CHAPTER 9

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

P. S. Rosenbloom, J. E. Laird, J. McDermott, A. Newell, and E. 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 implemented in *Soar*, look at the system's behavior to evaluate the claims of this work, and draw some conclusions.

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.

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

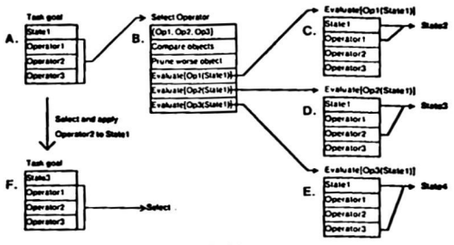

Fig. 1. A *Soar* subgoal structure. Each box represents one goal (the goal's name is above the box). The first row in each box is the current state for that goal. The remaining rows represent the operators that can be used for that state. Heavy arrows represent operator applications (and goals to apply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

state2, leading to a selection subgoal. The search will continue to deepen in this fashion until a terminal state is reached and evaluated. This leads to an exhaustive depth-first search for the best terminal state. Backtracking is not done explicitly, instead it implicitly happens whenever a subgoal terminates. A third weak method—depth-first search for the first desired state to be found—occurs when no evaluation information is available; that is, desired states can be recognized but no states can be evaluated.

In addition to the kinds of knowledge that lead to the well-known weak methods, additional search-control knowledge can be added to any problem space. The knowledge can be in the form of new object preferences or additional information that leads to new preferences. As more knowledge is added, the problem solving becomes more and more constrained until finally search is totally eliminated. This is the basic device in *Soar* to move toward a knowledge-intensive system. Each addition occurs simply by adding rules in the form of productions. Theoretically, *Soar* is able to move continuously from a knowledge-free solver (the default), through the weak methods to a knowledge-intensive system. It is possible to eliminate entire subspaces if their function can be realized by search-control knowledge in their superspace. For instance, if a subspace is to gather information for selecting between two operators, then it may be possible to encode that information directly as a search-control rule such as the following one from *R1-Soar:*

If there is an acceptable put-board-in-slot operator and an acceptable go-to-next-slot operator
Then the go-to-next-slot operator is worse than the put-board-in-slot operator.

Similarly, if a subspace is to apply an operator, then specific instances of that operator might be carried out directly by rules in the higher space.

Knowledge acquisition in *Soar* consists of the creation of additional rules by hand coding or by a mechanism that automatically *chunks* the results of successful goals [13], [12]. The chunking mechanism creates new production rules that allow the system to directly perform actions that originally required problem solving in subgoals. The conditions of a chunked rule test those aspects of the task that were relevant to satisfying the goal. For each goal generated, the architecture maintains a *condition list* of all data that existed before the goal was created and which were *accessed* in the goal. A datum is considered accessed if a production that matched it fires. Whenever a production is fired, all of the data it accessed that existed prior to the current goal are added to the goal's condition list. When a goal terminates (for whatever reason), the condition list for that goal is used to build the conditions of a chunk. The actions of the chunk generate the information that actually satisfied the goal. In figuring out the actions of the chunk, *Soar* starts with everything created in the goal, but then prunes away the information that does not relate directly to objects in any supergoal. What is left is turned into production actions.

New rules form part of search control when they deal with the selection among objects (chunks for goals that use the selection problem space), or they form part of operator implementation when they are chunks for goals dealing with problematic operators. Because *Soar* is driven by the goals automatically created to deal with difficulties in its performance, and chunking works for all goals, the chunking mechanism is applicable to all aspects of *Soar's* problem-solving behavior.

At first glance, chunking appears to be simply a caching mechanism with little hope of producing results that can be used on other than exact duplicates of tasks it has already attempted. However, if a given task shares subgoals with another task, a chunk learned for one task can apply to the other. Generality is possible because a chunk only contains conditions for the aspects that were accessed in the subgoal. This is an *implicit generalization*, by which many aspects of the context—the irrelevant ones—are automatically ignored by the chunk.[2]

IV. THE STRUCTURE OF *R1-SOAR*

The first step in building a knowledge-based system in *Soar* is to design and implement the base representation as a set of problem spaces within which the problem can be solved. As displayed in Fig. 2, *R1-Soar* currently consists of a hierarchy of ten task problem spaces (plus the selection problem space). These spaces represent a decomposition of the task in which the top space is given the goal to do the entire unibus configuration task; that is, to configure a sequence of modules to be put on a unibus. The other nine task spaces deal with subcomponents of this task. Each subspace implements one of the complex operators of its parent's problem space.

Each configuration task begins with a goal that uses the Unassigned Backplane problem space. This space has one operator for configuring a backplane that is instantiated with a parameter that determines which type of backplane is to be configured. The initial decision, of selecting which backplane to use next, appears as a choice between instances of this operator. Unless there is special search-control knowledge that knows which backplane should be used, no decision can be made. This difficulty (of indecision) leads to a subgoal that uses the selection problem space to evaluate the operators (by applying them to the original state and evaluating the resulting states). To do this, the evaluation operator makes recursive use of the Unassigned Backplane problem space.

The initial configuration of a backplane is accomplished in the five problem spaces rooted at the Configure Backplane space by: putting the backplane in a box (the Configure Box space), putting into the backplane as many modules as will fit (the Configure Modules space), reserving panel space in the cabinet for the module (Reserve Panel Space), putting the modules' boards into slots in the backplane (the Configure Boards space), and cabling the

[2] For comparisons of chunking to other related learning mechanisms such as memo functions, macrooperators, production composition, and analytical generalization, see [18] and [12].

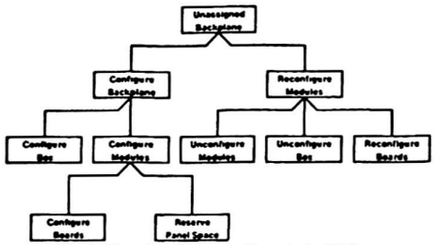

Fig. 2. The task problem-space hierarchy for *R1-Soar*.

backplane to the previous backplane (done by an operator in the Configure Backplane space that is simple enough to be done directly by a rule, rather than requiring a new problem space). Each of these problem spaces contains between one and five operators. Some of the operators are simple enough to be implemented directly by rules, such as the cable-backplane operator in the Configure Backplane space, or the put-board-in-slot, go-to-next-slot, and go-to-previous-slot operators in the Configure Boards space. Others are complex enough to require problem solving in new problem spaces, yielding the problem-space hierarchy seen in Fig. 2.

In addition to containing operators, each problem space contains the knowledge allowing it to recognize the satisfaction of the goal for that problem space. Several kinds of goal detection can occur: 1) recognition of a desired state, 2) satisfaction of path constraints (avoiding illegal sequences of operators), and 3) optimization over some criterion (such as maximizing the value of the result or minimizing its cost). All these different forms of goals are realized by appropriate production rules. For example, the Configure Backplane space simply has the following goal detection rule: if the modules have been placed in the backplane, and the backplane has been placed in a box, and the backplane has been cabled to the previous backplane (if there is one), then the goal is accomplished. In a more complicated case, the task of putting the boards from a module into slots in a backplane (the Configure Boards space) could be considered complete whenever all of the module's boards are associated with slots in the backplane. However, a two-board module can be configured by putting one board in the first slot and one in the last slot, or by putting the two boards into the first two slots, or by any one of the other combinatorial possibilities. For most modules it is desirable to put the boards as close to the front as possible to leave room for later modules (although there is one type of module that must go at the end of the backplane), so completed configurations are evaluated according to how much backplane space (first to last slot) they use. The goal is satisfied when the best completed configuration has been found.

In addition to the constraints handled by evaluation functions (such as using minimum backplane space), many other constraints exist in the configuration task that complicate the task of a problem-solving system. These include possible incompatibilities between boards and slots, the limited amounts of power that the boxes provide for use by modules (a new box may be required if more power is needed), components that are needed but not ordered, restrictions on the location of a module in a backplane (at the front or back), and limits on the electrical length of the unibus (for which a unibus repeater is required). *R1-Soar* pursues this complex configuration problem by searching for the best configuration that meets all of the constraints, and then trying to optimize the configuration some more by relaxing one of the constraints—the ordering relationship among the modules. This relaxation (occurring in the four spaces rooted at the Reconfigure Modules space) may allow the removal of backplanes that were added over and above those on the initial order. When possible, the modules configured in these backplanes are removed (the Unconfigure Modules space), placed into unused locations in other backplanes (the Reconfigure Boards space), and the extra backplanes are removed from their boxes (the Unconfigure Box space).

As described so far, *R1-Soar* forms a base-reasoning system because its representation and processing is in terms of the fundamental relationships between objects in the domain. The main mode of reasoning consists of search in a set of problem spaces until the goals are achieved. One part of this search can be seen clearly in the Configure Boards space. Given a module with two boards of width 6, and a nine-slot backplane with slot-widths of 4-6-6-6-6-6-6-6-4, a search proceeds through the problem space using the go-to-next-slot and put-board-in-slot operators. The search begins by making the easy decision of what to do with the first slot: it must be skipped because it is too narrow for either board. Then either one board can be placed in the second slot, or the slot can be skipped. If it is skipped, one board can be placed in the third slot, or it can be skipped, and so on. If, instead, a board is placed in the second slot, then it must go on to the third slot and decide whether to place the other board there or to skip the slot, and so on. All complete configurations are evaluated, and the path to the best one is selected. This is clearly not the most efficient way to solve this problem but it is sufficient.

R1-Soar becomes a more knowledge-intensive system as rules are added to guide the search through the problem space and to implement special cases of operators—although the complete operator is too complex for direct rule implementation, special cases may be amenable. Most of the hand-crafted knowledge in *R1-Soar* is used to control the search. In the Configure Boards space, all search is eliminated (at least for modules that go in the front of the backplane) by adding three search-control rules: 1) operators that configure boards of the same width and type are equal, 2) prefer the operator that configures the widest board that will fit in the current backplane slot, and 3) prefer an operator that puts a board into a slot over one that goes to the next slot. These rules convert the search in this problem space from being depth-first to algorithmic—at each step the system knows exactly what to do next. For the example above, the correct sequence is: go-to-next-slot, put-board-in-slot, go-to-next-slot, put-board-in-slot.

V. Results and Discussion

In this section we evaluate how well *R1-Soar* supports the three objectives given in the introduction by examining its performance on four configuration tasks.

1) There is one two-board module to be put on the unibus.

2) There are three modules to be put on the unibus. One of the already configured backplanes must be undone in order to configure a unibus repeater.

3) There are six modules to be put on the unibus. Three of the modules require panel space in the cabinet.

4) There are four modules to be put on the unibus. Three of the modules will go into a backplane already ordered, and one will go into a backplane that must be added to the order. Later, this module is reconfigured into an open location in the first backplane, allowing removal of the extra backplane from the configuration.

Most of the results to be discussed here are for tasks 1) and 2) which were done in earlier versions of both *Soar* and *R1-Soar* (containing only the Unassigned Backplane, Configure Backplane, Configure Box, Configure Modules, and Configure Boards spaces, for a total of 242 rules). Tasks 3) and 4) were run in the current versions of *Soar* and *R1-Soar* (containing all of the problem spaces, for a total of 266 rules).[3] Table I gives all of the results for these four tasks that will be used to evaluate the three objectives of this paper. The first line in the table shows that a system using a base representation can work, solving the rather simple task 1) after making 1731 decisions.

The first objective of this paper is to show that a general problem-solving system can work effectively at the knowledge-intensive end of the problem-solving spectrum. We examine three qualitatively different knowledge-intensive versions of *R1-Soar*, 1) where it has enough hand-crafted rules so that its knowledge is comparable to the level of knowledge in *R1* (before learning on the full version), 2) where there are rules that have been acquired by chunking (after learning on the base version), and 3) where both kinds of rules exist (after learning on the full version). The hand-crafted expertise consists solely of search control (operator selection) rules. The chunked expertise consists of both search-control and operator-application rules. In either case, this is expertise knowledge directly relating knowledge of the task domain to action in the task domain.

Table I shows the number of decisions required to complete each of the four configuration tasks when these three versions of *R1-Soar* are used. With hand-crafted search control, all four tasks were successfully completed, taking between 150 and 628 decisions. In the table, this is before learning on the full (search control) version. With just chunked search control, task 1) was accomplished in 7 decisions (after learning on the base version).[4] A total of 3 of the 7 decisions deal with aspects outside the scope of the unibus configuration task (setting up the initial goal, problem space, and state). *Soar* takes about 1.4 s per decision, so this yields about 6 s for the configuration task—within a factor of 2 of the time taken by *R1*. It was not feasible to run the more complex task 2) without search control because the time required would have been enormous due to the combinatorial explosion—the first module alone could be configured in over 300 different ways. Tasks 3) and 4) were also more complicated than task 1), and were not attempted with the base version. With both hand-crafted and chunked search control, tasks 1)–3) required between 7 and 16 decisions (after learning on the full version). Task 4) learning had problems of overgeneralization. It should have learned that one module could not go in a particular backplane, but instead learned that the module could not go in any backplane. More discussion on overgeneralization in chunking can be found in [13].

TABLE I
NUMBER OF DECISIONS TO COMPLETION FOR THE FOUR UNIBUS CONFIGURATION TASKS. THE BASE VERSION [TASK 1)] CONTAINS 232 RULES, THE PARTIAL VERSION [TASKS 1) AND 2)] CONTAINS 234 RULES, AND THE FULL VERSION CONTAINS 242 RULES [TASKS 1) AND 2)], OR 266 RULES [TASKS 3) AND 4)]. THE NUMBER OF RULES LEARNED FOR EACH TASK IS SHOWN IN BRACKETS IN THE DURING-LEARNING COLUMN

Task	Version	Before Learning	During Learning	After Learning
1	Base	1731	485[59]	7
	Partial	243	111[14]	7
	Full	150	90[12]	7
2	Partial	1064	692[109]	16
	Full	479	344[53]	16
3	Full	288	143[20]	10
4	Full	628		

In summary for the first objective, *R1-Soar* is able to do the unibus configuration task in a knowledge-intensive manner. To scale this result up to a full expert system (such as all of *R1*) we must know: 1) whether the rest of *R1* is similar in its key characteristics to the portion already done, and 2) the effects of scale on a system built in *Soar*. With respect to the unibus configuration task being representative of the whole configuration task, qualitative differences between portions of *R1* would be expected to manifest themselves as differences in amount of knowledge or as differences in problem-solving methods. The task that *R1-Soar* performs is atypical in the amount of knowledge required, but requires more knowledge, not less—15.7 rules per subtask for *R1-Soar*'s task, versus 10.3 for the entire task. The problem-solving methods used for the unibus configuration task are typical of the rest of *R1*—predominantly match, supplemented fairly frequently with multistep look ahead. With respect to the scaling of *R1-*

[3] The difference in number of task rules between these two versions is actually higher because a number of the default (nontask) rules needed by earlier versions of *Soar* is no longer necessary.

[4] For these runs it was assumed that the top-most goal in the hierarchy never terminates, and therefore is not chunked. If this assumption were changed, then the number of decisions with chunked search control would most likely be reduced to 1.

Soar up to *R1*'s full task, *Ops5*, from which *Soar* is built, scales very well—time is essentially constant over the number of rules, and linear in the number of modifications (rather than the absolute size) of working memory [6]. Additional speed is also available in the form of the *Ops83* production-system architecture, which is at least 24 times faster than Lisp-based *Ops5* (on a VAX-780) [5], and a production-system machine currently being designed that is expected to yield a further multiplicative factor of between 40 and 160 [5], for a combined likely speedup of at least three orders of magnitude.

The second objective of this paper is to show how base reasoning and expertise can be combined in *Soar* to yield more expertise and a smooth transition to search in problem spaces when the expertise is incomplete. Toward this end we ran two more before-learning versions of *R1-Soar* on tasks 1) and 2): the base version, which has no search-control rules, and the partial version, which has two hand-crafted search-control rules. The base version sits at the knowledge-lean end of the problem-solving spectrum; the partial version occupies an intermediate point between the base system and the more knowledge-intensive versions already discussed.

Task 1) took 1731 decisions for the base version, and 243 decisions for the partial version. Examining the trace of the problem solving reveals that most of the search in the base version goes to figuring out how to put the one module into the backplane. For the 9-slot backplane (of which 7 slots were compatible wih the module's two boards), there are (7 choose 2) = 21 pairs of slots to be considered. The two search control rules added in the partial version have already been discussed in the previous section: 1) make operators that configure boards of equal size be equal, and 2) prefer to put a board in a slot rather than skip the slot. These two rules reduce the number of decisions required for this task by almost an order of magnitude. With the addition of these two search control rules, the second task could also be completed, requiring 1064 decisions.

In summary, the base system is capable of performing the tasks, albeit very slowly. If appropriate search control exists, search is reduced, lowering the number of decisions required to complete the task. If enough rules are added, the system acts like it is totally composed of expertise knowledge. When such knowledge is missing, as some is missing in the partial version, the system falls back on search in its problem spaces.

The third objective is to show that knowledge acquisition via *Soar*'s chunking mechanism could compile computationally intensive problem solving into efficient rules. In *Soar*, chunks are learned for all goals experienced on every trial, so for exact task repetition (as is the case here), all of the learning occurs on the first trial. The *during learning* column in Table I shows how many decisions were required on the trial where learning occurred. The bracketed number is the number of rules learned during that trial. These results show that learning can improve performance by a factor of about 1.5–3, even the first time a task is attempted. This reflects a large degree of within-trial transfer of learning; that is, a chunk learned in one situation is reused in a later situation during the same trial. Some of these new rules become part of search control, establishing preferences for operators or states. Other rules become part of the implementation of operators, replacing their original implementations as searches in subspaces, with efficient rules for the particular situations.

In task 3), for example, three operator-implementation chunks (comprising four rules) were learned and used during the first attempt at the task. Two of the chunks were for goals solved in the Configure Boards space. Leaving out some details, the first chunk says that if the module has exactly one board and it is of width six, and the next slot in the backplane is of width six, then put the board into the next slot and move the slot pointer forward one slot. This is a macrooperator which accomplishes what previously required two operators in a lower problem space. The second chunk says that if the module has two boards, both of width six, and the current slot is of width four (too small for either board), and the two subsequent slots are of width six, then place the boards in those slots, and point to the last slot of the three as the current slot. The third chunk is a more complex one dealing with the reservation of panel space.

Comparing the number of decisions required before learning and after learning reveals savings of between a factor of 20 and 200 for the four unibus configuration tasks. In the process, between 12 and 109 rules are learned. The number of rules to be learned is determined by the number of distinct subgoals that need to be satisfied. If many of the subgoals are similar enough that a few chunks can deal with all of them, then fewer rules must be learned. A good example of this occurs in the base version of task 1) where most of the subgoals are resolved in one problem space (the Configure Boards space). Likewise, a small amount of general hand-crafted expertise can reduce significantly the number of rules to be learned. For task 1) the base version plus 59 learned rules leads to a system with 291 rules, the partial version plus 14 learned rules has 248 rules, and the full version plus 12 learned rules has 254 rules (some of the search control rules in the full version do not help on this particular task). All three systems require the same number of decisions to process this configuration task.

In summary, chunking can generate new knowledge in the form of search-control and operator-implementation rules. These new rules can reduce the time to perform the task by nearly two orders of magnitude. For more complex tasks the benefits could be even larger. However, more work is required to deal with the problem of overgeneralization.

VI. Conclusion

By implementing a portion of the *R1* expert system with the *Soar* problem-solving architecture, we have provided evidence for three hypotheses: 1) a general problem-solving architecture can work at the knowledge-intensive end 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.

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).

Edmund Orcioch received the B.S. degree in mathematics from Worcester State College, Worcester, MA, in 1975.

He joined Digital in 1980, and helped lead Digital's pioneering effort in the area of expert systems applications. He was the Chief Knowledge Engineer on XCON, Digital's premier expert system that configures custom computer systems, and later Chief Knowledge Engineer on ISA, an expert system that schedules customer orders. He spent the 1983/1984 academic year at Carnegie-Mellon University, Pittsburgh, PA, as Visiting Scientist in the AI Apprenticeship Program sponsored by the Digital Equipment Corporation. In addition to his project contributions, he was one of the designers of Digital's Corporate AI Training program. He developed and taught a course on *OPS5*, a rule-based language used to build expert systems; and he has conducted seminars on expert systems design and knowledge engineering. He is currently with the Intelligent Systems Technology Group in Digital's AI Technology Center where he is working in the areas of expert systems architectures and knowledge acquisition.