Control Issues in Parallel Rule-Firing Production Systems [*]

Daniel E. Neiman
Department of Computer and Information Sciences
University of Massachusetts
Amherst, MA 01003
DANN@CS.UMASS.EDU

Abstract

When rules are executed in a parallel production system, the goal of control is to ensure both that a high-quality solution is achieved and that processing resources are used effectively. We argue that the conventional conflict resolution algorithm is not suitable as a control mechanism for parallel rule-firing systems. The necessity for examining all eligible rules within a system imposes a synchronization delay which limits processor utilization. Rather than perform conflict resolution, we propose that rules should be executed asynchronously as soon as they become enabled, however, this approach leaves the problem of controlling the computation unsolved. We have identified three distinct types of control, *program sequencing*, *heuristic control*, and *dynamic scheduling*, which are required for efficient and correct parallel execution of rules. We discuss the issues involved in implementing each type of control without undue overhead within the context of our system, a parallel rule-firing system with an augmented agenda manager.

Introduction

The rule-based system is a fundamental paradigm in AI research which allows domain specific knowledge to be captured in rules and applied to situations in a data-directed manner. In order to increase the performance of such systems, recent research has explored the benefits of *rule-level* parallelism in which many rules are allowed to fire simultaneously [Ishida and Stolfo 1985, Schmolze 1989, Miranker, Kuo and Browne 1989]. Much of this research has been directed towards the problems of extracting parallelism from existing rule sets or ensuring that the parallel execution of rules produces a serializable result. Our research has focussed instead on the problem of *controlling* a parallel rule-firing system. We have found that within the context of such a system, the conventional conflict resolution control mechanism unnecessarily synchronizes rule-based computations, encourages the use of serializing control idioms, and cannot express strategies for ensuring efficient use of processing resources. In order to eliminate the synchronization delay imposed by conflict resolution and increase the average level of activity within our system, we employ a rule-firing policy called "fire-when-ready" which executes rules asynchronously. We discuss the effects of such a policy on the control of a rule-firing system and identify three distinct types of control activity which must occur during the parallel execution of rules: dynamic scheduling, heuristic discrimination between alternative activities, and algorithmic sequencing of computational phases:

- **Dynamic Scheduling:** Dynamic scheduling describes the process of scheduling rule activations so as to make effective use of limited processing resources while ensuring that the computation proceeds in a timely manner. We describe a set of heuristic rules derived from our research in parallel blackboard systems which can be applied to scheduling parallel rule firings.
- **Heuristic Control:** Heuristic control consists of the process of determining which of a set of alternative actions should be executed when more than one are appropriate to a given situation. The method of achieving heuristic control via conflict resolution is inherently serializing and we discuss methods of avoiding or reducing this overhead in asynchronous rule-firing systems.
- **Algorithmic control:** We define algorithmic control as the process by which common programming idioms such as iteration and program sequencing are implemented within a rule-based system. We present a solution to the program sequencing problem which is non-serializing and does not depend on either the matching process nor a conflict resolution proctocol.

In this paper, we describe methods of achieving each type of control within the context of our experimental vehicle, a parallel rule-firing production system with an augmented agenda manager. Finally, because the

[*]This work was partly supported by the Office of Naval Research under a University Research Initiative grant, number N00014-86-K-0764, NSF-CER contract DCR-8500332, and DARPA contract N00014-89-J-1877.

asynchronous execution of rules prohibits a dynamic analysis of rule interactions, we present a method of partially enforcing database consistency which is sufficient to allow our control idioms to be implemented without imposing undue overhead upon the system. In order to place our research in context, we present a brief survey of previous work on parallel rule-firing systems in the following section.

Related Work

In describing his research on the sources of parallelism within production systems, Gupta noted that performance could be greatly increased by executing rules in parallel and cited the SOAR system[Laird, Newell, and Rosenbloom 1987] as one which was potentially capable of executing all members of its conflict set in parallel [Gupta 1987]. Ishida and Stolfo first discussed the problem of maintaining consistency within a parallel rule-based system [Ishida and Stolfo 1985]. Their solution, which involved detecting references by rules to particular classes of working memory elements significantly restricted parallelism; effectively only one instance of each rule could fire simultaneously. More recently, Schmolze has devised an algorithm which uses both static analysis and a run time unification scheme to detect more precisely the rules within the conflict set which potentially interact [Schmolze 1989]; Schmolze's scheme allows multiple *instantiations* of rules to be executed in parallel. Both Schmolze and Ishida have proposed distributed parallel rule-firing systems. Ishida's system dynamically organizes itself to meet demand by copying rules to other processors [Ishida, Yokoo, and Gasser 1990]. Schmolze's system, PARS, assigns distinct rules to each processor and distributes working memory elements to each relevant rule; the rules fire asynchronously, using a communication protocol between the processors to ensure that consistency of working memory is maintained.

Recent work by Ishida has discussed design principles for parallel rule-based systems, dividing the problem into three components: interference analysis; the parallel firing algorithm which determines how rules should be distributed; and the parallel programming environment which provides language facilities for parallel rule execution [Ishida 1990]. Ishida's paper is one of the first to discuss control issues; he proposes dividing rules into separate *rule groups* and defining separate conflict resolution policies for each group, and dividing rules into two classes, control and heuristics; these two ideas also appear in our own work. Miranker has proposed allowing independent rules sets to execute asynchronously in parallel and has developed optimizing transformations for partitioning rules into *mutual exclusion* sets [Miranker, Kuo and Browne 1989], but has not addressed the problem of control in parallel asynchronous programs.

In our previous work in cooperative distributed problem solvers, we have performed considerable research in the control of parallel and distributed blackboard systems and have noted many similarities between the issues faced in distributed blackboard systems and those of rule-based systems[Corkill 1989, Decker et al. 1991]. Both types of systems consist of knowledge sources or productions acting upon a central data structure. Eligible knowledge sources are stored upon an agenda, and *agenda management* must be performed to ensure that the appropriate knowledge sources are executed in a timely fashion. It is natural to investigate whether it is possible to apply some of the techniques appropriate to large-grained blackboard systems in which the operators are relatively long-lived to the finer grained rule-based systems.

The Parallel Rule Firing System

At the University of Massachusetts, we have developed a parallel language based on the rule programming language OPS5 which supports both matching and rule-level parallelism [Neiman 1991a]. The system is implemented in TopCL[1], a parallel Common Lisp which supports futures and lightweight tasks called *threads.* Using this system and a set of benchmarks which display high levels of potential rule parallelism, we have observed speedups of 8 to 10 times over the serial case on a Sequent Symmetry shared memory multiprocessor with 16 processors.

Our initial use of this system was to experiment with a number of different rule firing policies and collect data on their relative performance. One of our first observations (also reported in [Miranker, Kuo and Browne 1989]) was that rule-firing in a parallel system should proceed, whenever possible, in an asynchronous manner, using what we call the "fire-when-ready" policy.

Previous rule-firing schemes have employed a synchronous rule-firing policy which performs conflict resolution, detects potential rule interactions, and creates a maximal set of executable rule instantiations which are then fired; the system must then wait for completion of all working memory changes. This synchronization delay can cause eligible rule instantiations to remain idle in the conflict set for significant lengths of time due to discrepancies in matching and execution time between rules. In our experiments, the *asynchronous* rule firing policy reduced conflict set latency time significantly, resulting in performance improvements of between a factor of 1.5 and an order of magnitude, depending on the benchmark under consideration. The speedup becomes more pronounced as the variance in production execution times in the benchmark increases. There is a trade-off between performance and control; the requirement of the "fire-when-ready" strategy that rules execute as soon as they become enabled eliminates the role of the conflict set in

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

determining the control policy of the system and detecting potential rule interactions [Neiman 1991b].

In order to contend with the control issues we have encountered, we are developing an agenda manager which is responsible for dynamically scheduling productions, interpreting meta-level specifications of program sequencing, and ensuring that pathological interactions do not exist between executing rules.

Control Issues

In this section, we discuss the three control areas we have identified, dynamic scheduling, heuristic, and algorithmic control and briefly outline the issues associated with achieving each type of control within the context of an asynchronous rule-firing system.

Dynamic Scheduling

In a production system in which many rules are eligible to fire at a given time, there is substantial competition for processing and matching resources. For example, in a modified version of the Toru-Waltz benchmark[2] we have seen as many as 124 rule instantiations eligible to fire concurrently on a 16 processor machine. In any implementation of a parallel OPS-like language, there will also be unavoidable contention for critical regions within the Rete net, the principal data structure used in the match process [Forgy 1979]. We have observed the following phenomena: first, when a standard conflict resolution scheme is used, many rules must be activated within a short span of time, causing massive contention for processors and shared resources as evidenced by increased latency periods on processor queues and increased wait time for locks in critical regions such as timetag allocation routines[3]. Second, when many instantiations of the same *type* of rule are active, they initiate similar working memory changes which must compete for access to matching resources, as evidenced by increased contention for locks on the memory nodes within the Rete net. Finally, rules which produce results necessary to the computation may not execute promptly due to contention for processors with less critical rules.

The problem of contention for resources is reduced, but not eliminated, by the use of the "fire-when-ready" policy. If rules are allowed to fire whenever they are eligible, the demand for processors and resources is spread out over a larger time period. Because the time spent in critical regions is very short, staggering the demand for resources eliminates much of the contention. To solve the remaining problems, we propose using a heuristic scheduling scheme which uses simple knowledge about the relative priorities of rules and their resource utilization: this scheme has been successfully applied by members of our research group to the scheduling of low-level blackboard activities [Decker et al. 1991]; it remains to be seen whether it will be effective for the finer-grained rule scheduling. Note that dynamic scheduling is only appropriate when the number of rules eligible to be executed exceeds the number of available processors; the heuristic rules can be then applied while waiting for resources to become free.

The following set of heuristic scheduling rules illustrate some of the goals of dynamic scheduling: increasing processor utilization, reducing contention for shared resources, and meeting deadlines:

- Develop triggering data first: Production systems are data-driven, thus, rules which produce data necessary for the activation of many rules should be executed as soon as possible. For example, in a circuit simulation program, this policy would schedule rule instantiations simulating earlier clock times first. In such cases, the *recency* criteria of conventional conflict resolution could result in starvation, preventing the necessary data from being asserted.

- Schedule goal-related rules first: The structure of parallel rule-based programs makes it necessary to divide computations into rules which perform goal-related tasks, and rules which perform bookkeeping functions, such as deleting obsolete working memory elements. While these latter increase the efficiency of the matching process by reducing node memory sizes, they do not further the computation and should be scheduled with a lower priority.

- Execute dissimilar rules: We have observed that the simultaneous execution of many instantiations of the same rule may lead to contention for system resources. When the level of activity exceeds a threshold, the matching process becomes effectively serial. While we have reduced this overhead using hashing techniques, contention should also be reduced by scheduling rules which have disparate righthand sides.

- Delete redundant rule instantiations: In a parallel asynchronous system, it is necessary to disable activities once they have become redundant or unnecessary in order to conserve processing resources. For example, in a goal-directed system consisting of an AND/OR hierarchy, unless the scheduler explicitly halts their execution, tasks related to a high-level goal may continue to execute indefinitely after that goal has been achieved, or has been shown to be unachievable.

[2]The Toru-Waltz program, written by Toru Ishida, demonstrates considerable rule parallelism and has been widely used as a benchmark in the literature on parallel rule-firing systems. For a description of the benchmark and analysis of the sources of parallelism, see [Neiman 1991a].

[3]In order to ensure that working memory elements are assigned unique timetags, the allocation of timetags and other necessary bookkeeping must be performed in a critical region.

Heuristic Control in Parallel Rule-firing Systems

We distinguish heuristic control from scheduling in that control is concerned not with the efficient use of resources, but rather with optimizing the quality of the eventual solution by selecting the best alternative among a set of applicable rules. The problem of heuristic control is complicated by the relatively short execution time of rules and the requirement that rules be executed asynchronously whenever possible. Because the 'fire-when-ready' policy executes productions before sufficient information is available to generate completely accurate control decisions, a computation must either generate only the correct productions to execute or be able to recover from the occasional spurious production execution. Space limits our discussion of heuristic control to the three following cases:

- Classes of problems in which conflict resolution need not be performed either because the nature of the problem ensures a one-to-one mapping between rules and subproblems or because the necessary focussing heuristics are already contained within the rules themselves.
- Algorithms in which executing conflicting rules can be viewed as a parallel search process whose cost is less than that of heuristic control and synchronization.
- Problems in which the number of conflicting rules grows exponentially and heuristic conflict resolution is unavoidable.

Execution of Independent Subtasks There is a large class of problems presenting significant opportunities for rule-level parallelism which do not require heuristic control. These problems possess the property that any rule which becomes enabled is both necessary to the computation and is not subsumed by any other rule instantiation. We have observed that problems which display *object* or *data* parallelism usually possess this property and are particularly well-suited for asynchronous rule execution. An example of a problem in this domain is the Toru-Waltz benchmark; our analysis of this program indicates that rule instantiations enter the conflict set monotonically, with each rule instantiation corresponding to a single labelling constraint. Each constraint is a unique entity which may be processed in a data-directed manner. Because the enabled rule instantiations do not conflict, but rather co-exist within the conflict set, no heuristic decisions are necessary, and the rules may be fired asynchronously.

Parallel Exploration of Alternatives The potential of parallel program to explore many alternatives concurrently is one of the most potent arguments for reducing the dependence on conflict resolution. Because rules are typically inexpensive to execute, the overhead of occasionally executing an inferior rule may well be less than the combined cost of waiting for quiescence within the conflict set and performing a heuristic evaluation function. Instead of viewing rules attending to the same subproblem as mutually exclusive and requiring conflict resolution, they can be viewed as operators generating the set of successor nodes to a node in a state space search. Executing all productions will lead to an exploration of all relevant search paths.

Unless an enumeration of all possible solutions to a subproblem is desired, parallel execution of rules which produce competing solutions requires a mechanism for ensuring that a unique solution is eventually produced. The principal reason for this is the avoidance of combinatorial growth of rules within the conflict set. If multiple potential solutions enter working memory, they may each enable successive rules, eventually causing saturation of the system. One possible solution is to implement a form of what Siler, *et al.* have labeled *memory conflict resolution* [Siler, Tucker, and Buckley 1987] in which working memory elements may be modified only if the changes monotonically converge towards a solution. We can produce a similar result in an asynchronous system by using a heuristic rule to propagate solutions of increasing quality in a data-directed fashion:

1. Create a working memory element of the type `(current-solution ^subproblem subproblem-id ^value nil)` to hold the final value.
2. For each rule firing which develops a potential solution, asynchronously assert a working memory element of the form: `(potential-solution ^subproblem subproblem-id ^value value)`.
3. Create a heuristic *join* rule which compares the value of each `potential-solution` to the value of `current-solution`and updates `current-solution` as necessary, while deleting the tentative solution[4].
4. Delete any rule instantiations and working memory elements created by the assertion of the previous solution.

This algorithm is particularly appropriate for real time systems in that it possesses the desirable feature of producing incrementally better results as time passes. Computations of this sort which converge on solutions despite incomplete or inconsistent memory states have been termed *functionally accurate* by Lesser and Corkill [Lesser and Corkill 1981].

Partitioning the Conflict Set for Heuristic Decisions The approach of performing a parallel search of alternatives by executing all eligible rules and merging the results depends on the set of eligible rules being manageably small. In cases in which the search space

[4]In an asynchronous system, multiple instantiations of this join rule may become eligible to fire simultaneously, causing a potential *clashing* situation. Our approach to the problem of enforcing consistency is discussed in a later section.

grows exponentially, rules must be collected within a conflict set so that heuristic pruning may be performed. In order to reduce the synchronization overhead associated with conflict resolution, we propose partitioning the conflict set and only performing conflict resolution between rule instantiations which are attending to the same subproblem.

The partitioning process presents two problems which are not explicitly present within the standard production system paradigm. Rules instantiations must be identified as being relevant to a particular subproblem so that they can be assigned to partitions, and it must be determined when all rules bearing on a subproblem have become enabled so that heuristic discrimination can begin.

Attaching rule instantiations to subproblems: Individual working memory elements may be matched by rules in different subproblems, and individual rules may operate on different sets of data; therefore rule instantiations cannot be easily partitioned on the basis of either the type of rule or the data matched [McDermott and Forgy 1978]. Except for problems displaying object parallelism in which the correlation between rule and object is easily distinguished, we take the approach of explicitly annotating rule instantiations.

We make the observation that any goal-directed activity is effectively traversing a state space. For any rule activation, a subset of the working memory elements referenced by the lefthand side of the rule denote an individual state within the state space, while the remaining elements represent global facts which remain static over the course of the computation.

In order to partition elements into a particular state, we assign each element a unique state identifier. In order to ensure that each rule instantiation accesses only working memory elements from either the current space or global data, we define a goal element which defines the type of the subproblem and which is augmented with a unique tag indicating the appropriate space. Each rule instantiation stimulated by a goal element is assigned to a conflict set corresponding to its state identifier. Because only rules in the same conflict set may interact, the contents of disparate conflict sets may be scheduled independently, minimizing synchronization delays.

Achieving quiescence within a partitioned conflict set: Before a heuristic decision can be made, all relevant rule instantiations must be present in the conflict set. In general, we can never guarantee that this is the case. Even if quiescence is achieved throughout the entire system, a rule firing at a later time may produce new information relevant to the current state. Thus, we make the assumption when testing for quiescence within a partitioned conflict set that all working elements relevant to the current state have been asserted when the activating goal is asserted, or are asserted in parallel with the activating goal element. We further assume that all global working memory elements not specific to the current state remain stable during the course of conflict resolution. If these assumptions hold, we can insure quiescence within a partitioned conflict set by waiting until each working memory element which is annotated as belonging to the current state has been asserted and propagated throughout the entire network.

This discussion has not fully addressed the problem of determining quiescence; for example, when performing an AND/OR traversal of a search space, it is necessary to determine when all competing or contributing subproblems have developed solutions – in situations such as these, the agenda manager must maintain a record of the goal hierarchy and check each task for quiescence.

Algorithmic Control

Although rule-based programs are data-driven, it is common practice to impose specific orderings on the rule firings in order to emulate programming idioms such as iteration, to implement particular algorithms such as search or goal-directed reasoning, or to partition the computation into discrete phases. Because of their dependance on the manipulation of conflict resolution and rule ordering, such idioms are rarely suitable for parallel rule execution. There are certain unavoidable serial constructs, for example, I/O operations and trivial iteration of operations over working memory elements. We argue that such operations are not essentially knowledge-based and are best performed using imperative constructs such as those provided in the OPS83 programming language[Forgy 1984].

One of the most problematic algorithmic constructs is that of program sequencing, that is, ensuring that a computation proceeds through a number of discrete processing phases. This is usually done by means of *mode* or *goal* working memory elements. Each rule in a given processing phase contains a reference to the mode element in its lefthand side. In order to change modes, a mode-changing rule is provided which is designed to fire only when no other rules within the phase are eligible. This particular construct poses a number of problems to the parallel programmer. First, it depends on the *specificity* condition of the standard conflict-resolution algorithm. When executing rules in parallel, conflict-resolution is not performed and the mode-changing production, being always enabled, may fire in parallel with other rules in the phase, disabling them prematurely. The mode-changing productions, therefore, should not be executed in parallel with any other rule.

To make matters worse, mode-changing is match intensive. Not only are large numbers of rules enabled by the addition of a new mode element, causing contention within the Rete net, but many partial matches within the Rete net must be retracted. We have observed that it is not unusual for a mode-changing rule which adds a single working memory element to take two orders of magnitude longer to fire than any other rule within the phase. Within the Toru-Waltz benchmark, for example, in a run of 370 rule firings, a single mode-changing rule consumes 25% of the run time. While we can reduce this execution time by enabling *match* parallelism for the duration of the production execution or allowing rules enabled by the mode change to execute asynchronously, it is clear that a construct which requires a single working memory change to activate a large number of rules is inherently serializing.

In order to remove the overhead of mode-changing while allowing the programmer to specify program sequencing, we have implemented a meta-level facility which allows the programmer to specify both a type and a rule group for each production. Each mode-changing production is explicitly tagged as a *mode-changer*. Our agenda manager ensures that no mode-changing rule will fire until all rules within a phase have executed and quiescence has been achieved.

To remove the matching overhead associated with mode-changing, we have created a new righthand-side action which allows a rule to communicate a new mode to the agenda manager. Because no matching is associated with the specification of the new mode, execution of mode-changing productions is extremely fast, and rules within the new computational phase are immediately executed by the agenda manager. During each computational phase, rule instantiations belonging to a subsequent phase may become enabled and enter the conflict set but will not execute until the mode has changed. This approach distributes the overhead of matching rules instantiations for subsequent phases over a set of parallel activities rather than the serial addition of a single working memory element.

Correctness of Algorithms

A problem which is closely related to the problem of algorithmic control is that of maintaining database consistency. When productions are allowed to execute in parallel, they may interact to produce results which could not be achieved by any serial ordering of rule firings.

A number of techniques have been developed for detecting rule interactions [Ishida and Stolfo 1985, Schmolze 1989, Ishida 1990]. These algorithms usually consist of a static analysis phase which is performed at compile time and a runtime component which dynamically examines all eligible rules in order to select a set of co-executable rules. The runtime component is relatively expensive, both because of the synchronization cost of accessing all eligible rules, and because it must perform unification of variables in order to precisely identify the rule interactions.

Our study of parallel rule-firing programs has indicated that rule interactions occur only rarely. Rather than accept the synchronization delays associated with a full analysis of rule interactions, we have chosen to only enforce a subset of the correctness criteria using a scheme of read/write locks on working memory elements. Each working memory element is assigned a write flag and a read counter. A working memory element whose write flag is set may not be modified or referenced by any rule instantiation. Any working memory element whose read counter is greater than zero cannot be modified by any rule. When a rule instantiation is selected from the agenda, it must check the write flags of elements it references and the read counters of elements it wishes to modify. If it is not safe to perform the desired operation, the rule instantiation is replaced on the agenda for future execution, otherwise the appropriate flags are set and counters are incremented.

The use of locks to enforce working memory consistency has the advantage of not requiring either static analysis of rule sets or runtime unification of variables, does not require synchronization of the conflict set, and possesses an extremely low overhead, on the order of one percent of rule execution time. However, when using this scheme, the agenda manager detects only a subset of potential rule interactions.

As it is not possible to lock a non-existent working memory element, interactions in which one of a pair of rules adds a working memory which is negatively referenced by the other may still occur. We have chosen to respond to this problem by creating specific language idioms and constructs which, instead of *guaranteeing* correctness, allow the *design* of correct programs. For example, in the *initialization* idiom, in which a rule checks for the existence of a working memory element and creates it if it does not exist, we have developed a variant of the `make` command which allows one and only one instance of a particular working memory element to be created, thus avoiding potential multiple executions of the initialization rule. So far, we have found that our locking mechanisms are both necessary and sufficient to allow the design of correctly written systems.

Conclusion

Executing productions in parallel in a rule-based system promises very high rates of rule execution, but only if the synchronization overhead introduced by conventional conflict resolution policies can be eliminated. We have taken the approach of executing productions asynchronously whenever the nature of the algorithm permits; this approach greatly increases processor utilization, but eliminates the principal control mechanism of rule-based systems. We have described three areas of control for which alternative mechanisms must be developed: dynamic scheduling of rules, heuristic control, and algorithmic sequencing of rule execution. To support these control activities, we have developed an agenda manager which provides support for enforcing consistency of the database, allows the user to specify rule types and groups for sequencing rule executions, and which allows both asynchronous and synchronous execution of rules. We are in the process of adding to the agenda manager the capability to perform dynamic scheduling using meta-level heuristics about rule priorities.

Acknowledgements

The author would like to thank his advisor, Victor Lesser, for his significant contributions to the ideas presented in this paper and his support and encouragement during the course of this research. I would also like to thank Jim Schmolze for his advice and feedback on the parallel OPS5 system, Kelly Murray for developing Top Level Common Lisp, and Penelope Sibun and members of the CDPS lab for their comments on this paper.

References

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, 1989.

Decker, K.; Garvey, A.; Humphrey, M.;and Lesser, V. Effects of Parallelism on Blackboard System Scheduling, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, August, 1991.

Forgy, C.L., *On the Efficient Implementation of Production Systems*, PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, February, 1979.

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

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

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.

Ishida, Toru, Methods and Effectiveness of Parallel Rule Firing, *6th IEEE Conference on Artificial Intelligence Applications*, March 5-9, 1990.

Ishida, Toru, Makoto Yokoo, and Les Gasser, An Organizational Approach to Adaptive Production Systems, AAAI-90, pp. 52-57.

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

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

McDermott, J., and C. Forgy, Production System Conflict Resolution Strategies, in *Pattern-Directed Inference Systems*, D. A. Waterman and Frederick Hayes-Roth, eds., Academic Press, 1978.

McDermott, J., Extracting Knowledge from Expert Systems, IJCAI-83, pp. 100-107.

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.

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

Neiman, Daniel, Control in Parallel Production Systems: A Research Prospectus, COINS TR 91-2, Computer and Information Sciences Dept., University of Massachusetts, 1991.

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

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

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),26,321-332.