R1-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture

PAUL S. ROSENBLOOM, JOHN E. LAIRD, JOHN McDERMOTT, ALLEN NEWELL, FELLOW, IEEE, AND EDMUND ORCIUCH

Abstract—**This paper presents an experiment in knowledge-intensive programming within a general problem-solving production-system architecture called *Soar*. In *Soar*, knowledge is encoded within a set of problem spaces, which yields a system capable of reasoning from first principles. Expertise consists of additional rules that guide complex problem-space searches and substitute for expensive problem-space operators. The resulting system uses both knowledge and search when relevant. Expertise knowledge is acquired either by having it programmed, or by a chunking mechanism that automatically learns new rules reflecting the results implicit in the knowledge of the problem spaces. The approach is demonstrated on the computer-system configuration task, the task performed by the expert system *R1*.**

Index Terms—**Chunking, computer configuration, deep and shallow reasoning, expert systems, general problem solving, knowledge acquisition, knowledge-intensive programming, problem spaces, production systems.**

I. INTRODUCTION

REPEATEDLY in the work on expert systems, domain-dependent *knowledge-intensive* methods are contrasted with domain-independent *general problem-solving methods* [8]. Expert systems such as *Mycin* [19] and *R1* [14] attain their power to deal with applications by being knowledge intensive. However, this knowledge characteristically relates aspects of the task directly to action consequences, bypassing more basic scientific or causal knowledge of the domain. We will call this direct task-to-action knowledge *expertise knowledge* (it has also been referred to as *surface knowledge* [3], [7]), acknowledging that no existing term is very precise. Systems that primarily use weak methods ([10], [15]), such as depth-first search and means-ends analysis, are characterized by their wide scope of applicability. However, they achieve this at the expense of efficiency, being seemingly unable to bring to bear the vast quantities of diverse task knowledge that allow an expert system to quickly arrive at problem solutions.

This paper describes *R1-Soar*, an attempt to overcome the limitations of both expert systems and general problem solvers by doing knowledge-intensive programming in a general weak-method problem-solving architecture. We wish to show three things: 1) a general problem-solving architecture can work at the knowledge-intensive (expert system) end of the problem-solving spectrum; 2) such a system can integrate basic reasoning and expertise; and 3) such a system can perform knowledge acquisition by automatically transforming computationally intensive problem solving into efficient expertise-level rules.

Our strategy is to show how *Soar*, a problem-solving production-system architecture ([9], [12]) can deal with a portion of *R1*—a large, rule-based expert system that configures Digital Equipment Corporation's VAX-11 and PDP-11 computer systems. A *base* representation in *Soar* consists of knowledge about the goal to be achieved and knowledge of the operators that carry out the search for the goal state. For the configuration task, this amounts to knowledge that detects when a configuration has been done and basic knowledge of the physical operations of configuring a computer. A system with a base representation is robust, being able to search for knowledge that it does not immediately know, but the search can be expensive.

Efficiency can be achieved by adding knowledge to the system that aids in the application of difficult operators and guides the system through combinatorially explosive searches. Expertise knowledge corresponds to this non-base knowledge. With little expertise knowledge, *Soar* is a domain-independent problem solver; with much expertise knowledge, *Soar* is a knowledge-intensive system. The efficient processing due to expertise knowledge replaces costly problem solving with base knowledge when possible. Conversely, incompleteness in the expertise leads back smoothly into search in the base system.

In *Soar*, expertise can be added to a base system either by hand crafting a set of expertise-level rules, or by automatic acquisition of the knowledge implicit in the base representation. Automatic acquisition of new rules is accomplished by *chunking*, a mechanism that has been shown to provide a model of human practice [16], [17], but is extended here to much broader types of learning.

In the remainder of this paper, we describe *R1* and *Soar*, present the structure of the configuration task as imple-

Manuscript received April 15, 1985. This work was supported by the Defense Advanced Research Projects Agency (DOD) under DARPA Order 3597, monitored by the Air Force Avionics Laboratory under Contracts F33615-81-K-1539 and N00039-83-C-0136, and by Digital Equipment Corporation. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US Government, or Digital Equipment Corporation.

P. S. Rosenbloom is with the Departments of Computer Science and Psychology, Stanford University, Stanford, CA 94305.

J. E. Laird is with the Xerox Palo Alto Research Center, Palo Alto, CA 94304.

J. McDermott and A. Newell are with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

E. Orciuch is with the Digital Equipment Corporation.

0162-8828/85/0900-0561$01.00 © 1985 IEEE

mented in *Soar*, look at the system's behavior to evaluate the claims of this work, and draw some conclusions.

II. *R1* and the Task for *R1-Soar*

R1 is an expert system for configuring computers [14]. It provides a suitable expert system for this experiment because: 1) it contains a very large amount of knowledge, 2) its knowledge is largely pure expertise in that it simply recognizes what to do at almost every juncture, and 3) it is a highly successful application of expert systems, having been in continuous use by Digital Equipment Corporation for over four years [1]. Currently written in *Ops5* [4], *R1* consists of a database of over 7000 component descriptions, and a set of about 3300 production rules partitioned into 321 subtasks. The primary problem-solving technique in *R1* is match–recognizing in a specific situation precisely what to do next. Where match is insufficient, *R1* employs specialized forms of generate and test, multistep look-ahead, planning in an abstract space, hill climbing, and backtracking.

Given a customer's purchase order, *R1* determines what, if any, modifications have to be made to the order for reasons of system functionality and produces a number of diagrams showing how the various components on the order are to be associated. In producing a complete configuration, *R1* performs a number of relatively independent subtasks; of these, the task of configuring unibus modules is by far the most involved. Given a partially ordered set of modules to be put onto one or more buses and a number of containers (backplanes, boxes, etc.), the unibus configuration task involves repeatedly selecting a backplane and placing modules in it until all of the modules have been configured. The task is knowledge intensive because of the large number of situation-dependent constraints that rule out various module placements. *R1-Soar* can currently perform more than half of this task. Since *R1* uses about one-third of its knowledge (1100 of its 3300 rules) in performing the unibus configuration task, *R1-Soar* has approximately one-sixth of the knowledge that it would require to perform the entire configuration task.

R1 approaches the unibus configuration task by laying out an abstract description of the backplane demands imposed by the next several modules and then recognizing which of the candidate backplanes is most likely to satisfy those demands. Once a backplane is selected on the basis of the abstract description, *R1* determines specific module placements on the basis of a number of considerations that it had previously ignored or not considered in detail. *R1-Soar* approaches the task somewhat differently, but for the most part makes the same judgments since it takes into account all but one of the six factors that *R1* takes into account. The parts of the unibus configuration task that *R1-Soar* does not yet know how to perform are mostly peripheral subtasks such as configuring empty backplanes after all of the modules have been placed and distributing boxes appropriately among cabinets. *R1* typically fires about 1000 rules in configuring a computer system; the part of the task that *R1-Soar* performs typically takes *R1* 80–90 rule firings, one-twelfth of the total number.[1] Since an order usually contains several backplanes, to configure a single backplane might take *R1* 20–30 rule firings, or about 3–4 s on a Symbolics 3600 Lisp machine.

III. *Soar*

Soar is a problem-solving system that is based on formulating all problem-solving activity as attempts to satisfy goals via heuristic search in problem spaces. A problem space consists of a set of *states* and a set of *operators* that transform one state into another. Starting from an initial state, the problem solver applies a sequence of operators in an attempt to reach a state that satisfies the goal (called a *desired* state). Each goal has associated with it a problem space within which goal satisfaction is being attempted, a current state in that problem space, and an operator which is to be applied to the current state to yield a new state. The search proceeds via *decisions* that change the current problem space, state, or operator. If the current state is replaced by a different state in the problem space—most often it is the state generated by the current operator, but it can also be the previous state, or others—normal within-problem-space search results.

The knowledge used to make these decisions is called *search control*. Because *Soar* performs all problem-solving activity via search in problem spaces, the act of applying search-control knowledge must be constrained to not involve problem solving. Otherwise, there would be an infinite regression in which making a decision requires the use of search control which requires problem solving in a problem space, which requires making a decision using search control, and so on. In *Soar*, search control is limited to *match*—direct recognition of situations. As long as the computation required to make a decision is within the limits of search control, and the knowledge required to make the decision exists, problem solving proceeds smoothly. However, *Soar* often works in domains where its search-control knowledge is either inconsistent or incomplete. Four difficulties can occur while deciding on a new problem space, state, or operator: there are no objects under consideration, all of the candidate objects are unviable, there is insufficient knowledge to select among two or more candidate objects, or there is conflicting information about which object to select. When *Soar* reaches a decision for which one of these difficulties occurs, problem solving reaches an *impasse* [2] and stops. *Soar's universal subgoaling* mechanism [9] detects the impasse and creates a subgoal whose purpose is to obtain the knowledge which will allow the decision to be made. For example, if more than one operator can be applied to a state, and the available knowledge does not prefer one over the others, an impasse occurs and a subgoal is created to find information leading to the selection of the appropriate one.

[1]This task requires a disproportionate share of knowledge—a sixth of the knowledge for a twelfth of the rule firings—because the unibus configuration task is more knowledge intensive than most of the other tasks *R1* performs.

Or, if an operator is selected which cannot be implemented directly in search control, an impasse occurs because there are no candidates for the successor state. A subgoal is created to apply the operator, and thus build the state that is the result of the operator.

A subgoal is attempted by selecting a problem space for it. Should a decision reach an impasse in this new problem space, a new subgoal would be created to deal with it. The overall structure thus takes the form of a goal–subgoal hierarchy. Moreover, because each new subgoal will have an associated problem space, *Soar* generates a hierarchy of problem spaces as well as a hierarchy of goals. The diversity of task domains is reflected in a diversity of problem spaces. Major tasks, such as configuring a computer, have a corresponding problem space, but so also do each of the various subtasks, such as placing a module into a backplane or placing a backplane into a box. In addition, problem spaces exist in the hierarchy for many types of tasks that often do not appear in a typical task–subtask decomposition, such as the selection of an operator to apply, the implementation of a given operator in some problem space, and a test of goal attainment.

Fig. 1 gives a small example of how subgoals are used in *Soar*. This is a subgoal structure that gets generated while trying to take steps in many task problem spaces. Initially (*A*), the problem solver is at state1 and must select an operator. If search control is unable to uniquely determine the next operator to apply, a subgoal is created to do the selection. In that subgoal (*B*), a *selection* problem space is used that reasons about the selection of objects from a set. In order to break the tie between objects, the selection problem space has operators to evaluate each candidate object.

When the information required to evaluate an operator (such as operator1 in the task space) is not directly available in search control (because, for example, it must be determined by further problem solving), the evaluation operator is accomplished in a new subgoal. In this subgoal (*C*), the original task problem space and state (state1) are selected. Operator1 is applied, creating a new state (state2). If an evaluation function (a rule in search control) exists for state2, it is used to compare operator1 to the other operators. When operator1 has been evaluated, the subgoal terminates, and then the whole process is repeated for the other two operators (operator2 and operator3 in *D* and *E*). If, for example, operator2 creates a state with a better evaluation than the other operators, it will be designated as better than them. The selection subgoal will terminate and the designation of operator2 will lead to its selection in the original task goal and problem space. At this point, operator2 is reapplied to state1 and the process continues (*F*).

Soar uses a monolithic production-system architecture—a modified version of *Ops5* [4] that admits parallel execution of all satisfied productions—to realize its search-control knowledge and to implement its simple operators (more complex operators are encoded as separate problem spaces that are chosen for the subgoals that arise when the

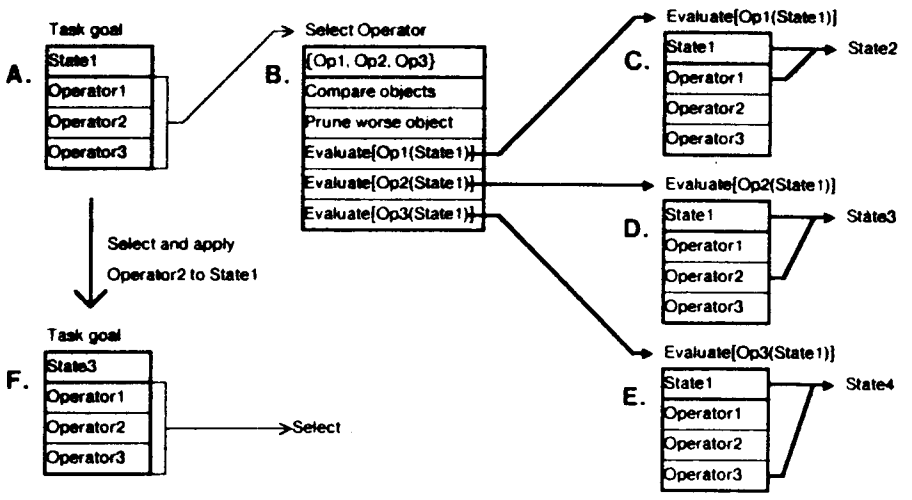


Fig. 1. A *Soar* subgoal structure. Each box represents one goal (the goal's name is above the box). The first row in each box is the current state for that goal. The remaining rows represent the operators that can be used for that state. Heavy arrows represent operator applications (and goals to apply operators). Light arrows represent subgoals to select among a set of objects.

operator they implement has been selected to apply). Production rules *elaborate* the current objects under consideration for a decision (e.g., candidate operators or states). The process of elaboration results in knowledge being added to the production system's *working memory* about the objects, including object substructures, evaluation information, and *preferences* relative to other candidate objects. There is a fixed decision process that integrates the preferences and makes a selection. Each decision corresponds to an elementary step in the problem solving, so a count of the number of decisions is a good measure of the amount of problem solving performed.

To have a task formulated in *Soar* is to have a problem space and the ability to recognize when a state satisfies the goal of the task; that is, is a desired state. The default behavior for *Soar*—when it has no search-control knowledge at all—is to search in this problem space until it reaches a desired state. The various weak methods arise, not by explicit representation and selection, but instead by the addition of small amounts of search control (in the form of one or two productions) to *Soar*, which acts as a *universal weak method* [10], [11], and [9]. These production rules are responsive to the small amounts of knowledge that are involved in the weak methods, e.g., the evaluation function in hill climbing or the difference between the current and desired states in means-ends analysis. In this fashion, *Soar* is able to make use of the entire repertoire of weak methods in a simple and elegant way, making it a good exemplar of a general problem-solving system.

The structure in Fig. 1 shows how one such weak method, steepest-ascent hill climbing—at each point in the search, evaluate the possible next steps and take the best one—can come about if the available knowledge is sufficient to allow evaluation of all of the states in the problem space. If slightly different knowledge is available, such as how to evaluate only *terminal* states (those states beyond which the search cannot extend), the search would be quite different, reflecting a different weak method. For example, if state2 in subgoal (*C*) cannot be evaluated, then subgoal (*C*) will not be satisfied and the search will continue under that subgoal. An operator must be selected for

of the problem-solving spectrum, 2) such a system can effectively integrate base reasoning and expertise, and 3) a chunking mechanism can aid in the process of knowledge acquisition by compiling computationally intensive problem solving into efficient expertise-level rules.

The approach to knowledge-intensive programming can be summarized by the following steps: 1) design a set of base problem spaces within which the task can be solved, 2) implement the problem-space operators as either rules or problem spaces, 3) operationalize the goals via a combination of rules that test the current state, generate search-control information and compute evaluation functions, and 4) improve the efficiency of the system by a combination of hand crafting more search control, using chunking, and developing evaluation functions that apply to more states.

References

[1] J. Bachant and J. McDermott, "R1 revisited: Four years in the trenches," *AI Mag.*, vol. 5, no. 3, 1984.

[2] J. S. Brown and K. VanLehn, "Repair theory: A generative theory of bugs in procedural skills," *Cogn. Sci.*, vol. 4, pp. 379–426, 1980.

[3] B. Chandrasekaran and S. Mittal, "Deep versus compiled knowledge approaches to diagnostic problem-solving," *Int. J. Man-Machine Studies*, vol. 19, pp. 425–436, 1983.

[4] C. L. Forgy, *OPS5 Manual.* Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. 81-135, 1981.

[5] C. Forgy, A. Gupta, A. Newell, and R. Wedig, "Initial assessment of architectures for production systems," in *Proc. Nat. Conf. Artif. Intell., Amer. Assoc. Artif. Intell.*, 1984.

[6] A. Gupta and C. Forgy, "Measurements on production systems," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. 83-167, Dec. 1983.

[7] P. E. Hart, "Directions for AI in the eighties," *SIGART Newsletter*, vol. 79, pp. 11–16, 1982.

[8] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, "An overview of expert systems," in *Building Expert Systems*, F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, Eds. Reading, MA: Addison-Wesley, 1983.

[9] J. E. Laird, "Universal subgoaling," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. 84-129, 1983.

[10] J. E. Laird and A. Newell, "A universal weak method," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. 83-141, June 1983.

[11] ——, "A universal weak method: Summary of results," in *Proc. 8th Int. Joint Conf. Artif. Intell.*, 1983.

[12] J. E. Laird, A. Newell, and P. S. Rosenbloom, *Soar: An Architecture for General Intelligence*, 1985, in preparation.

[13] J. E. Laird, P. S. Rosenbloom, and A. Newell, "Towards chunking as a general learning mechanism," in *Proc. Nat. Conf. Artif. Intell., Amer. Assoc. Artif. Intell.*, 1984.

[14] J. McDermott, "R1: A rule-based configurer of computer systems," *Artif. Intell.*, vol. 19, Sept. 1982.

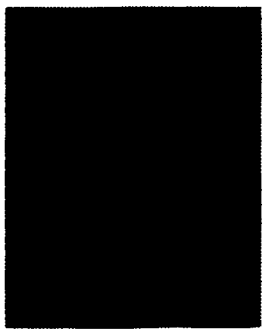
[15] A. Newell, "Heuristic programming: Ill-structured problems," *Progress in Operations Research, III*, J. Aronofsky, Ed. New York: Wiley, 1969.

[16] A. Newell and P. S. Rosenbloom, "Mechanisms of skill acquisition and the law of practice," in *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed. Hillsdale, NJ: Erlbaum 1981. Also in Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech Rep. 80-145, 1980.

[17] P. S. Rosenbloom, "The chunking of goal hierarchies: A model of practice and stimulus-response compatibility," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. 83-148, 1983.

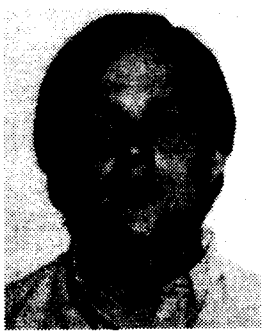
[18] P. S. Rosenbloom and A. Newell, "The chunking of goal hierarchies: A generalized model of practice," in *Machine Learning: An Artificial Intelligence Approach, Volume II*. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Los Altos, CA: Morgan Kaufmann, 1985, in press.

[19] E. H. Shortliffe, *Computer-Based Medical Consultation: MYCIN.* New York: Elsevier, 1976.



Paul S. Rosenbloom received the B.S. degree (Phi Beta Kappa) in mathematical sciences from Stanford University, Stanford, CA, in 1976 and the M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1978 and 1983, respectively, with fellowships from the National Science Foundation and IBM.

He did one year of graduate work in psychology at the University of California, San Diego, and was a Research Computer Scientist in the Department of Computer Science, Carnegie-Mellon University in 1983–1984. He is an Assistant Professor of Computer Science and Psychology at Stanford University. Primary research interests center around the nature of the cognitive architecture underlying artificial and natural intelligence. This work has included a model of human practice and developments toward its being a general learning mechanism (with J. E. Laird and A. Newell); a model of stimulus–response compatibility, and its use in the evaluation of problems in human–computer interaction (with B. John and A. Newell); and an investigation of how to do knowledge-intensive programming in a general, learning-problem solver. Other research interests have included the application of artificial intelligence techniques to the production of world-championship caliber programs for the game of Othello.®



John E. Laird received the B.S. degree in computer and communication sciences from the University of Michigan, Ann Arbor, in 1975 and the M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1978 and 1983, respectively.

He is a Research Associate in the Intelligent Systems Laboratory at Xerox Palo Alto Research Center, Palo Alto, CA. His primary research interest is the nature of intelligence, both natural and artificial. He is currently pursuing the structure of the underlying architecture of intelligence (with A. Newell and P. Rosenbloom). Significant aspects of this research include a theory of the weak methods, a theory of the origin of subgoals, a general theory of learning and a general theory of planning—all of which have been or are to be realized in the *Soar* architecture.

John McDermott, for a photograph and biography, see this issue, p. 522.

Allen Newell (SM'64–F'74) received the B.S. degree in physics from Stanford University, Stanford, CA, and the Ph.D. degree in industrial administration from Carnegie-Mellon University, Pittsburgh, PA, in 1957. He also studied mathematics at Princeton University, Princeton, NJ.

He worked at the Rand Corporation before joining Carnegie-Mellon University in 1961. He has worked on artificial intelligence and cognitive psychology since their emergence in the mid-1950's, mostly on problem solving and cognitive architectures, as well as list processing, computer architecture, human–computer interfaces, and psychologically based models of human–computer interaction. He is the U.A. and Helen Whitaker University Professor of Computer Science at Carnegie-Mellon University.

Dr. Newell received the Harry Goode Award of the American Federation of Information Processing Societies (AFIPS) and (with H. A. Simon) the A. M. Turing Award of the Association of Computing Machinery. He received the 1979 Alexander C. Williams Jr. Award of the Human Factors Society jointly with W. C. Biel, R. Chapman and J. L. Kennedy. He is a member of the National Academy of Sciences, the National Academy of Engineering, and other related professional societies. He was the First President of the American Association for Artificial Intelligence (AAAI).