Artificial Intelligence 47 (1991) 289–325
Elsevier

A preliminary analysis of the Soar architecture as a basis for general intelligence

Paul S. Rosenbloom*

Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, USA

John E. Laird

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA

Allen Newell

Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, USA

Robert McCarl

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA

Received May 1989

Abstract

Rosenbloom, P.S., J.E. Laird, A. Newell and R. McCarl, A primary analysis of the Soar architecture as a basis for general intelligence, Artificial Intelligence 47 (1991) 289–325.

In this article we take a step towards providing an analysis of the Soar architecture as a basis for general intelligence. Included are discussions of the basic assumptions underlying the development of Soar, a description of Soar cast in terms of the theoretical idea of multiple levels of description, an example of Soar performing multi-column subtraction, and three analyses of Soar: its natural tasks, the sources of its power, and its scope and limits

Introduction

The central scientific problem of artificial intelligence (AI) is to understand what constitutes intelligent action and what processing organizations are

* Much of the work on this article was done while the first author was affiliated with the Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

capable of such action. Human intelligence—which stands before us like a holy grail—shows to first observation what can only be termed *general intelligence*. A single human exhibits a bewildering diversity of intelligent behavior. The types of goals that humans can set for themselves or accept from the environment seem boundless. Further observation, of course, shows limits to this capacity in any individual—problems range from easy to hard, and problems can always be found that are too hard to be solved. But the general point is still compelling.

Work in AI has already contributed substantially to our knowledge of what functions are required to produce general intelligence. There is substantial, though certainly not unanimous, agreement about some functions that need to be supported: symbols and goal structures, for example. Less agreement exists about what mechanisms are appropriate to support these functions, in large part because such matters depend strongly on the rest of the system and on cost-benefit tradeoffs. Much of this work has been done under the rubric of AI tools and languages, rather than AI systems themselves. However, it takes only a slight shift of viewpoint to change from what is an aid for the programmer to what is structure for the intelligent system itself. Not all features survive this transformation, but enough do to make the development of AI languages as much substantive research as tool building. These proposals provide substantial ground on which to build.

The Soar project has been building on this foundation in an attempt to understand the functionality required to support general intelligence. Our current understanding is embodied in the Soar architecture [22, 26]. This article represents an attempt at describing and analyzing the structure of the Soar system. We will take a particular point of view—the description of Soar as a hierarchy of levels—in an attempt to bring coherence to this discussion.

The idea of analyzing systems in terms of multiple levels of description is a familiar one in computer science. In one version, computer systems are described as a sequence of levels that starts at the bottom with the device level and works up through the circuit level, the logic level, and then one or more program levels. Each level provides a description of the system at some level of abstraction. The sequence is built up by defining each higher level in terms of the structure provided at the lower levels. This idea has also recently been used to analyze human cognition in terms of levels of description [38]. Each level corresponds to a particular time scale, such as ~100 msec. and ~1 sec., with a new level occurring for each new order of magnitude. The four levels between ~10 msec. and ~10 sec. comprise the cognitive band (Fig. 1). The lowest cognitive level—at ~10 msec.—is the symbol-accessing level, where the knowledge referred to by symbols is retrievable. The second cognitive level—at ~100 msec.—is the level at which elementary deliberate operations occur; that is, the level at which encoded knowledge is brought to bear, and the most elementary choices are made. The third and fourth cognitive levels—at ~1 sec.

Rational Band	...	
	~10 sec.	Goal attainment
Cognitive Band	~1 sec.	Simple operator composition
	~100 msec.	Elementary deliberate operations
	~10 msec.	Symbol accessing
Neural Band	...	

Fig. 1. Partial hierarchy of time scales in human cognition.

and ~10 sec.—are the simple-operator-composition and goal-attainment levels. At these levels, sequences of deliberations can be composed to achieve goals. Above the cognitive band is the rational band, at which the system can be described as being goal oriented, knowledge-based, and strongly adaptive. Below the cognitive band is the neural band.

In Section 2 we describe Soar as a sequence of three cognitive levels: the memory level, at which symbol accessing occurs; the decision level, at which elementary deliberate operations occur; and the goal level, at which goals are set and achieved via sequences of decisions. The goal level is an amalgamation of the top two cognitive levels from the analysis of human cognition.

In this description we will often have call to describe mechanisms that are built into the architecture of Soar. The architecture consists of all of the fixed structure of the Soar system. According to the levels analysis, the correct view to be taken of this fixed structure is that it comprises the set of mechanisms provided by the levels underneath the cognitive band. For human cognition this is the neural band. For artificial cognition, this may be a connectionist band, though it need not be. This view notwithstanding, it should be remembered that it is the Soar architecture which is primary in our research. The use of the levels viewpoint is simply an attempt at imposing a particular, hopefully illuminating, theoretical structure on top of the existing architecture.

In the remainder of this paper we describe the methodological assumptions underlying Soar, the structure of Soar, an illustrative example of Soar's performance on the task of multi-column subtraction, and a set of preliminary analyses of Soar as an architecture for general intelligence.

1. Methodological assumptions

The development of Soar is driven by four methodological assumptions. It is not expected that these assumptions will be shared by all researchers in the field. However, the assumptions do help explain why the Soar system and project have the shapes that they do.

The first assumption is the utility of focusing on the cognitive band, as opposed to the neural or rational bands. This is a view that has traditionally

been shared by a large segment of the cognitive science community; it is not, however, shared by the connectionist community, which focuses on the neural band (plus the lower levels of the cognitive band), or by the logicist and expert-systems communities, which focus on the rational band. This assumption is not meant to be exclusionary, as a complete understanding of general intelligence requires the understanding of all of these descriptive bands.[1] Instead the assumption is that there is important work to be done by focusing on the cognitive band. One reason is that, as just mentioned, a complete model of general intelligence will require a model of the cognitive band. A second reason is that an understanding of the cognitive band can constrain models of the neural and rational bands. A third, more applied reason, is that a model of the cognitive band is required in order to be able to build practical intelligent systems. Neural-band models need the higher levels of organization that are provided by the cognitive band in order to reach complex task performance. Rational-band models need the heuristic adequacy provided by the cognitive band in order to be computationally feasible. A fourth reason is that there is a wealth of both psychological and AI data about the cognitive band that can be used as the basis for elucidating the structure of its levels. This data can help us understand what type of symbolic architecture is required to support general intelligence.

The second assumption is that general intelligence can most usefully be studied by not making a distinction between human and artificial intelligence. The advantage of this assumption is that it allows wider ranges of research methodologies and data to be brought to bear to mutually constrain the structure of the system. Our research methodology includes a mixture of experimental data, theoretical justifications, and comparative studies in both artificial intelligence and cognitive psychology. Human experiments provide data about performance universals and limitations that may reflect the structure of the architecture. For example, the ubiquitous power law of practice—the time to perform a task is a power-law function of the number of times the task has been performed—was used to generate a model of human practice [39, 55], which was later converted into a proposal for a general artificial learning mechanism [27, 28, 61]. Artificial experiments—the application of implemented systems to a variety of tasks requiring intelligence—provide sufficiency feedback about the mechanisms embodied in the architecture and their interactions [16, 51, 60, 62, 73]. Theoretical justifications attempt to provide an abstract analysis of the requirements of intelligence, and of how various architectural mechanisms fulfill those requirements [38, 40, 49, 54, 56]. Comparative studies, pitting one system against another, provide an evaluation of how well the respective systems perform, as well as insight about how the capabilities of one of the systems can be incorporated in the other [6, 50].

[1] Investigations of the relationship of Soar to the neural and rational bands can be found in [38, 49, 56].

The third assumption is that the architecture should consist of a small set of orthogonal mechanisms. All intelligent behaviors should involve all, or nearly all, of these basic mechanisms. This assumption biases the development of Soar strongly in the direction of uniformity and simplicity, and away from modularity [10] and toolkit approaches. When attempting to achieve a new functionality in Soar, the first step is to determine in what ways the existing mechanisms can already provide the functionality. This can force the development of new solutions to old problems, and reveal new connections—through the common underlying mechanisms—among previously distinct capabilities [53]. Only if there is no appropriate way to achieve the new functionality are new mechanisms considered.

The fourth assumption is that architectures should be pushed to the extreme to evaluate how much of general intelligence they can cover. A serious attempt at evaluating the coverage of an architecture involves a long-term commitment by an extensive research group. Much of the research involves the apparently mundane activity of replicating classical results within the architecture. Sometimes these demonstrations will by necessity be strict replications, but often the architecture will reveal novel approaches, provide a deeper understanding of the result and its relationship to other results, or provide the means of going beyond what was done in the classical work. As these results accumulate over time, along with other more novel results, the system gradually approaches the ultimate goal of general intelligence.

2. Structure of Soar

In this section we build up much of Soar's structure in levels, starting at the bottom with memory and proceeding up to decisions and goals. We then describe how learning and perceptual-motor behavior fit into this picture, and wrap up with a discussion of the default knowledge that has been incorporated into the system.

2.1. Level 1: Memory

A general intelligence requires a memory with a large capacity for the storage of knowledge. A variety of types of knowledge must be stored, including declarative knowledge (facts about the world, including facts about actions that can be performed), procedural knowledge (facts about how to perform actions, and control knowledge about which actions to perform when), and episodic knowledge (which actions were done when). Any particular task will require some subset of the knowledge stored in the memory. Memory access is the process by which this subset is retrieved for use in task performance.

The lowest level of the Soar architecture is the level at which these memory phenomena occur. All of Soar's long-term knowledge is stored in a single production memory. Whether a piece of knowledge represents procedural, declarative, or episodic knowledge, it is stored in one or more productions. Each production is a condition-action structure that performs its actions when its conditions are met. Memory access consists of the execution of these productions. During the execution of a production, variables in its actions are instantiated with values. Action variables that existed in the conditions are instantiated with the values bound in the conditions. Action variables that did not exist in the conditions act as generators of new symbols.

The result of memory access is the retrieval of information into a global working memory. The working memory is a temporary memory that contains all of Soar's short-term processing context. Working memory consists of an interrelated set of objects with attribute-value pairs. For example, an object representing a green cat named Fred might look like (object o025 ^name fred ^type cat ^color green). The symbol o025 is the identifier of the object, a short-term symbol for the object that exists only as long as the object is in working memory. Objects are related by using the identifiers of some objects as attributes and values of other objects.

There is one special type of working memory structure, the preference. Preferences encode control knowledge about the acceptability and desirability of actions, according to a fixed semantics of preference types. Acceptability preferences determine which actions should be considered as candidates. Desirability preferences define a partial ordering on the candidate actions. For example, a better (or alternatively, worse) preference can be used to represent the knowledge that one action is more (or less) desirable than another action, and a best (or worst) preference can be used to represent the knowledge that an action is at least as good (or as bad) as every other action.

In a traditional production-system architecture, each production is a problem-solving operator (see, for example, [42]). The right-hand side of the production represents some action to be performed, and the left-hand side represents the preconditions for correct application of the action (plus possibly some desirability conditions). One consequence of this view of productions is that the productions must also be the locus of behavioral control. If productions are going to act, it must be possible to control which one executes at each moment; a process known as conflict resolution. In a logic architecture, each production is a logical implication. The meaning of such a production is that if the left-hand side (the antecedent) is true, then so is the right-hand side (the consequent).[2] Soar's productions are neither operators nor implications. Instead, Soar's productions perform (parallel) memory retrieval. Each produc-

[2] The directionality of the implication is reversed in logic programming languages such as Prolog, but the point still holds.

tion is a retrieval structure for an item in long-term memory. The right-hand side of the rule represents a long-term datum, and the left-hand side represents the situations in which it is appropriate to retrieve that datum into working memory. The traditional production-system and logic notions of action, control, and truth are not directly applicable to Soar's productions. All control in Soar is performed at the decision level. Thus, there is no conflict resolution process in the Soar production system, and all productions execute in parallel. This all flows directly from the production system being a long-term memory. Soar separates the retrieval of long-term information from the control of which act to perform next.

Of course it is possible to encode knowledge of operators and logical implications in the production memory. For example, the knowledge about how to implement a typical operator can be stored procedurally as a set of productions which retrieve the state resulting from the operator's application. The productions' conditions determine when the state is to be retrieved—for example, when the operator is being applied and its preconditions are met. An alternative way to store operator implementation knowledge is declaratively as a set of structures that are completely contained in the actions of one or more productions. The structures describe not only the results of the operator, but also its preconditions. The productions' conditions determine when to retrieve this declarative operator description into working memory. A retrieved operator description must be interpreted by other productions to actually have an affect.

In general, there are these two distinct ways to encode knowledge in the production memory: procedurally and declaratively. If the knowledge is procedurally encoded, then the execution of the production reflects the knowledge, but does not actually retrieve it into working memory—it only retrieves the structures encoded in the actions. On the other hand, if a piece of knowledge is encoded declaratively in the actions of a production, then it is retrievable in its entirety. This distinction between procedural and declarative *encodings* of knowledge is distinct from whether the knowledge is declarative (represents facts about the world) or procedural (represents facts about procedures). Moreover, each production can be viewed in either way, either as a procedure which implicitly represents conditional information, or as the indexed storage of declarative structures.

2.2. Level 2: Decisions

In addition to a memory, a general intelligence requires the ability to generate and/or select a course of action that is responsive to the current situation. The second level of the Soar architecture, the decision level, is the level at which this processing is performed. The decision level is based on the memory level plus an architecturally provided, fixed, decision procedure. The

decision level proceeds in a two phase elaborate-decide cycle. During elaboration, the memory is accessed repeatedly, in parallel, until quiescence is reached; that is, until no more productions can execute. This results in the retrieval into working memory of all of the accessible knowledge that is relevant to the current decision. This may include a variety of types of information, but of most direct relevance here is knowledge about actions that can be performed and preference knowledge about what actions are acceptable and desirable. After quiescence has occurred, the decision procedure selects one of the retrieved actions based on the preferences that were retrieved into working memory and their fixed semantics.

The decision level is open both with respect to the consideration of arbitrary actions, and with respect to the utilization of arbitrary knowledge in making a selection. This openness allows Soar to behave in both plan-following and reactive fashions. Soar is following a plan when a decision is primarily based on previously generated knowledge about what to do. Soar is being reactive when a decision is based primarily on knowledge about the current situation (as reflected in the working memory).

2.3. Level 3: Goals

In addition to being able to make decisions, a general intelligence must also be able to direct this behavior towards some end; that is, it must be able to set and work towards goals. The third level of the Soar architecture, the goal level, is the level at which goals are processed. This level is based on the decision level. Goals are set whenever a decision cannot be made; that is, when the decision procedure reaches an impasse. Impasses occur when there are no alternatives that can be selected (*no-change* and *rejection* impasses) or when there are multiple alternatives that can be selected, but insufficient discriminating preferences exist to allow a choice to be made among them (*tie* and *conflict* impasses). Whenever an impasse occurs, the architecture generates the goal of resolving the impasse. Along with this goal, a new *performance context* is created. The creation of a new context allows decisions to continue to be made in the service of achieving the goal of resolving the impasse—nothing can be done in the original context because it is at an impasse. If an impasse now occurs in this subgoal, another new subgoal and performance context are created. This leads to a goal (and context) stack in which the top-level goal is to perform some task, and lower-level goals are to resolve impasses in problem solving. A subgoal is terminated when either its impasse is resolved, or some higher impasse in the stack is resolved (making the subgoal superfluous).

In Soar, all symbolic goal-oriented tasks are formulated in problem spaces. A problem space consists of a set of states and a set of operators. The states represent situations, and the operators represent actions which when applied to states yield other states. Each performance context consists of a goal, plus roles

for a problem state, a state, and an operator. Problem solving is driven by decisions that result in the selection of problem spaces, states, and operators for their respective context roles. Given a goal, a problem space should be selected in which goal achievement can be pursued. Then an initial state should be selected that represents the initial situation. Then an operator should be selected for application to the initial state. Then another state should be selected (most likely the result of applying the operator to the previous state). This process continues until a sequence of operators has been discovered that transforms the initial state into a state in which the goal has been achieved. One subtle consequence of the use of problem spaces is that each one implicitly defines a set of constraints on how the task is to be performed. For example, if the Eight Puzzle is attempted in a problem space containing only a slide-tile operator, all solution paths maintain the constraint that the tiles are never picked up off of the board. Thus, such conditions need not be tested for explicitly in desired states.

Each problem solving decision—the selection of a problem space, a state, or an operator—is based on the knowledge accessible in the production memory. If the knowledge is both correct and sufficient, Soar exhibits highly controlled behavior; at each decision point the right alternative is selected. Such behavior is accurately described as being algorithmic or knowledge-intensive. However, for a general intelligence faced with a broad array of unpredictable tasks, situations will arise—inevitably and indeed frequently—in which the accessible knowledge is either incorrect or insufficient. It is possible that correct decisions will fortuitously be made, but it is more likely that either incorrect decisions will be made or that impasses will occur. Under such circumstances search is the likely outcome. If an incorrect decision is made, the system must eventually recover and get itself back on a path to the goal, for example, by backtracking. If instead an impasse occurs, the system must execute a sequence of problem space operators in the resulting subgoal to find (or generate) the information that will allow a decision to be made. This processing may itself be highly algorithmic, if enough control knowledge is available to uniquely determine what to do, or it may involve a large amount of further search.

As described earlier, operator implementation knowledge can be represented procedurally in the production memory, enabling operator implementation to be performed directly by memory retrieval. When the operator is selected, a set of productions execute that collectively build up the representation of the result state by combining data from long-term memory and the previous state. This type of implementation is comparable to the conventional implementation of an operator as a fixed piece of code. However, if operator implementation knowledge is stored declaratively, or if no operator implementation knowledge is stored, then a subgoal occurs, and the operator must be implemented by the execution of a sequence of problem space operators in the subgoal. If a declarative description of the to-be-implemented

operator is available, then these lower operators may implement the operator by interpreting its declarative description (as was demonstrated in work on task acquisition in Soar [61]). Otherwise the operator can be implemented by decomposing it into a set of simpler operators for which operator implementation knowledge is available, or which can in turn be decomposed further.

When an operator is implemented in a subgoal, the combination of the operator and the subgoal correspond to the type of deliberately created subgoal common in AI problem solvers. The operator specifies a task to be performed, while the subgoal indicates that accomplishing the task should be treated as a goal for further problem solving. In complex problems, like computer configuration, it is common for there to be complex high-level operators, such as Configure-computer which are implemented by selecting problem spaces in which they can be decomposed into simpler tasks. Many of the traditional goal management issues—such as conjunction, conflict, and selection—show up as operator management issues in Soar. For example, a set of conjunctive subgoals can be ordered by ordering operators that later lead to impasses (and subgoals).

As described in [54], a subgoal not only represents a subtask to be performed, but it also represents an introspective act that allows unlimited amounts of meta-level problem-space processing to be performed. The entire working memory—the goal stack and all information linked to it—is available for examination and augmentation in a subgoal. At any time a production can examine and augment any part of the goal stack. Likewise, a decision can be made at any time for any of the goals in the hierarchy. This allows subgoal problem solving to analyze the situation that led to the impasse, and even to change the subgoal, should it be appropriate. One not uncommon occurrence is for information to be generated within a subgoal that instead of satisfying the subgoal, causes the subgoal to become irrelevant and consequently to disappear. Processing tends to focus on the bottom-most goal because all of the others have reached impasses. However, the processing is completely opportunistic, so that when appropriate information becomes available at a higher level, processing at that level continues immediately and all lower subgoals are terminated.

2.4. Learning

All learning occurs by the acquisition of chunks—productions that summarize the problem solving that occurs in subgoals [28]. The actions of a chunk represent the knowledge generated during the subgoal; that is, the results of the subgoal. The conditions of the chunk represent an access path to this knowledge, consisting of those elements of the parent goals upon which the results depended. The results of the subgoal are determined by finding the elements generated in the subgoal that are available for use in subgoals—an

element is a result of a subgoal precisely because it is available to processes outside of the subgoal. The access path is computed by analyzing the traces of the productions that fired in the subgoal—each production trace effectively states that its actions depended on its conditions. This dependency analysis yields a set of conditions that have been implicitly generalized to ignore irrelevant aspects of the situation. The resulting generality allows chunks to transfer to situations other than the one in which it was learned. The primary system-wide effect of chunking is to move Soar along the space-time trade-off by allowing relevantly similar future decisions to be based on direct retrieval of information from memory rather than on problem solving within a subgoal. If the chunk is used, an impasse will not occur, because the required information is already available.

Care must be taken to not confuse the power of chunking as a learning mechanism with the power of Soar as a learning system. Chunking is a simple goal-based, dependency-tracing, caching scheme, analogous to explanation-based learning [4, 36, 50] and a variety of other schemes [55]. What allows Soar to exhibit a wide variety of learning behaviors are the variations in the types of subgoals that are chunked; the types of problem solving, in conjunction with the types and sources of knowledge, used in the subgoals; and the ways the chunks are used in later problem solving. The role that a chunk will play is determined by the type of subgoal for which it was learned. State-no-change, operator-tie, and operator-no-change subgoals lead respectively to state augmentation, operator selection, and operator implementation productions. The content of a chunk is determined by the types of problem solving and knowledge used in the subgoal. A chunk can lead to skill acquisition if it is used as a more efficient means of generating an already generatable result. A chunk can lead to knowledge acquisition (or knowledge level learning [5]) if it is used to make old/new judgments; that is, to distinguish what has been learned from what has not been learned [52, 53, 56].

2.5. Perception and motor control

One of the most recent functional additions to the Soar architecture is a perceptual-motor interface [75, 76]. All perceptual and motor behavior is mediated through working memory; specifically, through the state in the top problem solving context. Each distinct perceptual field has a designated attribute of this state to which it adds its information. Likewise, each distinct motor field has a designated attribute of the state from which it takes its commands. The perceptual and motor systems are autonomous with respect to each other and the cognitive system.

Encoding and decoding productions can be used to convert between the high-level structures used by the cognitive system, and the low-level structures used by the perceptual and motor systems. These productions are like ordinary

productions, except that they examine only the perceptual and motor fields, and not any of the rest of the context stack. This autonomy from the context stack is critical, because it allows the decision procedure to proceed without waiting for quiescence among the encoding and decoding productions, which may never happen in a rapidly changing environment.

2.6. *Default knowledge*

Soar has a set of productions (55 in all) that provide default responses to each of the possible impasses that can arise, and thus prevent the system from dropping into a bottomless pit in which it generates an unbounded number of content-free performance contexts. Figure 2 shows the default production that allows the system to continue if it has no idea how to resolve a conflict impasse among a set of operators. When the production executes, it rejects all of the conflicting operators. This allows another candidate operator to be selected, if there is one, or for a different impasse to arise if there are no additional candidates. This default response, as with all of them, can be overridden by additional knowledge if it is available.

One large part of the default knowledge (10 productions) is responsible for setting up operator subgoaling as the default response to no-change impasses on operators. That is, it attempts to find some other state in the problem space to which the selected operators can be applied. This is accomplished by generating acceptable and worst preferences in the subgoal for the parent problem space. If another problem space is suggested, possibly for implementing the operator, it will be selected. Otherwise, the selection of the parent problem space in the subgoal enables operator subgoaling. A sequence of operators is then applied in the subgoal until a state is generated that satisfies the preconditions of an operator higher in the goal stack.

Another large part of the default knowledge (33 productions) is responsible for setting up lookahead search as the default response to tie impasses. This is accomplished by generating acceptable and worst preferences for the *selection* problem space. The selection problem space consists of operators that evaluate the tied alternatives. Based on the evaluations produced by these operators, default productions create preferences that break the tie and resolve the impasse. In order to apply the evaluation operators, domain knowledge must exist that can create an evaluation. If no such knowledge is available, a second impasse arises—a no-change on the evaluation operator. As mentioned earlier,

```
If there is an impasse because of an operator conflict
   and there are no candidate problem spaces available
then reject the conflicting operators.
```

Fig. 2. A default production.

the default response to an operator no-change impasse is to perform operator subgoaling. However, for a no-change impasse on an evaluation operator this is overridden and a lookahead search is performed instead. The results of the lookahead search are used to evaluate the tied alternatives.

As Soar is developed, it is expected that more and more knowledge will be included as part of the basic system about how to deal with a variety of situations. For example, one area on which we are currently working is the provision of Soar with a basic arithmetical capability, including problem spaces for addition, multiplication, subtraction, division, and comparison. One way of looking at the existing default knowledge is as the tip of this large iceberg of background knowledge. However, another way to look at the default knowledge is as part of the architecture itself. Some of the default knowledge—how much is still unclear—must be innate rather than learned. The rest of the system's knowledge, such as the arithmetic spaces, should then be learnable from there.

3. Example: multi-column subtraction

Multi-column subtraction is the task we will use to demonstrate Soar. This task has three advantages. First, it is a familiar and simple task. This allows the details of Soar not to be lost in the complexities of understanding the task. Second, previous work has been done on modeling human learning of subtraction in the Sierra architecture [71]. Our implementation is inspired by the Sierra framework. Third, this task appears to be quite different from many standard search-intensive tasks common in AI. On the surface, it appears difficult to cast subtraction within the problem-space framework of Soar—it is, after all, a procedure. One might also think that chunking could not learn such a procedure. However, in this example, we will demonstrate that multi-column subtraction can be performed by Soar and that important parts of the procedure can be learned through chunking.

There exist many different procedures for performing multi-column subtraction. Different procedures result in different behaviors, both in the order in which scratch marks—such as borrowing notations—are made and in the type of mistakes that might be generated while learning [72]. For simplicity, we will demonstrate the implementation of just one of the many possible procedures. This procedure uses a borrowing technique that recursively borrows from a higher-order column into a lower-order column when the top number in the lower-order column is less than the bottom number.

3.1. A hierarchical subtraction procedure

One way to implement this procedure is via the processing of a goal hierarchy that encodes what must be done. Figure 3 shows a subtraction goal

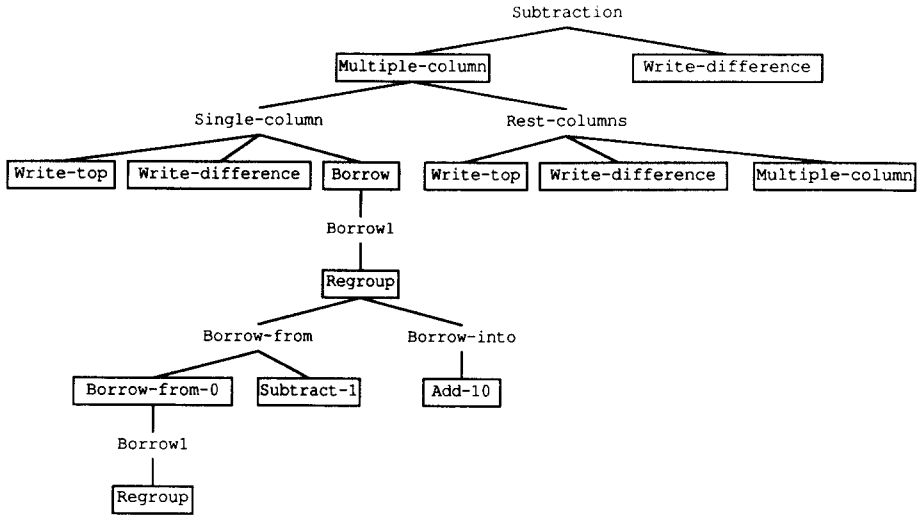

Fig. 3. A goal hierarchy for multi-column subtraction.

hierarchy that is similar to the one learned by Sierra.[3] Under each goal are shown the subgoals that may be generated while trying to achieve it. This Sierra goal hierarchy is mapped onto a hierarchy of operators and problem spaces in Soar (as described in Section 2). The boxed goals map onto operators and the unboxed goals map onto problem spaces. Each problem space consists of the operators linked to it from below in the figure. Operators that have problem spaces below them are implemented by problem solving in those problem spaces. The other operators are implemented directly at the memory level by productions (except for multiple-column and regroup, which are recursive). These are the primitive acts of subtraction, such as writing numbers or subtracting digits.

The states in these problem spaces contain symbolic representations of the subtraction problem and the scratch marks made on the page during problem solving. The representation is very simple and direct, being based on the spatial relationships among the digits as they would appear on a page. The state consists of a set of columns. Each column has pointers to its top and bottom digits. Additional pointers are generated when an answer for a column is produced, or when a scratch mark is made as the result of borrowing. The physical orientation of the columns on the page is represented by having "left" and "right" pointers from columns to their left and right neighbors. There is no inherent notion of multi-digit numbers except for these left and right relations between columns. This representation is consistent with the operators, which

[3] Sierra learned a slightly more elaborate, but computationally equivalent, procedure.

treat the problem symbolically and never manipulate multi-digit numbers as a whole.

Using this implementation of the subtraction procedure, Soar is able to solve all multi-column subtraction problems that result in positive answers. Unfortunately, there is little role for learning. Most of the control knowledge is already embedded in the productions that select problem spaces and operators. Within each problem space there are only a few operators from which to select. The preconditions of the few operators in each problem space are sufficient for perfect behavior. Therefore, goals arise only to implement operators. Chunking these goals produces productions that are able to compute answers without the intermediate subgoals.[4]

3.2. A single-space approach

One way to loosen up the strict control provided by the detailed problem-space/operator hierarchy in Fig. 3, and thus to enable the learning of the control knowledge underlying the subtraction procedure, is to have only a single subtraction problem space that contains all of the primitive acts (writing results, changing columns, and so on). Figure 4 contains a description of the

- *Operators*:
  Write-difference: If the difference between the top digit and the bottom digit of the current column is known, then write the difference as an answer to the current column.
  Write-top: If the lower digit of the current column is blank, then write the top digit as the answer to the current column.
  Borrow-into: If the result of adding 10 to the top digit of the current column is known, and the digit to the left of it has a scratch mark on it, then replace the top digit with the result.
  Borrow-from: If the result of subtracting 1 from the top digit in the current column is known, then replace that top digit with the result, augment it with a scratch mark and shift the current column to the right.
  Move-left: If the current column has an answer in it, shift the current column left.
  Move-borrow-left: If the current column does not have a scratch mark in it, shift the current column left.
  Subtract-two-digits: If the top digit is greater than or equal to the lower digit, then produce a result that is the difference.
  Subtract-1: If the top digit is not zero, then produce a result that is the top digit minus one.
  Add 10: Produce a result that is the top digit plus ten.
- *Goal Test*: If each column has an answer, then succeed.

Fig. 4. Primitive subtraction problem space.

[4] This work on subtraction was done in an earlier version of Soar that did not have the perceptual-motor interface described in Section 2. In that version, these chunks caused Soar to write out all of the column results and scratch marks in parallel—not very realistic motor behavior. To work around this problem, chunking was disabled for goals in this task during which environmental interactions occurred.

problem space operators and the goal test used in this second implementation. The operators can be grouped into four classes: the basic acts of writing answers to a single column problem (write-difference, write-top); borrow actions on the upper digits (borrow-into, borrow-from); moving from one column to the next (move-left, move-borrow-left); and performing very simple arithmetic computations (subtract-two-digits, subtract-1, add-10). With this simple problem space, Soar must learn the subtraction procedure by acquiring control knowledge that correctly selects operators.

Every operator in the subtraction problem space is considered for every state in the space. This is accomplished by having a production for each operator that generates an acceptable preference for it. The conditions of the production only test that the appropriate problem space (subtraction) is selected. Similar productions existed in the original implementation, except that those productions also contained additional tests which ensured that the operators would only be considered when they were the appropriate ones to apply.

In addition to productions which generate acceptable preferences, each operator has one or more productions which implement it. Although every operator is made acceptable for every state, an operator will actually be applied only if all of the conditions in the productions that implement it are satisfied. For example, write-difference will only apply if the difference between the top and bottom numbers is known. If an operator is selected, but the conditions of the productions that implement it are not satisfied, an impasse arises. As described in Section 2, the default response to this type of impasse is to perform operator subgoaling.

Figure 5 shows a trace of Soar's problem solving as it performs a simple two-column subtraction problem, after the learning of control knowledge has been completed. Because Soar's performance prior to learning on this problem is considerably more complicated, it is described after this simpler case. The

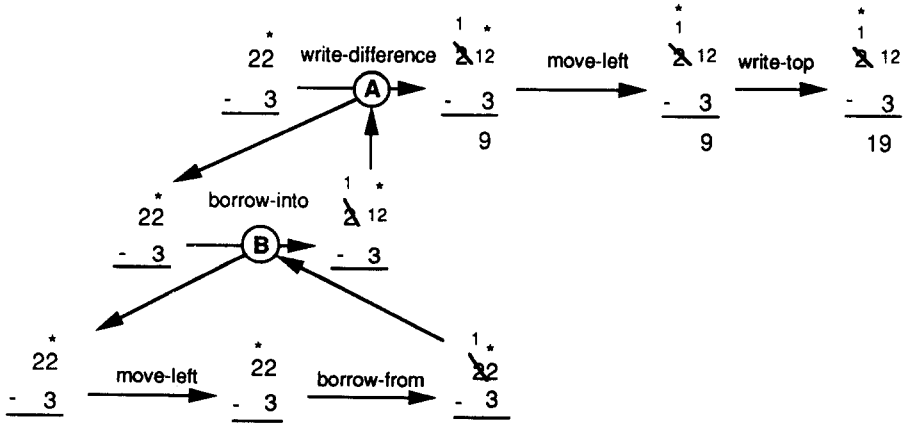

Fig. 5. Trace of problem solving after learning for 22 − 3.

top goal in this figure is to have the result of subtracting 3 from 22. Problem solving in the top goal proceeds from left to right, diving to a lower level whenever a subgoal is created in response to an impasse. Each state is a partially solved subtraction problem, consisting of the statement of the subtraction problem, a * designating the current column, and possibly column results and/or scratch marks for borrowing. Operator applications are represented by arrows going from left to right. The only impasses that occur in this trace are a result of the failure of operator preconditions—a form of operator no-change impasse. These impasses are designated by circles disrupting the operator-application arrows, and are labeled in the order they arise (A and B). For example, impasse A arises because write-difference cannot apply unless the lower digit in the current column (3) is less than the top digit (2).

For impasse A, operator subgoaling occurs when the subtraction problem space is selected in the subgoal. The preconditions of the write-difference operator are met when a state has been generated whose top digit has been changed from 2 to 12 (by borrowing). Once this occurs, the subgoal terminates and the operator applies, in this case writing the difference between 12 and 3. In this implementation of subtraction, operator subgoaling dynamically creates a goal hierarchy that is similar to the one programmed into the original implementation.

3.3. Performance prior to learning

Prior to learning, Soar's problem solving on this task is considerably more complicated. This added complexity arises because of an initial lack of knowledge about the results of simple arithmetic computations and a lack of knowledge about which operators should be selected for which states. Figure 6

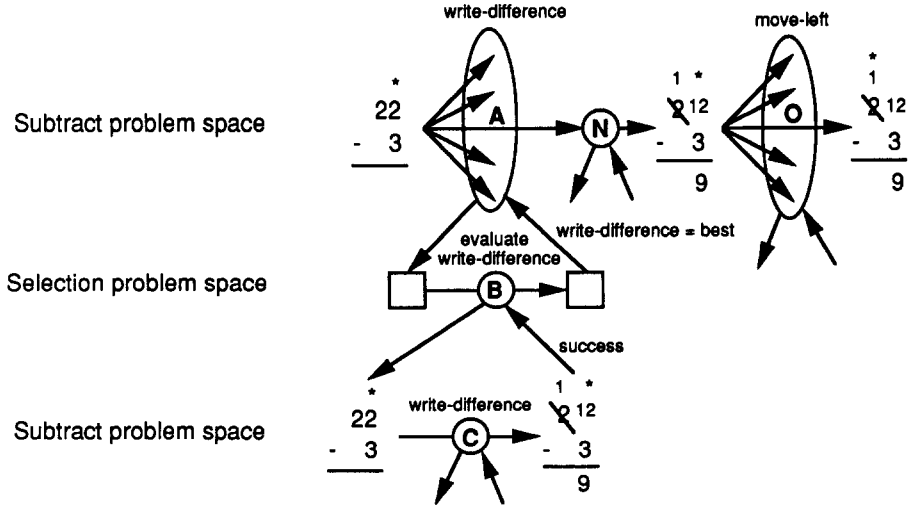

Fig. 6. Trace of problem solving before learning for 22 − 3.

shows a partial trace of Soar's pre-learning problem solving. Although many of the subgoals are missing, this small snapshot of the problem solving is characteristic of the impasses and subgoals that arise at all levels.

As before, the problem solving starts at the upper left with the initial state. As soon as the initial state is selected, a tie impasse (A) arises because all of the operators are acceptable and there are no additional preferences that distinguish between them. Default productions cause the selection space to be selected for this impasse. Within this space, operators are created to evaluate the tied operators. This example assumes that evaluate-object(write-difference) is selected, possibly based on advice from a teacher. Then, because there is no knowledge available about how to evaluate the subtraction operators, a no-change impasse (B) occurs for the evaluation operator. More default productions lead to a lookahead search by suggesting the original problem space (subtraction) and state and then selecting the operator that is being evaluated. The operator then applies, if it can, creating a new state. In this example, an operator subgoal impasse (C) arises when the attempt is made to apply the write-difference operator—its preconditions are not satisfied. Problem solving continues in this subgoal, requiring many additional impasses, until the write-difference operator can finally be applied. The lookahead search then continues until an evaluation is generated for the write-difference operator. Here, this happens shortly after impasse C is resolved. The system was given the knowledge that a state containing an answer for the current column is a (partial) success—such states are on the path to the goal. This state evaluation is then converted by default productions into an evaluation of "success" for the operator, and from there into a best preference for the operator. The creation of this preference breaks the operator tie, terminating the subgoals, and leading to the selection of the preferred operator (write-difference). The overall behavior of the system during this lookahead search is that of depth-first search—where backtracking occurs by subgoal termination—intertwined with operator subgoaling. Once this search is completed, further impasses (N) arise to actually apply the selected operator, but eventually, a solution is found.

One way in which multi-column subtraction differs from the classic AI search tasks is that the goal test is underspecified. As shown in Fig. 4, the goal test used here is that a result has been generated for each column of the problem. This determines whether some answer has been given for the problem, but is inadequate to determine whether the correct answer has been generated. The reason for this is that when solving a subtraction problem, the answer is in general not already available. It is theoretically (and practically) possible to use an addition procedure to test whether the subtraction procedure has generated the correct result. However, that corresponds to a deliberate strategy of "checking your work", rather than to the normal procedural goal test of determining whether the sequence of steps has been completed.

One consequence of having an underspecified goal test is that the combination of the problem space and goal test are not sufficient to ensure correct performance. Additional knowledge—the control knowledge which underlies the subtraction procedure—must also be provided in some form. VanLehn provided Sierra with worked-out examples which included the order in which the primitive external actions were to be performed [71]. The approach that we have taken is to provide advice to Soar [12] about which task operators it should evaluate first in the selection problem space. This ensures that the first answer generated during the lookahead search is the correct one.

3.4. Learning in subtraction

When chunking is used during subtraction problem solving, productions are created which reproduce the results of the subgoals in similar future situations. For the subgoals created because of tie impasses, the chunks create best preferences for the operators that led to the solution. These chunks essentially cache the results of the lookahead searches. A set of such chunks corresponds to a plan (or procedure)—they determine at every step what should be done—thus chunking converts Soar's behavior from search into plan (or procedure) following. When Soar is rerun on the same problem, the tie impasses do not arise and the solution is found directly, as in Fig. 5.

One important issue concerning the chunked productions is their generality. Does Soar only learn chunks that can apply to the exact same problem, or are the chunks general enough so that advice is no longer needed after a few subtraction problems have been completed? The answer is that the learned control chunks are quite general—so general that only one or two are required per operator. Once these chunks are acquired, Soar is able to solve perfectly all multi-column subtraction problems that have a positive answer. One sample control chunk for the borrow-into operator is shown in Fig. 7. Similar chunks are learned for each of the other major operators.

One reason for this generality is that operator subgoaling leads to a fine-grained goal hierarchy. There are a large number of relatively simple goals having to do with satisfying the preconditions of an operator. Because the problem solving for these goals is relatively minimal, the resulting chunks are quite general. A second reason for the generality of the learning is that the

```
If the super-operator is write-difference,
   and the bottom digit is greater than the top digit,
then make a best preference for borrow-into.
```

Fig. 7. A control chunk for borrow-into.

control chunks do not test for the specific digits used in the problems—if such tests were included, the chunks would transfer to many fewer problems.[5]

Though the control chunks that are learned are quite general, many specialized implementation chunks are also learned for the simple arithmetic operators. For example, the set of chunks that are eventually learned for the subtract-two-digits operator comprise a partial subtraction table for one- and two-digit numbers. Conceivably, these chunks could have been learned before multi-column subtraction is ever attempted—one may imagine that most of these simple digit manipulations are learned during earlier lessons on addition and single-column subtraction. Alternatively, these chunks can continue to be acquired as more multi-column subtraction problems are solved. The control chunks would all be acquired after a few trials, but learning of arithmetic knowledge would continue through later problems.

4. Analysis of Soar

There are a variety of analyses that could be performed for Soar. In this section we take our cue from the issues provided by the organizers of the 1987 Workshop on the Foundations of Artificial Intelligence [14]. We examine the set of tasks that are natural for Soar, the sources of its power, and its scope and limits.

4.1. Natural tasks

What does it mean for a task to be natural for an architecture? To answer this question we first must understand what a task is, and then what it means for such a task to be natural. By "task" we will mean any identifiable function, whether externally specified, or completely internal to the system. Computer configuration and maneuvering through an obstacle course are both tasks, and so are inheritance and skill acquisition. One way to define the idea of naturalness for a combination of a task and architecture is to say that a task is natural for an architecture if the task can be performed within the architecture without adding an extra level of interpretation within the software. This definition is appealing because it allows a distinction to be made between the tasks that the architecture can perform directly and those that can be done, but for which the architecture does not provide direct support. However, applying

[5] Chunking would include tests for the digits if their specific values were examined during the lookahead searches. However, the actual manipulation of the numbers is performed by the simple arithmetic operators: add-10, subtract-1 and subtract-two-digits. Before an operator such as write-difference is applied, an operator subgoal is created in which subtract-two-digits is selected and applied. The chunk for this subgoal reproduces the result whenever the same two digits are to be subtracted, eliminating the need for subtract-two-digits in such situations in the future. In the following lookahead searches, only pointers to the digits rather than the actual digits are ever tested, thereby leading to control chunks that are independent of the actual digits.

this definition is not without its problems. One problem is that, for any particular task, it is possible to replace the combination of an interpreter and its interpreted structures with a procedure that has the same effect. Some forms of learning—chunking, for example—do exactly this, by compiling interpreted structures into the structure of the interpreter. This has the effect of converting an unnatural task implementation into a natural one. Such a capability causes problems for the definition of naturalness—naturalness cannot be a fixed property of the combination of a task and an architecture—but it is actually a point is favor of architectures that can do such learning.

A second problem is that in a system that is itself built up in levels, as is Soar, different tasks will be performed at different levels. In Soar, tasks can be performed directly by the architecture, by memory retrieval, by a decision, or by goal-based problem solving. A task is implemented at a particular level if that level and all lower levels are involved, but the higher levels are not. For example, consider the task of inheritance. Inheritance is not directly implemented by the Soar architecture, but it can be implemented at the memory level by the firing of productions. This implementation involves the memory level plus the architecture (which implements the memory level), but not the decision or goal levels. Alternatively, inheritance could be implemented at the decision level, or even higher up at goal level. As the level of implementation increases, performance becomes more interpretive, but the model of computation explicitly includes all of these levels as natural for the system.

One way out of this problem is to have pre-theoretic notions about the level at which a particular task ought to be performable. The system is then natural for the task if it can be performed at that level, and unnatural if it must be implemented at a higher level. If, for example, the way inheritance works should be a function of the knowledge in the system, then the natural level for the capability is at the memory level (or higher).

In the remainder of this section we describe the major types of tasks that appear to us to be natural in Soar. Lacking any fundamental ways of partitioning the set of all tasks into principled categories, we will use a categorization based on four of the major functional capabilities of Soar: search-based tasks, knowledge-based tasks, learning tasks, and robotic tasks. The naturalness judgments for these task types are always based on assumptions about the natural level of implementation for a variety of subtasks within each type of task. We will try to be as clear as possible about the levels at which the subtasks are being performed, so that others may also be able to make these judgments for themselves.

4.1.1. Search-based tasks

Soar performs search in two qualitatively different ways: within context and across context. Within-context search occurs when Soar "knows" what to do at every step, and thus selects a sequence of operators and states without going

into a subgoal. If it needs to backtrack in within-context search, and the states in the problem space are internal (rather than states of the outside world), it can do so by reselecting a previously visited state. Within-context search corresponds to doing the task, without lookahead, and recovering if anything goes wrong. Across-context search occurs when the system doesn't know what to do, and impasses arise. Successive states in the search show up in successively lower contexts. Backtracking occurs by terminating subgoals. Across-context search corresponds to lookahead search, hypothetical scenario generation, or simulation.

Various versions of Soar have been demonstrated to be able to perform over 30 different search methods [21, 25, 26]. Soar can also exhibit hybrid methods—such as a combination of hill-climbing and depth-first search or of operator subgoaling and depth-first search—and use different search methods for different problem spaces within the same problem.

Search methods are represented in Soar as method increments—productions that contain a small chunk of knowledge about some aspect of a task and its action consequences. For example, a method increment might include knowledge about how to compute an evaluation function for a task, along with the knowledge that states with better evaluations should be preferred. Such an increment leads to a form of hill climbing. Other increments lead to other search methods. Combinations of increments lead to mixed methods.

The basic search abilities of making choices and generating subgoals are provided by the architecture. Individual method increments are at the memory level, but control occurs at the decision level, where the results of all of the method increments can be integrated into a single choice. Some search knowledge, such as the selection problem space, exists at the goal level.

4.1.2. Knowledge-based tasks

Knowledge-based tasks are represented in Soar as a collection of interacting problem spaces (as are all symbolic goal-oriented tasks). Each problem space is responsible for a part of the task. Problem spaces interact according to the different goal-subgoal relationships that can exist in Soar. Within each problem space, the knowledge is further decomposed into a set of problem space components, such as goal testing, state initialization, and operator proposal [77]. These components, along with additional communication constructs, can then be encoded directly as productions, or can be described in a high-level problem space language called TAQL [77], which is then compiled down into productions. Within this overall problem space organization, other forms of organization—such as object hierarchies with inheritance—are implementable at the memory level by multiple memory accesses. Task performance is represented at the goal level as search in problem spaces.

Several knowledge-based tasks have been implemented in Soar, including the R1-Soar computer configuration system [51], the Cypress-Soar and De-

signer-Soar algorithm design systems [60, 62], the Neomycin-Soar medical diagnosis system [73], and the Merl-Soar job-shop scheduling system [16].

These five knowledge-based systems cover a variety of forms of both construction and classification tasks. Construction tasks involve assembling an object from pieces. R1-Soar—in which the task is to construct a computer configuration—is a good example of a construction task. Classification tasks involve selecting from among a set of objects. Neomycin-Soar—in which the task is to diagnose an illness—is a good example of a classification task.[6] In their simplest forms, both construction and classification occur at the decision level. In fact, they both occur to some extent within every decision in Soar—alternatives must be assembled in working-memory and then selected. These capabilities can require trivial amounts of processing, as when an object is constructed by instantiating and retrieving it from memory. They can also involve arbitrary amounts of problem solving and knowledge, as when the process of operator-implementation (or, equivalently, state-construction) is performed via problem solving in a subgoal.

4.1.3. Learning tasks

The architecture directly supports a form of experiential learning in which chunking compiles goal-level problem solving into memory-level productions. Execution of the productions should have the same effect as the problem solving would have had, just more quickly. The varieties of subgoals for which chunks are learned lead to varieties in types of productions learned: problem space creation and selection; state creation and selection; and operator creation, selection, and execution. An alternative classification for this same set of behaviors is that it covers procedural, episodic and declarative knowledge [56]. The variations in goal outcomes lead to both learning from success and learning from failure. The ability to learn about all subgoal results leads to learning about important intermediate results, in addition to learning about goal success and failure. The implicit generalization of chunks leads to transfer of learned knowledge to other subtasks within the same problem (within-trial transfer), other instances of the same problem (across-trial transfer), and other problems (across-task transfer). Variations in the types of problems performed in Soar lead to chunking in knowledge-based tasks, search-based, and robotic tasks. Variations in sources of knowledge lead to learning from both internal and external knowledge sources. A summary of many of the types of learning that have so far been demonstrated in Soar can be found in [61].

The apparent naturalness of these various forms of learning depends primarily on the appropriateness of the required problem solving. Towards the natural end of the spectrum is the acquisition of operator selection productions, in

[6] In a related development, as part of an effort to map the Generic Task approach to expert system construction onto Soar, the Generic Task for classification by establish-refine has been implemented in Soar as a general problem space [17].

which the problem solving consists simply of a search with the set of operators for which selection knowledge is to be learned. Towards the unnatural end of the spectrum is the acquisition of new declarative knowledge from the outside environment. Many systems employ a simple store command for such learning, effectively placing the capability at the memory level. In Soar, the capability is situated two levels further up, at the goal level. This occurs because the knowledge must be stored by chunking, which can only happen if the knowledge is used in subgoal-based problem solving. The naturalness of this learning in Soar thus depends on whether this extra level of interpretation is appropriate or not. It turns out that the problem solving that enables declarative learning in Soar takes the form of an understanding process that relates the new knowledge to what is already known. The chunking of this understanding process yields the chunks that encode the new knowledge. If it is assumed that new knowledge should always be understood to be learned, then Soar's approach starts to look more natural, and verbatim storage starts to look more inappropriate.

4.1.4. Robotic tasks

Robotic tasks are performed in Soar via its perceptual-motor interface. Sensors autonomously generate working memory structures representing what is being sensed, and motor systems autonomously take commands from working memory and execute them. The work on robotics in Soar is still very much in its infancy; however, in Robo-Soar [30], Soar has been successfully hooked up to the combination of a camera and a Puma arm, and then applied to several simple blocks-world tasks.[7] Low-level software converts the camera signal into information about the positions, orientations and identifying characteristics of the blocks. This perceptual information is then input to working memory, and further interpreted by encoding productions. Decoding productions convert the high-level robot commands generated by the cognitive system to the low-level commands that are directly understood by the controller for the robot arm. These low-level commands are then executed through Soar's motor interface.

Given a set of operators which generate motor commands, and knowledge about how to simulate the operators and about the expected positions of blocks following the actions, Robo-Soar is able to successfully solve simple blocks-world problems and to learn from its own behavior and from externally provided advice. It also can make use of a general scheme for recovering from incorrect knowledge [23] to recover when the unexpected occurs—such as when the system fails in its attempt to pick up a triangular prism—and to learn to avoid the failure in the future. Robo-Soar thus mixes planning (lookahead

[7] The work on Robo-Soar has been done in the newest major release of Soar (version 5) [24, 63], which differs in a number of interesting ways from the earlier versions upon which the rest of the results in this article are based.

search with chunking), plan execution and monitoring, reactivity, and error recovery (with replanning). This performance depends on all of the major components of the architecture, plus general background knowledge—such as how to do lookahead search and how to recover from errors—and specific problem spaces for the task.

4.2. Where the power resides

Soar's power and flexibility arise from at least four identifiable sources. The first source of power is the universality of the architecture. While it may seem that this should go without saying, it is in fact a crucial factor, and thus important to mention explicitly. Universality provides the primitive capability to perform any computable task, but does not by itself explain why Soar is more appropriate than any other universal architecture for knowledge-based, search-based, learning, and robotic tasks.

The second source of power is the uniformity of the architecture. Having only one type of long-term memory structure allows a single, relatively simple, learning mechanism to behave as a general learning mechanism. Having only one type of task representation (problem spaces) allows Soar to move continuously from one extreme of brute-force search to the other extreme of knowledge-intensive (or procedural) behavior without having to make any representational decisions. Having only one type of decision procedure allows a single, relatively simple, subgoal mechanism to generate all of the types of subgoals needed by the system.

The traditional downside of uniformity is weakness and inefficiency. If instead the system were built up as a set of specialized modules or agents, as proposed in [10, 34], then each of the modules could be optimized for its own narrow task. Our approach to this issue in Soar has been to go strongly with uniformity—for all of the benefits listed above—but to achieve efficiency (power) through the addition of knowledge. This knowledge can either be added by hand (programming) or by chunking.

The third source of power is the specific mechanisms incorporated into the architecture. The production memory provides pattern-directed access to large amounts of knowledge; provides the ability to use strong problem solving methods; and provides a memory structure with a small-grained modularity. The working memory allows global access to processing state. The decision procedure provides an open control loop that can react immediately to new situations and knowledge; contributes to the modularity of the memory by allowing memory access to proceed in an uncontrolled fashion (conflict resolution was a major source of nonmodularity in earlier production systems); provides a flexible control language (preferences); and provides a notion of impasse that is used as the basis for the generation of subgoals. Subgoals focus the system's resources on situations where the accessible knowledge is

inadequate; and allow flexible meta-level processing. Problem spaces separate control from action, allowing them (control and action) to be reasoned about independently; provide a constrained context within which the search for a desired state can occur; provide the ability to use weak problem solving methods; and provide for straightforward responses to uncertainty and error (search and backtracking). Chunking acquires long-term knowledge from experience; compiles interpreted procedures into non-interpreted ones; and provides generalization and transfer. The perceptual-motor system provides the ability to observe and affect the external world in parallel with the cognitive activity.

The fourth source of power is the interaction effects that result from the integration of all of the capabilities within a single system. The most compelling results generated so far come about from these interactions. One example comes from the mixture of weak methods, strong methods, and learning that is found in systems like R1-Soar. Strong methods are based on having knowledge about what to do at each step. Because strong methods tend to be efficient and to produce high-quality solutions, they should be used whenever possible. Weak methods are based on searching to make up for a lack of knowledge about what should be done. Such methods contribute robustness and scope by providing the system with a fall-back approach for situations in which the available strong methods do not work. Learning results in the addition of knowledge, turning weak methods into strong ones. For example, in R1-Soar it was demonstrated how computer configuration could be cast as a search problem, how strong methods (knowledge) could be used to reduce search, how weak methods (subgoals and search) could be used to make up for a lack of knowledge, and how learning could add knowledge as the result of search.

Another interesting interaction effect comes from work on abstraction planning, in which a difficult problem is solved by first learning a plan for an abstract version of the problem, and then using the abstract plan to aid in finding a plan for the full problem [41, 57, 70, 69]. Chunking helps the abstraction planning process by recording the abstract plan as a set of operator-selection productions, and by acquiring other productions that reduce the amount of search required in generating a plan. Abstraction helps the learning process by allowing chunks to be learned more quickly—abstract searches tend to be shorter than normal ones. Abstraction also helps learning by enabling chunks to be more general than they would otherwise be—the chunks ignore the details that were abstracted away—thus allowing more transfer and potentially decreasing the cost of matching the chunks (because there are now fewer conditions).

4.3. Scope and limits

The original work on Soar demonstrated its capabilities as a general problem solver that could use any of the weak methods when appropriate, across a wide

range of tasks. Later, we came to understand how to use Soar as the basis for knowledge-based systems, and how to incorporate appropriate learning and perceptual-motor capabilities into the architecture. These developments increased Soar's scope considerably beyond its origins as a weak-method problem solver. Our ultimate goal has always been to develop the system to the point where its scope includes everything required of a general intelligence. In this section we examine how far Soar has come from its relatively limited initial demonstrations towards its relatively unlimited goal. This discussion is divided up according to the major components of the Soar architecture, as presented in Section 2: memory, decisions, goals, learning, and perception and motor control.

4.3.1. Level 1: Memory

The scope of Soar's memory level can be evaluated in terms of the amount of knowledge that can be stored, the types of knowledge that can be represented, and the organization of the knowledge.

Amount of knowledge. Using current technology, Soar's production memory can support the storage of thousands of independent chunks of knowledge. The size is primarily limited by the cost of processing larger numbers of productions. Faster machines, improved match algorithms and parallel implementations [13, 65, 66] may raise this effective limit by several orders of magnitude over the next few years.

Types of knowledge. The representation of procedural and propositional declarative knowledge is well developed in Soar. However, we don't have well worked-out approaches to many other knowledge representation problems, such as the representation of quantified, uncertain, temporal, and episodic knowledge. The critical question is whether architectural support is required to adequately represent these types of knowledge, or whether such knowledge can be adequately treated as additional objects and/or attributes. Preliminary work on quantified [43] and episodic [56] knowledge is looking promising.

Memory organization. An issue which often gets raised with respect to the organization of Soar's memory, and with respect to the organization of production memories in general, is the apparent lack of a higher-order memory organization. There are no scripts [59], frames [33], or schemas [1] to tie fragments of related memory together. Nor are there any obvious hierarchical structures which limit what sets of knowledge will be retrieved at any point in time. However, Soar's memory does have an organization, which is derived from the structure of productions, objects, and working memory (especially the context hierarchy).

What corresponds to a schema in Soar is an object, or a structured collection of objects. Such a structure can be stored entirely in the actions of a single production, or it can be stored in a piecemeal fashion across multiple productions. If multiple productions are used, the schema as a unit only comes into

existence when the pieces are all retrieved contemporaneously into working memory. The advantage of this approach is that it allows novel schemas to be created from fragments of separately learned ones. The disadvantage is that it may not be possible to determine whether a set of fragments all originated from a single schema.

What corresponds to a hierarchy of retrieval contexts in Soar are the production conditions. Each combination of conditions implicitly defines a retrieval context, with a hierarchical structure induced by the subset relationship among the combinations. The contents of working memory determines which retrieval contexts are currently in force. For example, problem spaces are used extensively as retrieval contexts. Whenever there is a problem solving context that has a particular problem space selected within it, productions that test for other problem space names are not eligible to fire in that context. This approach has worked quite well for procedural knowledge, where it is clear when the knowledge is needed. We have just begun to work on appropriate organizational schemes for episodic and declarative knowledge, where it is much less clear when the knowledge should be retrieved. Our initial approach has been based on the incremental construction, via chunking, of multi-production discrimination networks [53, 56]. Though this work is too premature for a thorough evaluation in the context of Soar, the effectiveness of discrimination networks in systems like Epam [7] and Cyrus [19] bodes well.

4.3.2. Level 2: Decisions

The scope of Soar's decision level can be evaluated in terms of its speed, the knowledge brought to bear, and the language of control.

Speed. Soar currently runs at approximately 10 decisions/second on current workstations such as a Sun4/280. This is adequate for most of the types of tasks we currently implement, but is too slow for tasks requiring large amounts of search or very large knowledge bases (the number of decisions per second would get even smaller than it is now). The principal bottleneck is the speed of memory access, which is a function of two factors: the cost of processing individually expensive productions (the *expensive chunks* problem) [67], and the cost of processing a large number of productions (the *average growth effect* problem) [64]. We now have a solution to the problem of expensive chunks which can guarantee that all productions will be cheap—the match cost of a production is at worst linear in the number of conditions [68]—and are working on other potential solutions. Parallelism looks to be an effective solution to the average growth effect problem [64].

Bringing knowledge to bear. Iterated, parallel, indexed access to the contents of long-term memory has proven to be an effective means of bringing knowledge to bear on the decision process. The limited power provided by this process is offset by the ability to use subgoals when the accessible knowledge is

inadequate. The issue of devising good access paths for episodic and declarative knowledge is also relevant here.

Control language. Preferences have proven to be a flexible means of specifying a partial order among contending objects. However, we cannot yet state with certainty that the set of preference types embodied in Soar is complete with respect to all the types of information which ultimately may need to be communicated to the decision procedure.

4.3.3. Level 3: Goals

The scope of Soar's goal level can be evaluated in terms of the types of goals that can be generated and the types of problem solving that can be performed in goals. Soar's subgoaling mechanism has been demonstrated to be able to create subgoals for all of the types of difficulties that can arise in problem solving in problem spaces [21]. This leaves three areas open. The first area is how top-level goals are generated; that is, how the top-level task is picked. Currently this is done by the programmer, but a general intelligence must clearly have grounds—that is, motivations—for selecting tasks on its own. The second area is how goal interactions are handled. Goal interactions show up in Soar as operator interactions, and are normally dealt with by adding explicit knowledge to avoid them, or by backtracking (with learning) when they happen. It is not yet clear the extent to which Soar could easily make use of more sophisticated approaches, such as non-linear planning [2]. The third area is the sufficiency of impasse-driven subgoaling as a means for determining when meta-level processing is needed. Two of the activities that might fall under this area are goal tests and monitoring. Both of these activities can be performed at the memory or decision level, but when they are complicated activities it may be necessary to perform them by problem solving at the goal level. Either activity can be called for explicitly by selecting a "monitor" or "goal-test" operator, which can then lead to the generation of a subgoal. However, goals for these tasks do not arise automatically, without deliberation. Should they? It is not completely clear.

The scope of the problem solving that can be performed in goals can itself be evaluated in terms of whether problem spaces cover all of the types of performance required, the limits on the ability of subgoal-based problem solving to access and modify aspects of the system, and whether parallelism is possible. These points are addressed in the next three paragraphs.

Problem space scope. Problem spaces are a very general performance model. They have been hypothesized to underlie all human, symbolic, goal-oriented behavior [37]. The breadth of tasks that have so far been represented in problem spaces over the whole field of AI attests to this generality. One way of pushing this evaluation further is to ask how well probem spaces account for the types of problem solving performed by two of the principal competing

paradigms: planning [2] and case-based reasoning [20].[8] Both of these paradigms involve the creation (or retrieval) and use of a data structure that represents a sequence of actions. In planning, the data structure represents the sequence of actions that the system expects to use for the current problem. In case-based reasoning, the data structure represents the sequence of actions used on some previous, presumably related, problem. In both, the data structure is used to decide what sequence of actions to perform in the current problem. Soar straightforwardly performs procedural analogues of these two processes. When it performs a lookahead search to determine what operator to apply to a particular state, it acquires (by chunking) a set of search control productions which collectively tell it which operator should be applied to each subsequent state. This set of chunks forms a procedural plan for the current problem. When a search control chunk transfers between tasks, a form of procedural case-based reasoning is occurring.

Simple forms of declarative planning and case-based reasoning have also been demonstrated in Soar in the context of an expert system that designs floor systems [47]. When this system discovers, via lookahead search, a sequence of operators that achieves a goal, it creates a declarative structure representing the sequence and returns it as a subgoal result (plan creation). This plan can then be used interpretively to guide performance on the immediate problem (plan following). The plan can also be retrieved during later problems and used to guide the selection of operators (case-based reasoning). This research does not demonstrate the variety of operations one could conceivably use to modify a partial or complete plan, but it does demonstrate the basics.

Meta-level access. Subgoal-based problem solving has access to all of the information in working memory—including the goal stack, problem spaces, states, operators, preferences, and other facts that have been retrieved or generated—plus any of the other knowledge in long-term memory that it can access. It does not have direct access to the productions, or to any of the data structures internal to the architecture. Nonetheless, it should be able to indirectly examine the contents of any productions that were acquired by chunking, which in the long run should be just about all of them. The idea is to reconstruct the contents of the production by going down into a subgoal and retracing the problem solving that was done when the chunk was learned. In this way it should be possible to determine what knowledge the production cached. This idea has not yet been explicitly demonstrated in Soar, but research on the recovery from incorrect knowledge has used a closely related approach [23].

The effects of problem solving are limited to the addition of information to

[8] The work on Robo-Soar also reveals Soar's potential to exhibit reactive planning [11]. The current version of Soar still has problems with raw speed and with the unbounded nature of the production match (the expensive chunks problem), but it is expected that these problems will be solved in the near future.

working memory. Deletion of working memory elements is accomplished by a garbage collector provided by the architecture. Productions are added by chunking, rather than by problem solving, and are never deleted by the system. The limitation on production creation—that it only occurs via chunking—is dealt with by varying the nature of the problem solving over which chunking occurs [56]. The limitation on production deletion is dealt with by learning new productions which overcome the effects of old ones [23].

Parallelism. Two principal sources of parallelism in Soar are at the memory level: production match and execution. On each cycle of elaboration, all productions are matched in parallel to the working memory, and then all of the successful instantiations are executed in parallel. This lets tasks that can be performed at the memory level proceed in parallel, but not so for decision-level and goal-level tasks.

Another principal source of parallelism is provided by the motor systems. All motor systems behave in parallel with respect to each other, and with respect to the cognitive system. This enables one form of task-level parallelism in which non-interfering external tasks can be performed in parallel. To enable further research on task-level parallelism we have added the experimental ability to simultaneously select multiple problem space operators within a single problem solving context. Each of these operators can then proceed to execute in parallel, yielding parallel subgoals, and ultimately an entire tree of problem solving contexts in which all of the branches are being processed in parallel. We do not yet have enough experience with this capability to evaluate its scope and limits.

Despite all of these forms of parallelism embodied in Soar, most implementations of the architecture have been on serial machines, with the parallelism being simulated. However, there is an active research effort to implement Soar on parallel computers. A parallelized version of the production match has been successfully implemented on an Encore Multimax, which has a small number (2–20) of large-grained processors [66], and unsuccessfully implemented on a Connection Machine [15], which has a large number (16 K–64 K) of small-grained processors [9]. The Connection Machine implementation failed primarily because a complete parallelization of the current match algorithm can lead to exponential space requirements. Research on restricted match algorithms may fix this problem in the future. Work is also in progress towards implementing Soar on message-passing computers [65].

4.3.4. Learning

In [61] we broke down the problem of evaluating the scope of Soar's learning capabilities into four parts: when can the architecture learn; from what can the architecture learn; what can the architecture learn; and when can the architecture apply learned knowledge. These points are discussed in Section 4.1, and need not be elaborated further here.

One important additional issue is whether Soar acquires knowledge that is at the appropriate level of generalization or specialization. Chunking provides a level of generality that is determined by a combination of the representation used and the problem solving performed. Under varying circumstances, this can lead to both overgeneralization [29] and overspecialization. The acquisition of overgeneral knowledge implies that the system must be able to recover from any errors caused by its use. One solution to this problem that has been implemented in Soar involves detecting that a performance error has occurred, determining what should have been done instead, and acquiring a new chunk which leads to correct performance in the future [23]. This is accomplished without examining or modifying the overgeneral production; instead it goes back down into the subgoals for which the overgeneral productions were learned.

One way to deal with overspecialization is to patch the resulting knowledge gaps with additional knowledge. This is what Soar does constantly—if a production is overspecialized, it doesn't fire in circumstances when it should, causing an impasse to occur, and providing the opportunity to learn an additional chunk that covers the missing case (plus possibly other cases). Another way to deal with overspecialized knowledge is to work towards acquiring more general productions. A standard approach is to induce general rules from a sequence of positive and negative examples [35, 45]. This form of generalization must occur in Soar by search in problem spaces, and though there has been some initial work on doing this [48, 58], we have not yet provided Soar with a set of problem spaces that will allow it to generate appropriate generalizations from a variety of sets of examples. So, Soar cannot yet be described as a system of choice for doing induction from multiple examples. On the other hand, Soar does generalize quite naturally and effectively when abstraction occurs [69]. The learned rules reflect whatever abstraction was made during problem solving.

Learning behaviors that have not yet been attempted in Soar include the construction of a model of the environment from experimentation in it [46], scientific discovery and theory formation [31], and conceptual clustering [8].

4.3.5. Perception and motor control

The scope of Soar's perception and motor control can be evaluated in terms of both its low-level I/O mechanisms and its high-level language capabilities. Both of these capabilities are quite new, so the evaluation must be even more tentative than for the preceding components.

At the low-level, Soar can be hooked up to multiple perceptual modalities (and multiple fields within each modality) and can control multiple effectors. The critical low-level aspects of perception and motor control are currently done in a standard procedural language outside of the cognitive system. The

resulting system appears to be an effective testbed for research on high-level aspects of perception and motor-control. It also appears to be an effective testbed for research on the interactions of perception and motor control with other cognitive capabilities, such as memory, problem solving, and learning. However, it does finesse many of the hard issues in perception and motor control, such as selective attention, shape determination, object identification, and temporal coordination. Work is actively in progress on selective attention [74].

At the high end of I/O capabilities is the processing of natural language. An early attempt to implement a semantic grammar parser in Soar was only a limited success [44]. It worked, but did not appear to be the right long-term solution to language understanding in Soar. More recent work on NL-Soar has focused on the incremental construction of a model of the situation by applying comprehension operators to each incoming word [32]. Comprehension operators iteratively augment and refine the situation model, setting up expectations for the part of the utterance still to be seen, and satisfying earlier expectations. As a side effect of constructing the situation model, an utterance model is constructed to represent the linguistic structure of the sentence. This approach to language understanding has been successfully applied to acquiring task-specific problem spaces for three immediate reasoning tasks: relational reasoning [18], categorical syllogisms, and sentence verification [3]. It has also been used to process the input for these tasks as they are performed. Though NL-Soar is still far from providing a general linguistic capability, the approach has proven promising.

5. Conclusion

In this article we have taken a step towards providing an analysis of the Soar architecture as a basis for general intelligence. In order to increase understanding of the structure of the architecture we have provided a theoretical framework within which the architecture can be described, a discussion of methodological assumptions underlying the project and the system, and an illustrative example of its performance on a multi-column subtraction task. In order to facilitate comparisons between the capabilities of the current version of Soar and the capabilities required to achieve its ultimate goal as an architecture for general intelligence, we have described the natural tasks for the architecture, the sources of its power, and its scope and limits. If this article has succeeded, it should be clear that progress has been made, but that more work is still required. This applies equally to the tasks of developing Soar and analyzing it.

Acknowledgement

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract N00039-86-C-0133 and by the Sloan Foundation. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. The views and conclusions contained in this document 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, the Sloan Foundation, or the National Institutes of Health.

We would like to thank Beth Adelson, David Kirsh, and David McAllester for their helpful comments on an earlier draft of this article.

References

[1] F.C. Bartlett, *Remembering: A Study in Experimental and Social Psychology* (Cambridge University Press, Cambridge, England, 1932).
[2] D. Chapman, Planning for conjunctive goals, *Artif. Intell.* **32** (1987) 333–377.
[3] H.H. Clark and W.G. Chase, On the process of comparing sentences against pictures, *Cogn. Psychol.* **3** (1972) 472–517.
[4] G. DeJong and R.J. Mooney, Explanation-based learning: an alternative view, *Mach. Learn.* **1** (1986) 145–176.
[5] T.G. Dietterich, Learning at the knowledge level, *Mach. Learn.* **1** (1986) 287–315.
[6] O. Etzioni and T.M. Mitchell, A comparative analysis of chunking and decision analytic control, in: *Proceedings AAAI Spring Symposium on Limited Rationality and AI*, Stanford, CA (1989).
[7] E.A. Feigenbaum and H.A. Simon, Epam-like models of recognition and learning, *Cogn. Sci.* **8** (1984) 305–336.
[8] D.H. Fisher and P. Langley, Approaches to conceptual clustering, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 691–697.
[9] R. Flynn, Placing Soar on the connection machine, Prepared for and distributed at the AAAI Mini-Symposium "How Can Slow Components Think So Fast" (1988).
[10] J.A. Fodor, *The Modularity of Mind* (Bradford Books/MIT Press, Cambridge, MA, 1983).
[11] M.P. Georgeff and A.L. Lansky, Reactive reasoning and planning, in: *Proceedings AAAI-87*, Seattle, WA (1987) 677–682.
[12] A. Golding, P.S. Rosenbloom and J.E. Laird. Learning general search control from outside guidance, in: *Proceedings IJCAI-87*, Milan, Italy (1987).
[13] A. Gupta and M. Tambe, Suitability of message passing computers for implementing production systems, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 687–692.
[14] C. Hewitt and D. Kirsh, Personal communication (1987).
[15] W.D. Hillis, *The Connection Machine* (MIT Press, Cambridge, MA, 1985).
[16] W. Hsu, M. Prietula and D. Steier, Merl-Soar: applying Soar to scheduling, in: *Proceedings Workshop on Artificial Intelligence Simulation, AAAI-88*, St. Paul, MN (1988) 81–84.
[17] T.R. Johnson, J.W. Smith Jr and B. Chandrasekaran, Generic Tasks and Soar, in: *Working Notes AAAI Spring Symposium on Knowledge System Development Tools and Languages*, Stanford, CA (1989) 25–28.
[18] P.N. Johnson-Laird, Reasoning by rule or model? in: *Proceedings 10th Annual Conference of the Cognitive Science Society*, Montreal, Que. (1988) 765–771.
[19] J.L. Kolodner, Maintaining order in a dynamic long-term memory, *Cogn. Sci.* **7** (1983) 243–280.

[20] J.L. Kolodner, ed., *Proceedings DARPA Workshop on Case-Based Reasoning*, Clearwater Beach, FL (1988).
[21] J.E. Laird, Universal subgoaling, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA (1983); also in: J.E. Laird, P.S. Rosenbloom and A. Newell, *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies* (Kluwer, Hingham, MA, 1986).
[22] J.E. Laird, Soar user's manual (version 4), Tech. Rept. ISL-15, Xerox Palo Alto Research Center, Palo Alto, CA (1986).
[23] J.E. Laird, Recovery from incorrect knowledge in Soar, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 618–623.
[24] J.E. Laird and K.A. McMahon, Destructive state modification in Soar, Draft V, Department of EECS, University of Michigan, Ann Arbor, MI (1989).
[25] J.E. Laird and A. Newell, A universal weak method, Tech. Rept. 83-141, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1983).
[26] J.E. Laird, A. Newell and P.S. Rosenbloom, SOAR: an architecture for general intelligence, *Artif. Intell.* **33** (1987) 1–64.
[27] J.E. Laird, P.S. Rosenbloom and A. Newell, Towards chunking as a general learning mechanism, in: *Proceedings AAAI-84*, Austin, TX (1984) 188–192.
[28] J.E. Laird, P.S. Rosenbloom and A. Newell, Chunking in Soar: the anatomy of a general learning mechanism, *Mach. Learn.* **1** (1986) 11–46.
[29] J.E. Laird, P.S. Rosenbloom and A. Newell, Overgeneralization during knowledge compilation in Soar, in: T.G. Dietterich, ed., *Proceedings Workshop on Knowledge Compilation*, Otter Crest, OR (1986).
[30] J.E. Laird, E.S. Yager, C.M. Tuck and M. Hucka, Learning in tele-autonomous systems using Soar, in: *Proceedings NASA Conference on Space Telerobotics*, Pasadena, CA (1989).
[31] P. Langley, H.A. Simon, G.L. Bradshaw and J.M. Zytkow, *Scientific Discovery: Computational Explorations of the Creative Processes* (MIT Press, Cambridge, MA, 1987).
[32] R.L. Lewis, A. Newell and T.A. Polk, Toward a Soar theory of taking instructions for immediate reasoning tasks, in: *Proceedings 11th Annual Conference of the Cognitive Science Society*, Ann Arbor, MI (1989).
[33] M. Minsky, A framework for the representation of knowledge, in: P. Winston, ed., *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975).
[34] M. Minsky, *The Society of Mind* (Simon and Schuster, New York, 1986).
[35] T.M. Mitchell, Generalization as search, *Artif. Intell.* **18** (1982) 203–226.
[36] T.M. Mitchell, R.M. Keller and S.T. Kedar-Cabelli, Explanation-based generalization: a unifying view, *Mach. Learn.* **1** (1986) 47–80.
[37] A. Newell, Reasoning, problem solving and decision processes: the problem space as a fundamental category, in: R. Nickerson, ed., *Attention and performance* **8** (Erlbaum, Hillsdale, NJ, 1980).
[38] A. Newell, *Unified Theories of Cognition* (Harvard University Press, Cambridge, MA, 1990).
[39] A. Newell and P.S. Rosenbloom, Mechanisms of skill acquisition and the law of practice, in: J.R. Anderson, ed., *Cognitive Skills and Their Acquisition* (Erlbaum, Hillsdale, NJ, 1981) 1–55.
[40] A. Newell, P.S. Rosenbloom and J.E. Laird, Symbolic architectures for cognition, in: M.I. Posner, ed., *Foundations of Cognitive Science* (Bradford Books/MIT Press, Cambridge, MA, 1989).
[41] A. Newell and H.A. Simon, *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
[42] N.J. Nilsson, *Principles of Artificial Intelligence* (Tioga, Palo Alto, CA, 1980).
[43] T.A. Polk and A. Newell, Modeling human syllogistic reasoning in Soar, in: *Proceedings 10th Annual Conference of the Cognitive Science Society*, Montreal, Que. (1988) 181–187.
[44] L. Powell, Parsing the picnic problem with a Soar3 implementation of Dypar-1, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1984).
[45] J.R. Quinlan, Induction of decision trees, *Mach. Learn.* **1** (1986) 81–106.
[46] S. Rajamoney, G.F. DeJong, and B. Faltings, Towards a model of conceptual knowledge

acquisition through directed experimentation, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 688–690.

[47] Y. Reich, Learning plans as a weak method for design, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, PA (1988).

[48] P.S. Rosenbloom, Beyond generalization as search: towards a unified framework for the acquisition of new knowledge, in: G.F. DeJong, ed., *Proceedings AAAI Symposium on Explanation-Based Learning*, Stanford, CA (1988) 17–21.

[49] P.S. Rosenbloom, A symbolic goal-oriented perspective on connectionism and Soar, in: R. Pfeifer, Z. Schreter, F. Fogelman-Soulie and L. Steels, eds., *Connectionism in Perspective* (Elsevier, Amsterdam, 1989).

[50] P.S. Rosenboom and J.E. Laird., Mapping explanation-based generalization onto Soar, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 561–567.

[51] P.S. Rosenbloom, J.E. Laird, J. McDermott, A. Newell and E. Orciuch, R1-Soar: an experiment in knowledge-intensive programming in a problem-solving architecture, *IEEE Trans. Pattern Anal. Mach. Intell.* **7** (1985) 561–569.

[52] P.S. Rosenbloom, J.E. Laird and A. Newell, Knowledge level leaning in Soar, in: *Proceedings AAAI-87*, Seattle, WA (1987) 499–504.

[53] P.S. Rosenbloom, J.E. Laird and A. Newell, The chunking of skill and knowledge, in: B.A.G. Elsendoorn and H. Bouma, eds., *Working Models of Human Perception* (Academic Press, London, 1988) 391–410.

[54] P.S. Rosenbloom, J.E. Laird and A. Newell, Meta-levels in Soar, in: P. Maes and D. Nardi, eds., *Meta-Level Architectures and Reflection* (North-Holland, Amsterdam, 1988) 227–240.

[55] P.S. Rosenbloom and A. Newell, The chunking of goal hierarchies: a generalized model of practice, in: R.S. Michalski, J.G. Carbonnell and T.M. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986) 247–288.

[56] P.S. Rosenbloom, A. Newell and J.E. Laird, Towards the knowledge level in Soar: the role of the architecture in the use of knowledge, in: K. VanLehn, ed., *Architectures for Intelligence* (Erlbaum, Hillsdale, NJ, 1990).

[57] E.D. Sacerdoti, Planning in a hierarchy of abstraction spaces, *Artif. Intell.* **5** (1974) 115–135.

[58] R.H. Saul, A Soar2 implementation of version-space inductive learning, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1984).

[59] R. Schank and R. Ableson, *Scripts, Plans, Goals and Understanding* (Erlbaum, Hillsdale, NJ, 1977).

[60] D. Steier, Cypress-Soar: a case study in search and learning in algorithm design, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 327–330.

[61] D.M. Steier, J.E. Laird, A. Newell, P.S. Rosenbloom, R. Flynn, A Golding, T.A. Polk, O.G. Shivers, A. Unruh and G.R. Yost, Varieties of learning in Soar: 1987, in: P. Langley, ed., *Proceedings Fourth International Workshop on Machine Learning*, Irvine, CA (1987) 300–311.

[62] D.M. Steier and A. Newell, Integrating multiple sources of knowledge in Designer-Soar: an automatic algorithm designer, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 8–13.

[63] K.R. Swedlow and D.M. Steier, Soar 5.0 user's manual, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1989).

[64] M. Tambe, Speculations on the computational effects of chunking, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1988).

[65] M. Tambe, A. Acharya and A. Gupta, Implementation of production systems on message passing computers: Simulation results and analysis, Tech. Rept. CMU-CS-89-129, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1989).

[66] M. Tambe, D. Kalp, A. Gupta, C.L. Forgy, B. Milnes and A. Newell, Soar/PSM-E: Investigating match parallelism in a learning production system, in: *Proceedings ACM/ SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems* (1988) 146–161.

[67] M. Tambe and A. Newell, Some chunks are expensive, in: J. Laird, ed., *Proceedings Fifth International Conference on Machine Learning* Ann Arbor, MI (1988) 451–458.

[68] M. Tambe and P.S. Rosenbloom, Eliminating expensive chunks by restricting expressiveness, in: *Proceedings IJCAI-89*, Detroit, MI (1989).

[69] A. Unruh and P.S. Rosenbloom, Abstraction in problem solving and learning, in: *Proceedings IJCAI-89*, Detroit, MI (1989).
[70] A. Unruh, P.S. Rosenbloom and J.E. Laird, Dynamic abstraction problem solving in Soar, in: *Proceedings Third Annual Aerospace Applications of Artificial Intelligence Conference*, Dayton, OH (1987) 245–256.
[71] K. VanLehn, *Mind Bugs: The Origins of Procedural Misconceptions* (MIT Press, Cambridge, MA, 1990).
[72] K. VanLehn and W. Ball, Non-LIFO execution of cognitive procedures, *Cogn. Sci.* **13** (1989) 415–465.
[73] R. Washington and P.S. Rosenbloom, Applying problem solving and learning to diagnosis, Department of Computer Science, Stanford University, CA (1988).
[74] M. Wiesmeyer, Personal communication (1988).
[75] M. Wiesmeyer, Soar I/O reference manual, version 2, Department of EECS, University of Michigan, Ann Arbor, MI (1988).
[76] M. Wiesmeyer, New and improved Soar IO, Department of EECS, University of Michigan, Ann Arbor, MI (1989).
[77] G.R. Yost and A. Newell, A problem space approach to expert system specification, in: *Proceedings IJCAI-89*, Detroit, MI (1989).