3. Laird, J. E. Soar User's Manual (Version 4). Tech. Rept. ISL-15, Xerox Palo Alto Research Center, 1986.

4. Laird, J. E., Newell, A., & Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33* (1987), 1-64.

5. Laird, J. E., Rosenbloom, P. S., & Newell, A. Towards chunking as a general learning mechanism. Proceedings of AAAI-84, Austin, 1984.

6. Laird, J. E., Rosenbloom, P. S., & Newell, A. "Chunking in Soar: The anatomy of a general learning mechanism". *Machine Learning 1* (1986), 11-46.

7. Lebowitz, M. Complex learning environments: Hierarchies and the use of explanation. In *Machine Learning: A Guide to Current Research*, T. M. Mitchell, J. G. Carbonell, & R. S. Michalski, Eds., Kluwer Academic Publishers, Boston, MA, 1986.

8. Mahadevan, S. Verification-based learning: A generalization strategy for inferring problem-decomposition methods. Proceedings of IJCAI-85, Los Angeles, CA, 1985.

9. Mitchell, T. M. "Generalization as search". *Artificial Intelligence 18* (1982), 203-226.

10. Mitchell, T. M. Toward combining empirical and anlytical methods for learning heuristics. In *Human and Artificial Intelligence*, A. Elithorn & R. Banerji, Eds., North-Holland Publishing Co., Amsterdam, 1984.

11. Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. "Explanation-based generalization: A unifying view". *Machine Learning 1* (1986), 47-80.

12. Rosenbloom. P. S., & Laird, J. E. Mapping explanation-based generalization onto Soar. Proceedings of AAAI-86. Philadelphia, 1986.

13. Rosenbloom, P. S.. Laird, J. E., & Newell, A. Knowledge level learning in Soar. Proceedings of AAAI-87. Seattle, 1987.

14. Rosenbloom, P. S., Laird, J. E., & Newell, A. The chunking of skill and knowledge. In *Working Models of Human Perception*, H. Bouma & B. A. G. Elsendoorn, Eds., Academic Press. London, 1988. In press.

15. Steier, D. M., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R., Golding, A., Polk, T. A., Shivers, O. G.. Unruh, A., & Yost, G. R. Varieties of Learning in Soar: 1987. Proceedings of the Fourth International Workshop on Machine Learning, Los Altos, CA, 1987.

CHAPTER 26

Meta-Levels in Soar

P. S. Rosenbloom, Stanford University, J. E. Laird, University of Michigan, and A. Newell, Carnegie Mellon University

Soar is an attempt to build an architecture capable of supporting general intelligence. In this article we present an analysis of Soar in terms of the concept of meta-level architecture. In the process, we provide a definition of what it means for something to be a meta-level architecture, and describe how Soar can be viewed as consisting of an infinite tower of meta-level architectures. Each of Soar's meta-level architectures is itself a complex structure that is constructed out of three types of processors: problem-space processors, production processors, and preference processors. In addition to the meta-level architectures themselves, Soar contains a learning mechanism which can modify some aspects of the meta-level architectures.

1. INTRODUCTION

The Soar architecture [1, 2, 3] grew out of an attempt to integrate a set of ideas about what should be included in an architecture for general intelligence. Such an architecture should provide the basis for a system that can work on a wide variety of tasks, employ a wide variety of problem solving methods, and improve its own performance via learning. Some demonstrations of the generality and power of the Soar architecture can be found in [1, 3, 4, 5].

The ideas that went into the design of the Soar architecture included such things as search in problem spaces [6], subgoals, production systems [7, 8], and learning by chunking [9], but did not at that time include the concepts of meta-level architecture [10], introspection [11, 12], or reflection [13]. However, as we now analyze Soar, several aspects of it bear a close resemblance to ideas being investigated in the area of meta-level architecture. In this article, we describe the relationship between Soar and the concept of meta-level architecture. In the process we hope to increase our understanding of both the structure of Soar and the space of meta-level architectures.

In the following sections we specify what we mean by the term "meta-level architecture", give an overview of the Soar architecture (in non-meta-level terms), describe the meta-level architecture embodied by Soar, and conclude.

---

*This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contracts N00039-83-C-0136 and F33615-81-K-1539, 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.

2. META-LEVEL ARCHITECTURE

There are two distinct but related views of the meta-level in the literature, one primarily declarative and the other primarily procedural. A summary of these two points of view can be found in [14], where they are referred to as the *autoepistemic* and *control* traditions. In this article we concentrate on a variation of the control tradition. In particular, we focus on the notion of levels of architecture rather than directly on the notions of quotation, designation, aboutness, or meta-knowledge. As levels of architecture are traversed, levels of quotation are introduced, but the two are not always in one-to-one correspondence.

The control tradition, as epitomized by 3-Lisp [15], is based on the classical computer science separation of the architecture of a processor from its contents (the programs and data that are processed by the architecture). In general, an architecture provides a processing level. For the purposes of this paper we will use a somewhat simplified notion of what this involves; specifically, the following primitive representational and action capabilities. An architecture provides the ability to represent a set of executable operations, data (operands and results), and a task to be performed (using the operations and data). The contents of a processing level are the operations, data, and task represented within the level's architecture. The architecture also provides a set of primitive actions for assembly, selection, and execution of operations (and data). Assembly involves the retrieval and composition of operations and operands. Selection involves the choice of a single operation and of its operands. Execution involves the application of the selected operation to the selected operands, yielding an alteration to the current computational state (the result). The architecture must employ its primitive acts in such a way that the specified task gets performed.

As an example, consider an interpreter-based version of Lisp. The interpreter is the architecture. It provides the ability to represent functions (the operations), list structures (the data), and a program which specifies the task to be performed. Assembly involves packaging together a function and the data to which it is to be applied. Selection involves determining which function is to be applied next, and to which data. Execution involves calling "eval" on the function and its data.

A processing level has a meta-level if the level's architecture can itself be decomposed into an architecture plus contents. The task at the meta-level is the implementation of the lower-level's architecture. In general there can be a hierarchy of meta-levels in which each higher meta-level defines the architecture of the level below it. The base-level is the lowest level in the hierarchy; that is, the level at which some task is performed other than the implementation of another level. The meta-level is the level at which the architecture of the base-level is defined. The meta^2-level is the level at which the architecture of the meta-level is defined, and so on.

According to the definitions provided so far, any time a level is implemented by another level, the implementation level is a meta-level for the original level. In systems such as 3-Lisp, it seems entirely appropriate to refer to the levels of implementation as meta-levels. However, in other systems, such as the Lisp-based implementation of the Ops5 production-system language [7], it does not seem appropriate. Additional restrictions on the use of the term meta-level have been proposed – such as the need for the meta-level to be declarative (see, for example, [11] and [10]) – but what differentiates these two systems is not so much the nature of the meta-level as it is the locus of the boundary that the theorist draws around the system to be described. In the Ops5 example, the implementation of the base-level's architecture is, from the theorist's standpoint, an unimportant detail that is outside the scope of the system – the nature of Ops5 does not fundamentally change when it is implemented in Bliss. In the 3-Lisp case, the system of theoretical interest includes both 3-Lisp and its architecture. In the remainder of this article we will distinguish between those levels which are meta-levels within Soar and those that are part of the extra-theoretic implementation of Soar. The extra-theoretic code comprises what is normally referred to as the Soar architecture – the fixed body of code which implements the overall Soar system.

The meta-level embodied by Soar is a composite level – a level that is composed of two or more components, each of which is itself a processing level. One type of processing level is used for selection, and two others are used as alternatives for assembly and execution. When there are composite levels it becomes important to distinguish between two slightly different meanings of the term "level": (1) a level in the meta-level hierarchy, such as the base-level and the meta-level, which provides a complete architecture for the next lower level; and (2) a processing level, which is provided by an architecture, but that may only provide one component of the architecture for the next lower level in the hierarchy. Referring to both of these as levels would be unnecessarily confusing, so we will henceforth reserve the term "level" for the first type, that is, levels that provide complete architectures. We will refer to a processing level by the alternative term "processor", with the understanding that a processor is a type of level.

3. SOAR

Soar is organized around the *Problem Space Hypothesis* [6], that all goal-oriented behavior is based on search in problem spaces. This applies to both obvious search-based tasks, such as chess, as well as to more algorithmic and knowledge-intensive tasks, such as the determination of the roots of an equation or the configuration of a computer. It also applies to both external task goals and internal system goals. Whatever type of goal it is, the problem space determines the set of legal states and operators that can be used during the processing to attain the goal. The states represent situations. There is an initial state representing the initial situation, and a set of desired states that represent the goal. An operator, when applied to a state in the problem space, yields another state in the problem space. The goal is achieved when one of the desired states is reached as the result of a string of operator applications starting from the initial state. In chess, the states represent legal configurations of the chess pieces on the board. In the initial state, all of the pieces are in their initial positions. In the desired states, the opponent's king is in checkmate. The operators represent legal moves. The game is won when a sequence of moves succeeds in transforming the initial state into a desired state.

Goals, problem spaces, states, and operators exist as data structures in Soar's working memory – a short-term declarative memory. Each goal defines a problem solving context – or context for short – which is a data structure in the working memory that contains, in addition to a goal, roles for a problem space, a state, and an operator. Problem solving is driven by the acts of selecting problem spaces, states, and operators for the appropriate roles in the context. Given a goal, a problem space is to be selected. Different problem spaces can be selected for different goals, or even tried as alternatives for a single goal. The selection of the problem space is followed by the selection of an initial state, and then of an operator to be applied to the initial state. When a problem space, state, and operator are all selected, the operator is applied to the state to yield a result state. Problem solving within the problem space can proceed with the selection of the result state, and the selection of an operator to apply to the result state. The goal is achieved when a desired state is selected.

Each of the deliberate acts of the Soar architecture – a selection of a problem space, a state or an operator – is accomplished via a two-phase decision cycle. First, during the

elaboration phase, the description of the current situation (that is, the contents of working memory) is elaborated with relevant information from Soar's long-term production memory. This memory is used for all long-term knowledge in Soar, including factual, procedural (operator application), and control information. The elaboration phase is a structurally monotonic process – it only adds elements to the working memory – in which successfully matched productions are fired in parallel until no more productions can fire (i.e., until quiescence). All of the control in Soar occurs at the problem solving level, not at the level of production-memory access – there is no conflict resolution process in Soar's production system.

The elaboration process involves both the addition of knowledge about existing objects and the creation of new objects. One important type of knowledge that may be added during the elaboration phase is knowledge of preferences. There is a fixed language of preferences which is used to describe the acceptability and desirability of the alternatives being considered for selection. Each preference contains fields that specify an object being considered for selection, the role for which the object is being considered (problem space, state, or operator), a set of problem solving contexts for which the preference is valid, a preference value (acceptable, reject, better, best, etc.), and possibly a reference object for comparison purposes. By using different preference values, it is possible to assert that a particular problem space, state, or operator is acceptable (should be considered for selection), rejected (should not be considered for selection), better than another alternative, and so on.

In the second phase of the decision cycle, the preferences in working memory are interpreted by a fixed decision procedure. If the preferences uniquely specify an object to be selected for a role in a context, then a decision can be made, and the specified object becomes the current value of the role. If the preferences for a decision are either incomplete or inconsistent, then the system does not know how to proceed on the goal, and an impasse occurs in problem solving. When an impasse occurs, a subgoal with an associated problem solving context is automatically generated for the task of resolving the impasse. The subgoal is a data structure that describes the impasse which caused the subgoal to be generated. The impasses, and thus their subgoals, vary from problems of selection (of problem spaces, states, and operators) to problems of generation (e.g., operator application). Most impasses in selection occur when there is more than one acceptable alternative and the preferences do not uniquely specify which one is best; for example, an impasse would occur in chess when more than one legal operator can be applied to a selected state and there is insufficient knowledge (encoded as preferences) about which of the operators should be selected. However, other impasses in selection can occur either when there are no acceptable alternatives or when the preferences state conflicting information. Operator application occurs via the same decision cycle on which selection is based. The act of applying the operator involves the assembly of a result state during the elaboration phase that immediately follows the selection of the operator. An impasse occurs in operator application if an operator is selected but no new state can be selected during the following decision cycle.

Given a subgoal, Soar can bring its full problem solving capability and knowledge to bear on resolving the impasse that caused the subgoal. For impasses in operator application, one approach is to perform a search for a state to which the operator can be applied (operator subgoaling). Another approach is to decompose the problem of applying the operator into a set of simpler operator applications in a sub-problem-space. This is not a critical capability in simple puzzle and game tasks, where the operators are all quite simple. However, in more demanding domains, where the operators can be quite complex, the ability to bring the system's full problem solving capabilities to bear can be critical. For example, in the Soar implementation of an expert system for computer configuration, there is a relatively complex operator called configure-backplane that involves, among other things, putting a computer backplane into a box, selecting and configuring a set of objects (called modules) into the backplane, and adding a cable to the backplane [16]. This operator is applied via four problem spaces over two levels of subgoals.

For impasses in selection, the usual response is to perform a look-ahead search to determine which of the alternatives is best. A problem space (called the selection space) is used which has an operator (called evaluate-object) which evaluates an alternative. If productions exist to directly implement this operator, preferences can be generated directly from the resulting evaluations. If not, an impasse occurs in operator application and a second-level subgoal is generated within the existing selection subgoal. In this subgoal, the problem of evaluating an alternative – for now, let's assume the alternative is an operator – is accomplished in three steps: (1) establishing the evaluation context (selecting the original problem space and state), (2) selecting and applying the operator to be evaluated, and (3) evaluating the resulting state. If the state can be evaluated, the evaluate-object operator has completed, and another of the alternatives can be evaluated. Otherwise the search continues with the attempt to select an operator for the result state. In the process, subgoals are generated at progressively deeper levels.

It should be emphasized that this form of look-ahead search is just one of a variety of types of processing that can conceivably be performed to determine what alternative to select. Depending on the task being pursued, and the knowledge available, other types of processing may be more appropriate. For example, another type of processing we have been actively exploring in Soar is a look-ahead search in a dynamically constructed abstract version of the problem space [17]. Abstract problem spaces are created by Soar from existing problem spaces and then used to more cheaply get insight into which alternatives it might be best to select.

When subgoals occur within subgoals, a goal hierarchy results (which also therefore defines a hierarchy of contexts). The top goal in the hierarchy is a task goal; for example, to win a chess game. The subgoals below it are all generated as the result of impasses in problem solving. A subgoal terminates when its impasse is resolved, even if there are many levels of subgoals below it (the lower ones were all in the service of the terminated subgoal, so they can be eliminated if it is resolved). On termination, all of the subgoal's working-memory elements are deleted by the working-memory manager except for those that are results of the subgoal.

Soar learns by a process of chunking that automatically acquires new productions that summarize the processing in a subgoal. The actions of the new productions are based on the results of the subgoal. The conditions are based on those aspects of the pre-goal situation that were relevant to the determination of those results. In relevantly similar situations in the future, the new production will fire during the elaboration phase, directly producing the required information. No impasse will occur, and problem solving can proceed smoothly. Chunking is thus a form of goal-based caching which avoids redundant future effort by directly producing a result that once required problem solving to determine.

Though chunking is a from of caching, it still achieves both significant generality and variation. Generality comes from including conditions for only those aspects of the situation upon which the results depended. The approach is similar to that taken in explanation-based generalization, where the appropriate features are determined by a form of goal regression [18, 19]. Variation in learning comes from the variation in the

types of goals that are encountered and the variation in the problem spaces used for the goals. The goal determines the role the new production will play in problem solving: a goal for a selection impasse yields a production that generates preferences (a search-control chunk); a goal for an operator-application impasse yields a chunk which directly implements the operator. The problem space, along with the actual search performed in the problem space, determines the contents of the new production.

4. SOAR'S META-LEVELS

Soar's meta-levels are built from three types of processors: problem-space processors, production processors, and preference processors (Figure 1). In a problem-space processor, operators are the executable operations, states are the data (operands and results), and the goal specifies the task to be performed. In a production processor, productions are the executable operations, working-memory elements are the data, and the task to be performed is to execute productions until quiescence. In a preference processor, the preferences in working memory are the executable operations. In Section 3, preferences are referred to as data structures that are processed by the decision procedure, rather than as executable operations. However, it is possible to view the preferences as operations that independently specify actions to be performed during the selection process.* The role of the decision procedure is then to provide the architecture for the preference processor. The context hierarchy plus the set of candidate objects are the data for the preference processor, and the task to be performed is a decision (either a selection or an impasse). These three types of processors – problem space, production, and preference – can get mixed into a meta-level hierarchy of unbounded height.

	Problem Space	Production	Preference
Operations	Operators	Productions	Preferences
Data	States	Working Memory	Contexts & Candidates
Task	Goal	Quiescence	Decision

FIGURE 1
Types of processors in Soar's meta-level hierarchy.

The base-level in Soar is a problem-space processor – the top context in the context hierarchy. Soar's meta-level defines the assembly, selection, and execution processes for this base-level. Because the overall picture is rather complex, we will describe this meta-level architecture in a stepwise fashion, first describing the portion of the meta-level that defines the execution of base-level operations, then incorporating the base-level's selection processes, and finally incorporating the assembly processes. Along the way we will also lay out the architectures of the production and preference processors embedded in the meta-level hierarchy. The section will be wrapped up with discussions about the role of chunking in Soar's meta-level's, and the tradeoff between uniformity and flexibility that is embodied in Soar's meta-levels.

*Given the preference semantics specified in [3], this is currently only approximately true. So, for now, this should be considered a goal rather than a statement of fact.

4.1. Execution

Base-level problem-space operators are executed by a pair of meta-level components: a production processor and a problem-space processor (Figure 2). One way to view this structure is that it provides two alternative meta-levels: the production meta-level and the problem-space meta-level. The production meta-level is entered at every opportunity, that is, every time an operator is to be executed. The productions in the production meta-level can examine the base-level goal, and from it get access to representations of the base-level problem space, state, and operator. It can also add information about existing objects, and assemble new objects. Operator execution by the production meta-level usually involves the examination of the operator and the state, and the assembly of a result state.

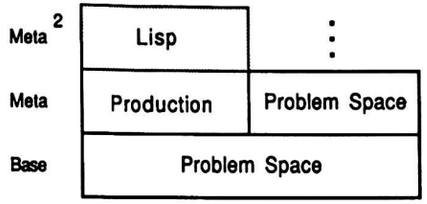

FIGURE 2
The meta-level hierarchy for execution.

If the production meta-level succeeds in executing the base-level operation, then the system returns immediately to the base-level. However, the production meta-level may fail to completely perform the execution act. Productions may or may not exist to execute any particular operator; there is no guarantee that the production meta-level will specify a complete architecture. When the production meta-level fails to completely execute an operator, an impasse generally occurs because there is no result state to select, and a subgoal is generated. The subgoal makes explicit information that was otherwise implicit in the architecture of the production meta-level – specifically, that quiescence occurred without a decision being made. It also includes additional information, such as a pointer to the parent goal and a description of the type of impasse that occurred.

When a subgoal is generated, the problem-space meta-level is used. Thus, this meta-level is entered not on opportunity but on need – when the production meta-level fails. The problem-space meta-level has a small amount of meta-level access to the production meta-level; specifically, it has access to the subgoal. However it cannot otherwise examine or modify the production meta-level. Instead, the problem-space meta-level acts as an alternative meta-level for the base-level. The problem-space meta-level allows difficult problem-space operators to themselves be executed via search in problem-solving contexts. Like the base-level, the problem-space meta-level has its own context (consisting of a goal, a problem space, a state, and an operator). To access the base-level context the problem-space meta-level goes indirect through the pointer from the subgoal to its parent goal (the base-level goal). The base-level problem space, state, and operator can all be accessed via pointers from this goal. Another way of viewing this is that the problem-space meta-level is always created with an extra level of quotation on the base-level context. This extra level of quotation, while often useful, can be explicitly removed when it is not needed – a common first step in the problem-space meta-level is to select the base-level state (or some subcomponent of it) as the initial state, thus allowing the problem-space meta-level to work on the base-level state without the intervening level of quotation. Using the problem-space meta-level for operator execution

can also be viewed as analogous to a reflective procedure call in 3-Lisp. Soar's run-time stack of reflective procedure calls is thus accessible as a subset of the goals in the context stack. There is no direct analogue in Soar of 3-Lisp's simple (within-level) procedure calls.*

The meta^2-level must provide the architecture for both the production and problem-space meta-levels. The architecture for the production meta-level is provided by the production interpreter. The interperter assembles executable operations by instantiating successfully matched productions, it selects all of the successfully assembled operations (the production instantiations), and it executes them to yield modifications to the production system's working memory. The production interpreter is implemented in Lisp. It is part of the fixed structure of Soar, and is not itself one of the meta-levels of the system.

The architecture for the problem-space meta-level is not shown in the figure. The implementation is done by recursing on the meta-level structure shown for the base-level. That is, the problem-space meta-level is implemented exactly as was the base-level. This recursive structure makes it possible to have a hierarchy with an arbitrary number of meta-levels. Though this provides great flexibility it also potentially leads to infinite regressions. In systems like 3-Lisp this infinite regression is terminated by having a second non-recursive architecture whose effect is the same as an infinite number of unmodified meta-levels (all of the levels start out with the same architecture). This architecture can then be used instead of the recursive architecture when it would have the same effect. The approach taken in Soar is similar, with the production meta-level serving the function of the non-recursive architecture. The regression bottoms out whenever the production meta-level can perform the requisite function.

Though the production meta-level is not guaranteed to provide a complete architecture for each level — Soar falls back on the problem-space meta-level when productions don't exist — Soar has been initialized with a set of productions that provide a complete default architecture for situations in which there is no specific knowledge about how to proceed. These productions prevent the actual generation of infinite, content-free, meta-level hierarchies by generating preferences for a default response when a subgoal occurs for which no problem space is suggested; that is, when Soar has absolutely no idea about how to proceed. For operator execution, the default response is to try operator subgoaling by searching for another state to which the operator can be applied, and then if that fails, to reject the operator so that another operator can be tried. For operator selection, the default is to try a look-ahead search to determine which operator to prefer, and then if that fails, to give all of the alternatives indifferent-preferences so that a random selection can occur among them.

4.2. Selection

Selection of base-level problem-space operators and states — actually of operators and states in all problem-space processors — is performed by a preference processor, as shown in Figure 3. In the figure, the "S" and "E" annotations show which processors at the meta-level implement which components of the base-level's architecture. For simplicity, when one meta-level processor implements both the selection and execution components of the level below it, the annotations are omitted.

*There is however an experimental mechanism that allows operators to be suspended, replaced, and later resumed in a single context without losing any of their subcontexts. By using preferences to control when operators are suspended and resumed it may be possible to implement a within-context operator-suboperator calling structure.

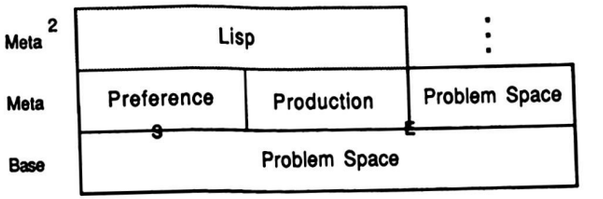

FIGURE 3
The meta-level hierarchy for selection and execution.

The selection process proceeds over multiple steps. First, a problem space is selected. This selection affects which operators and states can be selected by changing the context that valid preferences must match. The selection process then continues with the successive selection of a state and an operator. Not all of these selections must be made each time an operator is to be selected. Objects stay selected for a role until a new selection is made for that role, or for a higher one in the context. A new operator can be selected without affecting the problem space and the state, and a new state can be selected without affecting the problem space.

Preferences are selected and executed by the decision procedure, according to the fixed semantics described in [3]. The decision procedure is implemented in Lisp, is part of the fixed structure of Soar, and is not one of its meta-levels. As will be described in the next section, this fixity in the processing of preferences is counterbalanced by the flexibility available during the preference assembly process.

4.3. Assembly

Figure 4 shows the complete meta-level hierarchy in Soar. The assembly processes for both the base-level problem-space processor and the preference processor have been added. The "A" annotations in the figure show which meta-level processors implement the assembly processes. As before, annotations are omitted whenever one meta-level processor implements the complete architecture for a processor at the level below.

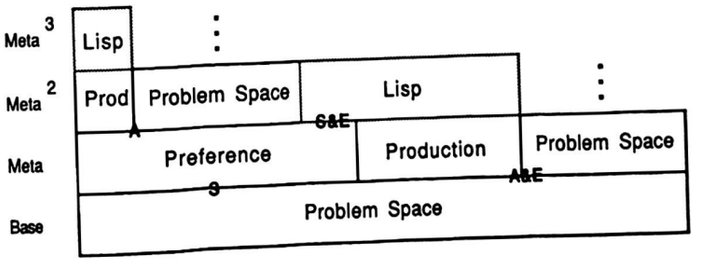

FIGURE 4
The complete meta-level hierarchy.

For the base-level problem-space processor, assembly occurs much in the same way as execution — operator execution is simply an assembly process (of the result state). Productions can assemble operators by adding the appropriate data structures to working memory. If a selection cannot be made, and thus an impasse is reached, operators

may be assembled via problem solving in the resulting subgoal; that is, in the problem-space meta-level. State assembly, except for the initial state of a goal, is accomplished directly by operator execution. Initial states are also assembled by production and problem-space processors, but are not based on the currently selected operator and state, because no operator or state is yet selected. Instead, initial-state assembly is usually based on the goal and the problem space. The complete architecture for the base-level can now be seen as being built upon a meta-level containing preferences, productions, and problem-space operators. Preferences are responsible for selection, while productions and problem-space operators take care of assembly and execution. This meta-level can be considered to consist of three independent processors, or we can again view this structure as defining two alternative meta-level architectures. One meta-level architecture consists of a preference processor and a production processor, while the other one consists of a preference processor and a problem-space processor. The working memory and preference processor are shared between the two architectures. This same meta-level structure is replicated at each level that implements a problem-space processor.

Preferences are assembled by productions and/or problem-space operators; that is, they are assembled by the same components that are used to assemble (and execute) problem-space operators. This indirectly provides the capability for arbitrary amounts of processing to be done in the service of controlling the selection of base-level operations: the selection of base-level operations is determined by the set of preferences that are represented explicitly in working memory, and the preference assembly process determines which preferences are made explicit. At the meta^3-level, the productions that assemble preferences are implemented by the same production interpreter that was used for the productions at the meta-level. All productions are in a single monolithic memory with a single interpreter. Likewise, the problem-space processor that assembles preferences is implemented in the same way as the problem-space processor used at the base-level and the meta-level — the system recurses.

The flexible control capability provided by the preference assembly process is available for all problem-space processors at all levels of the system. In Section 3, we described the multi-subgoal control searches that Soar often performs. Figure 5 shows a fragment of the meta-level hierarchy for such a search. At the base-level is a problem-space processor that is being controlled. At the meta-level, control occurs via preferences. The assembly of these preferences can occur via a production processor at the meta^2-level, if the requisite knowledge exists. However, if the requisite knowledge is not directly available in productions, a problem-space processor is used, usually in conjunction with the selection problem space. The evaluate-object operators in the selection problem space can be implemented at the meta^3-level by either a production processor or a problem-space processor. The meta^3-level problem-space processor works by first reestablishing the base-level problem solving context (the base-level problem space and state), and then trying to evaluate the operator by selecting it, applying it to the current state, and evaluating the resulting state. If the resulting state cannot be evaluated, then the whole process recurses (at even higher meta-levels) on the selection of an operator to apply to the result state. When the evaluation information finally becomes available, other base-level problem-space operators are evaluated until enough information is available to yield preferences that will result in the selection of one of the base-level operators.

4.4. Chunking

Soar's chunking mechanism was not included in the meta-level hierarchy in Figure 4 because it is not directly involved in assembly, selection, or execution for any of the

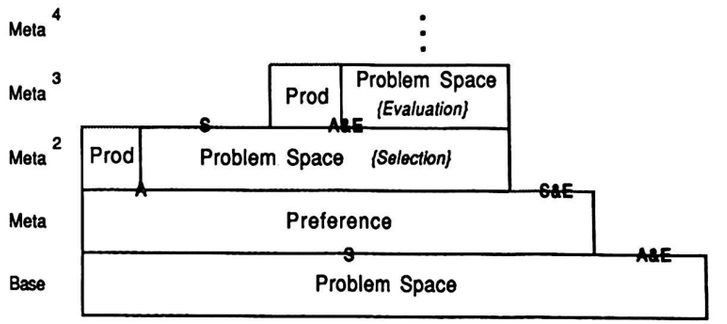

FIGURE 5
Fragment of meta-level hierarchy for control search.

processors in the hierarchy. However, it does indirectly affect production assembly by creating the productions that will later be used during the assembly of executable production instantiations.

One view of the role of chunking in Soar is that it incrementally compiles problem solving in meta-level problem-space processors into new productions. That is, it moves knowledge from a slow, deliberate, potentially problematic, problem-space meta-level to a fast, uncontrolled, nonproblematic, production meta-level. The production meta-level thus acts somewhat like a long-term cache for the problem-space meta-level. Over time, the chunking process gradually fills in the gaps in the production meta-level, reducing the need for Soar to use the problem-space meta-level, and thus reducing its need to climb higher in the meta-level hierarchy. This is particularly important for Soar, because it routinely goes to high meta-levels (that is, deep subgoals).

With any caching scheme, cache consistency is a potential issue. In Soar, there is no guarantee that the behavior resulting from use of the production meta-level will always be identical to what the problem-space meta-level would have done — changes to one of the meta-levels are not reflected automatically in changes to the other one. When cache consistency is important new productions must be learned that restore the relationship between the production and problem-space meta-levels.* However, the absence of cache consistency can itself sometimes be important. For example, it can allow productions to be learned which remember important transient aspects of the problem-space meta-level. This approach is used in [21] as the basis for the acquisition of new knowledge from transient external inputs.

4.5. Uniformity and Flexibility

One of the major trade-offs in Soar's meta-level architecture is in uniformity versus flexibility. Soar's meta-levels are unable to examine and/or modify its fixed architecture (the decision procedure, the production interpreter, etc.). As a result, it is not possible to allow levels to differ in arbitrary ways. Part of Soar's inflexibility is justified by the theory of intelligence on which Soar is based. The theory contains a number of universal statements, such as that all goal-oriented behavior occurs in problem spaces.

*This is not a solved problem, but some initial work on a related issue — recovery from the acquisition of overgeneral productions — is described in [20].

These universals are reflected in Soar as unmodifiable code.* However, the converse is not yet true – not all of the unmodifiable code is justified by general principles.

One of the main benefits of universals is that they assure some degree of uniformity. In Soar, uniformity exists across levels to the extent that they all make use of problem spaces, productions, and preferences. This uniformity allows all of the system's long-term knowledge and problem-solving capabilities to be brought to bear at every level. Productions learned at one level can be reused at any of the levels for which they are appropriate. Likewise, a problem space can be used at any level for which it is appropriate.

This uniformity also allows the sharing of working-memory elements between levels. The working-memory elements accessible by a level include all of the working-memory elements at the lower levels plus some additional data for that level (including the subgoal that led to the creation of the level). The sharing of working-memory elements across levels allows a meta-level to examine (and modify) data from multiple lower levels, facilitating communication between levels and enabling Soar to return directly to any lower level from any higher level without going through the intermediate levels (this is equivalent to automatic subgoal return at any level of the goal hierarchy).

Despite Soar's uniformity, quite different processing is possible at different meta-levels, and is in fact the norm. The across-level variation in the contents of working memory leads directly to the ability to selectively use productions and problem spaces across levels. A chess problem space may be in use at one level, while at another level a selection problem space is being used.

5. CONCLUSIONS

The major conclusions about the meta-levels embodied in Soar can be summarized as follows:

- Soar embodies a heterogeneous meta-level hierarchy, containing levels based on productions, problem spaces, and preferences.
- Soar's meta-level hierarchy is of unbounded height, with infinite regression avoided by the use of productions.
- Soar uses its meta-levels constantly, and to non-trivial heights in the hierarchy.
- Chunking moves knowledge from a problem-space meta-level to a production meta-level.
- The flexibility of Soar's meta-level hierarchy is limited by the fixed components of the architecture, but large degrees of variation remain.

*One of the lessons of Eurisko [22] was the necessity of protecting some core of an intelligent system from self-modification/destruction.

REFERENCES

[1] Laird, J. E. *Universal Subgoaling.* PhD thesis, Carnegie-Mellon University, 1983. (Available in Laird, J. E., Rosenbloom, P. S., & Newell, A. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies,* Hingham, MA: Kluwer, 1986).

[2] Laird, J. E. *Soar User's Manual (Version 4).* Technical Report ISL-15, Xerox Palo Alto Research Center, 1986.

[3] Laird, J. E., Newell, A., & Rosenbloom, P. S. Soar: An architecture for general intelligence. *Artificial Intelligence* 33, 1987. In Press.

[4] Laird, J. E., Rosenbloom, P. S., & Newell, A. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning* 1:11-46, 1986.

[5] Steier, D. M., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R., Golding, A., Polk, T. A., Shivers, O. G., Unruh, A., & Yost, G. R. Varieties of Learning in Soar: 1987. In *Proceedings of the Fourth International Machine Learning Workshop.* Irvine, 1987. In press.

[6] Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (editor), *Attention and Performance VIII.* Erlbaum, Hillsdale, N.J., 1980.

[7] Forgy, C. L. *OPS5 Manual* Computer Science Department, Carnegie-Mellon University, 1981.

[8] Newell, A. Production systems: Models of control structures. In Chase, W. G. (editor), *Visual Information Processing,* pages 463-526. Academic Press, New York, 1973.

[9] Rosenbloom, P. S., & Newell, A. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (editors), *Machine Learning: An Artificial Intelligence Approach, Volume II.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[10] Genesereth, M. An overview of meta-level architecture. In *Proceedings of AAAI-83,* pages 119-124. Washington, D.C., 1983.

[11] Batali, J. *Computational Introspection.* A. I. Memo 701, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February, 1983.

[12] Doyle, J. *A Model for Deliberation, Action, and Introspection.* PhD thesis, Massachusetts Institute of Technology, May, 1980.

[13] Smith, B. C. *Reflection and Semantics in a Procedural Language.* Technical Report MIT/LCS/TR-272, Laboratory for Computer Science, MIT, 1982.

[14] Smith, B. C. Varieties of self-reference. In J. Halpern (editor), *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge,* pages 19-43. Monterey, CA, 1986.

[15] Smith, B. C., & J. des Rivieres. *Interim 3-Lisp Reference Manual.* Technical Report ISL-1, Xerox, Palo Alto Research Center, 1984.

[16] Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A. & Orciuch, E. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7(5):561-569, 1985.

[17] Unruh, A., Rosenbloom, P. S., & Laird, J. E. Dynamic abstraction problem solving in Soar. 1987.

[18] Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. Explanation-based generalization: A unifying view. *Machine Learning* 1:47-80, 1986.

[19] Rosenbloom, P. S., & Laird, J. E. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI-86.* Philadelphia, 1986.

[20] Laird, J. E., Rosenbloom, P. S., & Newell, A. Overgeneralization during knowledge compilation in Soar. In T. G. Dietterich (editor), *Proceedings of the Workshop on Knowledge Compilation.* AAAI/Oregon State U., Otter Crest, 1986.

[21] Rosenbloom, P. S., Laird, J. E., & Newell, A. Knowledge level learning in Soar. In *Proceedings of AAAI-87*. Seattle, 1987. In press.

[22] Lenat, D. B. EURISKO: A program that learns new heuristics and domain concepts. The nature of heuristics III: program design and results. *Artificial Intelligence* 20:61-98, 1983.

Integrating Multiple Sources of Knowledge into Designer-Soar: An Automatic Algorithm Designer

D. M. Steier and A. Newell, Carnegie Mellon University

Abstract

Designing algorithms requires diverse knowledge about general problem-solving, algorithm design, implementation techniques, and the application domain. The knowledge can come from a variety of sources, including previous design experience, and the ability to integrate knowledge from such diverse sources appears critical to the success of human algorithm designers. Such integration is feasible in an automatic design system, especially when supported by the general problem-solving and learning mechanisms in the Soar architecture. Our system, Designer-Soar, now designs several simple generate-and-test and divide-and-conquer algorithms. The system already uses several levels of abstraction, generalizes from examples, and learns from experience, transferring knowledge acquired during the design of one algorithm to aid in the design of others.

1. Introduction

The frontier of artificial intelligence research has recently been described as "figuring out how to bring more kinds of knowledge to bear [18]". This paper addresses the question of how to bring more kinds of knowledge to bear in an automatic algorithm design system. A designer should be able to use knowledge about general problem-solving, algorithm design, implementation techniques, the application domain and prior experience. We describe a system, Designer-Soar, that both applies knowledge from these different sources and acquires knowledge for transfer to future problems. We adapt and extend techniques used in Designer [9], an initial implementation of an algorithm design system, and exploit the special properties of the Soar architecture [13].

The focus of this research is on the *design* of algorithms, rather than their *implementation*. We define algorithm design to be the process of sketching a computationally feasible technique for accomplishing a specified behavior [9]. Given such a sketch, a programmer may then proceed to an efficient implementation of the algorithm. Although we focus on the early design stages of the total programming process, we expect similar issues of multiple knowledge sources to arise in later stages as well.

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499, and monitored by the Air Force Avionics Laboratory. The research was also supported in part under a Schlumberger Graduate Fellowship to David Steier. 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, Schlumberger, or the U.S. Government.

2. The Need for Knowledge Integration in Algorithm Design

By *knowledge* we mean the information about some domain, abstracted from the representation used to encode it and the processing required to make it available [20]. A *knowledge source* is a system that provides access to a body of knowledge. A knowledge source has a specific *representation* of the knowledge, comprising both symbolic structures and the means for interpreting them to influence actions, when appropriate. The problem of integration of multiple sources of knowledge arises from the diversity of representations of the sources, each of which may differ from the representation used to select actions to attain the goals of the system. It is always much easier to design a system with a single source of knowledge, where the representation for action selection can be directly adapted to it.

The kinds of knowledge relevant to algorithm design are described below. Table 1 gives typical processes in design systems that apply this knowledge.

K1. Weak methods: Designing algorithms requires solving problems. Human problem solvers (but generally not automatic systems) usually manage to make some progress, even if they don't have all the knowledge necessary in the form of powerful domain specific techniques. They resort to *weak methods*, such as generating and testing many solutions, depth-first search, etc. Human algorithm designers show particularly heavy use of *means-ends analysis* [11]. They work to reduce the differences between the current state and the goal state, resulting in problem solving driven by difficulties and opportunities detected.

K2. High-level algorithm schemes: Algorithm designers usually begin to attack a problem using design schemes. A common example is *divide-and-conquer*: splitting a problem into subproblems, solving the subproblems separately, and merging the solutions to solve the original problem [24].

K3. Transformations: Once some procedural representation of the algorithm exists, other knowledge suggests ways to reformulate and refine the procedure into a better solution. One generally applicable transformation is *recursion formation* as used in [15]. Transformations more specific to particular situations have been collected in libraries of rules [3] or programming overlays that show correspondences between plans [21].

K4. Correctness: Knowledge suggesting transformations to apply is complemented by other knowledge asserting the application of a transformation will satisfy some design goals. Particularly in