When Processes Learn: Steps Toward Crafting an Intelligent Organization

Dan Zhu • Michael J. Prietula • Wen Ling Hsu
Department of Management Sciences, College of Business, University of Iowa, Iowa City, Iowa 52242
dan-zhu@uiowa.edu
Department of Commerce and Technology, Johns Hopkins University, Baltimore, Maryland 21201
prietula@jhu.edu
AT&T Laboratories, Murray Hill, New Jersey 07974
hsu@research.att.com

Two trends in information systems research provide an opportunity to add an additional link between information technology and organizational learning. First, there is an increasing penetration of information technology into the firm’s processes and structures. Second, research in artificial intelligence has given rise to the first generation of fully computational architectures of general intelligence. In this research note we explore a melding of these two trends. In particular, we present the crafting of an organizational process which can learn, and develop and apply a new set of organizational learning metrics to that process. The process is a simplification of a complex, parallel-machine production scheduling task performed in a local manufacturing firm. The system, Dispatcher-Soar, generally supports a symbolic, constraint propagation approach based, in part, on the reasoning methods of the human scheduler at the firm. The implementation of this process is based on a dispatching rule used by the expert. The behavior of Dispatcher-Soar centered around a small case study examining the effects of scheduling volume and learning on performance. Results indicated that the knowledge gained can reduce within-trial scheduling effort. An analysis of the generated knowledge structures (chunks) provided insight into how that learning was accomplished and contributed to process improvements. As the knowledge generated was in a form standardized to a common architecture, metrics were used to evaluate the production efficiency ($\eta_{prod}$), utility ($\eta_{util}$), and effectiveness ($\eta_{eff}$) of the accumulated organizational knowledge across trials.

(Information Systems; Organizational Learning; Artificial Intelligence; Machine Learning)

1. Introduction

In recent years, there has been renewed interest in what has been characterized under the general rubric of "organizational learning" (Argyris and Schon 1978, Cohen and Sproull 1996, Levitt and March 1988). Embedded within this phrase lies a large variety of perspectives and interpretations, ranging in contexts from market/economy-level analyses (Sunder 1995), to firm-level analyses (Argote and Eppe 1990), to analysis of organizational routines (Nelson and Winter 1982), to the analyses of individual decision makers performing an organizational task (Cohen and Bacdayan 1994). Regardless, at some level, in some manner, according to some metric, it is generally asserted that organizations (or organizational agents) can improve their performance over time. That is, organizations can learn.

The assertion that organizations can learn is neither new nor profound. In today’s rapidly shifting and competitive environments, the basic concept of a successful
firm almost presupposes an ability to adapt on a variety of levels (Garvin 1993). This adaptation may take the form of market- and industry-level decisions of expansions, mergers and acquisitions (Moore 1993), or focus on the design of fundamental processes (Hammer and Champy 1993). In order to learn, in order to adapt, critical aspects of the firm must be altered. The essence of learning is change.

For many organizations, change opportunities reside in the processes the business performs (Davidow and Malone 1992), and information technology (IT) can both improve and enable process changes as well as the processes themselves (Davenport 1993). Research on organizational processes and IT has emerged as relevant (to greater and lesser degrees) to researchers in organizational learning who take a general information processing perspective, such as relating IT to organizational “memories” (Stein and Zwass 1995), exploring issues of IT and organizational “cognition” (Boland et al. 1995), relating aspects of IT and organizational structuring and design (Huber 1990, Leidner and Elam 1995, Orlikowski and Robey 1991), and exploring issues of technological adoption and integration (Walton 1989, Zander and Kogut 1995). Relatedly, a great deal of work has been done in examining IT and organizational communication (Markus 1994, Sproull and Kiesler 1991). Overall, much of this work, which has centered around exploring or explaining the source of organizational learning, has a common and appropriate assumption: the presence of human decision makers in the processes.

The purpose of this paper is to investigate how IT may realize and impact aspects of organizational processes (and subprocesses) and learning without the presence of human decision makers. Realizing organizational processes in computational form may not only be fundamental for gains in efficiency and productivity in core activities (Winslow and Bramer 1994), but could have direct implications of the emerging conflicts between work demands and human performance limits (Moore-Ede 1993). We begin this investigation by crafting an observed organizational process as an intelligent agent capable of learning. From this perspective, organizational learning is specifically and unambiguously characterized as increases in task performance. The source of this organizational learning is the knowledge directly resulting from the underlying intelligent organizational processes. Processes run. Processes learn. Processes run better.

2. Assembling an Intelligent Process
The concept of an intelligent business process may suggest knowledge-based (expert) systems or their functional equivalents (Gonzalez and Dankel 1993). To create a truly intelligent business process of the sort we are describing, it is necessary to incorporate an architecture that goes beyond the capabilities of current form of knowledge-based systems. Although a knowledge-based system may capture the knowledge relevant to a process, it lacks the capability of autonomously adapting at sufficient levels to task variations. As the essential nature of an intelligent process presumes the capability to learn, learning is considered as a necessary component of an intelligent business process. In addition, the capability to learn should not be limited to a particular type of task; rather, that capability should be applicable (though possibly with greater or less success) to any type of learnable environment. That is, it should be a general model of learning and intelligence.1 A candidate architecture is Soar.

2.1. The Soar Architecture
Soar is a computational system for building intelligent systems and modeling of human cognition (Laird et al. 1987). Soar offers a symbol-oriented, state-space structure where all intelligent deliberation behaviors are characterized as search in problem spaces to reach a goal. A problem space consists of an initial state (a representation of the problem as symbol structures), one or more goal states to achieve (another set of symbol structures) signaling search termination, and collections

1 There are a variety of adaptive computational paradigms, such as neural networks (Zurada 1992), genetic algorithms (Koza 1992), mathematical approaches (Narendra and Thathachar 1989), families of logics (Ginsberg 1987), and a host of other machine learning techniques (Carbonell 1990). These, however, are generally not viewed as general symbolic cognitive architectures at the level of abstraction we are defining, as their states do not represent general cognitive content, nor do their architectures correspond to general cognitive components (Pylyshyn 1991).
of operators (search control knowledge) that detect and manipulate state symbol structures until search terminates. All Soar knowledge is stored in a permanent recognition memory as production rules.\(^2\) In addition, the focus of attention (setting the context for problem-solving deliberation efforts) is established in the form of a working memory, where state symbol structures reside and persist only as long as it is relevant to the current goals under consideration. Knowledge in a working memory acts as retrieval cues to knowledge in permanent memory. All retrieval, execution, and ancillary resolutions are handled directly by the Soar architecture via a uniform decision processes called the decision cycle.

The decision cycle in Soar is the basic “unit of cognitive effort” within which the Soar architecture responds to changes in working memory contents by invoking all candidate long-term memory productions in parallel (elaboration phase) and evaluating their collective preferences to determine subsequent changes to working memory elements (decision phase).\(^3\) Thus knowledge is invoked solely by pattern-recognition through the working memory, with the final resolution of their effects to be determined by low-level architecture semantic evaluations of a fixed preference structure. When this evaluation is complete, changes are made to the relevant working memory structures and the decision cycle starts again.

Successful problem solving in Soar involves finding a sequence of operators to transform the problem representation from the initial state representation to an acceptable goal state. However, within a given decision cycle, the results of the elaboration phase may not have generated sufficient knowledge to suggest an unambiguous set of preferences for the decision phase. When this occurs, an impasse ensues. Once an impasse has been detected, the architecture of Soar automatically sets up a new subgoal to be achieved, complete with its own problem space: the resolution of the impasse. Soar can now bring to bear its full problem solving architecture to resolve the subgoal (which may recursively result in other subgoaling behaviors). Subgoal problem solving in Soar terminates whenever sufficient knowledge is discovered to resolve the current (or any higher-level) subgoal contexts.

Learning in Soar occurs whenever subgoals are resolved. By associating the relevant componential change sequences of symbol structures with impasse resolution conditions, Soar produces new chunks of knowledge that can avoid similar impasse-producing states and invoke the sequence of operators that more directly result in achieving a goal (Laird et al. 1986). This is accomplished by Soar backtracing recursively in order to determine the relevant antecedent elements in working memory that lead to the resolution. Once a chunk is produced, it is immediately available for application within the same trial, if necessary; that is, Soar supports within-trial learning. The resulting accumulation of knowledge (i.e., chunks) can, depending on the nature of the task, result in dramatic reductions in deliberation effort, as more aspects of the task are solved directly through recognition. Formulated chunks are always available to service goals, as Soar retains all accumulated knowledge for possible invocation.

Finally, the development of Soar applications requires an explicit articulation of the task. For the Soar architecture, task knowledge is represented in terms of nine essential roles played in problem solving within the architecture: problem space proposal, state proposal, operator proposal, operator application, selection, goal testing, subgoal results, evaluation, and entailment.

Problem space proposal constructs define the conditions under which a problem space should be proposed. State proposal constructs propose (arbitrarily complex) working memory structures for the objects of interest to be included in the state representation for the problem space. Operator proposal constructs monitor state representations and propose operators when the specified conditions are satisfied. Operator application constructs specify the objects in working memory to edit when applying a given operator, the conditions under which the editing should take place, and when an operator should

---

\(^2\) Essentially, production rules are a type of data structure generally written in the form of “if (condition) then (action)” where (condition) describes the condition(s) under which the rule is fired, and (action) describes the events to ensue when the production rule is fired (Klahr et al. 1987).

\(^3\) The evaluation of preference semantics is done by the architecture and not under direct control of the knowledge productions that propose them. That is, each “component” of knowledge proposes changes to be made, but only the architecture has the ability to evaluate proposals.
terminate.\(^4\) Selection constructs are general ways to incorporate knowledge to generate preferences for competing objects (e.g., which operator to invoke next) as well as for explicit control of operator sequencing (e.g., explicit preferences on operator invocation). Goal testing constructs monitor whether the (possibly multiple alternative) conditions indicating the achievement of a problem space final state have been satisfied. Subgoal results constructs specify how to create the resulting state from an operator that has been applied in a subgoal.\(^5\) Evaluation constructs can explicitly influence behavior during search, such as determining what objects in working memory to be evaluated, under what circumstances to evaluate them, and the properties of the evaluation function. Entailment constructs change objects in working memory as do operators. Unlike persistent operator changes, if the conditions under which an entailment construct was invoked no longer exist in working memory, entailment changes to state representations are revoked.

2.2. The Scheduling Process
There is no generally accepted definition of a business process. Rather, there are a plethora of intuitive process and process-equivalent definitions, depending on the perspective taken when viewing organizations. For this report, we adopt a liberal interpretation of a process as "... a specific ordering of work activities across time and place, with a beginning, an end, and clearly identified inputs and outputs: a structure for action" (Davenport 1993, p. 5). The process selected is based on a logistical process of a large manufacturing environment located near Pittsburgh, Pennsylvania. A key component to this process was scheduling. Scheduling was not simply a task done by a software package, but a complex sub-process involving several sequential tasks unfolding over time, done by a single individual interacting with a variety of information resources to both gather information and resolve conflicting goals. A subset of the scheduling process was addressed.

To implement this process, it was necessary to gather a substantial amount of knowledge from the scheduler and to do an extensive analysis of the process at the site. The purpose was to begin to craft the initial components of a system which could eventually integrate the steps of the scheduling process previously done by several people, thus not only gaining process improvements in time and error reduction through this type of integration (Hammer and Champy 1993), but exploiting the inherent Soar's capability to resolve conflicting goals, to coordinate subtasks, and to learn from the resolution.

Knowledge about the process involves the representation of jobs, machines, constraints, available resources, and a profile of production processes. Knowledge about the problem solving methods primarily concerns a constraint propagation technique; that is, a way of propagating constraints to the global environment and the consequences of the local decisions taken during the problem solving activity. This also included knowledge about scheduling heuristics to guide the process.

In constructing a schedule, there are two kinds of constraints involved: hard constraints and soft constraints. Hard constraints are required constraints which dictate the admissibility of a schedule. For example, a job can be scheduled with another job at the same time (sharing the machine) only when they satisfy a number of compatibility constraints. Soft constraints are preferences which determine the comparative quality of schedules. For example, a particular job is preferred not to be scheduled on a certain time on a certain day. Obviously, some constraints may not be satisfied simultaneously as they may be in conflict with each other; therefore, it is essential to find the relative importance of constraints and relax where necessary.

The basic procedure involves a list scheduling algorithm (or critical path scheduling) combined with some simple, domain-specific heuristic dispatching rules obtained from the human scheduler. The listing algorithm lists the jobs in any order and then assigns them to the machines as machines become available, where the jobs are dispatched based on a multiple criteria priority metric. The scheduling process begins by placing all jobs on a priority queue of unscheduled...

\(^4\) Recall that operators, as search, involve manipulation of working memory elements, as state representations, until an acceptable goal state (representation) is achieved or an impasse occurs.

\(^5\) When that subgoaling activity eventually concludes (via a goal testing construct in the subgoal problem space), the final state representations resulting from the subgoaling activity are not done within the context of the subgoal problem space, but in the original "calling" problem space.
jobs. The greedy heuristics selected assigns jobs to the machines in a myopic fashion. Essentially, it sequentially selects the unassigned jobs from a priority queue to the machine with available capacity. The general heuristic constraint propagation procedure is summarized as follows:

Step 1: Generate a list of the jobs based on certain loading rules.

Step 2: Select a machine with the least amount of processing time allocated.

Step 3: Select a job from a list of all unscheduled jobs \( \{j_1, j_2, \ldots, j_n\} \).

Step 4: Apply the constraint set \( \{C_1, C_2, \ldots, C_k\} \) to the job from the list. If the job passes the constraint checking, then go to Step 5, otherwise, go to Step 3 to pick another job.

Step 5: Assign the job which passed the constraint checking to the corresponding slot of the machine \( M_i \) with the least amount of processing time allocated.

Step 6: If all the jobs have been assigned, then Stop, else go to Step 2.

Once the knowledge had been acquired and analyzed for consistency, compatibility, and coherence, the next step involved crafting it in terms of the target implementation system, Dispatcher-Soar.⁶

2.3. The Structure of Dispatcher-Soar

Dispatcher-Soar is a Soar system implementing an intelligent scheduling process incorporating a simple heuristic dispatching rule within the general constraint propagation procedure previously described. The basic Dispatcher-Soar structure partitions the type of scheduling problem into eleven primary task problem spaces plus a top-level space where all Soar activity begins. Figure 1 shows these problem spaces and their relationships (via impasse invocations). Referring to Figure 1, the initial space proposed is located at the top of the figure, the TOP-SPACE problem space. It defines the task state and proposes the basic operator for the process, DO-SCHEDULE.⁷ The DO-SCHEDULE operator is a complex operator and cannot be applied directly; therefore, when this task operator is selected in the TOP-SPACE space, an impasse occurs. As a result, the SCHEDULE-PS problem space is proposed to resolve the impasse as a subgoal.

In the SCHEDULE-PS space, three decisions have to be made, namely, selecting a machine from the available machines, selecting a job from a list of schedulable jobs, and forming the reservation group (i.e., groups of different, but compatible, parts), which correspond to the three common decision points in parallel machine scheduling. Consequently, three corresponding operators are proposed in the SCHEDULE space: SELECT-MACHINE, SELECT-JOB, and FORM-RESERVATION-GROUP. As there is not enough knowledge in the current space to implement any of these operators, the control will be directed to three subspaces. The Dispatcher-Soar first detects if a machine needs to be chosen. If so, Dispatcher-Soar will select a machine. Once the SELECT-MACHINE operator is chosen, the lack of knowledge in the problem on how to implement the operators causes an impasse and a subgoal is created, passing control to the SM-PS problem space.

In the SM-PS space, Dispatcher-Soar compares the amount of processing time allocated to each machine and chooses the machine with the least processing time allocated. After the selection of the machine, the subgoal is attained and the results are returned to the problem space that invoked the subgoal, SCHEDULE-PS. Following the selection of machine, a job is then selected. Similar selection/impasse events again occur in the SM-PS space for the SELECT-JOB operator, resulting in a subgoal being produced and the invocation of the SJ-PS problem space.

The goal of SJ-PS space is to search through the job list to find a candidate job that can be scheduled on chosen machine, subject to the set of constraints established. Accordingly, constraint propagation is initiated

⁶ Constraint satisfaction approaches (Tsang 1993) have a rich history in artificial intelligence (e.g., Kumar 1992, Mackworth and Freuder 1985, Stefik 1981a, b, Sussman and Steele 1980) and an emerging presence in scheduling (e.g., Fox 1987, Zweben and Fox 1994). The purpose of the approach taken in this paper is not to advance either constraint satisfaction or scheduling approaches, but to apply a simple version of a symbolic algorithm to instantiate a common organizational process and explore the implications of learning.

⁷ All operators are not explicitly shown in Figure 1. However, all of the problem spaces in Figure 1 actually represent problem spaces that implement operator calls from a higher-level problem spaces as indicated.
by the complex operator CPROP, which results in control being passed to the CPROP-PS problem space.

In the CPROP-PS space, four more operators are proposed to instantiate and support constraint propagation in Dispatcher-Soar: HARDCHECK for propagating hard constraints, SOFTCHECK for examining soft constraints, SEARCH-NEXT-JOB for finding jobs to schedule, and ELIMINATE-SCHEDULED-JOB for deleting scheduled jobs. Once proposed, each will cause a sub-goal to be generated and control will pass into their respective problem spaces: HPROP-PS, SPROP-PS, SEARCH-PS, and ESJ-PS. Each hard constraint and soft constraint has its own operators and subspace invocation in their respective problem spaces, depending on the constraints included in the representation.

Constraint propagation will result in either success or failure. If a job failed to satisfy one of the hard constraints, all the relevant information regarding this job has to be initialized to its original situation and the job is returned to the job list. Search for the next available job will begin by proposing the SEARCH-JOB operator, resulting in the invocation of the SEARCH-PS space. If a job satisfies both hard and soft constraints, it becomes the candidate job in the schedule. This job is assigned and eliminated from the job list by the ELIMINATE-SCHEDULED-JOB operator realized in the ESJ-PS space. After the job is selected, the reservation group needs to be formed. A reservation group is a group of jobs that, although not equivalent, are similar enough to be scheduled at the same time for the resource. This is implemented via the operators in the FRG-PS space.

In the FRG-PS space, it first must be determined if a compatible part is needed (i.e., if there is space available on the machine). If a part is indeed desired, Dispatcher-Soar will choose the next available job and compatible job, summoning CPROP-PS again. After a compatible job is selected, Dispatcher-Soar then computes the start
3. Exploring Dispatcher-Soar's Behavior

Recall that the general task involves a set of jobs \( J = \{J_1, J_2, \ldots, J_n\} \), where job \( J_i \) has processing time \( P_i \), are to be processed on \( m \) parallel machines. Each job is to be processed on one of the machines with single operation. Job-related information embodies job lot size, duration (the processing time required), requirements for auxiliary resources, and properties (e.g., type, dimensions) that determine compatibility of the parts scheduled. A small case study was developed to examine the effects schedule volume and learning on Dispatcher-Soar's performance and behavior. Volume was considered as it is one of the most important decisions in the process and interacts with constraints and job mix. Learning was explored not only to examine how a schedule-based process might be implemented, but also to determine how learning could contribute to solving problems of scale-up in the scheduling task.

Dispatcher-Soar was implemented in the Soar 5.2 environment (Laird et al. 1990) on a Digital Equipment Corporation DEC 3100 workstation. To facilitate the development of the Soar productions, a system called TAQL (Task Acquisition Language) was used to define the task in terms of problem space components (Yost and Altmann 1991). TAQL then compiles those component descriptions into the requisite sets of Soar productions. Dispatcher-Soar is comprised of 109 TAQL constructs that compile into 324 Soar productions (before learning). In Table 1, example TAQL constructs are shown with their English interpretation. Table 2 shows the distribution of TAQL constructs over the role they play in Dispatcher-Soar.

Each trial consisted of a Dispatcher-Soar program run. If it were a trial permitting learning, Dispatcher-Soar would generate chunks whenever it could and apply them when they were applicable within the same trial (as described in prior §2.1). If it was a trial with no learning permitted, learning was disabled, and Dispatcher-Soar carried out the tasks, but did not generate any chunks and, consequently, did not learn in the trial. At the end of each trial, program behavior traces and statistics were gathered, which included information on the total number of productions, the number of chunks that have been built (if learning was used), and the number of decision cycles. Decision cycles (§2.1) were used as a comparative measure of effort. This affords a normalized measurement across platforms expressed in terms of the architectural behavior (i.e., goals, knowledge, impasses) and not in terms of machine performance (i.e., seconds). The effort, as measured by number of decision cycles, varies according to the nature of the task, and how much of what was learned (generated chunks) can be reapplied.

In our study we chose to vary the tasks by volume, with the intention to explore whether or not increased volume implies increased effort for problem solving and, if so, where the effort would occur in the process. We also explored the impact of learning on problem solving effort in order to determine how and to what extent learning would decrease problem solving effort. Potentially there are other parameters that may affect the problem solving effort, such as the type and number of constraints. Our choice was based on representing and examining the impact of the most dominant constraint first.

Two levels of job volume, low (6 jobs) and high (12 jobs), were crossed with Dispatcher-Soar runs with and without learning enabled.\(^8\) Three machines were simulated with a single intra-resource constraint: large jobs (i.e., physical size of an individual product) cannot be scheduled back to back. Within the two volume levels, the mix and order of the jobs were the same for all runs. The job mix (large, small) reflected the relative occurrence in the observed plant.

\(^8\)These numbers are about an order of magnitude less than the real scheduling task. The deficit is in the number of jobs to be scheduled, as the real task involves substantial production runs over three machines, but in the size of the set of alternative jobs to be considered. We simplified the task as the purpose was to perform in-depth analyses of Dispatcher-Soar behaviors and to illustrate the fundamental components of the approach.


Table 1  Examples of Dispatcher-Soar TAOQ Constructs

<table>
<thead>
<tr>
<th>TAQL Construct Type</th>
<th>Dispatcher-Soar Example (some are abbreviated)</th>
<th>English Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Space Proposal</td>
<td>(propose-space ps *schedule &lt; function (apply operator do-schedule) :space Schedule-PS)</td>
<td>Propose the Schedule-PS problem space for the purpose of applying an operator called do-schedule.</td>
</tr>
<tr>
<td>State Proposal</td>
<td>(propose-initial-state isp *schedule-ps :space Schedule-PS :use superstate)</td>
<td>In the Schedule-PS space, do not create a whole new state, but use the state representations from the problem space that called it (the superstate).</td>
</tr>
<tr>
<td>Operator Proposal</td>
<td>(propose-operator po *select-machine :space Schedule-PS :op (select-machine *current-lehr (cl)) :when ((goal *state (s) *quiescence t) :state (s) *phase select-machine-phase *current-lehr (cl)))</td>
<td>If there is a goal that is flagged as quiescent (waiting) for state (s), and that state is in the select-machine-phase with an associated machine (cl), then suggest the select-machine operator and include a pointer to the current-lehr.</td>
</tr>
<tr>
<td>Operator Application</td>
<td>(apply-operator ao *select-lehr :space SM-PS :op (select-lehr) (edit :what state :replace (phase :by select-job-phase) :replace (select-machine :by yes) :when ((state *pointer-lehr (pl)) (lehr (pl) *next none)) :replace (current-lehr (cl) :by (pl) :when ((goal *operator (o) *state (s)) (operator (o) *result t *pointer-lehr (pl))))))</td>
<td>Apply the select-lehr operator according to the guidance in the edit-clause. The edit-clause specifies how to change elements in the state representation of the SM-PS space. The when-clauses define the conditions under which to make changes. In the example, the first when-clause applies to the first two replace-clauses, while the second when-clause applies only to the third.</td>
</tr>
<tr>
<td>Selection</td>
<td>(compare schedule *machine *jog :space Schedule-PS :op1 select-machine :op2 select-job relation better)</td>
<td>In the Schedule-PS problem space, if the named two operators are being considered, prefer the select-machine operator over the select-job operator.</td>
</tr>
<tr>
<td>Goal Testing</td>
<td>(goal-test-group ggt *select-machine :group-type success :space SM-PS :test (select-machine-done :when ((state *select-machine yes))))</td>
<td>When the current state in the SM-PS space has the select-machine attribute with the value &quot;yes,&quot; then declare that a final state (goal) has been achieved in the state.</td>
</tr>
<tr>
<td>Subgoal Results</td>
<td>(result-superstate rs *compute-times :group-type success :space CTime-PS :terminate when ((superstate *compute-t-done yes)) (edit :what superstate :replace (compute-times :by finished)))</td>
<td>When the final state in the CTime-PS space has been achieved, effect changes in the state that invoked the compute-times operator (i.e., called in the superstate).</td>
</tr>
</tbody>
</table>

4. Results

From a manufacturing perspective, this study explores the contribution of machine learning to the general goal of creating manufacturing processes that are intelligent (Zweber and Fox 1994). The overall results of the learning study are presented in Figure 2. From the figure, three observations can be made. First, without learning, increasing the volume of scheduling substantially increases the effort in terms of the decision cycles (Dc) required for Dispatcher-Soar to solve the problem. Sec-
Table 2  Distribution of TAQL Constructs Types within Problem Space

<table>
<thead>
<tr>
<th>Problem Spaces</th>
<th>PSP</th>
<th>SP</th>
<th>OP</th>
<th>OA</th>
<th>SEL</th>
<th>GTEST</th>
<th>SUBG</th>
<th>Total Constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task-PS</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>SCHEDULE-PS</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SM-PS</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SJ-PS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>CPROP-PS</td>
<td>3</td>
<td>3</td>
<td>13</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>HPROP-PS</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>SEARCH-PS</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>ESJ-PS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>FRG-PS</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
<td>2</td>
<td></td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>CTIME-PS</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>ASSIGN-PS</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Total Constructs 12 13 37 15 9 12 11 109


Learning reduces effort, as designed. But where does that learning occur? The problem space construct reflects as well-known hypotheses concerning the conceptual structuring of task environment (Newell and Simon 1972). Soar realizes problem spaces as a functional structuring of the task environment, where all learning in Soar takes place in the context of specific problem spaces. Figure 3 shows the distribution of effort (in terms of decision cycles) by problem space for the Low and High volume cases.

In examining Figure 3, note how the general patterns of the two traces are similar. This is to be ex-
expected as the task environment changed in terms of one parameter, job volume. Overall, the mean percentage change (reduction in effort) for the Low volume case was 59.1 percent ($\Delta D_e = -659$) while the mean percentage change for the High volume case was 70.2 percent ($\Delta D_e = -1549$). The highest percentage and absolute gains in learning occur in the SCHEDULE-PS problem space, where improvement for the Low volume case is 95.3 percent ($\Delta D_e = -267$) and improvements for the High volume case is 96 percent ($\Delta D_e = -586$). In the Low volume case, secondary absolute gains are found in FRG-PS at 80.2 percent ($\Delta D_e = -122$) and the CPROP-PS at 72.6 percent ($\Delta D_e = -109$). In the High volume case, secondary absolute gains are also made in CPROP-PS at 77.5 percent ($\Delta D_e = -287$) and in FRG-PS at 81.8 percent ($\Delta D_e = -270$). Thus, learning was able to exploit regularities in most problem spaces (indicated by a reduction in $D_e$), with specific problem spaces accounting for much of the effort reduction.

An understanding of how this occurred can be gained by examining the source and application of knowledge generated during task performance—the generation and application of chunks. A chunk, as it is knowledge in the form of a production rule, has an “if then” structure, and is created from working memory elements and preferences. For example, chunk P152 from one of the runs is:

(STATE (S1) "DUMMY-ATT TRUE" CURRENT-JOB (P1) CURRENT-LEHR (L1))
(OPERATOR (O1) NAME CHECK-WIDTH)
(LEHR (L1) NAME L1)
(JOBN (P1) SCHEDULED NO MARK1 UNCHECKED P-STATE UNKNOWN WIDTH 31)

(STATE (S1) "DUMMY-ATT TRUE" CURRENT-JOB (P1) CURRENT-LEHR (L1))
(OPERATOR (O1) NAME CHECK-WIDTH)
(LEHR (L1) NAME L1)
(JOBN (P1) SCHEDULED NO MARK1 UNCHECKED P-STATE UNKNOWN WIDTH 31)

The clauses before the separator symbol (→) specify the antecedent conjunctive elements that must match in working memory, while the clause(s) after the separator symbol are the actions to propose if the antecedent conditions are met.6

The first clause identifies the chunk (SP P152). The second clause starts with a object identifier for the clause (a GOAL object) and a symbolic pointer ((G1)) which is external to the chunk and therefore acts as a free variable. In the GOAL clause, there are two attributes, one pointing to a STATE object ((S1)) and the other pointing to an OPERATOR object ((O1)), where both objects are represented in the next two clauses (indicated by the matching symbols (S1) and (O1) respectively). The STATE and OPERATOR objects (general type of Soar objects), both refer to problem-specific objects and attributes: LEHR (machine resource) and JOB (job to be scheduled on the resource). Other attributes are used for flags and other Soar-specific conditions. A translation of chunk P152 is:

IF

in the context of achieving some goal (g1)
the name of the proposed operator is check-width,
and the state has a current job which has block width with 31,
and the name of the current machine is L1,
and job has not been checked,
and job has not been scheduled yet,

THEN

Mark a best preference for the operator that change the flag p-state with pass

6 All elements of working memory are represented as object-attribute-value structures, where objects can have multiple attribute-value pairs, attribute names are preceded by a carat (\(^\) ) followed by their values, and values of objects can be explicit constants, or symbolic pointers (indicated by brackets, ( )) to other objects (thus permitting complex structures to be represented).
Table 3: Analysis of Knowledge Chunks Produced by Problem Space for Low and High Volume Cases

<table>
<thead>
<tr>
<th>Problem Space</th>
<th>Number of Unique Chunks Created</th>
<th>Percent (% of Total Chunks)</th>
<th>Total Number of Chunk Firings</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEDULE-PS</td>
<td>308</td>
<td>700</td>
<td>43.6%</td>
</tr>
<tr>
<td>SM-PS</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>SJ-PS</td>
<td>42</td>
<td>102</td>
<td>5.9%</td>
</tr>
<tr>
<td>CROP-PS</td>
<td>130</td>
<td>265</td>
<td>18.4%</td>
</tr>
<tr>
<td>HPROP-PS</td>
<td>77</td>
<td>76</td>
<td>10.9%</td>
</tr>
<tr>
<td>ESJ-PS</td>
<td>32</td>
<td>32</td>
<td>4.5%</td>
</tr>
<tr>
<td>SEARCH-PS</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>FRG-PS</td>
<td>106</td>
<td>205</td>
<td>15.0%</td>
</tr>
<tr>
<td>CTIME-PS</td>
<td>12</td>
<td>12</td>
<td>1.7%</td>
</tr>
<tr>
<td>ASSIGN-PS</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>TOTALS</td>
<td>706 chunks</td>
<td>1390 chunks</td>
<td>100%</td>
</tr>
</tbody>
</table>

Chunk P152 will be stored in recognition memory and Dispatcher-Soar continues to run. If another job with the same critical attribute (width) and value (32) is encountered, applying the CHECK_WIDTH operator does not cause an impasse as it did in Figure 3 on decision cycle 29. Instead, the created chunk P152 will be activated from memory and applied to the current context.

In Table 3 a comparative analysis of chunk production and use is given for the Low and High volume cases. The number of chunks produced in each problem space by case is shown in the first column category, while the relative percentages are given in the second column category. The third column category reveals how many times chunks were applied. From Table 3, it can be seen that for both cases chunks are generated in the problem spaces with subsequent reduction in effort (i.e., SCHEDULE-PS, CROP-PS, FRG-PS). Chunks were applied in a similar fashion. The only exception is the upper level space, SCHEDULE-PS. The reason for this is that the production of the chunks significantly reduces the need to "come up" to the top level to learn the sequence of high-level operators. That sequence did not change. What changed was the sequence of events responding to particular jobs.10

As chunks represent the fundamental units of learning, it affords an important opportunity to construct knowledge metrics to aid in the analysis of the relative efficiency and effectiveness of the chunks within the context of process behavior and performance. Three such knowledge metrics have been defined (Prietula et al. 1993) and are summarized in Table 4. The first knowledge metric measures production efficiency ($\eta_{prod}$).

Table 4: Knowledge Metrics for Dispatcher-Soar Analysis

<table>
<thead>
<tr>
<th>Metric</th>
<th>Computational Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production Efficiency</td>
<td>$\eta_{prod}$</td>
<td>Proportion of total chunks generated that were eventually applied in performing the task.</td>
</tr>
<tr>
<td>Knowledge Utility</td>
<td>$\eta_{util}$</td>
<td>Average number of times applied chunks were used in performing the task.</td>
</tr>
<tr>
<td>Knowledge Effectiveness</td>
<td>$\eta_{eff}$</td>
<td>Average contribution of knowledge chunks to the reduction of problem solving effort.</td>
</tr>
</tbody>
</table>

10 Because of the hierarchical nature of problem space invocations and chunk production, it is possible that an operation may also be subsumed to a higher level after learning. For example, if a chunk is built that anticipates a certain set of conditions for an operator in a problem space, that operator (through the acquired chunks) may be applied directly and avoid a once-needed subgoal. Therefore, deliberation after learning can shift among problem spaces.
and reflects the proportion of produced chunks that were eventually applied in the same trial. First, a constraint on the summand is given as follows:

\[
[\mathcal{A}(k)] = \begin{cases} 
1, & \text{if chunk } k \text{ is applied at least once within the trial,} \\
0, & \text{if chunk } k \text{ is not applied within the trial.}
\end{cases}
\]

The metric \(\eta_{\text{prod}}\) ranges from 0.0 (when none of the produced chunks were applied in the trial) to an upper limit of 1.0 (when all of the produced chunks were applied at least once in the trial). In essence, it is a measure of the proportion of the knowledge applied. Dispatcher-Soar for the Low volume case generated 706 chunks and used (at least once) 88, yielding a \(\eta_{\text{prod}} = 0.12\). For the High volume case, Dispatcher-Soar generated 1390 chunks and applied 100, producing a \(\eta_{\text{prod}} = 0.07\). Thus, Dispatcher-Soar in the Low volume case applied a proportionally greater number of the chunks it produced than in the High volume case. Thus, the High and Low volume cases basically used a similar set of chunks (in terms of the problem space in which they were created and applied) indicating some similar "core" knowledge was learned across tasks.

A second knowledge metric, \(\eta_{\text{util}}\), estimates the knowledge utility of the chunks applied. \(\eta_{\text{util}}\) is calculated by dividing the total number of times the chunks instantiations were applied (\(N_{\text{appl}}\)) by the number of chunks produced that were eventually applied at least once. \(\eta_{\text{util}}\) can theoretically range from 0.0 (no produced chunks were applied in the same trial) to an arbitrary upper bound of multiple instantiations of produced chunks. In essence, it is the average firing per applied chunk. An analysis of the chunk firings (summarized in prior Table 3) revealed that a subset of the created chunks accounted for all applications in the trial. For the Low volume case, Dispatcher-Soar used 88 chunks (of the 706 created) a total of 178 times, yielding a \(\eta_{\text{util}} = 2.02\) firings/chunk. For the High volume case, Dispatcher-Soar applied 100 chunks (of the 1390 created) a total of 657 times, yielding a \(\eta_{\text{util}} = 6.57\) firings/chunk. Although Dispatcher-Soar experienced a decline in general production efficiency (\(\eta_{\text{prod}}\)) with the higher volume, there was a greater level of activity for each chunk applied (\(\eta_{\text{util}}\)). Consistent with the production efficiency observation, the core knowledge is more active in the High volume case, because the High volume case has more recurrences of a situation to which the core knowledge can be applied.

The third knowledge metric, \(\eta_{\text{eff}}\), measures the knowledge effectiveness of the applied chunks and is determined by dividing the effort saved (expressed as the reduction in decision cycles, \(D_c\)) by the number of unique chunks produced and eventually applied in the trial. In the numerator, \(D_c^e\) represents the number of decision cycles required to solve the case without learning, while \(D_c^l\) is the number of decision cycles required with learning. \(\eta_{\text{eff}}\) defines a "decision cycle saved per chunk" number that reflects the average within-trial contribution for the chunks applied. \(\eta_{\text{eff}}\) can theoretically range from 0.0 (no reduction in \(D_c^e\)) and approach \(\eta_{\text{eff}} = \frac{D_c^e}{\sum_{1 \leq k \leq N} [\mathcal{A}(k)]}\) as \(D_c^e\) declines through learning. \(\eta_{\text{eff}}\) can be interpreted as the average amount of "work" done (as \(D_c\)) per unit (chunk) applied. In the Low volume case, there was a reduction of 659\(D_c\) through the application of 88 unique chunks, yielding a \(\eta_{\text{eff}} = 7.48\)\(D_c\) per chunk. Each unique chunk generated (and applied) effectively contributed a 7.48 reduction in \(D_c\). In the High volume case, there was a total improvement of 1549\(D_c\), through the application of 100 unique and applied chunks, yielding a \(\eta_{\text{eff}} = 15.49\)\(D_c\) per chunk. With the higher volume, Dispatcher-Soar got more use (\(\eta_{\text{util}}\)) and more effective use (\(\eta_{\text{eff}}\)) of its chunks. In the High volume case, Dispatcher-Soar leverages more of the core knowledge it has learned. With these metrics, we have found that in a scheduling process as this, there is are highly repetitive components and, once the relevant core knowledge is gained regarding these components, the problem solving effort can be reduced.

5. Conclusion
We have demonstrated how a mechanism for general intelligence can be used to realize a simple scheduling process capable of learning. On a more specific level, we have also demonstrated how a particular approach to a scheduling process, a constraint-propagation approach, can be realized in this architecture and how that approach can support a self-directed process improvement under problem scale-up. The process changed itself, albeit in a small and constrained manner. What changed was the sequence of events responding to particular
jobs. This corresponds to learning to recognize a situation, crafting a template of a preferred approach at a higher level, and applying this template over and over again. In our study, most of the high-level behavior is contained in problem space SCHEDULE-PS as core knowledge.

Scheduling, like many distributed processes, involves the articulation and resolution of constraints. The value in the approach lies in the ability to learn from the resolution and improve the process. As such, the foundation is provided for an explicit articulation of an intelligent scheduling process, the generation of organizational knowledge about scheduling (or the scheduling mix) from that process, and the organizational learning resulting from that scheduling knowledge.

The scheduling example also demonstrates how interdependent tasks with potentially conflicting goals are addressed. In this architecture, conflict is resolved based on knowledge and the default mechanisms of the architecture itself. Thus, if a conflict arises in task processes, it is articulated as a goal that must be achieved (i.e., resolved). As such, the full power of the architecture can be brought to bear on the goal. If process-resolution knowledge exists (e.g., negotiation strategies), then it can be applied toward that goal. In fact, the process-resolution knowledge can be augmented by the acquisition of new knowledge gained from each negotiation.

The results have shown existence and form of knowledge and learning in a business process. The example demonstrated how an intelligent business process could gain core knowledge and apply it under problem scale-up to reduce learning effort. In addition, the uniform architecture permitted explicit comparative measurement and evaluation of knowledge. This was accomplished through the development and application of a set of new organizational learning metrics. These metrics were applied to processes to measure aspects of process performance as well as process learning with considerably more precision than heretofore possible. We necessarily come to the hindmost question of a research report: So what?

The work reported in this study anticipates the eventual merging of two streams of information systems research, the automation of business processes, and the infusion of intelligence into automation, influencing a third stream, organizational learning. When information systems learn, things change. The perspective taken in this paper affords seamless integration into some theoretical positions, confusion from others, and opposition to a few.

In general, the perspective is sympathetic with views that cast the individual agent as the unit of analysis for organizational learning and behavior (Simon 1991). Brevity restricts examination to the first (i.e., sympathetic) category. For example, issues of organizational acquisition of skills (Nass 1994) and knowledge creation (Nonaka 1994) are explicitly represented in the Dispatcher-Soar productions and the generated chunks. Grammatical metaphors applied to organizational processes (Pentland 1994) are, of course, isomorphic to the expression of the computational architecture. The unified architecture of cognition upon which the Dispatcher-Soar model is based provides one of the most theoretically complete models of deliberation in cognitive science (Newell 1990); this is prima facie relevance for the emerging field of managerial and organizational cognition (Carley, Prietula and Lin 1996, Walsh 1995). Procedural memory interpretations of organizational routines (Cohen and Bacdayan 1994) are equivalent to the chunks produced by the Soar architecture. As the articulation of the processes are in the form of a computational system, it is amenable to interpretation (at some level) as a theory of the firm processes and behavior (Carley 1995). Furthermore, the nature of the computational approach is sympathetic in spirit, if not form, to more general methods of defining knowledge interchange formats (e.g., Genesereth and Fikes 1992, Gruber 1993). Finally, issues in coordination theory regarding parallel distributed computation (e.g., Malone and Crowston 1994) are influencing the form of process specification (Malone et al. 1993). It would be interesting to examine a process specification that could be expanded to include distributed agents continuously exchanging and coordinating knowledge of processes and of the coordination task itself.

We are moving into an era of information systems and organization theory, where systems learn, communicate, cooperate, collaborate, and change to accomplish organizational tasks and achieve (or define) organizational goals. The general similarities between organizations and computer systems have not gone
unnoticed (Malone 1990). When intelligent and adaptive components of organizations are computer systems, the envelope of comparison expands to examine (more validly) the similarities between organizations and cognition, though not necessarily human cognition. Then things get interesting.

Simon (1979) has noted that social science research often models its form on the natural sciences, such as physics (e.g., search for the economic equivalent for the laws of motion) or biology (e.g., search for metaphors from evolutionary biology), suggesting the latter as a more meaningful altar than the former. This has been reflected in the popularity of recent perspectives of organizational evolution and ecology (Aldrich 1979, Hannan and Freeman 1984, Nelson and Winter 1982). Yet, the development of strong computational forms suggest that new metaphors might be required, as computational systems and organizations are neither entirely natural, nor entirely bound by natural laws—they are artificial systems (Simon 1996).

When systems are artificial, the laws of behavior—the laws of how things work—may change fundamentally in a very short period of time (e.g., via new legislation, new technology). Artificial systems need not be burdened with vestigial structures and processes that laid claim to prior reproductive (so to speak) successes, and their proliferation has not retained the constancy of low modal complexity found in the biological domain (diluting the biological metaphor). The argument is not that organizations change instantly (nor should they). In fact, institutional theorists have provided us with numerous examples of institutional (read, artificial) constraints on change, such as legacy programs in information systems (Tapscott and Caston 1993) and grand-father clauses and rules in legislation (DiMaggio and Powell 1991). However, in actual organizations, even though change may be slowed by the presence of these institutional factors, the process of change in artificial systems reflects an attention to strategy not present in biological systems. As the rules change, it is unlikely that unqualified fundamental and universal laws of organizational behavior, structure and interaction will be found (diluting the physics metaphor).

What does seem to be happening is an acceleration and infusion of information processing components in organizations and markets. Structures and routines may disappear as optimal, self-organizing forms are constantly sought that negotiate among themselves, inventing new processes, contracts, decisions, and products. Time scales of context and analyses are being reduced by orders of magnitudes. Firms are competing in terms of their programs (and programmers) and their ability to exchange symbols (data, information, knowledge) at light speed. Properties of processes' behaviors are both contained in their individual structures and in the emergent nature of their interactions.

In practice, firms are starting to realize that knowledge matters (Davenport 1996). If knowledge matters, and knowledge is viewed as an asset and as a component process, then the future is likely to include more knowledge-generating engines as constituent elements of business processes. Traditional views of organizational learning might do well to consider embracing models of cognition, computation, and artificial intelligence, for it is from those arenas where innovative organizational research is emerging (e.g., Burton and Obel 1995, Gasser and Huhns 1989, Glance and Huberman 1994, Levitt et al. 1994).

The interesting thing in this work (as we see it) is not what we have learned about manufacturing (indeed, it was not constructed for that purpose), but what we have demonstrated about manufacturing learning. Like our biologist colleagues, we may be facing new issues of organizational ontology in determining what entities actually exist in our organizations, and "of what the natures and activities of such entities are" (Eldredge and Grene 1992, p. 2). Perhaps the chimerical metaphor of choice is a computational one, where Darwin's Mechanism is now replaced by Turing's Machine.\footnote{We would like to thank Dr. David Steier at the Price-Waterhouse Research Center and our other colleagues in the Soar Group, School of Computer Science, Carnegie Mellon University, for their helpful comments and ideas as this project evolved. This research was partially supported by NSF grant IRI-9634383 and an Old Gold Summer Grant from the University of Iowa. We also thank the anonymous reviewers for their valuable and specific suggestions to improve the manuscript.}

References
Argyris, C. and D. Schon, Organizational Learning, Addison-Wesley, Reading MA, 1978.


