From: AAAI-90 Proceedings. Copyright ©1990, AAAI (www.aaai.org). All rights reserved.

An Organizational Approach to Adaptive Production Systems

Toru Ishida **Makoto Yokoo**
NTT Communications and
Information Processing Laboratories
1-2356, Take, Yokosuka-shi, 238-03, Japan
ishida/yokoo%nttkb.ntt.jp@relay.cs.net

Les Gasser
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
gasser@pollux.usc.edu

Abstract

Recently-developed techniques have improved the performance of production systems several times over. However, these techniques are not yet adequate for continuous problem solving in a dynamically changing environment. To achieve adaptive real-time performance in such environments, we use an *organization of distributed production system agents*, rather than a single monolithic production system, to solve problems. *Organization self-design* is performed to satisfy real-time constraints and to adapt to changing resource requirements. When overloaded, individual agents *decompose* themselves to increase parallelism, and when the load lightens the agents *compose* with each other to free hardware resources. In addition to increased performance, generalizations of our composition/decomposition approach provide several new directions for organization self-design, a pressing concern in Distributed AI.

Introduction

To improve the efficiency of production systems, high-speed matching algorithms, such as RETE [Forgy, 1982], TREAT [Miranker, 1987], and optimization algorithms [Ishida, 1988] have been investigated. Two kinds of parallel processing techniques have also been proposed: *parallel matching* [Stolfo, 1984; Gupta *et al.*, 1985; Acharya *et al.*, 1989] to speed-up matching processes and *parallel firing* [Ishida *et al.*, 1985; Ishida, 1990; Tenorio *et al.*, 1985; Moldvan, 1986] to reduce the total number of sequential production cycles. The motive for all of these studies is to speed up production systems several times over. However, these techniques are not yet adequate for continuous problem solving systems.

Typical examples can be found in *real-time expert systems*, where new techniques are required to adapt the systems to dynamically changing environments [Laffey *et al.*, 1988]. To satisfy real-time constraints, various *agent-centered approaches* are currently being studied. Lesser *et al.* [1988] discussed *approximate processing techniques.* Hayes-Roth *et al.* [1989] introduced *adaptive intelligent systems* that reason about and interact with other dynamic entities in real-time. These approaches attempt to meet deadlines by improving the decision-making of individual agents. In this paper we take an *organization-centered approach*, where problems are solved by a society of distributed problem-solving agents. This approach aims to achieve adaptive real-time performance through reorganization of the society. In addition to improving adaptability, our technique provides several insights and general mechanisms for organizational adaptation, a pressing concern in DAI [Gasser *et al.*, 1989a]. Moreover, it has the advantage of being grounded in a well-understood body of theory and practice: parallel production systems.

To explore the effectiveness of the organization-centered approach, we are studying the adaptive load balancing problem in which a particular problem solver shares a collection of processor resources with other problem solvers (and so has an opportunity for adapting its levels of resource use). Problem solving requests arrive at the organization continuously, at variable rates. Meaningful results are required within a (possibly changing) time limit. When the problem-solver is embedded in an open community of other problem solvers, it does not suffice to simply decompose to maximal parallelism – the collective must adapt itself to take advantage of resources as needed, *but it must also adaptively free up resources for others while continuing to operate.*

To achieve this goal, we first extended *parallel production systems*, where global control exists, into *distributed production systems*, with distributed control. We then introduced *organization self-design* (OSD) [Corkill, 1982; Durfee *et al.*, 1987; Gasser *et al.*, 1989a,b] into these distributed production systems. In previous research, reorganization mechanisms typically changed agent roles or inter-agent task ordering. In this paper, we added new reorganization primitives: *composition and decomposition of agents.* When problem solving requests arrive frequently, and make it difficult for the organization to meet its deadlines, agents autonomously *decompose* themselves so that parallelism increases. In contrast, when the organi-

zational load decreases, two agents *compose* (combine with each other) to save hardware resources. As a result, both real-time constraints and efficient resource utilization are satisfied through composition and decomposition of the agents.

Production Systems

To establish our terminology, we give a brief overview of production systems. A *production system* is defined by a set of *rules* or *productions* called *production memory (PM)*, together with an assertion database called *working memory (WM)* that contains a set of *working memory elements (WMEs)*. Each rule comprises a conjunction of *condition elements* called the *left-hand side (LHS)* of the rule, and a set of actions called the *right-hand side (RHS)*. *Positive condition elements* are satisfied when a matching WME exists, and *negative condition elements* are satisfied when no matching WME is found. An *instantiation* of the rule is a set of WMEs that satisfy the positive condition elements. The RHS specifies assertions to be added to or deleted from the WM[1].

A *data dependency graph of production systems* [Ishida *et al.*, 1985; Ishida, 1990] is constructed from the following four primitives:

A *production node*, which represents a set of instantiations. Production nodes are shown as circles in Figure 1 and 2.

A *working memory node*, which represents a set of WMEs. Working memory nodes are shown as squares in Figure 1 and 2.

A *directed edge from a production node to a working memory node*, which represents the fact that a production node modifies a working memory node. More precisely, the edge labeled '+' ('-') indicates that a WME in a working memory node is added (deleted) by firing an instantiation in a production node.

A *directed edge from a working memory node to a production node*, which represents the fact that a production node refers to a working memory node. More precisely, the edge labeled '+' ('-') indicates that a WME in a working memory node is referenced by positive (negative) condition elements when creating an instantiation in a production node.

Interference exists among rule instantiations when the result of parallel execution of the rules is different from the results of sequential executions applied in any order; it must be avoided by synchronization.

Figure 1 shows an example of OPS5 rules and their data dependency graph. In this example, if either `ruleA` or `ruleB` is fired first it destroys the other rule's

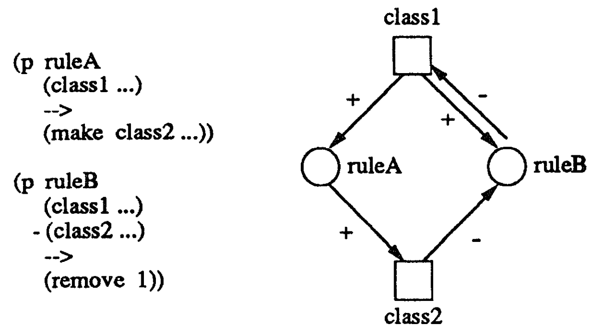


Figure 1: Data Dependency Graph

preconditions; therefore, interference may occur when firing both rules in parallel. If the two rules are distributed to different agents, the agents have to synchronize to prevent firing those rules in parallel.

Distributed Production Systems

Overview

A *distributed production system* is defined as a set of production system agents, each of which contains and fires some of the problem solving rules of the overall system. Each agent comprises the following three components:

A *problem solver*, which continuously repeats the *problem solving cycle* described later in this section. In parallel production systems, multiple rules are simultaneously fired but globally synchronized at the conflict resolution phase [Ishida *et al.*, 1985; Ishida, 1990]. In distributed production systems, on the other hand, rules are asynchronously fired by distributed agents. Since no global control exists, interference among the rules is prevented by local synchronization between individual agents.

Problem solving knowledge, contained in the PMs and WMs. For simplifying the following discussion, we assume no overlap between PMs in different agents, and assume the union of all PMs in the organization is sufficient to solve the given problem. Each agent's WM contains only WMEs that match the LHS of that agent's rules. Since the same condition elements can appear in different rules, the WMs in different agents may overlap. The union of WMs in an organization logically represents all the facts necessary to solve the given problem. In practice, since agents asynchronously fire rules, WMs can be temporarily inconsistent.

Organizational knowledge, representing relationships among agents. Each agent knows only about the others with whom it has data dependency or interference relationships (called its *neighbors*—see below). Since agents asynchronously perform reorganization,

[1]In this paper, we assume that each WME contains unique information. Operations adding duplicated WMEs are ignored.

organizational knowledge can be temporarily inconsistent across agents.

Organizational Knowledge

Organizational knowledge consists of the following three elements:

Dependencies:
Each agent knows which rules in the organization have data dependency relationships with its own rules. We say that **ruleA** *depends on* **ruleB** if **ruleA** refers to a working memory node that is changed by **ruleB**. We describe this as **depends(ruleA, ruleB)**. The data dependency knowledge of **agentP** is represented as:

$$DEPENDENCY_{agentP} = \{(\texttt{ruleA}, \texttt{ruleB}) \mid (\texttt{ruleA} \in PM_{agentP} \vee \texttt{ruleB} \in PM_{agentP}) \wedge \texttt{depends(ruleA, ruleB)}\}$$

Interference:
Each agent knows which rules in the organization may interfere with its own rules. Various interference analysis techniques are reported in [Ishida, 1990]. We describe the interference of **ruleA** and **ruleB** as **interfere(ruleA, ruleB)**. The interference knowledge of **agentP** is represented as:

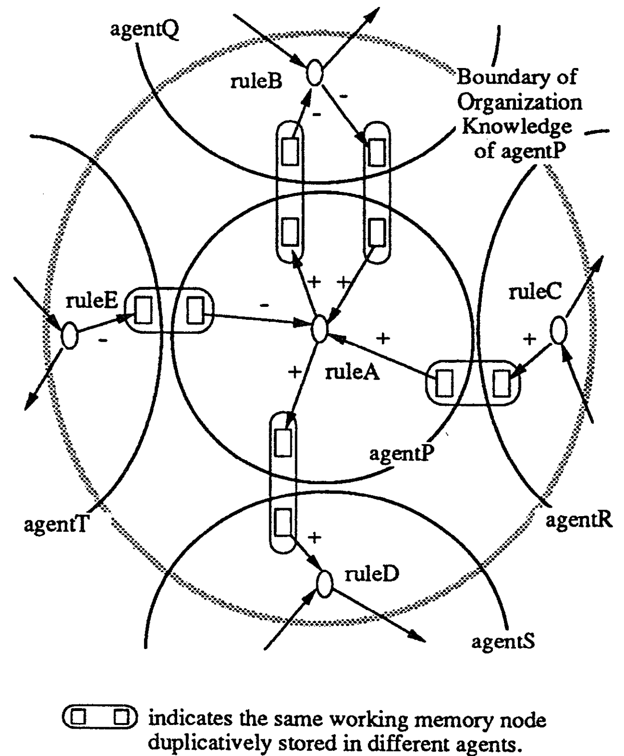
$$INTERFERENCE_{agentP} = \{(\texttt{ruleA}, \texttt{ruleB}) \mid (\texttt{ruleA} \in PM_{agentP} \vee \texttt{ruleB} \in PM_{agentP}) \wedge \texttt{interfere(ruleA, ruleB)}\}$$

Though an individual agent's execution cycle is sequential, potential interference among its own rules is analyzed for future distribution of those rules.

Locations:
Each agent, say **agentP**, knows the location of rules, say **ruleA**, appearing in its own data dependency and interference knowledge. We describe this as **appears(ruleA, agentP)**. The neighbor knowledge of **agentP** is represented as:

$$LOCATION_{agentP} = \{(\texttt{ruleA}, \texttt{agentQ}) \mid \texttt{appears(ruleA, agentP)} \wedge \texttt{ruleA} \in PM_{agentQ}\}$$

Figure 2 illustrates the organizational knowledge of **agentP**. For example, since **ruleA** and **ruleB** interfere with each other, **agentP** has to synchronize with **agentQ** when executing **ruleA**. Also, **ruleA**'s WM modification has to be transferred to **agentS**. We call **agentQ** a *neighbor* of **agentP** when **agentQ** has data dependency or interference relationships with **agentP**. From this definition, as illustrated in Figure 2, **agentP**'s organizational knowledge refers only to its *neighbors*.



$DEPENDENCY_{agentP}$ = {(ruleA, ruleC) (ruleD, ruleA) (ruleA, ruleE)}
$INTERFERENCE_{agentp}$ = {(ruleA, ruleB)}
$LOCATION_{agentP}$ = {(ruleA, agentP) (ruleB, agentQ) (ruleC, agentR) (ruleD, agentS) (ruleE, agentT)}

Figure 2: Organizational Knowledge

Problem Solving Cycle

We define a *problem solving cycle of distributed production system agents* by extending the conventional *Match-Select-Act cycle* to accommodate inter-agent data transfers and synchronization. Temporary inter-agent inconsistency caused by distribution is handled locally using temporary synchronization via *rule deactivation*. (We assume the preservation of message ordering.) The cycle is:

1. *Process messages:*
   When receiving a *synchronization request message* (e.g., **deactivate(ruleA)**), return an *acknowledgment message* and deactivate the corresponding rule (**ruleA**) until receiving a *synchronization release message* (**activate (ruleA)**). When receiving a *WM modification message*, update the local WM to reflect the change made in the other agent's WM.

2. *Match:*
For each rule, determine whether the LHS matches the current WM.
3. *Select:*
Choose one instantiation of a rule (e.g., `ruleB`) that is not deactivated.
4. *Request synchronization:*
Using interference knowledge, send synchronization request messages (`deactivate(ruleB)`) to the agents requiring synchronization. Await acknowledgment from all synchronized agents[2]. After complete acknowledgment, handle all WM modification messages that have arrived during synchronization. If the selected instantiation is thereby canceled, send synchronization release messages and restart the problem solving cycle.
5. *Act:*
Fire the selected rule instantiation (`ruleB`). Using the data dependency knowledge of `agentP`, inform dependent agents with WM modification messages.
6. *Release synchronization:*
Send synchronization release messages (`activate (ruleB)`) to all synchronized agents.

Organization Self-Design (OSD)

Reorganization Requests

To start reorganization, two kinds of *reorganization requests* are sent to all agents of the organization. *Decomposition requests* are issued when the organization cannot meet deadlines. *Composition requests* are issued to release resources when the organization-wide load is light. For these purposes, the behavior of the organization must be continuously observed.

Decomposition requests initiate division in heavily-loaded agents. Decomposition continues until parallelism increases, response times are shortened, and decomposition requests disappear. Conversely, composition requests initiate combining each two lightly-loaded agents into one. Composition continues until the organization's load increases and composition requests disappear. Both kinds of requests can be issued simultaneously.

Reorganization Process

To control the reorganization processes, we added to each agent an *organization self-designer*, which performs reorganization at the end of each problem solving cycle. We describe below how one agent (e.g., `agentP`) decomposes itself into two agents(e.g., `agentP` and `agentQ`). During reorganization, rules, WMEs, dependency and interference knowledge are transferred from `agentP` to `agentQ`, but not modified. Location knowledge is modified and changes are propagated to other agents.

1. *Create a new agent:*
`agentP` creates a new agent, `agentQ`, which immediately starts problem solving cycles.
2. *Select rules to be transferred:*
`agentP` selects *active* rules to be transferred (e.g., `ruleA`) to `agentQ`. `agentP` sends synchronization request messages (`deactivate(ruleA)`) to `agentQ`. Currently, half of the active rules are arbitrarily selected and transferred, but we are refining a theory of rule selection based on maximizing intra-agent rule dependencies and minimizing inter-agent communication.
3. *Request synchronization:*
`agentP` sends synchronization request messages to *neighbors* for all rules that have data dependency or interference relationships with rules to be transferred (e.g., `deactivate(ruleB)` is sent if `depends(ruleA, ruleB)`, `depends(ruleB, ruleA)` or `interfere(ruleA, ruleB)`)[3]. `agentP` waits for complete acknowledgment (resolving deadlock as before).
4. *Transfer rules:*
`agentP` transfers rules (`ruleA`) to `agentQ`, updates its own location knowledge, and propagates any changes to its *neighbors*.
5. *Transfer WMEs:*
`agentP` copies WMEs that match the LHS of the transferred rules (`ruleA`) to `agentQ`[4]. A bookkeeping process follows in both agents to eliminate duplicated or unneeded WMEs.
6. *Transfer dependency and interference knowledge:*
`agentP` copies its dependency and interference knowledge to `agentQ`. Both agents do bookkeeping to eliminate duplicated or unneeded organizational knowledge[5].
7. *Release synchronization:*
`agentP` sends synchronization release messages (`activate(ruleA)` to `agentQ` and `activate(ruleB)` to all synchronized *neighbors*). This ends reorganization.

[2] Deadlock is a possibility. When acknowledge messages are not received, synchronization release messages are sent and the problem solving cycle is restarted.

[3] This is to assure that WM modification and synchronization request messages related to rules to be transferred are not sent to agentP during the reorganization process.

[4] More precisely, to avoid reproducing once-fired instantiations, not only WMEs but also conflict sets are transferred to agentQ. Before transferring the conflict sets, however, agentP has to maintain its WM by handling the WM modification messages that have arrived during the synchronization process.

[5] Unneeded data dependency and interference knowledge are the tuples that include none of the agents' rules. Unneeded location knowledge is the tuples that include none of the rules that appear in the agents' data dependency and interference knowledge.

An agent (e.g., **agentP**) can compose with another agent by a similar process. First, **agentP** sends *composition request messages* to its *neighbors.* If some agent, say **agentQ**, acknowledges, **agentP** transfers all rules and organizational knowledge to **agentQ** and destroys itself. The transfer method is the same as that for decomposition.

During the reorganization process, neighboring agents deactivate rules that have data dependency or interference relationships with transferred rules. However, neighboring agents can concurrently perform other activities including firing and transferring rules that are not deactivated. This localization helps agents to modify the organization incrementally.

Experimental Evaluation

To evaluate the effectiveness of our approach, we implemented a simulation environment and solved the *Waltz labeling problem*: 36 rules solve the problem that appears in Figure 3-17 in [Winston, 1977] with 80 rule firings.

At initiation, only one agent, with all problem-solving and organizational knowledge, exists in the organization. We assume the organization knowledge for the initial agent is prepared by analyzing its problem solving knowledge before execution. Problem-solving requests continuously arrive at the agent; older pending requests are processed with higher priority. The load of each agent is represented by a *firing ratio*: the ratio of the number of rule firings to the number of problem solving cycles. Reorganization is performed as follows. (Global parameters are adjustable.)

When the organization cannot solve a problem within a predefined time limit, say 20 problem solving cycles, decomposition requests are sent to the organization. We use experimentally-generated firing ratio thresholds to trigger reorganization. Agents whose firing ratio is greater than 80% start decomposing. Upon decomposition, rules are arbitrary divided and distributed between two agents. When the organization-wide ratio is less than 60%, composition requests are sent to the organization. Agents whose firing ratio is less than 30% compose with each other. These thresholds were experimentally found to provide a good balance between adaptiveness and sensitivity, but further study is warranted.

Figure 3 shows the simulation results. The line chart indicates response times normalized by problem solving cycles, and the step chart represents the number of agents in the organization. In Figure 3(a), problem solving requests arrive at constant intervals. In Figure 3(b), the frequency of requests is changed periodically. From these figures, we can conclude the following:

Adaptiveness of the organization:

In Figure 3(a), the organization reaches a stable state. Since several composition and decomposition cycles are performed, the firing ratios of the resulting agents are equalized. In Figure 3(b), we can see

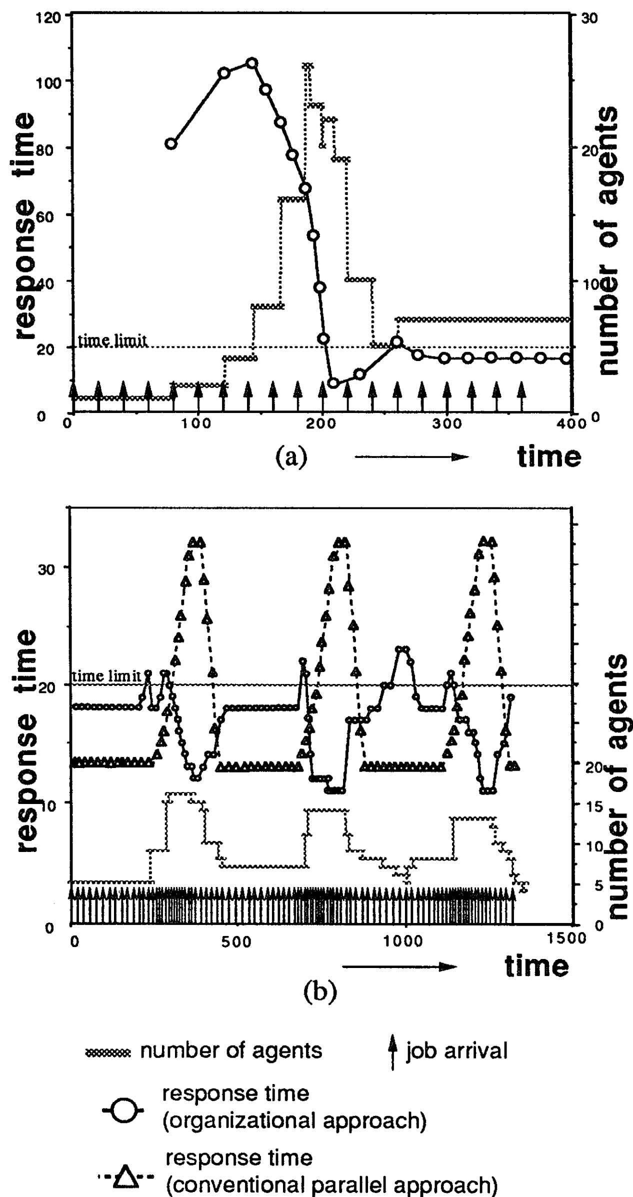


Figure 3: Simulation Results

the number of agents at the busiest peak slightly decreases over time. Both charts show that the society of agents has gradually adapted to the situation through repeated reorganization.

Real-time problem solving:
The average number of agents in Figure 3(b) is 8.95. We compared the organization response times to the performance of 9 permanent agents with no self-design (which is the conventional parallel production system approach, shown as the dashed line in Figure 3(b)). Differences between the dashed and solid lines demonstrate how the organizational approach is effective for adaptive real-time problem solving. However, the effect of reorganization lags the change in problem load. To improve the capability to meet deadlines, time limits must be set shorter than actual deadlines, and load increases must be detected as early as possible.

Efficient resource utilization:
In Figure 3(b), the number of required agents varies from 4 to 17. It is obvious that the organizational approach is more economical than the conventional parallel approach that permanently reserves 17 agents. The resource saving effect of the organizational approach is also supported by the fact that 9 permanent agents (which require almost the same processing resources as the organizational approach) cannot meet deadlines.

Conclusion

Techniques for building problem-solving systems that can adapt to changing problem solving requests and deadlines are of great interest. This paper has presented an approach that relies on reorganization of a collection of problem-solvers to track changes in deadlines and problem solving requests. It exploits an adaptive trade-off of parallelism for time by making new agents and continually reallocating problem-solving knowledge. The importance of this approach goes beyond the adaptive performance we have illustrated.

With additional decision-making meta-knowledge, this approach can become a more general organization self-design technique. It also has the advantage of being grounded in a well-understood body of theory and practice: parallel production systems. In the current version, composition/decomposition decisions are made solely on the basis of firing ratios, and the choice of rules to transfer is made arbitrarily. Allocation decisions could instead be based on the semantics of rules (i.e., distribution based on the kinds of tasks that need more resources). Partial knowledge transfer among *existing* agents can be combined with composition and decomposition to provide a flexible and distributed task-sharing system.

Within our existing formulation, avenues for future research include the implementation and the evaluation of this approach on actual message passing multiprocessor systems, evaluating the impact of reorganization overheads, finer threshold sensitivity analyses, techniques for incrementally acquiring reorganization strategy in more dynamic contexts, and applying this approach to adaptively overcoming local faults and inconsistency among agents.

Acknowledgments

The basic ideas in this paper were elaborated during Les Gasser's visit at NTT Communications and Information Processing Laboratories. The authors wish to thank Kunio Murakami and Ryohei Nakano for their support to our joint research project, and Nick Rouquette for helpful comments.

References

[Acharya *et al.*, 1989] A. Acharya and M. Tambe, "Production Systems on Message Passing Computers: Simulation Results and Analysis," *International Conference on Parallel Processing*, pp. 246-254, 1989.

[Corkill, 1982] D.D. Corkill, *A Framework for Organizational Self-Design in Distributed Problem Solving Networks*, PhD Dissertation, COINS-TR-82-33, University of Massachusetts, 1982.

[Durfee *et al.*, 1987] E. H. Durfee and V. R. Lesser, "Using Partial Global Plans to Coordinated Distributed Problem Solvers," *IJCAI-87*, pp. 875-883, 1987.

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

[Gasser *et al.*, 1989a] L. Gasser and M. N. Huhns, "Themes in Distributed AI Research," in L. Gasser and M. N. Huhns, Editors, *Distributed Artificial Intelligence*, Volume II, London:Pitman, 1989.

[Gasser *et al.*, 1989b] L. Gasser, N. Rouquette, R. Hill and J. Lieb, "Representing and Using Organizational Knowledge in DAI Systems," in L. Gasser and M. N. Huhns, Editors, *Distributed Artificial Intelligence*, Volume II, London:Pitman, pp. 55-78, 1989.

[Gupta *et al.*, 1988] A. Gupta, C. L. Forgy, D. Kalp, A. Newell and M. Tambe, "Parallel OPS5 on the Encore Multimax," *International Conference on Parallel Processing*, pp. 271-280, 1988.

[Hayes-Roth *et al.*, 1989] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett and A. Seiver, "Intelligent Monitoring and Control," *IJCAI-89*, pp. 243-249, 1989.

[Ishida *et al.*, 1985] T. Ishida and S. J. Stolfo, "Towards Parallel Execution of Rules in Production

System Programs," *International Conference on Parallel Processing*, pp. 568-575, 1985.

[Ishida, 1988] T. Ishida, "Optimizing Rules in Production System Programs," *AAAI-88*, pp. 699-704, 1988.

[Ishida, 1990] T. Ishida, "Methods and Effectiveness of Parallel Rule Firing," *IEEE Conference on Artificial Intelligence Applications*, pp. 116-122, 1990.

[Laffey *et al.*, 1988] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, "Real-Time Knowledge-Based Systems," *AI Magazine*, Vol. 9, No. 1, pp. 27-45, 1988.

[Lesser *et al.*, 1988] V. R. Lesser, J. Pavlin and E. H. Durfee, "Approximate Processing in Real Time Problem Solving," *AI Magazine*, Vol. 9, No. 1, pp. 49-61, 1988.

[Miranker, 1987] D. P. Miranker, "TREAT: A Better Match Algorithm for AI Production Systems," *AAAI-87*, pp. 42-47, 1987.

[Moldovan, 1986] D. I. Moldovan, "A Model for Parallel Processing of Production Systems," *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 568-573, 1986.

[Stolfo, 1984] S. J. Stolfo, "Five Parallel Algorithms for Production System Execution on the DADO Machine," *AAAI-84*, pp. 300-307, 1984.

[Tenorio *et al.*, 1985] F. M. Tenorio and D. I. Moldovan, "Mapping Production Systems into Multiprocessors," *International Conference on Parallel Processing*, pp. 56-62, 1985.

[Winston, 1977] P. H. Winston, *Artificial Intelligence*, Addison-Wesley, 1977.