on conflict resolution and the serial nature of the OPS5 recognize-match-act cycle.

More recent research has demonstrated that executing rules in parallel can greatly increase the performance of production systems. Unlike matching parallelism which is largely transparent to the programmer, parallel rule execution requires significant changes to the rules and control structures of the system. In the case of these systems which allow multiple productions to fire simultaneously, the issue of which rules can co-execute must be considered. Current techniques for identifying and managing interactions between rules require a resource intensive syntactic analysis.

Control of parallel production systems is problematic. For example, experiments which I have performed using a parallel version of OPS5 indicate that in systems which execute many productions concurrently, the standard control mechanism, conflict resolution, turns out to be a bottleneck. The standard conflict resolution algorithm requires that all working memory changes and entries/deletions to the conflict set be completed before a rule is selected, this causes the maximum level of parallelism to be greatly reduced. If *no* control mechanism is applied, then rules may interfere with each other, may fire redundantly, or may exceed the computing capacity of even a parallel machine. The problem of control is further complicated by the relatively short execution times of productions which tends to preclude lengthy deliberation. The question of controlling a highly parallel production system without incurring unacceptable overhead is considered and a number of approaches are proposed, including a functionally accurate/cooperative control algorithm, a concurrent control scheme which monitors asynchronous production firings, and a meta-language for expressing control requirements.

The proposed research develops the thesis that successful development of parallel production systems requires a careful analysis of the nature of the algorithms being implemented, a re-evaluation of existing conflict resolution techniques, construction of algorithms with a tolerance for temporary inconsistencies in working memory, development of a vocabulary for describing temporal interdependancies between rules

and data, and language mechanisms for supporting parallel rule execution.

Contents

1 Introduction 5

2 OPS5: A Case History of a Rule-based Language 8
- 2.1 Control of OPS5 Programs . . . 10
- 2.2 The Rete Net . . . 11

3 Research in Parallel Production Systems 12
- 3.1 Compilation of the Rete Net . . . 13
- 3.2 Parallelism in OPS5 . . . 13
- 3.3 Production Parallelism . . . 14
- 3.4 Node and Intra-node Parallelism . . . 15
- 3.5 Extremely Fine Grained Parallelism within the Rete Net . . . 17
- 3.6 Action Parallelism . . . 17
- 3.7 Application Parallelism . . . 17
- 3.8 Parallel Execution of Rules . . . 20
  - 3.8.1 Achieving Serializable Behavior in a Parallel Program . . . 20
  - 3.8.2 Parallel Rule Firing with Fuzzy Logic . . . 22
- 3.9 Architectures for Production Systems . . . 24
  - 3.9.1 DADO . . . 24
  - 3.9.2 Implementation of OPS5 on Non-Von . . . 25
  - 3.9.3 CUPID and DRete . . . 25
  - 3.9.4 Message Passing Architectures . . . 26
  - 3.9.5 Shared Memory Architectures . . . 26

4 Parallelism in OPS5 – Research to Date 26
- 4.1 Implementation of a Parallel OPS5 . . . 27
- 4.2 Implementation Environment . . . 28
- 4.3 Benchmarks . . . 28
  - 4.3.1 Experiment 1: Explicit Synchronization . . . 29
  - 4.3.2 Experiment 2: Synchronization via Conflict Set . . . 30
  - 4.3.3 Experiment 3: Asynchronous Production Execution . . . 31
- 4.4 Summary of Experiments . . . 31

5 Proposed Research 33
- 5.1 Control Issues . . . 34
  - 5.1.1 Definitions of Control . . . 34
  - 5.1.2 Removing the Conflict Set Bottleneck . . . 35
- 5.2 Controlling Parallel Production Systems . . . 35
  - 5.2.1 Concurrent Control . . . 37
  - 5.2.2 Algorithms for Functionally Accurate Computations . . . 40

5.2.3 Meta-Rules 42
5.3 Research Contribution 43
5.4 Related Work 44
5.4.1 Parallel Blackboard Systems 44
5.5 Research Program 46
5.5.1 Models of Rule Interactions 46
5.5.2 Development of Algorithms for Parallel Control 47
5.5.3 Development of an Intelligent Controller for Parallel Rule Execution 47
5.5.4 Development of a Parallel Production System 47
5.6 Results 48

6 Conclusion 48

1 Introduction

Production systems are a popular method for implementing expert (or knowledge-based) systems and have been proposed as cognitive models of intelligent activity. The main source of power of the production system formalism is that it allows a rule to represent a single 'fact' or unit of knowledge in a discrete form. This encapsulation of knowledge simplifies the tasks of knowledge acquisition, learning, program modification, interactive transfer of expertise, and explanation. The principle disadvantage of production systems is that they tend to be slow due to the high overhead required to match each rule against all the relevant elements of the knowledge base in order to determine if that rule is eligible to fire. As the number and complexity of rules and the size of working memory increase, performance decreases proportionally, limiting the practical applications of production systems.

Various algorithms (e.g. Rete and TREAT) have been proposed to speed up the matching process. In general, these algorithms take advantage of the observation that the world (as represented in working memory) changes slowly and therefore the overhead of the match process can be reduced by storing the results of previous partial matches. Various studies have been performed to estimate the speedup resulting from parallelizing these algorithms. These algorithms appear to be very suitable for parallelism, yet parallelizing them yields disappointing results when benchmarked using *existing* rule-based systems[Gupta87]. Briefly, one of the reasons for these disappointing results is that the benefits of parallelism within the matching process are directly related to the number of productions affected by each working memory change; due to the focussed nature of existing expert systems this number tends to be small. Various researchers [Ishida85, Miranker89, Schmolze89, Nii89] have pointed out that existing rule-based systems on which the benchmarks are based were not designed with parallelism in mind, and that, in fact, the standard control structures used in rule-based systems act to limit parallelism.

Research has recently focussed on systems which can better exploit parallelism by allowing multiple rules to fire concurrently. In order to fully understand the implications of such systems, consider the typical algorithms employed at the rule level in production systems and the purposes for which they are employed. Applications typically use rules for either recognition or data retrieval, as well as for low-level control. The need for performance (and thus parallelism) is particularly acute in systems which contain large numbers of rules operating on a large working memory database, or in systems which operate under real-time constraints in which response time is critical.

While production systems impose no *a priori* control structure on the sequencing of rule executions, it is possible, by executing productions in a controlled fashion, to model virtually any conventional AI paradigm: search, forward or backward chaining, goal-directed reasoning, etc... Each rule execution acts as an operator in each of these

algorithms, and the process of matching can be viewed as the process of updating the state of the system and identifying enabled operators. The benefit of a parallel rule-based system depends considerably on the ability of the particular application to support parallel decomposition of operations.

In a system which allows concurrent production execution, it is necessary to devise algorithms which will determine when rules can be run in parallel without developing pathological interactions between productions which would result in mutual disabling of rules, false firings, and the creation of anomalous working memory states which could not appear in the corresponding serial version of the production system. Various methods have been developed which perform syntactic analyses of rules or rule instantiations and determine whether interactions exist. These methods, when performed statically, serve to limit the available parallelism, and when performed dynamically, are sufficiently time-consuming so as to impose a severe control overhead on the parallel system. When rules are determined to interact, the solution is to *synchronize* the rules by selecting only one for execution.

One hypothesis that will be explored in this research is that most undesireable rule interactions can be avoided if the rule base is designed for parallel execution with an understanding of the role that each rule plays in the overall algorithm. It is inevitable, however, that rules will occasionally conflict, for example, cases will arise in which multiple sources of knowledge apply to the same situation. The synchronization approach proposed in this work is to generate a meta-language describing the intention of each rule instantiation so that conflicts can be resolved in keeping with the goals of the high-level paradigm.

Synchronization is not the only control problem encountered in a parallel production system. Not only is control serializing, but most conventional control mechanisms are simply not appropriate to parallel production systems.

In a typical production system, control is synonymous with conflict resolution. In order to decide between alternative activities, the scheduler first identifies all possible actions, evaluates their relative merits, and selects a rule to execute. In a serial system, the cost of the evaluation function is justified by the increase in efficiency caused by applying the correct operator. In a parallel rule-based system, the distinction between conflict resolution and control becomes more sharply defined[Miranker89]. In a parallel system, multiple alternatives should no longer be considered mutually exclusive and the necessity for choosing between them decreases. Ideally, a problem solver could investigate all potential solution paths simultaneously, reaching the best solution in minimum time. In a realistic system, however, the number of processing resources will be limited and some measure of control must be applied to ensure that the resources are used effectively. A parallel rule-based system, therefore, requires a sophisticated control mechanism which not only controls the sequencing of operations, but which also determines when and if rules should fire, monitors progress towards a solution, and allocates resources effectively. The implications of these observations

are that the conventional model of a production system as consisting of three separate stages of *match*, *conflict-resolve*, and *act* must be extensively modified if high degrees of parallelism are to be achieved.

Finally, the desire to provide the maximum possible rate of production execution is antithetical to the control goals described above. The need to perform conflict resolution implies that the system must first reach quiescence so that no further potential operators will enter the conflict set, and this synchronization delay decreases the rate of rule firing. If conflict resolution is eliminated, then rules may be executed as soon as they enter the conflict set (asynchronously). Experiments which involved executing productions asynchronously indicate that this execution strategy results in significantly improved utilization of processing resources, but at the cost of possible temporary inconsistencies within the conflict set and the loss of opportunities for control. This research will investigate techniques for executing productions asynchronously and identify the tradeoffs between maximum execution rates and control requirements. Many of the techniques developed to allow asynchronous production execution will also be applicable to distributed systems which present many of the same problems with temporal inconsistencies due to communications delays in updating conflict sets and working memory[Schmolze90].

To summarize the above: current production systems provide no support for programming in parallel and, in fact, encourage programming idioms which tend to serialize production execution. The existence at UMass of a parallel Lisp running on a multiprocessor gives us the opportunity to develop, study, and benchmark production system algorithms specifically designed to run in parallel at both the implementation and application levels. It is the goal of my research to determine what class of problems are best suited for parallel production execution, which problems require some degree of control, and to understand the nature of the control algorithms which will permit the highest degree of parallelism without sacrificing clarity and expressive ability.

Outline of Proposal: A certain amount of background would be helpful in understanding the concepts presented in this proposal. Section 2 will present a brief description of the production system language OPS5. Section 3 discusses previous approaches to increasing the speed of production systems by compilation of the pattern-matching routines, construction of special-purpose architectures, and parallelizing the rule matching and firing process. In Section 4, I discuss the current state of my research in developing algorithms to support parallel production execution and the implementation which supports this research. Finally, Section 5 describes my proposed research directions, how they relate to previous work, and their contributions to the field.

2 OPS5: A Case History of a Rule-based Language

The programming language OPS5 was written by C. Forgy at Carnegie-Mellon University as the fifth in a series of production system languages[1] [Forgy81]. It achieved a great deal of popularity largely because of its use in the R1 project[McDermott80], its general availability as a public domain program, and its efficiency due to the use of the Rete net pattern matcher (described in Section 2.2).

An OPS5 program consists of *rules* matching against a *working memory*. Working memory consists of a set of facts. Each fact is represented as a linear set of attribute-value pairs associated with a class, i.e.

```
(class ^att val ^att val ^att val ...).
```

Working memory elements are created using the *make* command, deleted using the *remove* command and modified using the *modify command*. The modify command simply does a remove followed by a make which deletes and recreates the working memory element. At creation time, each working memory element is given a unique timetag which identifies that element; two otherwise identical working memory elements created at different times are assigned distinct timetags. A rule consists of a lefthand side (LHS) pattern and a righthand side (RHS) set of actions. (The terms lefthand and righthand side are due to the fact that productions have historically been written as LHS -> RHS). The lefthand side consists of a series of patterns. The rule is considered eligible to fire when there exists one or more sets of working memory elements such that there is one working memory element in the set for every positive pattern in the LHS, and there is no working memory element in working memory that matches any negated pattern. LHS patterns consist of conjuctions; the only way to program disjunctions (`IF A or B THEN ...`) is to code them as multiple productions.

The righthand side of a production consists of actions. These actions can consist of changes to working memory, I/O operations, or arbitrary function calls. Because one of the goals of parallelizing OPS is to increase the potential for parallelism in the matching process by increasing the throughput in working memory, I will make the assumption in my discussions that the righthand side consists exclusively of changes to working memory unless otherwise stated. Because the matching process is by far the most expensive operation in processing a working memory change, this assumption virtually eliminates the distinction between the matching and execution phases in the production system.

An example of a small OPS5 rule set is shown in Figure 1.

[1] OPS reportedly stands for Official Production System.

```
(literalize cat name state action)

(literalize see cat obj)

(literalize attack attacker victim)

;If cat is hungry and cat sees food, cat will eat food.

(p hungry_cat
  (cat ^name <kitty> ^state hungry)
  (see ^cat <kitty> ^obj food)
 -->
  (modify 1 ^action eat))

;If cat is hungry and cat sees critter, cat will try to eat critter.

(p hunting_cat
   (cat ^name <kitty> ^state hungry)
   (see ^cat <kitty>  ^obj  << pigeon duck fish >>  <victim>  )
  -(attack ^attacker <kitty>)
-->
   (modify 1 ^action pounce)
   (make attack ^attacker <kitty> ^victim <victim>) )

;If cat has nothing better to do, it will purr.

(p happy_cat
   (cat   ^name <kitty>  ^action <> purr  )
  -(cat   ^name <kitty>  ^state << aggressive hungry >> )
 -->
   (modify 1 ^action purr))

;Cats are territorial beasts

(p aggressive_cat
  (cat ^name <kitty> ^state <> aggressive)
  (see ^cat <kitty> ^obj cat)
 -->
  (modify 1 ^state aggressive ^action hiss))

;Cats have no respect for expensive furniture and houseplants.

(p playful_cat
  (cat ^name <kitty> ^state playful)
  (see ^cat <kitty> ^obj  << string houseplant hallucination >> <victim> )
 -(attack ^attacker <kitty>)
 -->
  (make attack ^attacker <kitty> ^victim <victim>))
```

Figure 1: A "Complete" Cognitive Model of *Felis Domesticus*

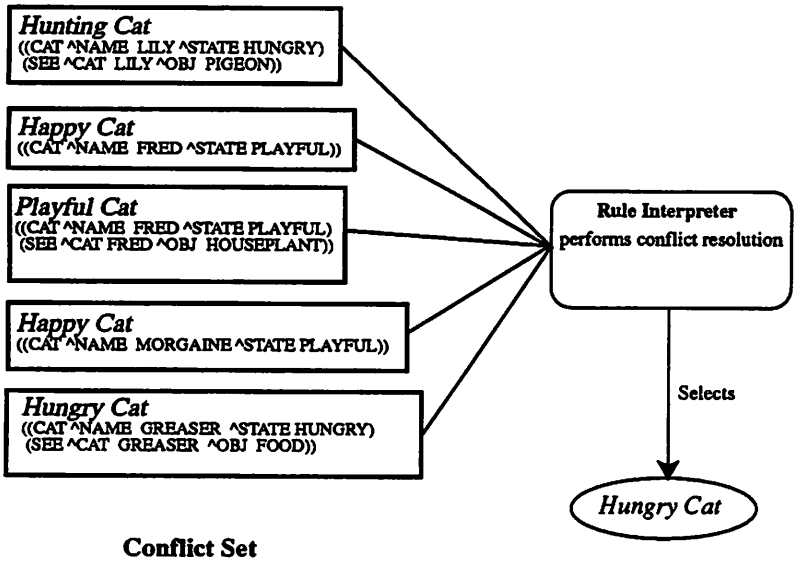


Figure 2: An example of conflict resolution.

2.1 Control of OPS5 Programs

A rule-based system is data-driven; the rules which are considered to be eligible to fire depend entirely on the state of working memory. Because only one rule can execute at a time in a serial system, if more than one rule is eligible to fire, the production system must perform *conflict resolution.* During conflict resolution, all eligible rules are examined and the one which is perceived to be most useful according to the conflict resolution algorithm is fired (see Figure 2.1). Because of the lack of imperative control mechanisms, programmers of production systems frequently manipulate the conflict set in order to obtain a specific sequence of rule executions. As productions fire, they change working memory which in turn changes the contents of the conflict set. Therefore, conflict resolution must be performed after each production execution.

Conflict resolution algorithms are typically optimized to be fast and heuristic, using only syntactic information which can be quickly accessed [McDermott78]. The conflict resolution algorithms in OPS5, MEA and LEX, are typical in this respect; they select rule instantiations based primarily on the creation time of working memory elements and the number of condition elements in the lefthand side of a production. The use of more sophisticated meta-rules or scheduling algorithms [Davis80, Hayes-Roth85] is difficult due to the inability of OPS5 to express meta-level patterns. The issue of control in OPS5 and possible modifications to the language will be addressed in greater detail in a later section of this proposal.

2.2 The Rete Net

In production systems, most of the processing time is spent determining which rules are eligible to fire. In OPS5, this process consists of matching the lefthand sides of productions against working memory. When a set of working memory elements is found such that there is a working memory element for every non-negated condition element in the lefthand side and there exist no elements which match negated condition elements, the rule is eligible to fire. As a principle bottleneck in rule firing, this matching process should be as fast as possible. The Rete net is an efficient implementation of a pattern matcher based on the following observations:

- Working memory changes only incrementally from cycle to cycle.
- Many productions in a rule base are frequently structurally similar and may share one or more terms.

The first observation implies that it should be possible to store partial matches and only match against those working memory elements which change, rather than implementing the naive approach of comparing each production against all of working memory after each set of working memory changes. Sharing of tests between productions reduces the total number of comparisons that must take place.

The matching process works by passing tokens consisting of one or more working memory elements through the net, performing tests on them at each node. The 'top' of the Rete net is composed of *alpha* nodes which consist of simple tests on the class of the working memory element and specific fields. This part of the network possesses no memory and resembles a conventional discrimination net; tokens are passed to suceeding nodes in the network only if the tests at the current node succeed. Alpha tests are not very time-consuming and parallelizing their execution does not lead to large improvements in performance.

Beta tests are responsible for unifying variable values between fields of a condition element (intra-element tests) or between two condition elements (inter-element tests). Each of the beta nodes has two inputs and two memories, one associated with each input. As a token arrives at a beta node, it is stored in memory and tested against the *opposite* memory to see if one or more consistent bindings can be achieved. If so, a new token is constructed from the incoming token and the stored token. This new token is then propagated through the beta node's out list (a list of successor nodes). The memories associated with the beta nodes store partial matches, making it unnecessary to repeat the entire computationally expensive unification process after each working memory modification. The cost of executing a beta node is proportional to the size of the memory against which the incoming token is tested. The two main beta nodes are the **AND** and **NOT** nodes. Beta nodes present numerous opportunities for parallelism; for example, multiple beta nodes can be executed in parallel, or, if the architecture supports sufficiently fine-grained processing, an incoming token can be compared to each corresponding token in memory simultaneously.

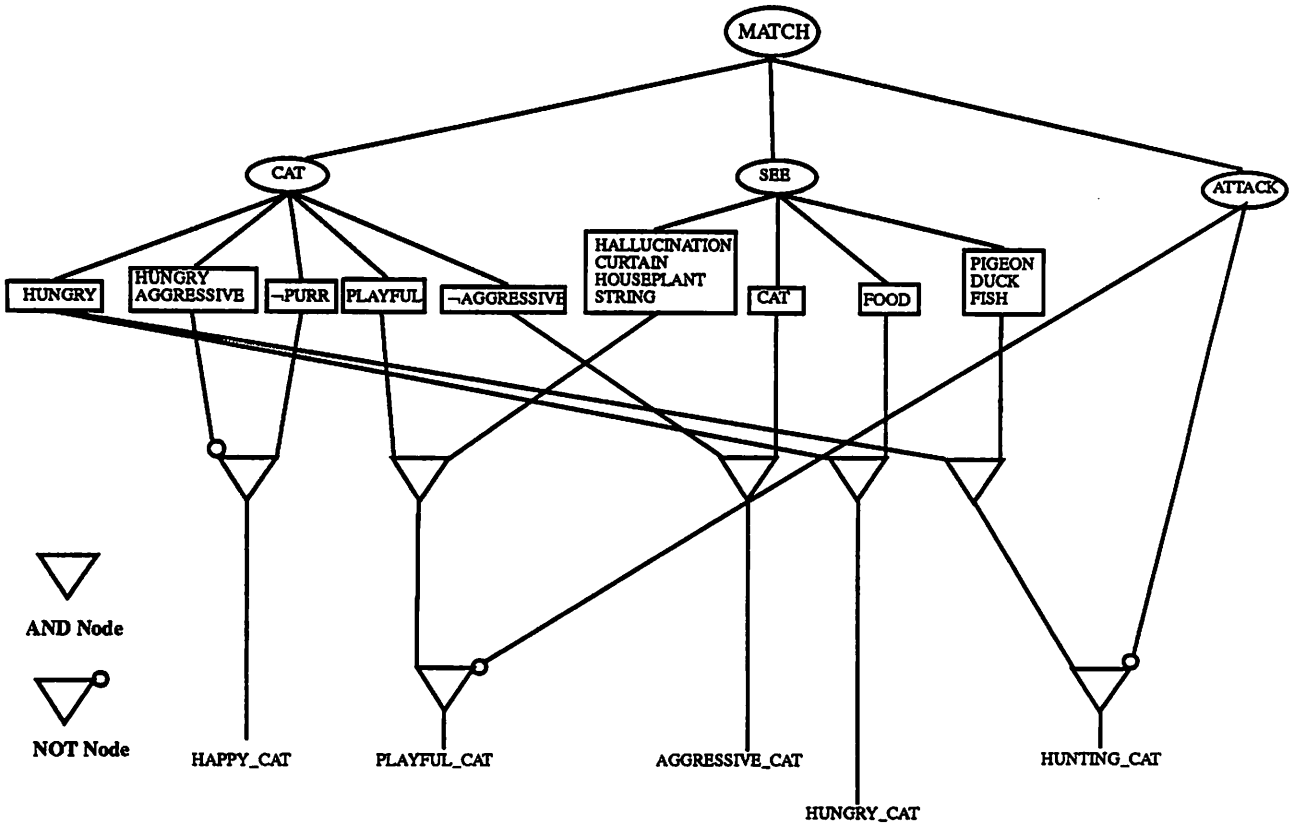


Figure 3: The Rete net for a simple OPS5 program

At the bottom of the Rete net is a series of *production* nodes; when a token arrives at one of these nodes, the production corresponding to the node is placed in the conflict set, instantiated with variable bindings from the incoming token. The production node has no memory, thus only one production firing ever results from a given combination of working memory elements.

Figure 2.2 shows the Rete net for the simple OPS5 example.

The implementation of the Rete net is quite complex and a complete discussion is beyond the scope of this document. For a more thorough reference see [Forgy82], or, for an implementation specific description of the public domain version of OPS5, see [Neiman87].

3 Research in Parallel Production Systems

A considerable amount of research has been done on increasing the speed of production systems and of OPS5 in particular. The research divides into a number of categories:

- Increasing the efficiency of pattern matching through compilation.
- Specialized architectures customized for rule matching and execution.
- Parallelism.

These categories are not necessarily distinct; in particular, many of the architectures proposed for production systems have incorporated parallelism in their design.

3.1 Compilation of the Rete Net

The Rete net algorithm, as originally coded, was interpreted. The tests for a particular node were evaluated at run time and applied to incoming tokens. In his original paper on the Rete net[Forgy79], Forgy described an approach towards compiling the pattern matching network directly into assembler which he then used in implementing the programming language OPS83. Compilation of rules allows the production system to run substantially faster than the interpreted version. The ops5c compiler, a compiled version of the TREAT algorithm written by Miranker was reported to provide speedups of 50-200 times [Miranker90].

There are disadvantages to the compiled approach to pattern matching; typically, new productions can not be incrementally added to the system, debugging of rules is more difficult (because less information about the state of the network can be accessed), and the compilation itself is time-consuming. While compilation can reduce the time required to match productions against working memory substantially, there still exists an upper bound on the rule execution speed which can be achieved using a uniprocessor simply because the righthand side of a rule can contain an arbitrary number of actions, including relatively slow I/O operations and calls to arbitrary Lisp code.

It is not clear whether the compiled implementations of the pattern matching algorithms allow for low-level parallelism, although such an approach should be effective for applications in which the average size of node memories or number of node activations are large. Increased performance can certainly be achieved by executing multiple productions in parallel; the effectiveness of the parallelism will probably also be increased due to decreased time spent in critical regions.

3.2 Parallelism in OPS5

One of the principle studies on parallelism in OPS5 has been done by Anoop Gupta in CMU[Gupta84]. In a very thorough analysis, he demonstrated that the average speedup in production systems due to parallelism would be much less than expected (on the order of 10's rather than 1000's). There is little question that this analysis holds for *existing* applications of production systems, however, it can be argued that the analysis is based principally on these existing systems, and that architectures can

be developed which provide opportunites for much greater degrees of parallelism. The unfortunate implication of this conclusion is that rather than gaining a speedup from a simple change in machines and implementation language; the programmer of the production system must explicitly think in terms of exploiting parallelism.

In his work, Gupta identifies a number of types of parallelism which can occur within the execution of a production system and analyzes their effect on the performance of the system. These types of parallelism are:

- Application parallelism
- Production parallelism
- Action parallelism
- Node parallelism
- Intra-node parallelism

The above levels of parallelism describe a hierarchy in which each level essentially implies all the levels above it. Each level of parallelism adds a certain degree of speedup to an application. Thus, the system with the highest performance would be one which employs parallelism at all levels. The following discussion briefly describes each type of parallelism and indicates the assumptions that Gupta makes when estimating the effect of that type of parallelism on the performance of an OPS5 system.

3.3 Production Parallelism

Gupta defines *production parallelism* as the assigning of processors to each production and the matching of each production affected by a working memory change at the same time. It does not employ parallelism at lower levels of the implementation. According to Gupta's analysis, the effect of production parallelism on performance is very small, in fact, approximately a factor of two. The reasons are as follows: first, only a small number of productions (on the average, 26, in Gupta's test set) are affected by any one working memory change. This would seem to indicate an average speedup of 26; however, there are further factors limiting parallelism in the canonical rule-based architecture. Once the productions are matched, conflict resolution must still be performed, therefore the matching process cannot terminate until the last production has been entered into the conflict set. Because the expressions which determine whether rules can fire can be arbitrarily complex, and because matching is performed by propagating tokens through a potentially unbalanced tree-like data structure, the time required for productions to match is typically unbalanced.[2] Some

[2]But see [Oflazer84].

productions enter the conflict set considerably later than other instantiations enabled by the same changes to working memory. This causes a loss of parallelism of a factor of five. Additional losses are incurred because of loss of sharing in the Rete net and the overhead due to parallelism.

Several of these assumptions can be questioned. In situations which possess *action parallelism* (see below), the number of working memory elements and thus the number of productions affected might be much larger. In an asynchronous system, it may not be necessary to examine the conflict set before executing productions, in fact, the conflict set may turn out to be unnecessary (or undesirable) in a parallel system. Without the conflict set bottleneck, an additional factor of 5.1 (according to Gupta) could be obtained.

3.4 Node and Intra-node Parallelism

In *node* and *intra-node* parallelism, the execution of each node in the Rete net is assigned to a separate processor. If a node in the network has multiple descendants, all of the subsequent nodes can be evaluated concurrently (see Figure 3.4). If a particular working memory change can affect many productions then the branching factor of the associated nodes will be high and so will the speedup provided by node parallelism.

Node parallelism assumes that no other process affects the data of a particular node while it is executing. Intra-node parallelism allows the same node to be executed by several processors simultaneously; this gain in parallelism avoids long delays during which access to a node must be restricted, but incurs a cost in terms of contention for the resources of the node and problems in keeping the memory of the node consistent during parallel memory accesses.

In his analysis of node and intra-node parallelism, Gupta assumes the fairly simple pattern matching operations available in OPS5. In OPS5, the number of operations computed at a given node is only an order of magnitude greater than the overhead required to schedule and execute parallel processors. Very little computation occurs within a particular node, perhaps fifty to one hundred instructions, on average. In order to be beneficial, parallelism must be achieved with very little overhead.

The speedup gained by node parallelism alone is therefore only marginal and acquired only by optimizing the scheduling mechanism both in hardware and software. If the complexity of computations performed at each node increases, the advantage to be gained from node parallelism also increases. It might be possible to increase this complexity either by enhancing our pattern matching language to allow more sophisticated expressions or by greatly increasing the size of the memories stored at each node, a natural consequence of applying production systems to applications requiring very large databases.

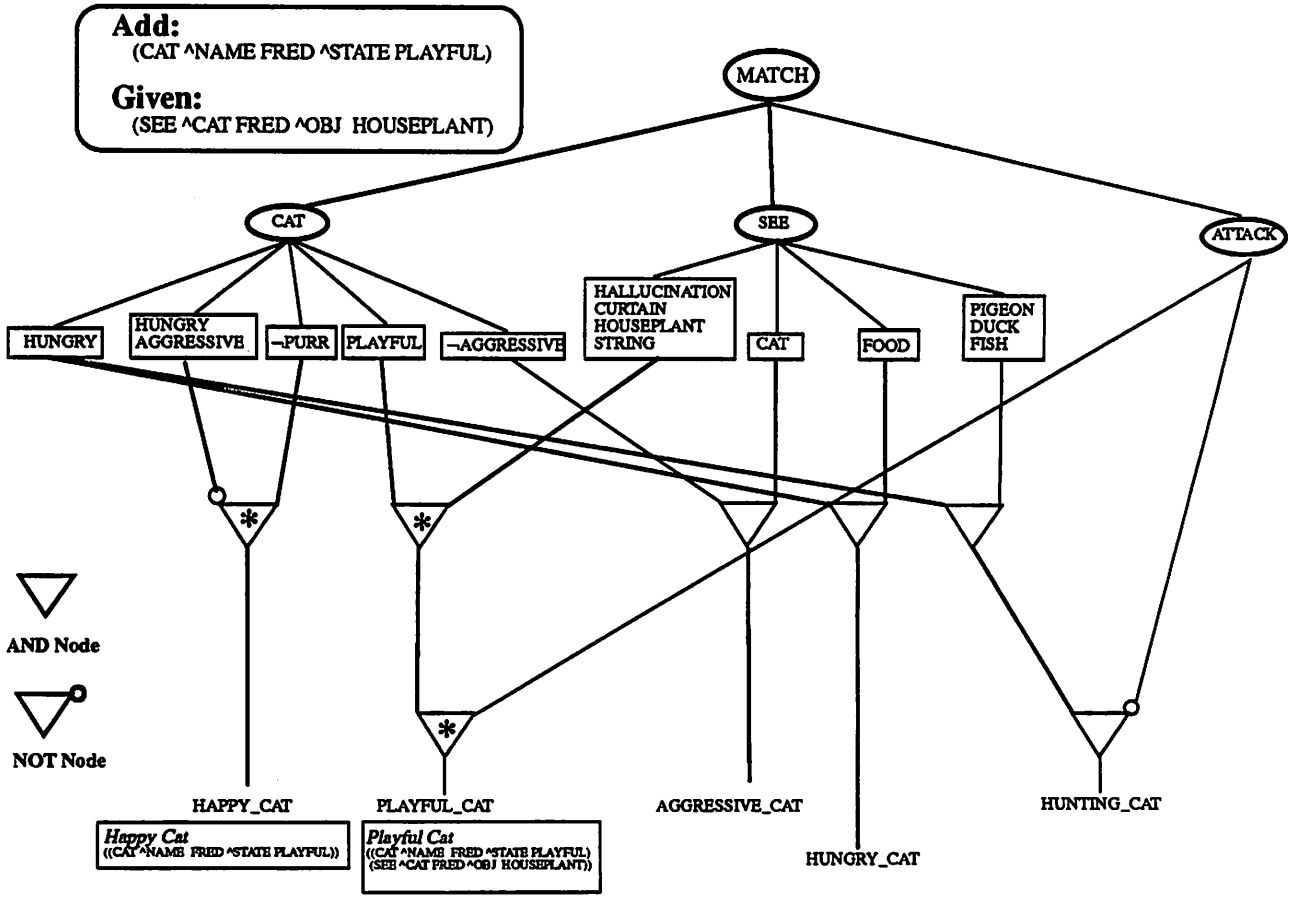


Figure 4: Node parallelism for a single working memory change in the cat example.

3.5 Extremely Fine Grained Parallelism within the Rete Net

When a token enters a two input AND node, it is compared against all elements in the opposite memory. This operation can certainly be executed in parallel, however the tests are so simple that parallelism can only be beneficial at the finest levels of parallelism. This approach is taken by Kelly and Seviora using DRete on the CUPID architecture[Kelly89]. While their experiments indicate a very high degree of speed up in the matching process, it's not clear that a corresponding increase in speed would be achieved in a full implementation.

3.6 Action Parallelism

Action parallelism is the changing of multiple working memory elements simultaneously. In OPS5, action parallelism is equivalent to executing all the elements of the righthand side of a production in parallel, or executing the righthand sides of multiple productions in parallel. By allowing action level parallelism, the number of productions affected per matching cycle increases, as well as the number of node activations (see Figure 3.6). With the use of action parallelism, the potential for increasing speedup within the match process is proportional to the number of rules executing concurrently and the average number of righthand actions in each rule. The level of programming complexity encountered by both the OPS5 implementor and the OPS5 programmer in systems which allow action parallelism is considerable, At the implementation level, there are all the problems of intra-node parallelism as well as possible consistency errors due to race conditions in the network and multiple simultaneous writes to memory nodes. At the programming level, there are the problems of non-serializable behavior, that is, behavior which could not have been achieved had all the working memory changes taken place in a serial order. To avoid implementation level errors, the system must provide mechanisms for locking memory nodes and synchronizing conflicting actions, as a result the contention for resources within the net can potentially degrade system performance drastically as the number of parallel actions increases.

3.7 Application Parallelism

Gupta notes that the SOAR architecture appears to be capable of a type of parallelism which he calls *application parallelism*[3] The SOAR system makes the assumption that all rules in the conflict set can be fired simultaneously, thus providing for potentially large degrees of production parallelism and high degrees of parallel node activation.

[3]Also called production parallelism or rule parallelism by various researchers.

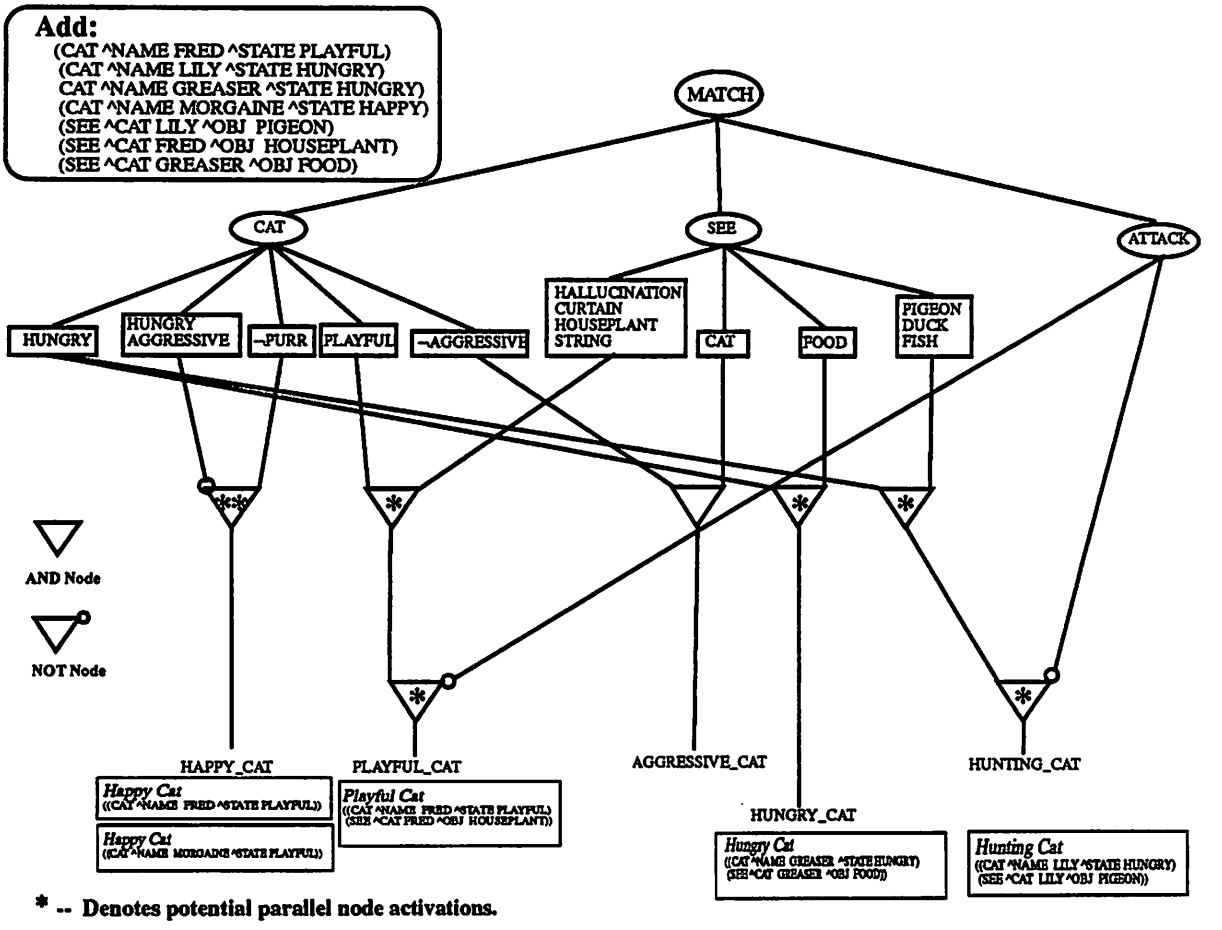


Figure 5: Action parallelism combined with node parallelism greatly increases the number of concurrent node activations.

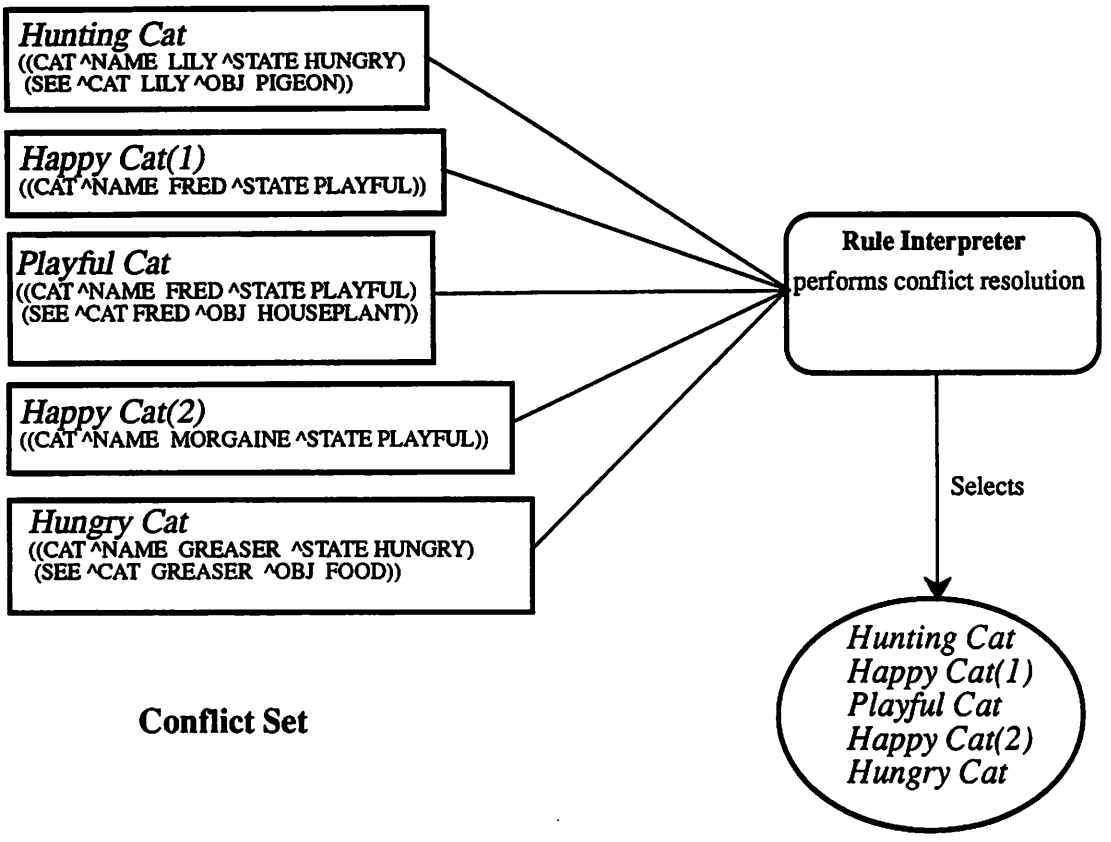


Figure 6: Application parallelism allows all the instantiations in the conflict set to be executed concurrently.

This property of SOAR has been extensively studied due to its promise of considerable parallelism [Tambe88, Nayak88].

Application parallelism allows individual rules which co-exist in the conflict set to be executed concurrently (see Figure 3.7).

In OPS5, the righthand side of productions contain mostly modifications to working memory, therefore application parallelism is similar in nature and effect to the previously described action parallelism except that the concurrently executing productions need not be referring to, or modifying the same working memory elements. Therefore, problems of contention for resources in the Rete net are reduced and the potential speedup in the matching process is significantly increased.

The speedup offered by application parallelism is directly proportional to the number of productions which can be usefully be concurrently executed. But this number is dependent only on the application; a system which is executing a large number of loosely coupled tasks may be able to maintain a very high level of rule activations. It is this aspect of parallelism in production systems which offers the

most hope for substantial speedup. But there are many questions which remain unanswered. For instance:

- What classes of problems require (or at least allow) application parallelism?
- How can you insure that parallel applications do not interfere with each other?
- Is the Rete net the appropriate data structure for problems which run essentially independent of each other?
- How do you synchronize and pass data between nearly independent applications?

3.8 Parallel Execution of Rules

Ishida and Stolfo have described an approach which proposes that productions which coexist in the conflict set be fired in parallel [Ishida85]. This approach is similar in concept to the application parallelism mentioned by Gupta. They describe two major problems in this approach, sychronizing concurrent production firings to avoid interference between productions, and decomposing problems to achieve maximum parallelism. They propose algorithms for detecting interference. Tellingly, these algorithms produced disappointing results on most benchmarks until the benchmarks were rewritten in a less serial form. On one (rewritten) problem, they report an expected speedup of 7.5 on a 32 processor system, not including possible speedups due to node and intra-node level parallelism. The algorithms derived by Ishida and Stolfo for detecting interactions between productions are static, and are performed on the rulebase before execution. In work which builds upon that of Ishida and Stolfo, Schmolze has developed three algorithms which contain both static and runtime components and more precisely determine when rules can co-execute without violating serialization constraints, thus producing larger subsets of productions which can be co-executed [Schmolze89].

3.8.1 Achieving Serializable Behavior in a Parallel Program

When productions run in parallel, the possibility exists that they will interfere with each other. Schmolze identifies two types of rule interactions, disabling and clashes. A production *disables* another production when it causes a change in working memory which removes the second production from the conflict set[4] (see Figure 7.).

Productions *clash* when they cause conflicting changes in working memory. For example, when production A adds a working memory element, and production B

[4]Strictly speaking, it is an *instantiation* of a production rather than the production itself which is placed in the conflict set, but for brevity, I'll use "production" to mean " instantiation of a production" when it won't cause confusion.

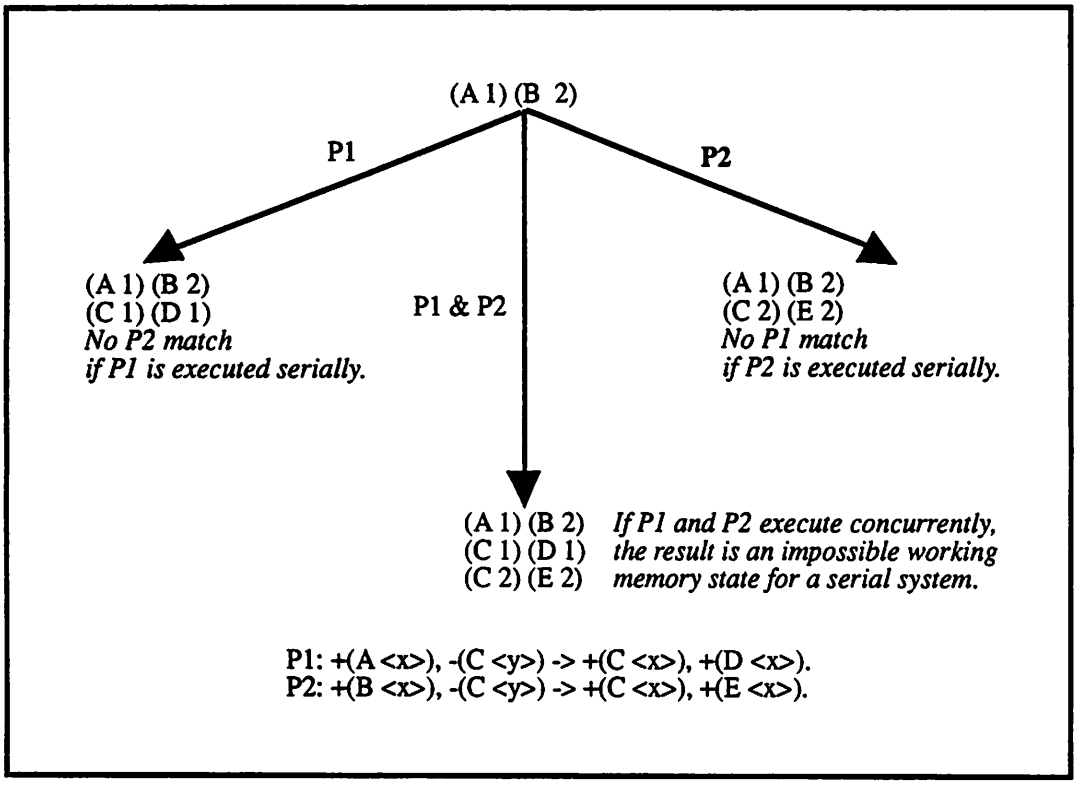


Figure 7: Mutually disabling productions cannot be executed concurrently.

deletes it, the final state of working memory depends on the order in which the productions fire (see Figure 8.).

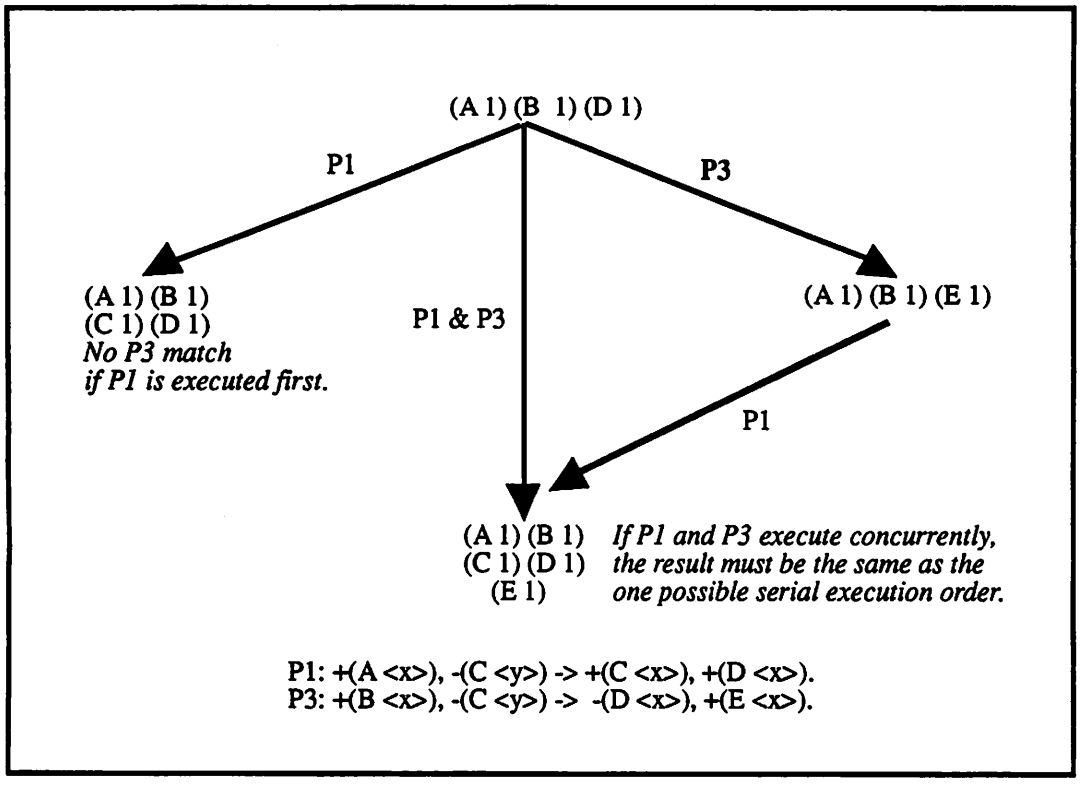
If productions can interfere with each other, then the results of running them in parallel will not necessarily be the same as running them serially, and the answers achieved by such a system may not be deterministic. In order to guarantee serializable results, Schmolze develops algorithms which analyze rules for potential disabling/clashing behavior and uses this information to *synchronize* the conflict sets. A conflict set is said to be synchronized if it possesses no instantiations which can either clash with or disable each other. Schmolze reports on three algorithms of varying precision for identifying rule sets which can be synchronized. Each algorithm has a static phase which examines the rule set, and a runtime *Select* phase which processes the conflict set and produces a subset of co-executable productions. The trade-off between the algorithms is between speed and precision. Static analysis is imprecise, because values have not yet been determined for many of the variables used in the matching process making it impossible to determine all the possible non-serializing relationships. A static analysis, therefore, must err on the side of safety and prevent rules from co-executing which only *potentially* interact. An algorithm which examines actual instantiations within the conflict set at runtime can be more precise, but dynamic detection of serialization violations increases the cost of conflict resolution and thus reduces the speedup obtained from parallel execution of the productions.

While the dynamic analysis of the conflict set does allow potentially co-executable productions to be precisely identified, it also limits potential parallelism in that it requires that an instantiation be compared with all other instantiations in the conflict set. This implies that the system must achieve quiescience, that is, that there be no matching taking place during the Select process. The quest for the maximum synchronization sets of executable productions thus effectively prohibits asynchronous production execution.

Miranker discusses the issue of asynchronous production execution and proposes an approach in which the production system is partitioned into sets of rules such that each rule can execute concurrently with each rule within its partion, and each separate partition can be executed asynchronously with respect to any other partition[Miranker89]. This approach still leaves the somewhat difficult task of *statically* assigning each rule to its correct partition.

3.8.2 Parallel Rule Firing with Fuzzy Logic

An alternative approach to resolving conflicts between executing productions has been taken by Siler, et. al. in the programming language FLOPS (Fuzzy Logic Production System)[Siler87]. In the FLOPS system, all eligible productions are executed concurrently. There is no conflict resolution and no backtracking. Instead, a *memory conflict* algorithm is employed which resolves contradictions in memory using weakly monotonic fuzzy logic. Each rule generates both values for attributes and confidence



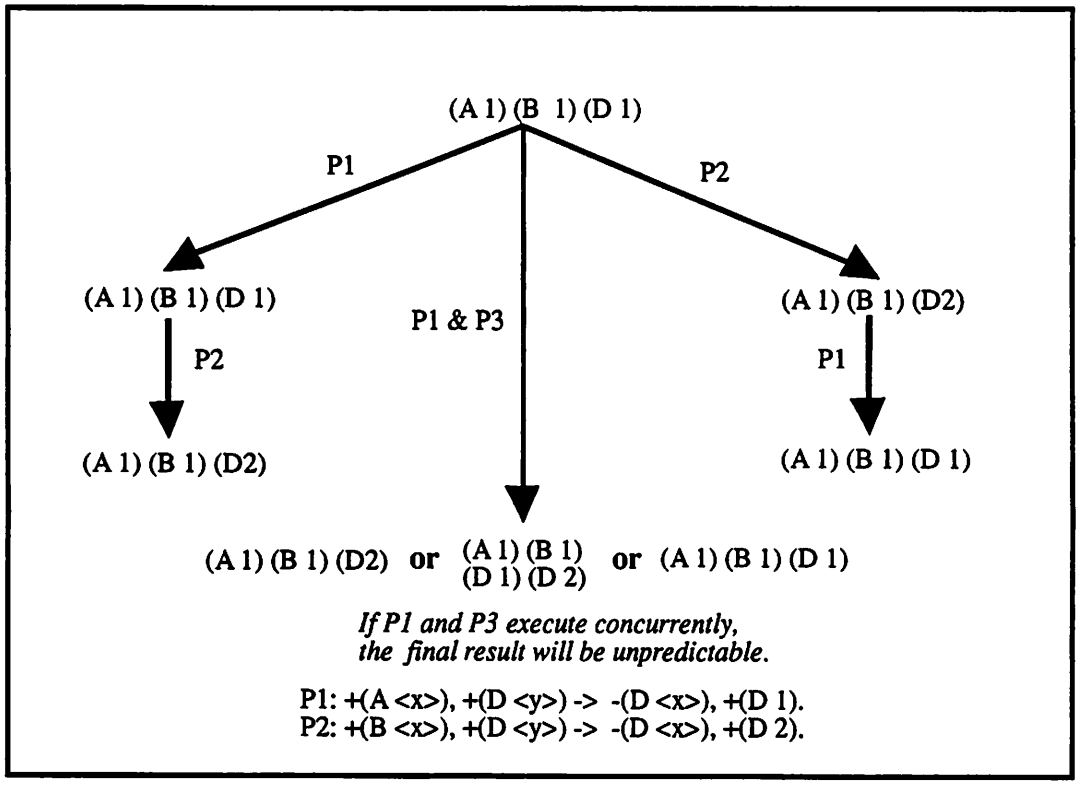


Figure 8: An example of clashing productions.

levels. If a rule produces an attribute value with a confidence level greater or equal to the existing value, then the previous attribute value is replaced with the most recent value. Naturally, this approach depends on the ability to generate meaningful and accurate confidence values. In order to ensure program correctness, parallel rule firing with fuzzy logic still requires that the state of working memory be independent of the order in which rules are executed.

3.9 Architectures for Production Systems

The previous section described a number of algorithms for incorporating parallel processing into production systems. Pragmatically, the algorithm chosen depends almost as much on the available hardware as it does on the inherent parallelism within the problem area. A number of machine architectures have been proposed for the rapid execution of production systems; they range from uniprocessors to machines with thousands of processors which support extremely fine-grained parallelism. There is by no means universal agreement on the correct degree of granularity for these architectures; the questions of how many processors, and how powerful, are closely tied to the degree and location of maximum potential parallelism within the production system. This section will discuss a number of proposed architectures for executing production systems.

3.9.1 DADO

The DADO machine [Stolfo84] is an attempt to develop a parallel tree-structured architecture which will efficiently execute expert system programs. The prototype DADO2 machine has 1023 8-bit processors *each* producing approximately 0.5 MIPS. The tree architecture minimizes communication costs; each node is responsible for transmitting to the nodes immediately below it, and propagating results from lower nodes upwards through the tree. Each node is implemented using a microprocessor with a small (16K) amount of memory. The DADO architecture can operate in either a semi-SIMD mode (in which the single instructions are function calls rather than machine language calls) or MIMD in which nodes execute autonomously. The DADO architecture can apparently support most levels of parallelism present in rule-based systems by assigning tasks to different levels of the hierarchy. To implement production parallelism, each production is assigned to a processing element (PE) at a fixed level of the tree. Processing Elements below the PE assigned to production matching are assigned to specific working memory elements. In the ideal case, the DADO architecture should produce matches independant of the number of productions or working memory elements. Because it is unlikely that there will be enough processors to map to each production and working memory element, multiple assignments can be made, causing some decrease in performance. The algorithms available for DADO can be tailored to the nature of the production system program being executed, for

example some programs may not have a significant amount of production parallelism but may have extremely large lefthand sides to productions; thus more processors might be allocated to matching working memory [Stolfo84].

There has been a certain amount of controversy regarding the DADO architecture, particularly whether the power of the large number of DADO processors could be utilized given the properties of OPS5 programs as analyzed by Gupta [Gupta84, Stolfo84a]. The results of the DADO2 project have been reported in [Stolfo87]. Work is now proceeding on DADO4, an architecture which 15 high-speed 16-bit RISC processors each running at approximately 12.5 MIPS; an OPS5 implementation should be running on DADO4 by summer 1990[5].

3.9.2 Implementation of OPS5 on Non-Von

The Non-Von architecture, a massively parallel multiple-SIMD machine developed at Columbia University, has also been considered as a vehicle for executing OPS5 [Hillyer86]. The key to Non-Von's performance is the heterogenous nature of its architecture. Working memory elements are assigned to small processing elements (SPEs) and operations which refer to attributes of the working memory elements are performed associatively. Operations at a greater level of granualarity are carried out in the large processing elements (LPEs). The architecture contains a large number of SPEs (on the same order as the average number of working memory elements in the standard production system), and a much smaller number of LPEs (approximately 32). Benchmarks based on simulations of the Rete algorithm using data gathered from existing expert systems promise upwards of 850 production executions per second as compared to 1-5 on a Lisp-based interpreter running on hardware of equivalent cost (a VAX 11/780). Whether this performance would actually be achieved by a working prototype is not known, as the project has been discontinued.

3.9.3 CUPID and DRete

Another approach to fine-grained parallelism has been taken by Kelly and Seviora with the distributed Rete (DRete) algorithm designed for the Cupid architecture [Kelly89]. This architecture consists of a matching processor networked to a host. The host performs conflict resolution; the matching processor performs the matching actions. The CUPID architecture consists of a large number of small processors. The underlying approach is that of very fine granularity. Each beta node in the Rete net has to perform a number of comparisons proportional to the number of tokens in that node. The DRete splits each node so that a copy exists for each token stored in that node's memory. This allows each comparison to be performed on each node in parallel, thus allowing each beta node to proceed in essentially unit time. There is, however, an overhead associated with generating new copies of nodes for new tokens

[5]Stolfo – Personal communication.

as they are propagated through the net. The effectiveness of the DRete algorithm increases as the number of tokens stored in each node increases.

3.9.4 Message Passing Architectures

Multi-processors with distributed memory are not ideally suited for executing production systems because the communication costs required to transmit updates to working memory largely eclipse the advantages gained by parallel processing at the node levels. These architectures are most suited for large grained parallelism in which each processor contains its own working memory and productions and works on separate tasks. Communication costs are decreasing, however, and distributed memory architectures are becoming more effective. They are particularly attractive because they provide more processors at less cost than the more expensive shared memory machines. Research on executing production systems on message passing computers is described in [Tambe89, Acharya, Schmolze90].

3.9.5 Shared Memory Architectures

The work by Gupta predicted fairly low levels of concurrent node activation and a relatively high overhead associated with scheduling fine-grained parallelism. The conclusion reached was that the preferred architecture for executing a parallel production system (based on studies of existing systems) is a shared memory system containing no more than 64 high-speed processors augmented with a hardware scheduler for allocating processors to node activations.

An advantage of the shared memory architecture is that it is currently available state-of-the art technology for which production systems can be written without being preceeded by massive hardware development projects. Architectures such as the Connection Machine which employ very large numbers of processors are available, but typically provide weak processing elements and an SIMD control flow; while some research has been done concerning the implementation of a production system on such an architecture[Brooks85, Morgan88], the mapping is not straightforward, particularly if the lefthand side does variable binding and unification.

The choice of computer architectures is strongly influenced by the estimated degree of parallelism within the matching process. As algorithms and applications are developed for production systems which display increased levels of parallelism, the demand for processors and processing capacity should increase.

4 Parallelism in OPS5 – Research to Date

The preceding section has described a number of algorithms and architectures for implementing parallel production systems. With few exceptions (e.g. [Gupta88]),

these experiments were carried out using simulated parallelism on existing production systems designed for parallel execution. In order to experiment with an actual parallel production system, I have added mechanisms to OPS5 to support rule and node level parallelism[Neiman90a].

The implementation of parallel OPS5 is intended to serve several purposes:

- To provide a working implementation of a parallel production system which could provide significant speedups in appropriately constructed programs.
- To provide a series of benchmarks for parallelism at various degrees of granularity within a production system.
- To provide an experimental vehicle for testing different program structures and control strategies for parallel production systems.

The results of the experiments with the parallel OPS5 and future directions are presented in the following section.

4.1 Implementation of a Parallel OPS5

The availability of a shared memory multiprocessor and a Lisp which supports parallel programming has made possible the construction of a version of OPS5 which can provide parallelism at the action, node, intra-node, and application levels. Only extremely fine-grained levels of parallelism cannot be implemented. The system has been tested on a number of small benchmark programs including a simple circuit simulator, an OPS5 program for solving Rubik's cube, and a couple of implementations of Waltz's algorithm for line labelling.

The implementation of the parallel OPS5 is based on a public domain OPS5 written by C. Forgy at Carnegie-Mellon University. There are several advantages and disadvantages to using an existing implementation of OPS5. The most obvious advantage is the savings in implementation time. The Common Lisp OPS5 already contains the necessary parser, compiler, and matching algorithms. In addition, a large part of the production system-using community is familiar with or has access to this implementation; this will make the description of the parallel mechanisms easier to follow. The disadvantage of using the public domain OPS5 is that it is largely undocumented and was written in a highly optimized form for some previous version of Lisp. The result is a body of Lisp code which is not up to currently accepted Lisp programming standards. The code is, at times, very difficult to follow and the exact algorithms used to implement the nodes of the Rete net must be determined by very careful reading of the code, not to mention considerable experimentation. One of the principle disadvantages of using the existing OPS5 code is the unfortunate use of globals (specials) which are used to pass information between functions instead of parameters, undoubtedly for efficiency purposes. These globals must be eliminated

from all code which is expected to run concurrently. Another disadvantage of the OPS5 code is the implementation method used for node memories. At the time of the OPS5 implementation, most Lisps did not support structures or arrays. Thus, the memory nodes are represented as unstructured lists. The use of structures would allow faster and more sophisticated access methods to be used, as well as allowing changes to the node structures to be made more easily.

The OPS5 language itself is not without shortcomings; it possesses a somewhat awkward syntax, cannot easily express disjunctions, and does not allow the expression of meta-level rules. The final disadvantage of using an existing language is that one inherits a programming methodology which is extremely well-established; this discourages innovation and causes features to be retained which might not be optimal for parallel execution.

The research program described in the following section is not compromised by these features of OPS5: the goal of the research is to study properties of production systems and languages which require the support of parallelism, rather than the implementation of a commercial quality programming system for rule-based languages. While the current version of OPS5 has proven suitable for adapting to parallelism, the disadvantages noted above may make it advisable to substantially rewrite and modify the implementation during the course of the research.

4.2 Implementation Environment

The parallel OPS5 was programmed on a Sequent shared-memory multiprocessor using Top Level Common Lisp[6], an implementation of concurrent Common Lisp. The use of a shared memory multiprocessor allows a reasonably fine-grained approach to parallelism with low communication costs. The Sequent only supports a limited number of high performance processors (in this case, 16), so considerations of processor utilization are fairly important.

The programming language TopCL is a standard implementation of Common Lisp with mechanisms for invoking different levels of parallelism using lightweight processes (threads) and futures.

4.3 Benchmarks

The principle benchmark program for the parallel OPS5 is a very simple circuit simulator. While unimpressive in terms of circuit simulation technology, this application displays the desirable property of task independence, and does not require an extensive knowledge-engineering effort. Conceptually, a circuit simulator is event-driven, with each device capable of being simulated in parallel without reference to other entities in the system. For each type of device in the system, the simulator contains

[6]TopCL is a trademark of Top Level, Inc.

a production which "knows" how to simulate the behavior of that device. The circuit is represented by working memory elements describing the devices, the connections, and the signal values seen on inputs and outputs. At each time quantum (which is equivalent to a production execution cycle), each device is simulated by the execution of one production. Each simulation is followed by a propagation phase in which the outputs of each device are propagated to the appropriate device inputs. The problem displays parallelism in that each device can be simulated independently, but has a sequential component in that devices must be simulated at time *t* before their value at time *t+1* can be computed.

The initial test set consisted of fifteen devices containing a total of twenty seven inputs and thus required fifteen productions to execute in the simulation phase and twenty seven in the propagation phase. The system was configured to run using only production-level parallelism. Due to the relative independence of the productions in this system and their ability to execute concurrently, my initial predictions were that the performance of the system would be very nearly linear, with the speed of execution being roughly proportional to the number of processors and some penalty due to contention for shared resources within the Rete net.

4.3.1 Experiment 1: Explicit Synchronization

In the first experiment (see Figure 9A), the two phases of the program, simulation and propagation, were synchronized using a working memory element of type *mode* – a conventional OPS5 programming technique. Two 'demon' productions detected when it was time to change the mode of the system from simulate to propagate and vice-versa. The assumption was made that all productions in the conflict set were capable of being executed concurrently and therefore no runtime checking for potential conflicts was performed [Schmolze89]. If the conflict set contained more instantiations than there were processors in the system, the surplus rules were placed on a process queue and executed as processors became available. The rule execution mechanism achieved synchronization by explicitly waiting until all productions had completed execution and the system had achieved quiescence before re-examining the conflict set.

The results of the first experiment were disappointing, the speedup due to parallelism was only a factor of three and the utilization of processors was poor. My analysis of the system indicated that the fault lay with the mode-changing productions. These productions, by necessity, did not share the conflict set with any other productions and could only be executed serially. These mode-switching productions are responsible for deleting and adding the mode working memory elements. Inside the Rete net, these elements act as gates which prevent the matching process from proceeding past a given point and adding instantiations into the conflict set. Thus, execution of the mode-switching productions initiate considerable matching activity, cause many productions to be instantiated, and consume a disproportionate amount

of processing resources. While node parallelism reduced the length of the serial bottleneck, it could not eliminate it entirely. Therefore, a second experiment was devised to increase the level of asynchronous behavior in the program by removing the explicit working memory-based control.

4.3.2 Experiment 2: Synchronization via Conflict Set

In the second experiment (Figure 9B), the conflict set was used as an explicit synchronization mechanism. The observation was made that the computation was logically divided into phases, with all the rules composing one phase capable of executing in parallel. The purpose of the mode-changing technique is to prevent instantiations from one phase from being prematurely inserted into the conflict set and being executed out of order. However, because production parallelism allows all instantiations in the conflict set to execute simultaneously, the instantiations belonging to the next phase can be safely added to the conflict set, avoiding the necessity for explicit mode-changing. Effectively, this approach to synchronization allows a greater proportion of the matching process for the next phase of the computation to take place during the current phase. Note that because each instantiation in the conflict set contains a separate copy of the working memory elements which caused it to be instantiated, once a production begins execution, it cannot be disabled by succeeding changes to working memory.

In this second experiment, the speed of processing increased by an additional factor of two, however, processor utilization remained low. Analysis of the results of this experiment revealed that the bottleneck was the delay imposed by the necessity for achieving quiescence in the system before the next round of productions can be executed. That is, an instantiation of a production may be eligible to fire, not conflict with any other existing instantiation or executing rule and yet remain in the conflict set for a considerable length of time until the entire previous round of production firings are completed.

Assuming that all productions in the conflict set can be executed simultaneously and that no conflict resolution need be performed, the time that a production A remains in the conflict set is equal to the amount of time between the insertion of A and the time that the system reaches quiescence and all actions affecting working memory are completed. This time is dependent on a number of factors: the number of productions being concurrently executed, the number and type of righthand side actions to be performed, and the number of processors available.

For example, consider the case of a 16 processor machine, attempting to execute 17 productions concurrently. Assume each production contains roughly the same number of righthand side actions, and therefore takes roughly the same amount of time, t, to execute. Assume also that each production causes one instantiation to be placed within the conflict set, each of which could be executed without conflicting with any other. After time t, 16 instantiations are present in the conflict set, the 17th

production is being executed, and 15 processors are idle. If the righthand side (RHS) of the productions contain a significant number of actions, produce output, or perform calls to the operating system, the time t could become quite large. If the RHS actions consist solely of working memory changes, applying the surplus processors towards low-level node parallelism can reduce t, however, my experience has been that the degree of effective node parallelism in problems which support multiple production firings is fairly low.

4.3.3 Experiment 3: Asynchronous Production Execution

The final version of the experiment (Figure 9C.) was optimized for maximum asynchronous behavior. Given the realization that any time that an eligible production spent in the conflict set was time wasted, a new scheduling policy called "fire when ready" was devised. In this scheme, the conflict set was continually monitored; whenever a new production entered, it was immediately fired. The OPS5 code implementing the simulator had to be substantially re-written in order to support this level of parallelism. This approach to scheduling did not employ conflict resolution, so it was no longer possible to guarantee that rules would be executed in any given order. Therefore, the rules had to be rewritten to be self-synchronizing, that is, to examine the appropriate working memory elements for ordering information (explicit timetags) to ensure that the simulation of devices proceeded in the correct temporal order.

Because the computation was asynchronous, it was possible that some working memory elements (representing inputs) could be modified before all the devices connected to those inputs had been simulated. To avoid this problem, separate working memory elements representing successive inputs to a device were created, rather than modifying the existing elements. This approached created a potential for explosive memory growth and required an architecture in which working memory management was knowledge based, and elements were only deleted when it could be guaranteed that they would no longer be needed.

This asynchronous approach to rule firing produced very high levels of processor utilization (nearly 100%), and a speedup of 8-10 over the strictly serial case. Considerable extra matching and production execution took place to synchronize the computation and to garbage collect unneeded working memory elements; unlike the previous experiments, additional processors would have resulted in increased performance.

4.4 Summary of Experiments

The experiments described above demonstrated that performance can be greatly increased in a production system by eliminating the conflict resolution stage and executing productions asynchronously in parallel. This improvement was gained at

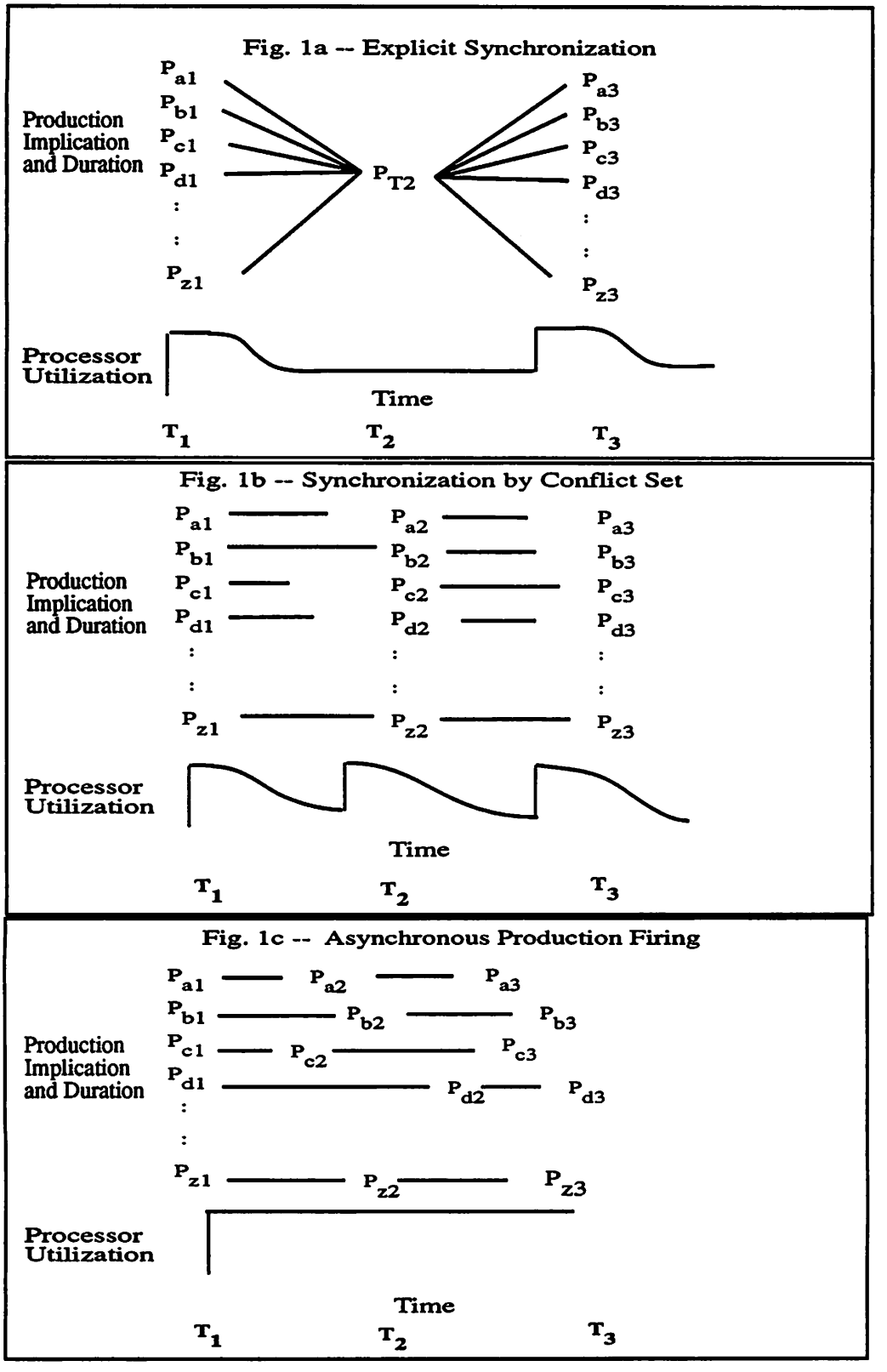


Figure 9: Controlling Production-Level Parallelism

the cost of considerable extra programming effort. The asynchronous version of the benchmark program was significantly more difficult to write and debug than the conventional "lockstep" versions. Furthermore, the circuit simulator did not perform any meaningful kind of search activity and research has to be done to determine if these results can be applied to "real" AI applications. In cases in which a fully asynchronous model is not possible, we would like to develop methods for reducing the overhead inherent in control. This research is the subject of the following section.

5 Proposed Research

Summary of Research Goals: In the course of my research, I intend to perform a study of the nature of parallel production systems, concentrating in particular on the semantics of rule interactions, and the control, language, and design issues raised by parallel and parallel-asynchronous execution of rules. The goals of this research are briefly summarized below:

- Identify the limitations of current methods for extracting production level parallelism using syntactic analysis of rules.
- Create a taxonomy of potential rule interactions, the implications of the rule interactions, and the action to be taken by the scheduler when the interactions are detected.
- Demonstrate rule-based language idioms for supporting well-known AI paradigms in parallel systems without excessive serialization.
- Analyze the performance of asynchronous execution for rule-based implementation of various AI algorithms such as search and task-based programming.
- Analyze the validity of existing working memory representations and matching algorithms for parallel production execution.
- Generate a meta-rule syntax for specifying rule and data interactions necessary for resolving conflicts and scheduling parallel rule execution.
- Generate an algorithm for a controller which will monitor instantiations as they arrive in the conflict set and execute productions according to the constraints imposed by the meta-rules, while sychronizing conflicting rules in a knowledge-based manner.
- Construct a production system language which provides robust support for parallel data manipulation and incorporates the previously mentioned meta-rule syntax and parallel controller.

5.1 Control Issues

Much of the proposed research centers around the problem of *control*; that is, the problem of imposing a structure to a parallel rule-based computation without incurring prohibitive levels of overhead. These control issues are discussed in the following sections:

5.1.1 Definitions of Control

Control exists at a number of different levels in a production system. Low-level, or imperative control, consists of sequencing from episode to epsisode in a deterministic fashion, in the way that control is transferred from statement to statement in a conventional language such as C or Pascal. At this level of control we have loops, conditionals, calls to subroutines, and the straightforward execution of sequential program statements.

In a serial production system, control usually consists of the process of looking at all the computations which can potentially be performed (that is, all the items in the conflict set) and deciding which one is most appropriate under the current circumstances. A common programming trick is to manipulate working memory to force the conflict resolution routine to select a particular sequence of productions in order to emulate a particular low-level control idiom. Emulation of these constructs in a production system tends to lead to a serial program structure.

At a more interesting level of sophistication, we can think of control as the process of deciding "what to do next" based on a consideration of the goals of a system, its current world state, and the possible actions available to it. Control can be arbitrarily complex; decisions can be made using a fixed control policy which examines only local state information, or a system can use multiple control policies and select its next actions according to longterm global strategies. Naturally, the more complex the control strategy, the longer the decision process takes.

Part of the need for control comes because serial systems can only perform one action at a time; therefore, in a resource-limited situation, the best alternative should be tried first. In a parallel system, it is possible to investigate multiple solution paths simultaneously which may reduce the need for sophisticated control strategies. However, given a realistic number of processors and the combinatorial nature of search spaces, the number of solutions which can be tried is likely to be less than the number of alternatives.

Control, as it is conventionally implemented, is a serializing process. If one has to examine all the actions available, then there must be a synchronization step during which all the alternatives can be examined. The problem of reducing the inherent overhead of control is addressed in the next section.

5.1.2 Removing the Conflict Set Bottleneck

There is near-universal agreement in the literature that the bottleneck in OPS5 processing is the match stage, based on Forgy's contention that matching can consume as much as 90% of the total computation time [Forgy79]. However, as the experiments described above indicate, this is not the whole story. The necessity for performing some kind of conflict resolution imposes a bottleneck on the matching process which may make it irrelevant how much parallelism takes place in the match. Because the conflict resolution cannot take place until all productions have matched, the minimum cycle time is the time of the longest match.

In general, a production system is most efficient when it can execute as many rules in parallel as possible, thus maintaining the number of node activations and thus the maximum degree of parallelism within the matcher. Any time that a useful production spends in the conflict resolution set is time that that same production could be executing, thus, performing conflict resolution as a separate serial step between episodes of rule firing and matching is detrimental to parallelism [Miranker90]. The experiments in parallel rule execution indicate that the highest performance will be achieved by an asynchronous control strategy in which productions are executed as soon as they are enabled [Neiman90]. Because this 'fire-when-ready' policy executes productions before sufficient information is available to generate completely accurate control decisions, the computation must either generate only the correct productions to execute or be able to recover from the occasional erroneous production execution.

5.2 Controlling Parallel Production Systems

The most satisfactory way of eliminating the sequential overhead of conflict resolution is to eliminate the *need* for conflict resolution; that is, to design the system so that it proceeds in a focussed manner with only relevant productions being enabled and no productions being mutually interfering or disabling. While it was possible to achieve this kind of completely asynchronous performance in the circuit simulator benchmark, it was only done at the cost of much additional complexity and loss of comprehensibility in the program. More compellingly, the circuit simulator does not perform search, therefore, any rule which become eligible to fire can be fired without redundancy.

The conflict set plays a critical role in controlling most existing rule-based programs; for example, low level control of OPS5 programs depends on firing productions in an order based on explicit properties of the the conflict resolution algorithm. Removing this control property of OPS5 would not be a bad thing – as discussed previously, lack of a reasonable control mechanisms is an archaic feature of OPS5 not necessarily present in later systems such as OPS83. Without some notion of a conflict set, however, developing rule-based systems would be much more complex. The ideal approach of executing productions as soon as they are enabled is only appropriate

if productions never conflict or perform the same task. Otherwise, the system must contend with problems of redundant rules consuming processing resources, cluttering working memory, and disabling more useful rules. In any application which performs any kind of search (and virtually all AI applications can be viewed as performing search) it is necessary to maintain some degree of control over the computation. For example:

- Assume that more than one production is eligible in a given situation, but only one action should be executed. After executing the first eligible production, the remainder should be suppressed as they enter the conflict set.
- If a production is eligible, but is not guaranteed to produce the desired results, it is necessary to monitor the results of the production execution and execute other productions as necessary.
- Executing multiple productions in parallel could result in inconsistent states – we want to recognize these states and eliminate them.
- If executing a parallel search, possibly composed of multiple pipelined chains of inference, once a satisfactory answer is achieved, we want to terminate all extant processes to free up computing resources.
- If the number of eligible rules becomes greater than the number of available processors, then the most useful or highly rated rules should be run first.

A number of methods of maintaining some level of control over a computation while eliminating the strict need for global control and conflict resolution are presented below. The following definitions will be useful in the ensuing discussion.

Goals: In systems in which control is necessary, we use the convention of creating *goals* to direct the computation. The creation of a goal is a signal that any processing which potentially contributes to that goal should be given higher priority. The process of satisfying a goal can lead to the creation of subgoals, that is, goals which have to be satisfied before the original goal can be satisfied. In systems which display uncertainty, a computation may be selected which could *potentially* lead to the satisfaction of the goal, and satisfaction of sub-goals are likely, but not certain to lead to the achievement of the super-goal. There may be multiple ways of achieving (or possibly achieving) any given goal.

Tasks: A task can be considered as a series of one or more sub-computations, each of which is represented by one or more goals or sub-goals. These subcomputations may have to be performed serially, or they may allow for partial or complete parallelism. A production can be associated with a particular task according to the working memory

elements which enabled it. For example, in a vehicle monitoring system, a task might be to track a single vehicle. All the productions which execute in the process of tracking this vehicle would be identified with this task.

Quiescence A production system can be said to be *quiescient* when all working memory changes have been processed and all possible instantiations have been entered into the conflict set. Determining when a system has become quiescient is simple in a serial system (one just waits for a function call to complete), however, in a system which employs any degree of parallelism it is necessary to develop synchronization mechanisms which monitor active processes and determine when they complete. Synchronization can be expensive; one or more processors may have to be assigned just to identify quiescence in a timely fashion.

Match Episodes A *match episode* is the process of identifying all productions affected by a single working memory change. If each working memory change is associated with a single task or computation, then any productions stimulated during a match episode will be relevant to the same problem and can be considered by the same conflict resolution routine or process (See Figure 10). This is a simplifying assumption which reduces to a large extent the number of interactions between tasks, and eliminates many potential clashing and disabling behaviors.

Note that a single match episode can enable multiple productions. Each production instantiation enters into the conflict set when the working memory change propagates through the matcher to the appropriate bottom node; the time differential between the instantiations depends on the complexity of the matches, the total number of matching productions, and whether the matching process makes use of internal parallelism or proceeds serially breadth first or depth first.

A single production execution can execute many working memory changes, each one representing a separate match episode. The easiest way to define quiescence in the system is to wait until a production has completely executed, however, it can be seen by examining Figure 10, part B., that this can result in significant delays in evaluating and executing instantiations. By executing all the righthand side actions in parallel, this delay can largely be eliminated, but this is only an option if there are no interactions between the righthand side actions (see Figure 10, part C). Finally, multiple productions can execute concurrently, creating multiple simultaneous match episodes; determining T in this case is difficult as it must be determined when all executing productions relating to a particular computation have terminated.

5.2.1 Concurrent Control

Given that the canonical conflict resolution schemes tend to serialize a computation, it is worth asking whether control can take place in parallel with the computation.

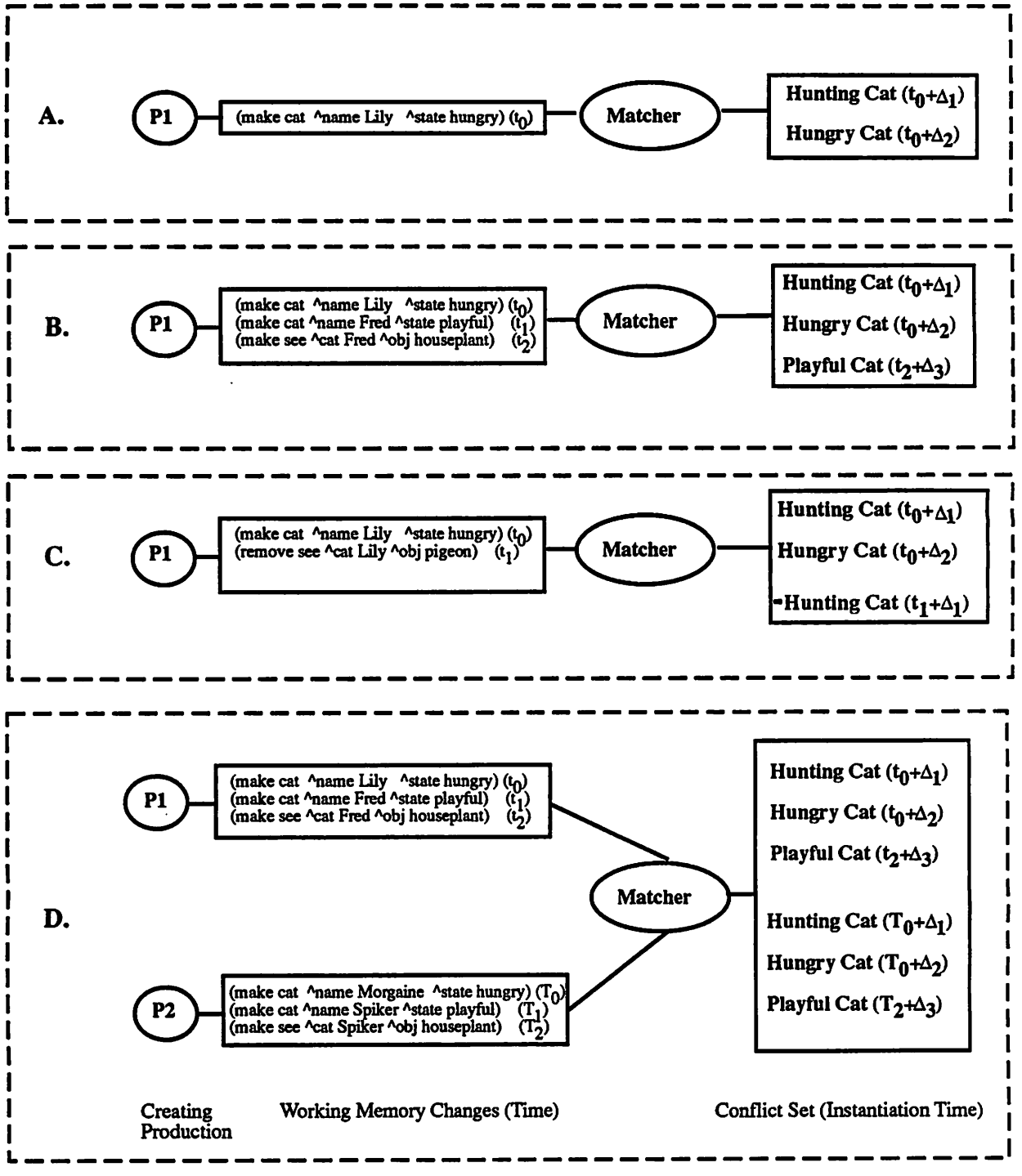


Figure 10: Examples of match episodes.

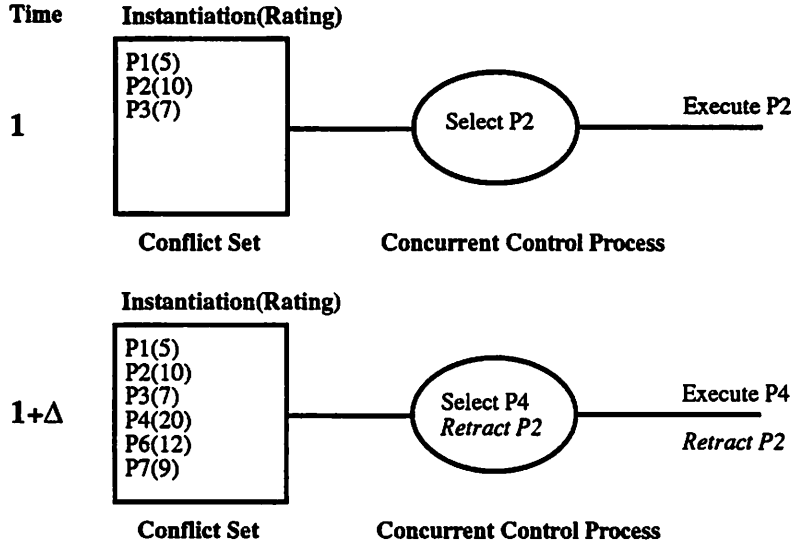


Figure 11: Concurrent Conflict Resolution

A few researchers have proposed pipelining the conflict resolution phase with the production matching phase, but given the superficial nature of most conflict resolution algorithms, this would not result in much of a speedup. A more effective approach is to perform asynchronous process monitoring. In this paradigm, a conflict resolution process is assigned to a particular match episode. As instantiations are matched, the process incrementally selects those which are to run. If a rule is fired, and a better choice later enters into the conflict set, then the earlier rule execution could be halted, or, with the appropriate bookkeeping, 'unwound' in a manner not unlike retracting an incorrect assumption in a truth-maintenance system (see Figure 11). Note that with a functionally accurate approach (discussed below), the only reason for retracting production firings may to remove unneeded working memory elements.

An advantage of this approach is that it allows the "fire-when-ready" approach to rule execution while avoiding complete saturation of processing capacity with useless or redundant instantiations. Because a separate process monitors each production cycle, the conflict resolution scheme used at each stage of the computation can be situation specific. The conflict resolution process can be made sensitive to the current state of working memory and, in one extreme architecture, can be implemented as a separate concurrent rule set (see Figure 12). The disadvantages are the complexity of programming the parallel conflict resolving mechanism, the necessity for extra bookkeeping to monitor all executing instantiations, and the potentially large overhead

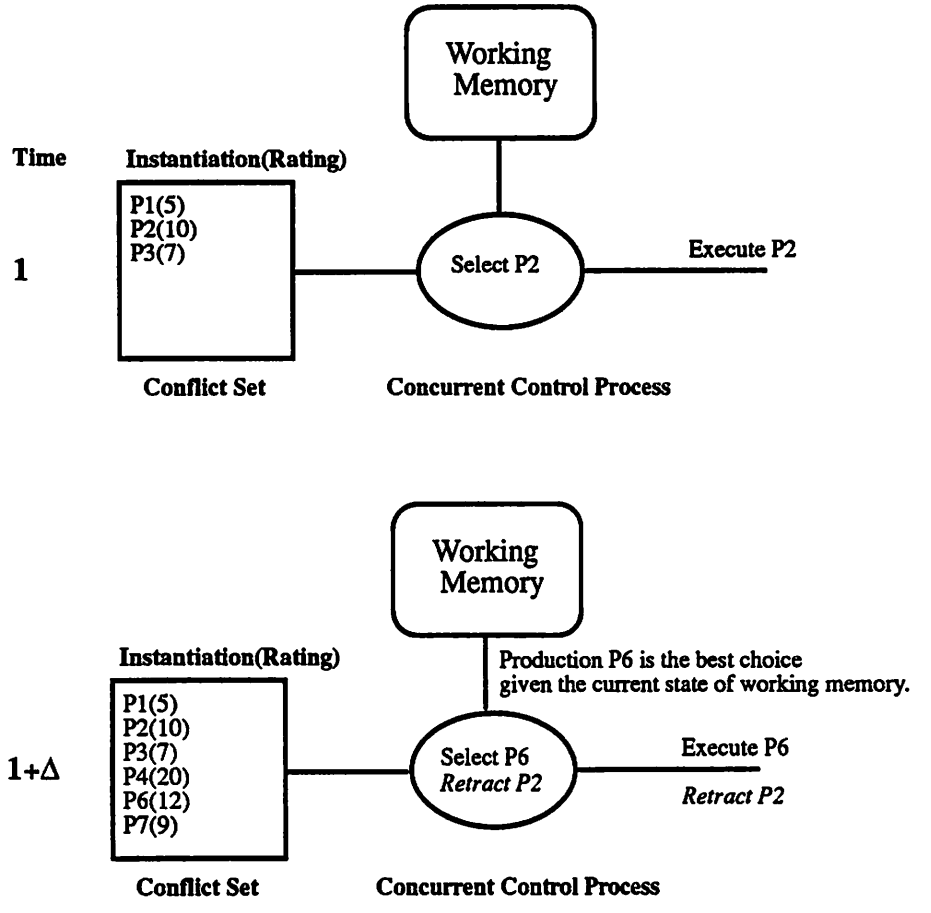


Figure 12: Concurrent Conflict Resolution with Meta-rules

required to retract incorrect working memory changes. Because the assumption is made that all the instantiations in the conflict set being monitored are relevant to the same problem, this approach implies an architecture which supports multiple conflict sets and task-oriented productions.

5.2.2 Algorithms for Functionally Accurate Computations

To say that a computation is *functionally accurate* is to be able to guarantee that it converges on a correct solution despite transient inconsistencies in the knowledge base. A computation which can guarantee this property despite inconsistencies due to lack of coordination between concurrent processes is said to be *Functionally Accurate/Cooperative* (FA/C) [Lesser81].

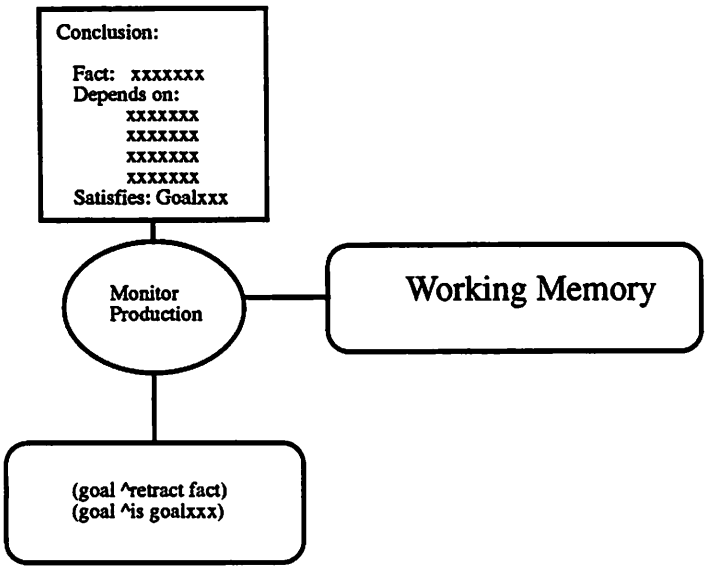


Figure 13: An active approach to FA/C computation.

To be able to demonstrate that a computation is functionally accurate, it must be possible to *enumerate* the expected inconsistencies and errors resulting from a lack of control or coordination and to *prove* that the computation successfully reaches a correct solution despite them. Many of the standard failure modes of parallel production systems (disabling, clashing, etc..) have been enumerated by Stolfo, Schmolze and others.

A functionally accurate computation must either tolerate errors or repair them. To reduce the burden on the creator of the system, this process must be made as automatic as possible. The production system architecture is remarkably well-suited to the FA/C approach in that it is, by nature, data-driven. Therefore, a correctly structured computation can recognize when a goal has failed to be achieved and retry. Two approaches to developing an FA/C computation are presented in figures 13 and 14.

In the first approach, the system takes an active role in ensuring the consistent nature of the computation by explicitly establishing conditions which validate a conclusion and monitoring the database to ensure that these conditions do not change. This is similar in principle to establishing protection axioms in a conventional planning system. If suitable representations can be established in working memory, then a single demon production (or class of productions) could be used to perform the monitoring task and retrigger productions if necessary. This active approach to FA/C requires that the programmer be able to determine the critical data in the computation and specify what changes in working memory would invalidate the results.

The second approach to FA/C relies on robust design to ensure that the correct

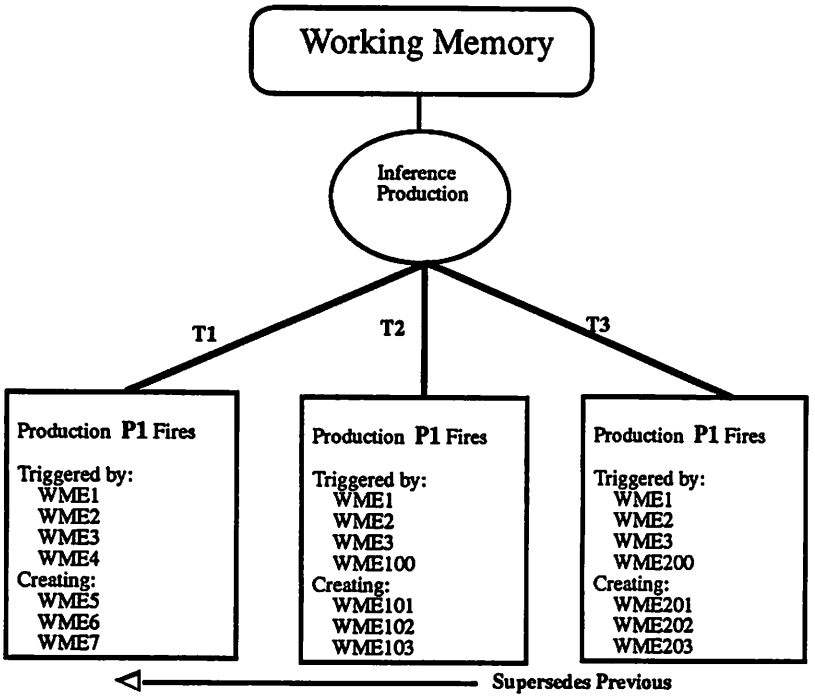


Figure 14: A passive approach to FA/C computation.

solution is reached. If working memory changes, then any productions which are responsible for drawing inferences from the changed data are re-executed. If the system performs "pure" forward chaining inference, than the conflict set could actually be used as a truth maintenance mechanism. When a production is withdrawn from the conflict set, the working memory elements that it has added can be withdrawn in turn.

Some mechanism must be employed to make sure that looping behavior does not occur in which the same rule is fired over and over without results. In-depth checks should be reserved for the point at which the results of the algorithm consume scarce resources or perform some non-reversible "real-world" action (e.g. before you actually spend money, fire a missile, prescribe a drug...). In order to automate the FA/C checking, representations will have to be devised to allow the controller to recognize states which are invalid or undesirable.

5.2.3 Meta-Rules

The conflict resolution scheme used in OPS5 is essentially an institutionalized kludge. Control is based solely upon syntactic characteristics of rules, and it falls upon the programmer to insure that the rules are written in such a way that they execute in the correct order. The use of meta-rules [Davis80] allows a more knowledge intensive

approach to control. Although the use of meta-rules does not necessarily eliminate the conflict set, meta-rules can be developed which contain knowledge about parallel control [Decker90]. The idea of using the same representation for control as for domain directed knowledge is not new [Hayes-Roth85], however care would have to be taken to ensure that the control rules were suitable to parallelism; if not, then the serial bottlenecks implicit in control would merely have been transferred to the rule component of the system.

Development of meta-rules for parallel control will involve developing a vocabulary containing predicates which will allow the meta-rules to refer to the contents of rules, relationships between rules, the number and nature of instantiations in the "conflict" set, the status of each current match episode, and the current contents of working memory, among others.

5.3 Research Contribution

Why is the work described here interesting, and why, in particular, should it be of interest to people working in A.I.?

First, Gupta's basically pessimistic results (and others) have discouraged many researchers from attempting to apply parallelism to A.I. techniques[Kibler85]. The reasoning seems to be that even if we *could* build systems that contained thousands of reasonably powerful processors, most of them would remain idle most of the time. Therefore, it is more worthwhile to attempt to devise faster algorithms for serial machines or which only require low degrees of parallelism.

So it is important to examine the contention that the use of parallelism in A.I. systems is limited. If it does turn out, as seems intuitively true, that large numbers of processors are useful when attempting to devise an intelligent system, then we'd like to be able to define the situations in which large degrees of parallelism are appropriate, and algorithms by which this parallelism can be exploited.

In order to apply production systems to interesting real-time applications it will be necessary to employ parallelism. Currently, it is very difficult to write production systems that employ production level parallelism. The methodologies that I develop should ease the task of system design. Because the detection, avoidance, or toleration of inter-production interactions is an integral part of such a system, the process of incrementally adding new productions to the system should become easier than it is existing rule-based languages.

Finally, any system operating in the real world is going to find itself making assumptions about the state of the world which no longer are true when it comes time to act [Schoppers87, D-McDermott78]. There's no way of placing the entire world in a critical region. The only way to cope with a dynamic environment is to be functionally accurate. A system which is trying to produce coherent results while working on many concurrent tasks employing large numbers of processing elements in a dynamic environment is going to find it necessary to continually monitor its goals,

internal state, and surroundings to ensure that its actions remain relevant, consistent and timely. This continuous monitoring represents a first step towards truly self-aware systems which is one of the principle goals of artificial intelligence.

5.4 Related Work

Control of parallel rule-based systems is a relatively new topic in A.I. Much of the relevant work in this area has been done using blackboard systems [Nii89, Corkill89]. Nii, et al., mention the serializing nature of global control schemes. Corkill proposes using the FA/C paradigm to detect and repair inconsistencies in the database caused by non-atomic actions in the knowledge source firings. Research on controlling parallel production systems are being undertaken by Schmolze who is studying asynchronous production systems [Schmolze90] and Gamble who is studying methods of avoiding serializing control constructions using Swarm, a parallel language which allows formal proofs of correctness[Gamble90].

5.4.1 Parallel Blackboard Systems

Blackboard systems resemble production systems in many ways: they consist of multiple knowledge sources acting upon a central database; they are primarily data-driven; and they employ a central scheduler which is responsible for conflict resolution and knowledge source invocation. The primary differences are that knowledge sources in a blackboard system tend to be of a large granularity and may take a substantial amount of time to execute, the database is structured in a (usually) hierarchical fashion, and the instantiation of knowledge sources is not necessarily performed strictly by pattern-matching, so applicability of knowledge sources is not guaranteed.

Many of the issues which arise when parallelizing production systems also arise when attempting to apply multiprocessing to blackboard systems.

Blackboard systems can potentially support the following levels of parallelism:

- Implementation of low-level blackboard operations (object creation, deletion, retrieval, and modification).
- Internal parallelism within each knowledge source(KS); this results in an increase in speed of a specific KS, but the degree of speedup is dependent on the nature of the task being performed by the knowledge source.
- Knowledge source parallelism in which multiple KSs are executed as separate processes.
- Control parallelism in which the scheduling of knowledge sources is carried out in parallel with the execution of the knowledge sources.

In the work on Cage and Poligon described by [Nii89], a number of observations are made. One of the principle concerns of this work is eliminating the *serializing* effects of global control. When a global control mechanism is used, the scheduling processor must evaluate the state of the system and decide which action to take next. While this process of reflection takes place within the control processor, no work is being done elsewhere in the system. Because the global controller needs complete knowledge of the state of the system, it may be forced to wait for all extant processes to complete, thus increasing the delay imposed by the central control scheme. The Cage and Poligon projects attempted to determine whether a knowledge-based approach could eliminate global control, or alleviate its serializing delays.

As in rule-based systems, the approach which promises the greatest potential speedup for blackboard systems includes executing multiple knowledge sources in parallel, possibly in conjunction with other levels of parallelism. Ensuring the consistency of the database during concurrent KS execution is difficult in a blackboard system[Fennell77]. If the executing KSs access mutual data items, then the value of blackboard objects can change between the time when a precondition is computed and when a KS is executed. This may result in the creation of inconsistent items on the blackboard, or the unnecessary firing of knowledge sources.

Corkill addresses this problem in [Corkill89], noting that simply locking individual slots in blackboard objects is not sufficient to ensure consistency. Postponing execution of KSs which might semantically interact with a currently executing knowledge source is unreasonable and is likely to seriously reduce parallelism. The argument in this case is that an intelligent scheduler would require knowledge about every potential interaction between every knowledge source instantiation (KSI); not only is this computationally expensive, but in cases in which the KSI behavior is strongly dependent on the data, predicting potential interactions can lead to over-conservative scheduling and loss of concurrency.

The "dataflow" model of programming in which new versions of blackboard objects are created to reflect changes, and existing objects are never modified represents a potential solution, but greatly increases memory and processing costs. Another approach is to treat a KS computation as an atomic operation, using locks on blackboard objects and regions in order to avoid conflicts while minimizing interference with other KSs.

Production system differ from blackboard systems in a number of ways which make the control issues analogous rather than identical. Productions automatically enter the conflict set whenever they become enabled while a blackboard system must explicitly execute a (potentially expensive) precondition to determine if a knowledge source is eligible to fire. The blackboard data structure is much more structured than the production system's working memory. This allows a more precise approach to locking and updating resources. This advantage is largely negated by the relatively long duration of knowledge source executions which can result in significant losses

of parallelism if resources have to be locked. The long lifetime of knowledge source executions allows more time for deliberation over control issues; a typical production system rule must be executed as soon as it is enabled or it will spend more time in the conflict set than it does executing.

Rules represent (or are intended to represent) very small focussed "chunks" of knowledge, potentially many of which are applicable to a given situation. Knowledge sources are more monolithic and may not be able to achieve high degrees of parallelism unless the KS supports internal parallelism.

5.5 Research Program

The following section contains a list of the tasks involved in completing the proposed thesis and their expected start dates and durations. All the dates are subject to change as the research proceeds.

5.5.1 Models of Rule Interactions

Description of Work Previous research in parallel production systems have focussed on clashing and disabling rule interactions, terms which are borrowed from database systems. There are significant differences, however, between the function of working memory and a database's memory. In addition, the operations which affect a production system's knowledge base are, in most cases, algorithmic and purposeful. The phase of the research will attempt to model the causes of rule interactions and identify the actions to be taken by the scheduler in each event.

Contribution We have ways of identifying rules which interact *over the course of one conflict set iteration* but we have no real model of why rules which affect the same data are occurring concurrently. Is it a case of two types of knowledge applying to the same problem? Is one production instantiation redundant? Are the productions mutually exclusive? Can we model multiple production firings as operators in an AND/OR tree? What is the role of conventional conflict resolution techniques in a parallel rule-firing system?

Without a clear model of the semantics behind a production instatiation, it will be difficult to make the correct control decisions required to determine when and if that production should be executed. As a result of this phase of the research, it will be possible to develop language constructs which avoid many of the more common production interactions and will generate a better model of those remaining interactions.

5.5.2 Development of Algorithms for Parallel Control

Description of Work The level of control present in a parallel production system ranges from the tightly controlled "lockstep" processing model which examines the conflict set after quiescence and selects only non-interacting rules to the fully asynchronous "demon" model in which productions clamor to be executed as soon as they are satisfied.

A number of approaches for controlling (or not controlling) these parallel rule-based systems have been discussed. These include totally asynchronous rule execution with functionally accurate monitoring, concurrent control monitoring with incremental conflict resolution, meta-rules, and predictions of future matching quality. The selection of the appropriate "conflict resolution" algorithm depends on both the high-level model of the computation (search, forward-chaining, pattern-recognition) and the low-level nature of the data. This phase of the research will attempt to define efficient algorithms for controlling a parallel production system and will define the computation models and situations for which they are appropriate.

Contribution One of the major contributions of this work is to determine whether it is possible to control a parallel rule-based system without imposing undesireable serialization or unreasonable complexity upon the system.

5.5.3 Development of an Intelligent Controller for Parallel Rule Execution

Description of Work This phase of the research will concentrate on the design of a sophisticated controller which will intelligently schedule productions to execute according to their expected interactions with other productions, incoming data, and the current state of the computation.

Contribution We have observed that the highest efficiency results when productions are executed as soon as they are enabled, however, as in any parallel system, certain operations must be synchronized in order to avoid data inconsistency. I would like to show that in a "knowledge-based" system, sufficient information is available to allow a sophisticated scheduler to flexibly decide when rules are eligible to fire.

5.5.4 Development of a Parallel Production System

Contribution The proposed work requires an experimental vehicle to test assertions about program constructs and control algorithms.

Description of work The parallel rule-based language will have a syntax essentially similar to OPS5, but will support the major levels of parallelism: production,

node, and action level parallelism. The principle innovation in the language will be the control mechanism which will differ in several ways from the conventional recognize-select-act control cycle and the data constructs required to support consistent parallel execution of rules.

Provisions should be made for switching between parallelism modes, or operating serially for benchmarking purposes. The probable need to experiment with the control mechanisms, syntax, and semantics of the language to optimize it for parallel operation makes Lisp a logical choice for an implementation language. Documentation and an accompanying technical report describing the language will be provided.

5.6 Results

If research progresses as described in this document, the following results should be obtained:

- A list of the causes of rule interactions in terms of the underlying language idioms.
- Algorithms and language constructs for implementing common AI paradigms in a parallel rule-based language. When possible, analyses of expected performance will be performed.
- A description of a controller which is responsible for controlling the execution of productions and maintaining database consistency.
- A programming language which implements the language constructs and controller.
- Benchmarks which demonstrate the effectiveness of the control algorithms in maximizing parallel rule executions.

6 Conclusion

The benefits of parallelism in productions systems appear to be limited only by the degree of parallelism inherent in the application itself. By decomposing applications into semi-independent tasks, each composed of one or more production firings, the level of parallelism can be shown to increase dramatically. At the same time, the increased number of potential interactions betwen productions, and the increased complexity of parallel systems impose a burden on the implementor. In order to take full advantage of the available parallelism, new control mechanisms must be devised which do not present serializing bottlenecks or unacceptable overhead. The research proposed here will focus on new techniques for creating, analyzing, and controlling highly parallel rule-based systems.

References

[Acharya] Acharya, A., Milind Tambe, "Production Systems on Message Passing Computers: Simulation Results and Analysis", School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213-3890.

[Brooks85] Brooks, Ruven and R. Lum, "Yes, An SIMD Machine Can Be Used for AI", IJCAI85, pp. 73-79.

[Corkill89] Corkill, Daniel D., "Design Alternatives for Parallel and Distributed Blackboard Systems", in *Blackboard Architectures and Applications*, V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, eds., Academic Press, pp. 99-136.

[Davis80] Davis, Randall, "Meta-Rules: Reasoning about Control", *Artificial Intelligence* 15 (1980), pp. 179-222.

[Decker90] Decker, K., A. Garvey, M. Humphrey, V. Lesser, "Effects of Parallelism on Blackboard System Scheduling", COINS Dept., *Proceedings of the Fourth AAAI Blackboard Workshop*, August, 1990.

[Durfee87] Durfee, E. H. and V.R. Lesser, "Planning to Meet Deadlines in a Blackboard-based Problem Solver", COINS Technical Report 87-07, University of Massachusetts, February 1987.

[Fennell77] Fennell, Richard D., Victor R. Lesser, "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II", *IEEE Transactions on Computers*, Vol. C-26, No. 2, February 1977, pp. 98-111.

[Forgy79] Forgy, C.L., *On the Efficient Implementation of Production Systems*, Dept. of Computer Science, Carnegie-Mellon University, 1979.

[Forgy81] Forgy, C.L., "OPS5 User's Manual". Technical Report CMU-CS-84-135, Dept. of Computer Science, Carnegie-Mellon University, July 1981.

[Forgy82] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence*, 19 , 1982, pp. 17-37.

[Forgy84] Forgy, C.L., "The *OPS83* Report", Technical Report CMU-CS-84-133, Department of Computer Science, Carnegie-Mellon University, May 1984.

[Gamble90] Gamble, Roseanne Fulcomer, "A Methodology for Developing Correct Rule-based Programs for Parallel Implementation", Thesis Proposal, Washington University, Sever Institute of Technology, 1990.

[Gupta84] Gupta, Anoop, "Implementing OPS5 Production Systems on DADO", CMU-CS-84-115, Department of Computer Science, Carnegie-Mellon University, 1984.

[Gupta87] Gupta, Anoop, *Parallelism in Production Systems*, Morgan Kaufman Publishers, Inc., Los Altos, CA, 1987.

[Gupta87a] Gupta, Anoop, Charles Forgy, et al., "Results of Parallel Implementation of OPS5 on the Encore Multiprocessor", CMU-CS-87-146, Computer Science Dept., Carnegie-Mellon University, 1987.

[Gupta88] Gupta, Anoop, Milind Tambe, Dirk Kalp, Charles Forgy, and Allen Newell, "Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis", International Journal of Parallel Processing, Vol 17, No. 2, April, 1988.

[Gupta89] Gupta, A., C. Forgy, A. Newell, "High-Speed Implementations of Rule-Based Systems", *ACM Transactions on Computer Systems*, Vol. 7, No. 2, May 1989, pp. 119-146.

[Gupta89a] Gupta, A. and C. Forgy, "Static and Run-Time Characteristics of OPS5 Production Systems", *Journal of Parallel and Distributed Computing*, 7, 64-95, 1989.

[Harvey89] Harvey, Wilson, Dirk Kalp, Milind Tambe, David McKeown, and Allen Newell, "Measuring the Effectiveness of Task-Level Parallelism for High-Level Vision", CMU-CS-89-125, March 27, 1989.

[Hayes-Roth85] Hayes-Roth, B. "A Blackboard Architecture for Control", *Artificial Intelligence* Vol. 26 (1985) 251-321.

[Hillyer86] Hillyer, B.K., and D. E. Shaw, "Execution of OPS5 Production Systems on a Massively Parallel Machine", *Journal of Parallel and Distributed Processing*, August, 1988.

[Ishida85] Ishida, T. and Stolfo, S., "Towards the Parallel Execution of Rules in Production System Programs", *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 568-575, 1985.

[Kalp85] Kalp, Dirk, et al., *Parallel OPS5 User's Manual*, CMU-CS-88-187, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[Kelly89] Kelly, Michael, and Rudolph Seviora, "An Evaluation of DRete on CUPID for OPS5 Matching", *IJCAI-89*, pp. 84-90.

[Kibler85] Kibler, D. and Conery, J., "Parallelism in AI Programs", IJCAI, 1985.

[Laird87] Laird, J.E., A. Newell, and P.S. Rosenbloom, "Soar: An Architecture for General Intelligence", *Artificial Intelligence* 33:1-64,1987.

[Lehr85] Lehr, Theodore, "The Implementation of a Production System Machine", CMU-CS-85-126, Department of Computer Science, Carnegie-Mellon University, May, 1985.

[Lesser81] Lesser, V. and Corkill, D., "Functionally Accurate, Cooperative Distributed Systems", *IEEE Transactions on Man, Machine, and Cybernetics*, Vol. SMC-11, No. 1, January 1981.

[D-McDermott78] McDermott, Drew, "Planning and Acting", *Cognitive Science*, Vol. 2, pp. 71-109, 1978.

[McDermott78] McDermott, J. and C. Forgy, "Production System Conflict Resolution Strategies", in D.A. Waterman and F. Hayes-Roth, eds., *Pattern-Directed Inference Systems*, New York, Academic Press, 1978, pp. 177-199.

[McDermott80] McDermott, J., "R1: A Rule-Based Configurer of Computer Systems", Technical Report CMU-CS-80-119, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA.

[McDermott83] McDermott, J., "Extracting Knowledge from Expert Systems", IJCAI-83, pp. 100-107.

[Miranker89] Miranker,Daniel, Chin-Ming Kuo, and James C. Browne, "Parallelizing Transformations for a Concurrent Rule Execution Language", TR-89-30, Department of Computer Science, University of Texas at Austin, October, 1989.

[Miranker90] Miranker, Daniel P., "An Algorithmic Basis for Integrating Production Systems and Large Databases", *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA, February, 1990.

[Morgan88] Morgan, Keith, "BLITZ: A Rule-Based System for Massively Parallel Architectures", In *Proceedings of 1988 ACM Conference for Lisp and Functional Programming*, Snowbird, Utah, 1988.

[Nayak88] Nayak, Pandurang, Anoop Gupta, Paul Rosenbloom, "Comparison of the Rete and Treat Production Matchers for Soar (A Summary)", *AAAI-88.*

[Neiman87] Neiman, Daniel, "Adding the Rete Net to Your OPS5 Toolbox", *AI Expert*, January, 1987, pp. 42-49.

[Neiman90] Neiman, Daniel, "Control Issues in Parallel Production Systems", Technical Report in preparation, COINS, 1990.

[Neiman90a] Neiman, Daniel, "Parallel OPS5 User's Manual and Technical Report", COINS Technical Report 91-1, Computer and Information Sciences Dept., University of Massachusetts, 1991.

[Nii89] Nii, H. Penny, Aiello Nelleke, James Rice, "Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems", in *Distributed Artificial Intelligence, Vol. II*, Les Gasser and Michael N. Huhns, eds., Morgan Kaufman Publishers, Inc., 1989, pp. 319-384.

[Oflazer84] Oflazer, Kemal, "Partitioning in Parallel Processing of Production Systems", *Proceedings of the IEEE International Conference on Parallel Processing*, 1984.

[Okuno88] Okuno, H., A. Gupta, "Parallel Execution of OPS5 in QLISP", *Proceedings of the Fourth Conference on Artificial Intelligence Applications*, March 1988, pp. 268-273.

[Schmolze89] Schmolze, James G.,"Guaranteeing Serializable Results in Synchronous Parallel Production Systems", Technical Report 89-5, Department of Computer Science, Tufts University, October, 1989.

[Schmolze90] Schmolze, James G. and S. Goel, "A Parallel Asynchronous Distributed Production System", *AAAI-90*, pp. 65-71.

[Schoppers87] Schoppers, M. J., "Universal Plans for Reactive Robots in Unpredictable Environments", *Proceedings of the 10th Internation Joint Conference on Artificial Intelligence*, pp. 852-859.

[Selfridge60] Selfridge, Oliver G. and Ulric Neisser, "Pattern Recognition", *Scientific American*, August, 1960, pp. 60-68.

[Siler87] Siler, William, Douglas Tucker, and James Buckley, "A Parallel Rule Firing Fuzzy Production System with Resolution of Memory Conflicts by Weak Fuzzy Monotonicity, Applied to the Classification of Multiple Objects Characterized by Multiple Uncertain Features", *International Journal of Man-Machine Studies*, (1987) Vol. 26, 321-332.

[Stolfo84] Stolfo, Salvatore, Daniel Miranker, "DADO: A Parallel Processor for Expert Systems", *Proceedings of the 1984 Int. Conference on Parallel Processing*, August , 1984, pp. 74-82.

[Stolfo84a] Stolfo, Salvatore, "Five Parallel Algorithms for Production System Execution on the DADO Machine", *Proceedings of the National Conference on Artificial Intelligence, AAAI-84*, pp. 3

[Stolfo87] Stolfo, Salvatore J., "Initial Performance of the DADO2 Prototype", *Computer*, january, 1987, pp. 75-82.

[Tambe88] Tambe, M., D. Kalp, A. Gupta, C. Forgy, B. Milnes, and A. Newell, "Soar/PSM-E: Investigating Match Parallelism in a Learning Production System", *Proceedings of Parallel Programming Environments, Application Languages, and Systems(PPEALS)*, July, 1988.

[Tambe89] Tambe, Milind, Anurag Acharya, Anoop Gupta, "Implementation of Production Systems on Message Passing Computers: Techniques, Simulation Results, and Analysis", CMU-CS-89-129, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[Tenorio85] Tenorio, M. and D. Moldovan, "Mapping Production Systems into Multiprocessors", *Proceedings of the IEEE International Conference on Parallel Processing*, 1985, pp. 56-62.