Organization Self-Design of Distributed Production Systems

Toru Ishida, Les Gasser, and Makoto Yokoo

***Abstract*—*Organization* has emerged as a key concept for structuring the activities of collections of problem-solvers. *Organization self-design (OSD)* has been studied as an adaptive approach to long term, strategic work-allocation and load-balancing. In this paper, we introduce two new reorganization primitives, *composition and decomposition*. They change the population of agents and the distribution of knowledge in an organization. To create these primitives, we formalize *organizational knowledge*, which represents knowledge of potential and necessary interactions among agents in an organization. We develop computational OSD techniques for agents with architectures based on production systems to take advantage of the well-understood body of theory and practice.**

We first extend *parallel production systems*, where global control exists, into *distributed production systems*, where problems are solved by a society of agents using distributed control. We then introduce OSD into distributed production systems to provide adaptive work allocation. Simulation results demonstrate the effectiveness of our approach in adapting to changing environmental demands. In addition to introducing advanced techniques for flexible OSD, our approach impacts production system design, and improves our ability to build production systems that can adapt to changing real-time constraints.

***Index Terms*—Adaptive problem solving, organization self-design, parallel and distributed processing, production system, real-time problem solving.**

I. Introduction

It has been clear for some time that organization is a powerful concept for thinking about how to structure the interactions of collections of problem solvers. Understanding the concept of organization and developing techniques for adaptive reorganization are pressing concerns in distributed artificial intelligence (DAI) [3]. Several conceptual approaches to organization have been introduced, including treating organization as 1) a long term, strategic load-balancing technique [5], 2) a structural set of control and communication relationships among agents [28], 3) sets of interaction patterns among agents [19], [12], 4) sets of commitments and expectations among agents, [3], [14], [20], or 5) collections of settled and unsettled questions about knowledge and action [13].

The comparative information processing performance of rigid organization structures was studied by Malone [28]. However, since no single organization is appropriate in all situations, organization self-design (OSD) has been proposed to allow an organization of problem solvers to adapt itself to dynamically changing situations [5]. In this paper, we further explore the process of OSD, and, in doing so, we examine some new ideas about the nature and representation of organizations which are the foundations of OSD. We address OSD by introducing the following new concepts:

- *Organizational knowledge:* To perform either domain problem solving or reorganization, agents need *organizational knowledge*, which represents both the necessary interactions among agents and their organization. However, the kind of organizational knowledge required for reorganization has not been thoroughly investigated in prior research. In this paper, we formalize organizational knowledge as a collection of *agent–agent relationships* and *agent–organization relationships*, which represent how agents' local decisions affect both other agents' decisions and the behavior of the entire organization.
- *Reorganization primitives:* In previous research, reorganization mechanisms typically changed agent roles or inter-agent task ordering [5], [7] [9]. In this paper, however, we take the approach of formalizing reorganization primitives, which can perform OSD through repeated application. The new reorganization primitives, *composition and decomposition of agents*, dynarnically change inter-agent relationships, the knowledge agents have about one another, the size of the agent population, and the resources allocated to each agent.

Up to now, OSD has been investigated using comparatively complex agents, such as blackboard-based agents. However, here we discuss OSD using a problem-solving model based on production systems, to take advantage of a well-understood body of theory and practice, while retaining general applicability. Production systems have the advantage of providing a formal characterization of both the knowledge needed to solve a problem and the ways in which parts of that knowledge interact. In addition, production rules can be used as general abstractions of organizational and problem-solving processes of many kinds. (For example, Zisman has provided a well-known application of production systems to modeling asynchronous organizational work and problem-solving [34].) Though we use production systems here as a theoretical and modeling foundation, our concepts of OSD and organizational knowledge can be generalized to apply to other problem-solving models and other types of problem solvers.

In addition to advancing OSD techniques, our approach impacts production system design. Previous research, attempted

Manuscript received July 1, 1991. This paper is the extended version of the authors' previous AAAI conference papers [25], [15].

T. Ishida and M. Yokoo are with NTT Communication Science Laboratories, Sanpeidani, Inuidani, Seika-cho, Soraku-gun, Kyoto, 619-02, Japan.

L. Gasser is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 9106253.

to improve the efficiency of production systems by investigating high-speed matching algorithms, such as RETE [10] and TREAT [29]. Two kinds of parallel approaches have also been studied: *parallel matching* [1], [16] to speedup matching processes and *parallel firing* [22], [31], [23], [24] to reduce the total number of sequential production cycles. However, the motive for all of these studies is to speed up production systems several times over, and not necessarily to make them more adaptive or reactive, e.g., to follow changing environmental demands or resource constraints. Thus, the published techniques are not yet fully adequate for real-time expert systems [26]. The OSD approach, proposed in this paper, can complement other approaches currently being developed for real-time expert systems, such as approximate processing techniques [27] and adaptive intelligent systems [18]. These approaches attempt to meet deadlines by improving the decision-making of individual agents. On the other hand, the OSD approach, where problems are solved by a society of distributed problem-solving agents, aims to achieve adaptive real-time performance through the reorganization of the society. Various simulation results show the effectiveness of our OSD approach for building adaptive real-time systems with production system architectures.

II. Overview of Approach

We begin our approach to OSD with a general problem-solving model based on parallel production systems, in which global control exists. Next, we extend this into distributed production systems with distributed control. Finally, we introduce OSD primitives and an OSD architecture into distributed production systems, and test their performance.

We are interested in OSD for *problem-solving organizations*, whose products are solutions to *individual problem-solving requests* that are issued from the organization's *environment*. Several types of change in the relationship between a problem solving organization and its environment can create pressure for reorganization. These include: 1) demands for change in the organizational performance level (e.g., shorter or longer response time requirements or new quality levels), 2) change in the level of demand for certain solution types (e.g., more or fewer problem-solving requests per unit time, or changes in the mix of problem types), and 3) changes in the level of demand for resources that the organization shares with others in its environment.

No single organization can adequately handle all problems and environmental conditions. For example, suppose there are three agents in an organization, each of which fires one production rule for solving each problem request, the three agents work in a pipelined fashion (because their rules are sequentially dependent), and the communication delay among agents is equal to one production cycle. Thus, the total throughput cycle time for satisfying a single request is 5. In this case, however, a single agent organization would perform better—it would incur no communication overhead, and would take only 3 cycles for satisfying a single request. On the other hand, if there were ten problem-solving requests, the response time of the last request would be 14 cycles in the three agent organization, while it would be 30 in the single agent case.

In our model, problem-solving requests issued from the environment arrive at the organization continuously, and at variable rates. To respond, the organization must supply meaningful results within specified time limits, which are also set by the environment and which also may vary. These variations are changing conditions to which the organization must adapt using organizational knowledge and OSD primitives.

Fig. 1 describes the process of OSD. Composition and decomposition are repeatedly performed as follows [25]:

- *Decomposition* divides one agent into two. Decomposition is performed when environment demands too much from the organization (e.g., high arrival rates of problem-solving requests), such that the organization finds it difficult to meet its response requirements with its available resources. More precisely there are two cases. In the first case, agents decompose to increase *intra*-problem parallelism. This happens when the structure of the problem-solving rules being applied contains concurrency, and agents cannot meet deadlines because of the complexity of the given problem. In the other case, agents decompose to increase *inter*-problem parallelism. Even if there is no possible concurrency among rules, decomposition can increase the organizational throughput when multiple problem requests can be processed in a pipelined fashion.
- *Composition* combines two agents into one. As with decomposition, two cases exist. In the first case, the organization is embedded in an open community with other organizations, and it must save community-wide computing resources for cost-effective problem solving. In this case, it is not sufficient just to continuously utilize the maximal available parallelism—the collective must also adaptively *free up* computing resources for use by others, and it can do this through composition. In the other case, agents compose to reduce response times. This need arises when communication overhead cannot be ignored. Because of the inter-agent communication overhead, maximal decomposition does not necessarily yield either minimal response time or maximal organizational throughput. Composition may actually reduce response time, even though parallelism decreases, where coordination overhead (i.e., communication and synchronization) is high.

Both composition and decomposition force reorganize actions by modifying the distribution of problem-solving and organizational knowledge in the organization, and by modifying the particular association between resources and problem-solving knowledge. In general, decomposition increases the overall level of resources used, while composition decreases resource use. Composition and decomposition can occur concurrently in different parts of the organization. The relative balance of composition and decomposition activities during any period is a result of the interaction between a set of reorganization rules that govern reorganization, and the conditions in the organization and in the environment during that period.

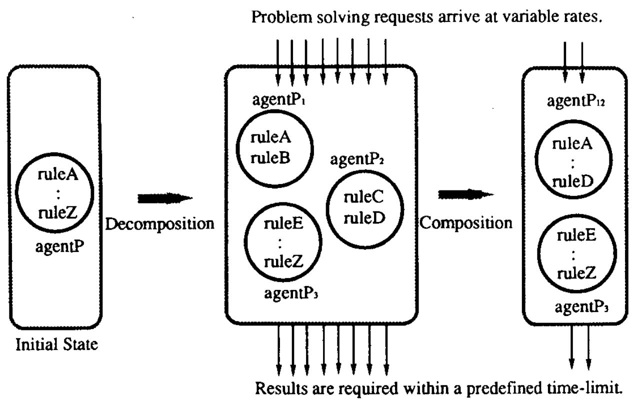


Fig. 1. Composition and decomposition.

III. Production Systems

To establish our terminology, we must 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 (WME's). Each rule comprises a conjunction of condition elements called the left-hand side (LRS) 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 WME's that satisfy the positive condition elements. The RHS specifies assertions to be added to or deleted from the WM.[1] The production system interpreter repeatedly executes the following cycle of operations:

1) *Match:* For each rule, determine whether the LHS matches the current environment of the WM.
2) *Select:* Choose exactly one of the *matching instantiations* of the rules according to some predefined criterion. This is called a *conflict resolution strategy.*
3) *Act:* Add to or delete from the WM all assertions as specified by the RHS of the selected rule.

A *data dependency graph for production systems* [22]–[24] is constructed from the following four primitives:

- A *production node*, which represents a set of instantiations. Production nodes are shown as circles in figures.
- A *working memory node*, which represents a set of WME's. Working memory nodes are shown as squares in figures.
- 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 specifically, the edge labeled "+" ("−") indicates that a WME in a working memory node is added (deleted) by firing an instantiation in a production node.

[1] In this paper, we assume that each WME contains unique information. Operations adding duplicated WME's are ignored. Several commercial production systems take this approach [4].

```
(p ruleA
   (class1 ...)
   -->
   (make class2 ...))

(p ruleB
   (class1 ...)
   - (class2 ...)
   -->
   (remove 1))
```

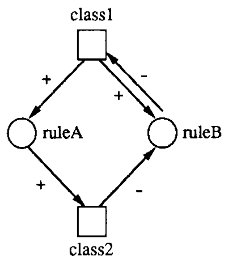


Fig. 2. Data dependency graph.

- 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. Interference must be avoided by synchronization. Various methods for detecting interference are reported in [23] and [24]. In this paper, we utilize compile-time analysis because run-time analysis is too expensive in multiagent situations. In compile time analysis, interference can be identified when multiple rules destroy other rules' preconditions in a cyclic fashion.

Fig. 2 shows an example of OPS5 rules and their data dependency graph. In Fig. 2, if either `ruleA` or `ruleB` is fired first it destroys the other rule's 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 the rules from being fired in parallel and thus maintain consistency.

IV. Distributed Production Systems

A. Architecture

Fig. 3 illustrates three types of agent. A *production system agent*, illustrated in Fig. 3(a), consists of a *production system*

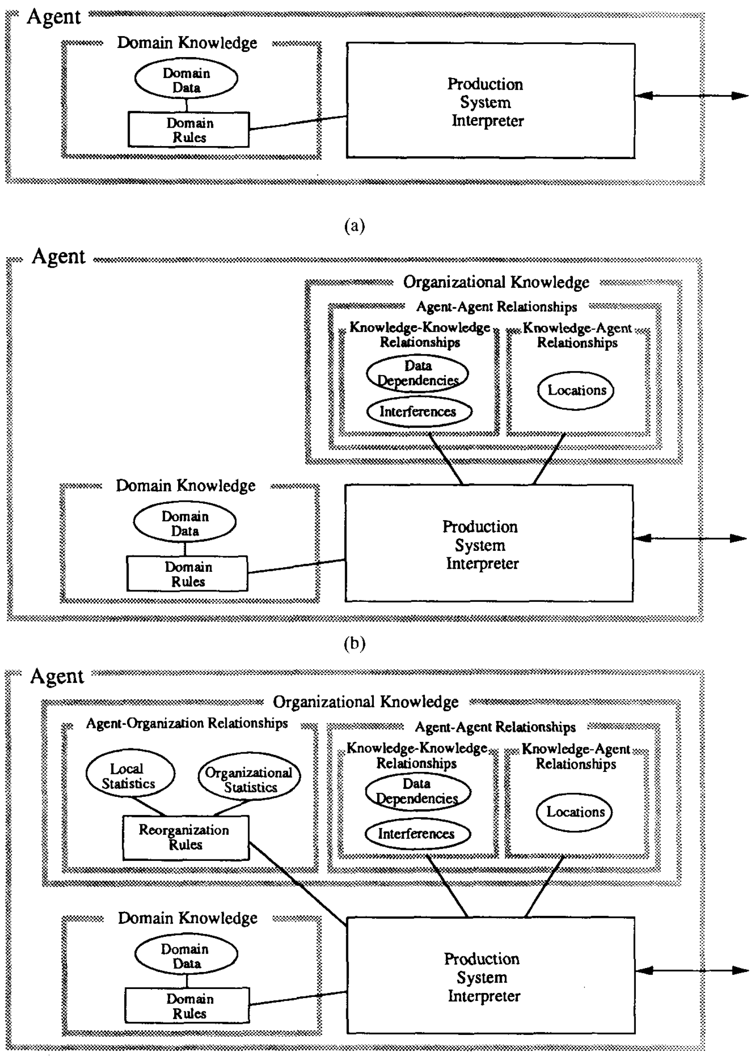

Agent
Domain Knowledge
Domain Data
Domain Rules
Production System Interpreter
(a)
Agent
Organizational Knowledge
Agent-Agent Relationships
Knowledge-Knowledge Relationships
Data Dependencies
Interferences
Knowledge-Agent Relationships
Locations
Domain Knowledge
Domain Data
Domain Rules
Production System Interpreter
(b)
Agent
Organizational Knowledge
Agent-Organization Relationships
Local Statistics
Organizational Statistics
Reorganization Rules
Agent-Agent Relationships
Knowledge-Knowledge Relationships
Data Dependencies
Interferences
Knowledge-Agent Relationships
Locations
Domain Knowledge
Domain Data
Domain Rules
Production System Interpreter
(c)

Fig. 3. Agent architecture. (a) Production system agent. (b) Distributed production system agent. (c) Self-organizable distributed production system agent.

interpreter and *domain knowledge*, in which the PM represents domain rules and the WM represents *domain data*.

Fig. 3(b) represents the architecture of distributed production system agents, each of which contains a part of the domain knowledge. Such agents must communicate with other agents for data transfer and for synchronization. Thus, each agent requires organizational knowledge, which represents *agent–agent relationships*. A distributed production system agent comprises the following three components:

- A *production system interpreter* that continuously repeats the *production cycle*, described in Section IV-C. In a parallel production system, multiple rules are simultaneously fired but globally synchronized at the Select phase [22]–[24]. In a collection of distributed production system agents, 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.
- *Domain knowledge* is contained in the PM, which represents *domain rules*, and WM, which represents *domain data*. To simplify the following discussion, we assume no overlap between PM's in different agents, and assume that the union of all PM's in the organization is sufficient to solve the given problem. Each agent's WM contains only WME's that match the LHS of that agent's rules. Since the same condition elements can appear in different rules, the WM's in different agents may overlap. The union of WM's in an organization logically represents all data necessary to solve the given problem. In practice, since agents asynchronously fire rules, WM's can be temporarily inconsistent.
- *Organizational knowledge* represents relationships among agents, and we call these *agent–agent relationships*.

Agents that have such relationships with a particular agent are called that agent's neighbors. Agent–agent relationships are initially obtained by analyzing domain knowledge at compile time, and are dynamically maintained during the process of OSD. Since agents asynchronously perform reorganization, organizational knowledge can be temporarily inconsistent across agents. Taken together, these relationships structure the actions of each agent at any moment; they provide a decentralized knowledge-based specification of the instantaneous organizational form.

Self-organizable distributed production system agents are discussed in Section V.

B. Organizational Knowledge: Agent–Agent Relationships

Agent–agent relationships can be seen as the aggregation of two more primitive types of relationships: *knowledge–knowledge relationships*, which represent interactions within domain knowledge, and *knowledge–agent relationships*, which represent how domain knowledge is distributed among agents. *Knowledge–knowledge* relationships consist of data dependencies and interferences among domain rules as follows:

- *Data dependencies:* Each agent knows which domain 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, rule B )`. The data dependency knowledge of `agentP` is represented as follows:

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

- *Interferences:* Each agent knows which rules in the organization may *interfere with* its own rules. We describe the interference of `ruleA` and `ruleB` as `interfere(ruleA,ruleB )`. The interference knowledge of `agentP` is represented as follows:

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

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

On the other hand, knowledge-agent relationships are represented by the locations of domain rules:

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

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

Fig. 4 illustrates the organizational knowledge of `agentP`. Large solid circles indicate the boundaries of individual agents. Long, narrow ovals that connect agents indicate interaction paths among agents; the two rectangles within each oval indicate the WME's communicated between agents via that interaction path, and duplicated in both agents. "+"and "-"indicate data dependencies as described in Section III-B.

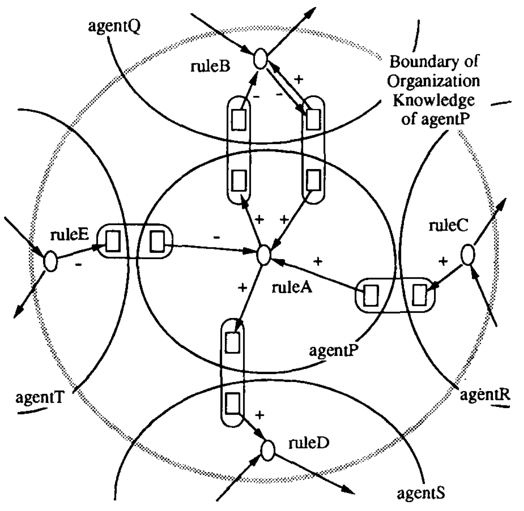
In the example in Fig. 4, since `ruleA` and `ruleB` interfere with each other, `agentP` has to synchronize with `agentQ` when executing `ruleA`. Also, `ruleA`'s WM modification has to be transferred to agents. We call `agentQ`, `agentR`, `agentS`, and `agentT` *neighbors* of `agentP` because they have agent–agent relationships with `agentP`. From this definition, as illustrated in Fig. 4, `agentP`'s organizational knowledge refers only to its *neighbors*.

C. Production Cycle

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

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

To avoid deadlock, we prioritize interfering rule pairs at compile time. This idea is borrowed from [32]. Let `ruleA` and `ruleB` interfere with each other, and let `ruleB` be given a higher priority. Then, `ruleB` can be fired without



(□ □) indicates the same working memory node duplicatively stored in different agents.

DEPENDENCY$_{agentP}$ = {(ruleA, ruleB) (ruleB, ruleA) (ruleA, ruleC) (ruleD, ruleA) (ruleA, ruleE)}
INTERFERENCE$_{agentP}$ = {(ruleA, ruleB)}
LOCATION$_{agentP}$ = {(ruleA, agentP) (ruleB, agentQ) (ruleC, agentR) (ruleD, agentS) (ruleE, agentT)}

Fig. 4. Organizational knowledge.

synchronization as long as it is not deactivated. However, when firing `ruleA`, `ruleB` has to be deactivated through synchronization. This approach can avoid interference through one-directional synchronization, and thus can reduce half of the synchronization overhead. Deadlock may still occur when agents are prioritized in a cyclic fashion, i.e., `ruleA` requires `ruleB` be deactivated, `ruleB` requires `ruleC` be deactivated, and `ruleC` requires `ruleA` be deactivated. However, since interference is analyzed at compile time, we can easily prioritize rules such that loops are not created. Thus, this approach can avoid deadlocks among distributed production system agents.

V. Organization Self-Design (OSD)

A. Architecture

Fig. 3(c) represents the agent architecture for self-organizable distributed production system agents. OSD is performed in the following way: Upon initiation, only one agent, containing all domain and organizational knowledge, exists in the organization. We assume organizational knowledge for the initial agent is prepared by analyzing its domain knowledge before execution. Problem-solving requests continuously arrive at the agent; older pending requests are processed with higher priority.

For effective reorganization, agents should invoke the reorganization primitives appropriate for each situation. For this purpose, we extend the organizational knowledge (in Section IV-B) to include *agent–organization relationships*, which represent how agents' local decisions affect organizational behavior or, in other words, how well the organization is meeting its response goals. However, since multiple agents asynchronously fire rules and perform reorganization, knowing the exact status of the entire organization is difficult. Under the policy of obtaining better decisions with maximal locality, we first introduce *local* and *organizational statistics*, which can be easily obtained, and then define *reorganization rules* using those statistics to select an appropriate reorganization primitive when necessary. Since the reorganization rules are also production rules, OSD and domain problem solving are arbitrarily interleaved. In our implementation, however, we assume higher priority is given to the reorganization rules during the Select phase of the production cycle. This mechanism is analogous to the integration of control and domain knowledge source activations in systems such as BB1 [17], or to integrated metalevel reasoning in DVMT [8].

Decomposition is triggered when the environmental conditions (problem-solving demand on the organization and required response-time) exceed the organization's ability to respond, given its current form and resource level. Excessive demand at the organization level is translated into excessive local demand in particular regions of the organization, measured using the local organizational statistics. At this point, particular agents with excessive local demand are divided into multiple agents, and additional computational resources are assigned to them. Decomposition continues until parallelism increases and response improves. Composition is performed when under-utilized resources can be released for use by other organizations, or to improve local performance by reducing coordination overhead. When two agents, taken together, contain an oversupply of resources, they are combined into one agent via composition. Composition repeats until no more composition is possible under the conditions of meeting deadlines. Since the aims of composition and decomposition are independent, both kinds of reorganization can be performed simultaneously in different parts of the organization. In this way, both problem-solving and organization self-design are treated as decentralized processes.

B. Organizational Knowledge: Agent–Organization Relationships

Agent–organization relationships consist of local statistics, organizational statistics, and reorganization rules:

- *Local statistics:* We introduce *firing ratio* to represent the level of activity of each agent. Let P be a predefined period (normalized by production cycles) for measuring statistics, and F be the number of rule firings during P. Then the firing ratio R can be represented by F/P. When $R = 1.0$ (i.e., there are no idle production cycles over the measurement interval P), agents are called *busy*, while when $R < 1.0$, agents can be assigned additional tasks. To avoid the need for frequent communication among agents, however, we do not assume that agents need to know other agents' local statistics.

- *Organizational statistics:* We assume each agent can know by periodically-broadcast messages whether the organization is currently meeting deadlines. Let $T_{response}$ be the most recently observed response time (that is, time taken to complete the most recent task), and $T_{deadline}$ be the predefined time limit of the task. When $T_{response} > T_{deadline}$, the performance of the organization should be improved, while when $T_{response} < T_{deadline}$, the organization can release resources.
- *Reorganization rules:* By using local and organizational statistics, the following rules are provided for each agent to initiate reorganization. These rules are tested during the production cycle.

R1) Perform decomposition if
$T_{deadline} < T_{response}$ and
$R = 1.0$

R2) Perform composition if
$T_{deadline} > T_{response}$ and
$2R < T_{deadline}/T_{response}$

R3) Perform composition if
$R < 0.5$.

R1 initiates busy agents to perform decomposition, when the organization cannot meet its deadline. R2 initiates agents to perform composition, when the organization can keep its deadline. Composition is performed even if agents are fully busy, when $T_{response}$ is enough lower than $T_{deadline}$.

R3 is introduced to take account of communication overhead. Suppose problem solving requests initially arrive frequently, and subsequently decrease. Initially, R1 is repeatedly applied, maximizing the parallelism to increase organizational throughput. Later, even though the frequency of requests decreases, R2 may not be valid because the communication overhead may not allow agents to meet deadlines. Thus, R3 is necessary to merge lightly loaded agents even when $T_{response}$ exceeds $T_{deadline}$. This merging lowers coordination cost in the overall problem pipeline, and so improves performance.

C. Reorganization Process

Reorganization is triggered by the firing of a reorganization rule during the normal production cycle. We describe below how one agent (e.g., `agentP`) decomposes itself into two agents (e.g., `agentP` and `agentQ`). During reorganization, domain rules, WME's, dependency, and interference knowledge are transferred from `agentP` to `agentQ` without any modification. However, location knowledge is modified due to the relocation of domain rules and changes are propagated to neighboring agents.

1) *Create a new agent:* `agentP` creates a new agent, `agentQ`, which immediately starts production cycles.
2) *Select domain rules to be transferred:* `agentP` selects domain rules to be transferred (e.g., `ruleA` to `agentQ`. Currently, we arbitrarily transfer half of the active rules, but we are refining a theory of rule selection based on maximizing the intra-agent rule dependencies and minimizing inter-agent communication.
3) *Request synchronization:* `agentP` sends a synchronization request message for each rule to be synchronized (e.g., `deactivate(ruleA)`) to `agentQ`. `agentP` also sends synchronization request messages to its neighbors, i.e., all the domain rules that have data dependency or interference relationships with rules to be transferred[2] (e.g., `deactivate(ruleB)` is sent if `depends(ruleA, ruleB)`, `depends(ruleB,ruleA)` or `interfere(ruleA,ruleB)`). `agentP` waits for complete acknowledgment. While waiting for acknowledgment, `agentP` processes messages as in Step 1 of the production cycle described in Section IV-C.
4) *Tansfer rules:* `agentP` transfers rules `(ruleA)` to `agentQ`, updates its own location knowledge, and propagates the change to its neighbors.
5) *Transfer WME's:* `agentP` copies WME's that match the LHS of the transferred rules `(ruleA)` to `agentQ`.[3] A bookkeeping process follows in both agents to eliminate duplicated or unneeded WME's.
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.[4]
7) *Release synchronization:* `agentP` sends synchronization release messages `(activate(ruleA)` to `agentQ`, and `activate(ruleB)` to all synchronized *neighbors*). This ends reorganization.

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 domain and organizational knowledge to `agentQ` and destroys itself. The transfer method is the same as that for decomposition.

During the reorganization process, deadlock never occurs, because reorganization does not block other agents' domain problem solving and reorganization. Furthermore, though neighboring agents are required to deactivate domain rules that depend on or interfere with the transferred rules, they can concurrently perform other activities including firing and transferring rules that are not deactivated. This localization helps agents to modify the organization incrementally.

VI. Experimental Evaluation

To evaluate the effectiveness of our approach, we implemented a simulation environment and executed the *Waltz labeling program*: 36 rules solve the problem that appears in [33] with 80 rule firings. Our experiments begin with one agent

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

[3]More precisely, to avoid reproducing once-fired instantiations, not only WME's 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 before the synchronization is completed.

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

that contains all problem-solving knowledge. Its organizational knowledge is trivial in that references are to itself, since it has no neighboring agents.

A. Simulation Excluding Overheads

Figs. 5 and 6 show the simulation results. In these figures, communication and reorganization overheads are ignored. The line chart indicates response times normalized by production cycles. The step chart represents the number of agents in the organization. The time limit ($T_{deadline}$) is set at 20 production cycles, while the statistics measuring period (P) is set at 10 production cycles. In Fig. 5, problem solving requests arrive at constant intervals, while in Fig. 6, the frequency of requests is changed periodically. From these figures, we can conclude the following:

1) *Adaptiveness of the organization:* In Fig. 5, around time 100, the response time far exceeds the time limit. Thus, the organization starts decomposition. Around time 200, the number of agents has increased to 26, the response time drops below the time limit, and the organization starts composition. After fluctuating slightly, the organization finally reaches a stable state with the number of agents settling at 6. Since composition and decomposition have been repeatedly performed, the firing ratios of the resulting agents are almost equal. In Fig. 6, we can see the number of agents at the busiest peak decreases over time. Both charts show that the society of agents has gradually adapted to the situation through repeated reorganization.
2) *Real-time problem solving:* The average number of agents in Fig. 6 is approximately 9. We compared response times of our organizational approach which flexibly selects the number of agents, to those of the conventional parallel approach using 9 permanent agents. Differences in results from these two approaches demonstrate that while the conventional approach uses the same average number of agents, it cannot respond to meet deadlines when problem demand increases. Thus, the organizational approach is more effective for adaptive real-time problem solving. However, the effect of reorganization does lag behind the change in environmental demand. For improved capability to meet response requirements, the time limits must be set shorter than the actual deadlines and increases in agent activity should be detected as early as possible.
3) *Efficient resource utilization:* As shown in Fig. 6, the conventional parallel approach requires 17 permanent processors to meet deadlines. Thus, the organization-centered approach, which requires around 9 processors on average, is more economical.

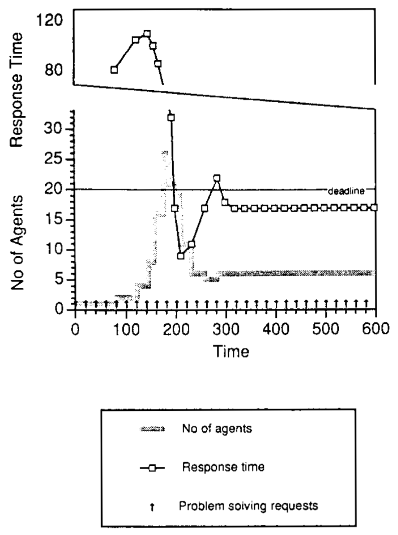


Fig. 5. Simulation results (constant intervals)

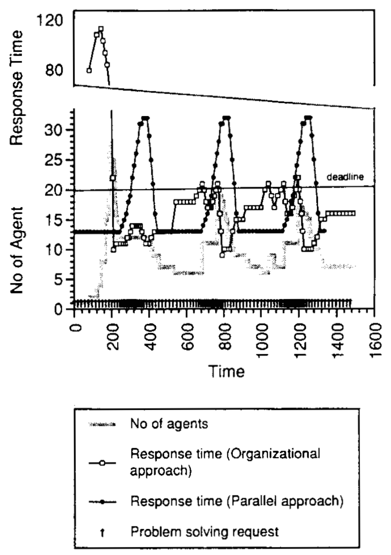


Fig. 6. Simulation results (changed intervals).

B. Simulation Including Overheads

Figs. 7 and 8 describe the results obtained from the same situation conditions as given in Fig. 5, but they include communication and reorganization overheads. But what are reasonable assumptions for communication and computation speeds? In the iPSC/2, a typical message-passing multicomputer, the communication overhead of sending a 2 kilobyte message across the diameter of a 128 node machine is 840 μs, and the transfer rate for a 64 kilobyte message is 2.6 megabyte/s [2]. For computation, a state-of-the-art production system takes from several to several tens of milliseconds for one production cycle on an HP9000/370, a Motorola 68030 based workstation [30]. Since one production cycle creates several messages, each of which contains a few WME's, the communication overhead in a good message-passing machine can be estimated as at most one production cycle. However, we also have taken into account communication overhead to cover cases in which wider-area and somewhat slower networks such as Ethernet or public telecommunication networks are used for

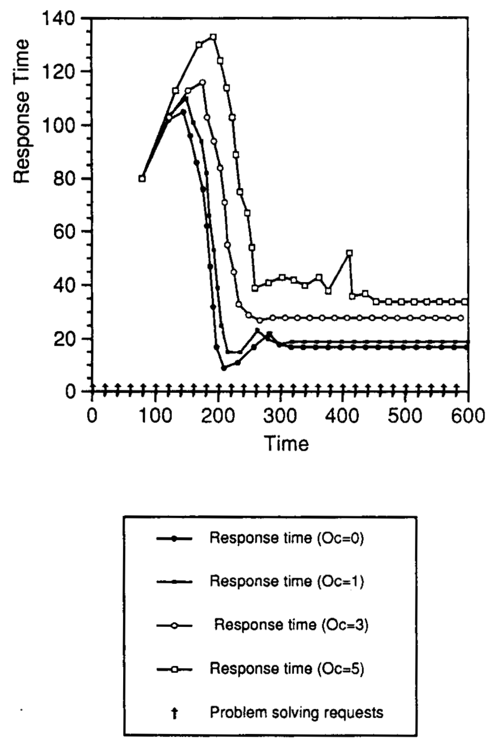


Fig. 7. Simulation results (communication overheads).

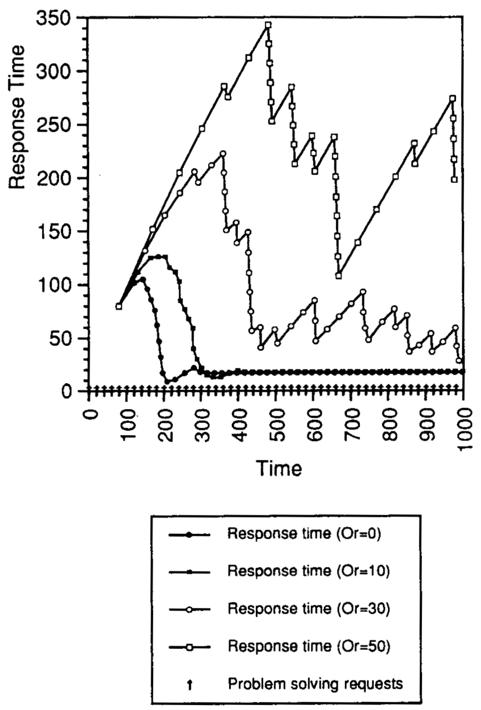


Fig. 8. Simulation results (reorganization overheads).

distributed problem solving. Let O_c be the average network latency represented in terms of production cycles. Then, agents can utilize other agents' results no sooner than O_c cycles later. We simulated situations in which O_c was equivalent to 1, 3, or 5 production cycles to assess the effect of communication overheads.

Reorganization overheads cannot be ignored even in message passing machines, depending on how many rules, WME's, and conflict sets are to be transferred.[5] Let O_r be the reorganization overhead in terms of production cycles. O_r of our example program costs at most 10 production cycles, during which we can transfer all rules of the Waltz labeling program and WME's for 10 pending problem-solving requests. However, we have simulated cases where O_r is equivalent to 10, 30, or 50 production cycles to observe the general influence of reorganization overheads on OSD for distributed production systems. The major results obtained from these simulations are as follows:

- *Influences of communication overhead:* Fig. 7 considers communication overhead but does not include reorganization overhead. When $O_c = 1$, the organization can meet its deadline, but when $O_c = 3$ or more, the organization fails to satisfy the real-time constraint. This is because communication overhead delays problem solving, and this also destablizes the organization. The organization fluctuates in two cases. Agents may decompose themselves rapidly so that $T_{response}$ becomes much less than $T_{deadline}$. This triggers R2 and causes agents to start composition. The other case occurs even when $T_{response}$ exceeds $T_{deadline}$, i.e., the firing ratio of agents significantly decreases because of the communication overheads. In this case, R3 is satisfied. The chances of the latter case increase with the communication overhead.
- *Influences of reorganization overhead:* Fig. 8 considers reorganization overhead but does not include communication overhead. Unlike the communication overhead, reorganization overhead is temporary and thus should not affect the stability of the organization. When $O_r = 10$, the organization soon reaches a stable state. However, when reorganization overhead becomes larger, such as $O_r = 30$ or more, the organization oscillates and never seems to become stable. The reason is as follows. Since reorganizing agents cannot fire rules during the decomposition process, their firing ratios temporarily decrease. Firing ratios of neighboring agents also decrease because no new WME is transferred from the reorganizing agents. As a result, R3 is fired in the neighboring agents to start composition, and thus the organization oscillates.

In summary, communication overhead is not a problem in current message passing machines. Furthermore, ongoing research on message passing machines has been reduced by the communication overhead by an order of magnitude [6]. However, in the future, communication overhead can be a problem when using wider networks to perform distributed

[5]When the RETE match algorithm [10] is employed, building the RETE networks in newly generated agents requires additional costs. However, we can ignore this by assuming the TREAT match algorithm [29], in which networks are built dynamically in each production cycle.

problem solving (cf., [21]). Reorganization overhead is also not a problem in this example, but it might cause oscillation if it is too large. Further research is required,[6] but one way to avoid oscillation due to reorganization would be to decrease the sensitivity of OSD by enlarging the period of measuring statistics (P).

VII. Discussion

A. Generalizing the OSD Approach

1) Generalizing the Environmental Change: This paper has presented a particular abstract model for one type of organizational self-design, in which a collection of agents adapts itself to changes in a particular set of environmental conditions. We can generalize elements of this model, to serve as the basis for other types of organizational adaptation to other kinds of environmental change, such as new quality requirements as follows.

To adapt to new quality requirements, the problem-solving organization must decide which quality level it should achieve, and then revise its behavior to achieve that new level. To do this, it needs a *quality level decision making procedure* that it can use to reason about the appropriate solution quality level, and *quality-manipulation mechanisms* for changing the quality level of its solutions. Such mechanisms may include revising its search space by refining the specification of the goal state (e.g., by constraining it further), or applying a more detailed set of operators. If we assume that more complete searching, possibly of a larger search space, leads to better quality solutions, then our OSD mechanisms can be invoked to create greater decompositions as the required quality levels increase. For a given response-time demand, greater decomposition will lead to higher solution quality, and vice-versa. Our initial research results of this approach have appeared in [15].

2) Generalizing the Reorganization Primitives: In our current OSD approach, performance may improve because extra resources have been supplied. However, extra resources need not only come from the environment in the form of new agents—they can also come from underused capacity of existing agents, and from recovering resources wasted in poorly organized communication and interaction structures. Similarly, underutilized resources need not only be returned to the environment—they can be returned to the organization itself in the form of improved organization structure. To do this, our decomposition primitive can be generalized so as to include additional decision making knowledge about whether to decompose$_1$ by creating a new agent (like hiring a new employee), or to decompose$_2$ by transferring knowledge to an existing agent (reallocation of skill). Our composition primitive can be generalized to include additional knowledge about whether to compose$_1$ by destroying an entire agent (like firing an employee), or to compose$_2$ by accepting partial knowledge from that agent (reallocation of skill).

[6]Hogg and Huberman [21] have studied similar problems in the abstract. They verify the possibility of chaotic behavior in systems with long communication delays, and suggest an approach to controlling chaos based on rewarding agents with good decision making performance. However, their scheme takes both the boundary and the decision capability of an agent to be fixed, whereas in our formulation, an agent is a flexible entity, and it is less clear where to assign credit or blame for poor performance over the longer term.

Transfers of large collections of knowledge, even among preexisting agents, could be expensive. One remedy would be to trade space for time, by giving each preexisting agent the entire collection of rules, and using location knowledge (Section IV-B) as the basis for deciding which rules within any agent were usable at any time. This approach would be a dynamic extension of the static approach to organization based on capability constraints used in [8]. In this way, reallocation of rules during composition$_2$ or decomposition$_2$ would be accomplished by using the already-existing mechanism of simply updating organizational knowledge. Transfers of local data (WME's) are unavoidable in any reorganization scheme.

B. Characterizing the OSD Approach

Conceptual foundations for organization may be characterized as a spectrum with two poles [14]. Purely *individualist* approaches (generally the standard in DAI) build an organization with individual, pre-existing agents with relatively fixed internal structures and one locus of action. These agents interact with each other under some set of internal or external constraints, and it is the constraints that provide organization. In purely *social-interactionist* approaches, neither the structure of the individual agents nor the nature of the organization is necessarily fixed. Instead, agents and organization are both treated as flexible constructions, carved out of a fabric of distributed interactions. Relationships between problem-solving knowledge, resources, and the loci of action are variable. Agents might thus be distributed and concurrent entities, and their boundaries and contents might change. Organization consists of emergent *patterns* of interaction, and is relative to the observer's viewpoint.

Approaches to implementing organization have generally included two sorts. *Structure-based organizations* use fixed interaction structures or capability restrictions to configure actions (e.g., by establishing roles among identical problem solvers by using capability constraints, as in [8]). Structure-based organizations are changed by changing the structural properties of the organization, such as the number or types of agents, their interaction structures (e.g., inter-agent connections) or by modifying agent capability constraints. In *knowledge-based organizations*, the particular distribution and use of knowledge configures actions (e.g., flexible networks of default knowledge proposed in [13]). Modifying the knowledge that agents have—e.g., about the beliefs, goals, or capabilities of another agent—or changing the distribution of knowledge in the group, causes changes in the possible patterns of action, and thus changes in the organization.

As a foundation for our model of organizational adaptation, we have taken a hybrid approach to conceptualizing and implementing organization. From the structure-based perspective, our reorganization primitives manipulate the contents of the agent population, including the number of agents and the resources they use, but depend upon the fixed and pre-calculated dependency relationships expressed in problem-solving knowledge. From the knowledge-based perspective, our reorganization primitives modify the distribution of both problem-solving knowledge and organizational knowledge. Our conceptual approach is individualist, in the sense that at any moment there is a fixed collection of agents each of which has a stable internal architecture. But our approach is also social-interactionist. In most distributed problem-solving and multiagent systems, the boundaries of agents are treated as fixed. Our scheme for OSD involves creating and destroying agents, as well as transferring organizational and problem-solving knowledge among agents.

In effect, our overall problem-solving system can be seen as a large and flexible fabric of knowledge, resources, and action, *out of which agents actively and flexibly construct and reconstruct themselves* by adding and subtracting resources and by changing agent–knowledge boundaries. In our OSD approach, it is the overall collection of problem-solving knowledge that is fixed—not the definition of agents. Agents, resources, and distributions are flexible and open to adaptation as the circumstances of the organization change. This represents a new approach to the nature of both agents and organization—an approach that conforms more closely to the social view of agents and organization articulated in [14]. It appears to offer the promise of increased organizational flexibility.

VIII. Conclusion

Techniques for building problem-solving systems that can adapt to changing environmental conditions are of great interest. We have presented an approach that relies on the reorganization of a collection of problem-solvers to track changes in response requirements, problem solving requests, and resource requirements. The approach exploits an adaptive tradeoff of resources and organization form to satisfy for time and performance constraints. Agents are created and destroyed, and domain knowledge is continually reallocated. To extend the possible architectures for OSD, composition and decomposition have been introduced as new reorganization primitives. Organizational knowledge has been formalized to represent interactions among agents and their organization. Overall, these developments provide a rich ground for future development of the concepts and implementation of organization in DAI systems. Future research involves the implementation and evaluation of more generalized versions of our approach, implementation on actual message passing multiprocessor systems, and the investigation of techniques for incrementally acquiring organizational knowledge in more dynamic contexts.

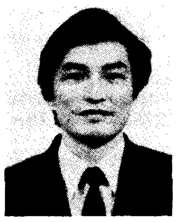
Acknowledgment

The authors wish to thank K. Murakami and R. Nakano for supporting our joint research. We also appreciate several helpful discussions with K. Kuwabara and A. Bond, and the comments of N. Rouquette on an earlier draft.

References

[1] A. Acharya and M. Tambe, "Production systems on message passing computers: Simulation results and analysis," in *Proc. Int. Conf. Parallel Processing*, 1989, pp. 246–254.
[2] L. Bomans and D. Roose, "Benchmarking the iPSC/2 Hypercube Multiprocessor," *Concurrency: Practice and Exper.*, vol. 1, pp. 3–18, 1989.
[3] A. Bond and L. Gasser, *Readings in Distributed Artificial Intelligence*. San Mateo, CA: Morgan Kaufman, 1988.
[4] B. D. Clayton, *ART Programming Tutorial*, Inference Corp., 1987.
[5] D. D. Corkill, "A framework for organizational self-design in distributed problem solving networks," Ph.D. dissertation, COINS-TR-82-33, Univ. of Massachusetts, 1982.
[6] W. J. Dally, "Directions in concurrent computing," in *Proc. Int. Conf. Comput. Design*, 1986, pp. 102–106.
[7] R. Davis and R. G. Smith, "Negotiation as a metaphor for distributed problem solving," *Artif. Intell.*, vol. 20, pp. 63–109, 1983.
[8] E. H. Durfee, V. R. Lesser, and D. D. Corkill, "Coherent cooperation among communicating problem solvers," *IEEE Trans. Comput.*, vol. C-36, pp. 1275–1291, 1987.
[9] E. H. Durfee and V. R. Lesser, "Using partial global plans to coordinated distributed problem solvers," in *Proc. IJCAI-87*, 1987, pp. 875–883.
[10] C. L. Forgy, "RETE: A fast algorithm for the many pattern / many object pattern match problem," *Artif. Intell.*, vol. 19, pp. 17–37, 1982.
[11] M. Fox, "An organizational view of distributed systems," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-11, Jan. 1981.
[12] L. Gasser, "The integration of computing and routine work," *ACM Trans. Office Inform. Syst.*, vol. 4, no. 3, pp. 205–225, July 1986.
[13] L. Gasser, N. Rouquette, R. Hill, and J. Lieb, "Representing and using organizational knowledge in DAI systems," in *Distributed Artificial Intelligence, Vol. II*, L. Gasser and M. N. Huhns, Eds. London, England: Pitman, 1989, pp. 55–78.
[14] L. Gasser, "Social conceptions of knowledge and action," *Artif. Intell.*, pp. 107–138, Jan. 1991.
[15] L. Gasser and T. Ishida, "A dynamic organizational architecture for adaptive problem solving," in *Proc. AAAI-91*, 1991, pp. 185–190.
[16] A. Gupta, C. L. Forgy, D. Kalp, A. Newell, and M. Tambe, "Parallel 0P55 on the Encore Multimax," in *Proc. Int. Conf. Parallel Processing*, 1988, pp. 271–280.
[17] B. Hayes-Roth, "A blackboard architecture for control," *Artif. Intell.*, vol. 26, pp. 251–321, 1985.
[18] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett, and A. Seiver, "Intelligent monitoring and control," in *Proc. IJCAI-89*, 1989, pp. 243–249.
[19] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artif. Intell.*, vol. 8, no. 3, pp. 323–364, 1977.
[20] C. Hewitt, "Open systems semantics for distributed artificial intelligence" *Artif. Intell.*, pp. 79–106, Jan. 1991.
[21] T. Hogg and B. A. Huberman, "Controlling chaos in distributed systems," Tech. Rep. SSL-90-52, Dynamics of Computation Group, Xerox Palo Alto Research Center, Palo Alto, CA, 1990.
[22] T. Ishida and S. J. Stolfo, "Toward parallel execution of rules in production system programs," in *Proc. Int. Conf. Parallel Processing*, 1985, pp. 568–575.
[23] T. Ishida, "Methods and effectiveness of parallel rule firing," in *Proc. IEEE Conf. Artif. Intell. Appl.*, 1990, pp. 116–122.
[24] ——, "Parallel rule firing in production systems," *IEEE Trans. Knowledge Data Eng.*, vol. 3, no. 1, pp. 11–17, 1991.
[25] T. Ishida, M. Yokoo, and L. Gasser, "An organizational approach to adaptive production systems," in *Proc. AAAI-90*, 1990, pp. 52–58.
[26] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, "Real-time knowledge-based systems," *AI Mag.*, vol. 9, no. 1, pp. 27–45, 1988.
[27] V. R. Lesser, J. Pavlin, and E. H. Durfee, "Approximate processing in real time problem solving," *AI Mag.*, vol. 9, no. 1, pp. 49–61, 1988.
[28] T. W. Malone, "Modeling coordination in organizations and markets," *Management Sci.*, vol. 33, no. 10, pp. 1317–1332, 1987.
[29] D. P. Miranker, "TREAT: A better match algorithm for AI production dystems," in *Proc. AAAI-87*, 1987, pp. 42–47.
[30] D. P. Miranker, B. J. Lofaso, G. Farmer, A. Chandra, and D. Brant, "On a TREAT based production system compiler," in *Proc. 10th Int. Conf. Expert Syst.*, Avignon, France, 1990.
[31] D. I. Moldovan, "A model for parallel processing of production systems," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 1986, pp. 568–573.
[32] J. G. Schmolze and S. Goel, "A parallel asynchronous distributed production system," in *Proc. AAAI-90*, 1990, pp. 65–71.
[33] P. H. Winston, *Artificial Intelligence*. Reading, MA: Addison-Wesley, 1977.

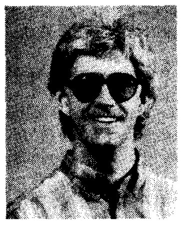
[34] M. D. Zisman, "Using production systems for modeling asynchronous concurrent processes," in *Pattern-Directed Inference Systems*, D. Waterman, Ed. New York: Academic, 1978.



Toru Ishida received the B.E., M.Eng., and D.Eng. degrees from Kyoto University, Kyoto, Japan, in 1976, 1978, and 1989, respectively.

He is currently with NTT Communication Science Laboratories, Kyoto, Japan. From 1983 to 1984, he was a visiting research scientist at the Department of Computer Science, Columbia University. Since 1983 he has been working in the area of production systems and their applications. His current research interests include parallel and distributed artificial intelligence.

Dr. Ishida is a member of the Information Processing Society of Japan, the Japanese Society for Artificial Intelligence, and AAAI.



Les Gasser received the B.A. degree in English from the University of Massachusetts in 1976, and the M.S. and Ph.D. degrees in Information and Computer Science from the University of California, Irvine, in 1979 and 1984, respectively.

He is currently on the faculty of the Department of Computer Science at the University of Southern California, Los Angeles. His main research interest is artificial intelligence at the social level—theoretical foundations and practical strategies for Distributed Artificial Intelligence (DAI). He has published two books and over 30 technical articles, and has consulted internationally on DAI research and system development projects.

Dr. Gasser is a member of the USC Robotics Institute and is a member of AAAI, the Association for Computing Machinery, SIGART, SIGCAS, and the IEEE Computer Society.

Makoto Yokoo received the B.E. and M.Eng. degrees from Tokyo University, Tokyo, Japan, in 1984 and 1986, respectively.

He is currently with NTT Communication Science Laboratories, Kyoto, Japan. He was a visiting research scientist at the Department of Electrical Engineering and Computer Science, the University of Michigan from 1990 to 1991. Since 1987 he has been working in the area of distributed artificial intelligence. His current interests include distributed artificial intelligence and constraint satisfaction.

Mr. Yokoo is a member of the Information Processing Society of Japan, the Japanese Society for Artificial Intelligence.