An Analysis of Soar as an Integrated Architecture*

John Laird, Mike Hucka, Scott Huffman
Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110

Paul Rosenbloom
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292

In this article we provide a brief analysis of the Soar architecture [Laird *et al.*, 1987; Laird *et al.*, 1990; Rosenbloom *et al.*, 1991a] with respect to the issues listed in the preface of this collection.

1 Integration Requirements

Soar was originally designed to be an architecture that supported a wide range of methods for problem solving. We also had the goal that it be extensible to cover the full range of tasks performed by humans. Originally, we did not attempt to closely model human behavior, but instead, used human behavior as a guideline for general intelligence. Only recently have we taken the structure of Soar as a theory for human cognition [Lewis *et al.*, 1990; Newell, 1990]. Because of our goals of generality and flexibility across a wide variety of domains, integration has always been a primary concern in the design of Soar. Toward that end, Soar has evolved over the last eight years to include automatic subgoaling, learning, and interaction with external environments. As we have been extending its capabilities, we have also been demonstrating it on tasks that require integration of many capabilities, including expert systems, cognitive modeling, robotic control, and natural language understanding.

2 Background Influences

Soar is inspired by many previous research efforts in artificial intelligence and psychology. The concept of problem spaces and heuristic search that were demonstrated in GPS [Ernst and Newell, 1969] are central to Soar. Soar's long-term memory is a production system that has its roots both in psychology and AI [Newell, 1973]. Soar grew out of the instructable production system project at Carnegie Mellon University [Rychener and Newell, 1978; Rychener, 1983] and was originally built upon Xaps2 [Rosenbloom and Newell, 1987] and then OPS5 [Forgy, 1981]. Soar's subgoaling mechanism is a generalization of earlier work on impasse-driven Repair Theory [Brown and VanLehn, 1980], while chunking has its roots in the study of human practice and skill acquisition [Newell and Rosenbloom, 1981].

3 Architecture Components

Soar can be described at three levels: the knowledge level [Newell, 1982; Rosenbloom *et al.*, 1991b], the problem space level [Newell *et al.*, 1991], and the symbol level. A knowledge level description of a system abstracts away from the underlying structure and thus does not provide any insight into integration issues except possibly that the system can produce behavior that one would expect requires the integration of multiple capabilities.

*This research was sponsored by grants NCC2-517 and NCC2-538 from NASA Ames.

3.1 Problem Space Level

At the problem space level, Soar is described in terms of the variety of problem spaces it uses on a task, and within a problem space, the states and operators it uses to solve problems within that space. At this level, we can identify problem spaces specialized for external interaction, natural language processing, design, and so on. There are also general problem spaces that support computations required in the specialized problem spaces. For example, Soar has a `selection` problem space that is used for meta-level control, and a set of problem spaces for performing arithmetic operations.

Thus, Soar does not have predefined modules for the various tasks of an intelligent agent, such as natural language understanding, natural language generation, design, etc. Instead, the architecture supports the problem spaces that contain the knowledge relevant to these tasks. These problem spaces are themselves dynamic, as new operators can be created through learning, and even new problem spaces can be created through experience.

The integration of problem spaces is determined by Soar's subgoaling mechanism, which automatically creates subgoals in response to impasses. An impasse arises when the knowledge that is directly available within the problem is insufficient (or inconsistent) for performing the basic problem space functions, such as selecting and applying operators. The purpose of the subgoal is to eliminate the impasse through problem solving in a problem space. Thus, problem spaces are used to perform the functions of another problem space that could not be performed directly.

For example, if a robot is attempting to clean up a room, it may be in a problem space with operators that include `pickup-cup`, `find-basket`, and `drop-cup`. With insufficient control knowledge, it will encounter impasses when trying to decide which operator to select, and thus create a subgoal and use a meta-level control problem space such as the `selection` problem space to break the tie. Once the problem solving in the subgoal has determined which operator is best for the current situation, possibly through a abstract look-ahead search, one of the operators will be selected, say `pickup-cup` in this case. This operator involves many coordinated actions, and thus, an impasse will arise because it can not be performed directly. Within the subgoal, the operators within a problem space for controlling the robot's arm can be used to pick up the cup.

Integration of different problem spaces, and thus the different knowledge and capabilities embedded within them, occurs in response to the demands of the task. If a task can be performed in a single problem space, it will be. On the other hand, if, for example, during an attempt to understand natural language, an ambiguity arises that requires an internal simulation of a robotic action or perceptual processing, it will occur within a subgoal. The processes of detecting an ambiguity, establishing a subgoal, and enabling the selection of a problem space for the subgoal all happen automatically via

architectural mechanisms. However, the processes of specifying which problem space to select in the subgoal, converting the ambiguity in the initial space into a problem expressed in a form understandable by the subgoal's problem space, and backwards converting the subgoal results to structures usable in the initial problem space are nonautomatic, and must be mediated by knowledge added on top of the architecture.

One additional feature of Soar is that it not only performs a run-time integration of its knowledge as demanded by the task, but it also proceeds to compile that integration through learning so that in the future the knowledge is available directly, without need to invoke the subgoals. For example, within natural language understanding, there are separate problem spaces for processing syntactic and semantic constraints. During the processing of a sentence, these are used as necessary for disambiguation, thus integrating the knowledge at run-time. Soar's learning mechanism then compiles the use of these two sources of knowledge, thus moving the fragments of knowledge from the spaces in which they originally resided to the space in which they are needed, and creating rules that combine both syntactic and semantic knowledge. In the future, the processing will be "recognitional," using the rules directly without any subgoals [Lehman *et al.*, forthcoming].

3.2 Symbol Level

The whole purpose of Soar's symbol level processing is to support the primitive functions necessary to carry out the problem space level. Thus, the symbol level must provide access to knowledge relevant to the current problem solving situation and methods for combining that knowledge for performing the problem space functions, such as selecting and applying operators. The architecture also provides the primitive support for interaction with external environments by defining the interfaces for input and output. Finally, the architecture directly supports the acquisition of new knowledge.

Soar's symbol level architecture is described in detail in Laird *et al.* (1990) and here we give a brief overview, taken in part from that source. Figure 1 presents an overview of the symbol level, including the memories, processes that work on the contents of those memories, and the flow of information between them.

The rectangles are memories, while the triangles are processes that take as their input the contents of the memories to which they are attached. Starting from the top-left, the *preference memory* receives input from the external environment through *perception modules* as well as directly from productions. Preference memory does not directly cue the retrieval of data from *recognition memory*, but is first processed by the *decision procedure*, which computes changes to *working memory* based on the contents of preference memory.

Recognition memory contains the permanent knowledge that is encoded as *productions*. Productions are simple "if-then" rules whose conditions test the conditions of working memory and whose actions create preferences. Productions in recognition memory are *matched* against working memory and if successful, add preferences to preference memory.

Preferences that propose new *context* objects (problem spaces, states, and operators) are also added to working memory so that they can serve as retrieval cues for additional preferences that will determine which object to select as current. The main flow of information is shown by the dark arrows where match computes changes to preference memory and

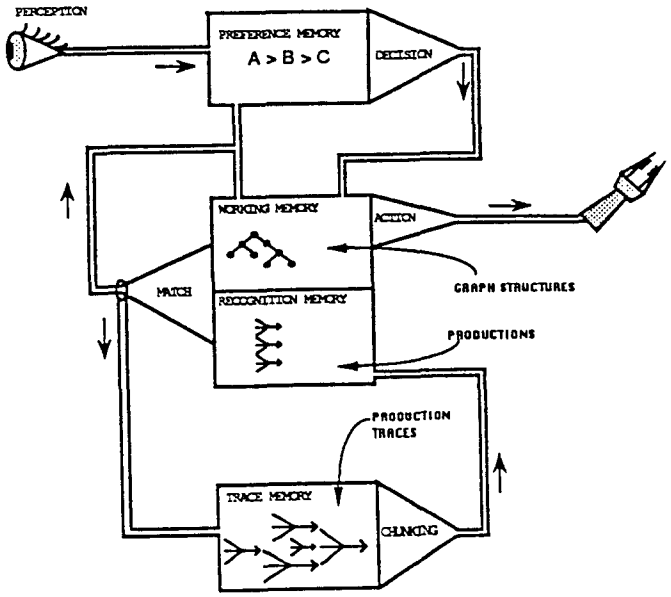

Figure 1: Soar's Symbol Level Architecture

decision computes changes to working memory, which in turn leads to changes in match. The contents of working memory are also processed by *action*, which sends motor commands to the system's effectors. As productions are matched and fired, the instantiated production traces are sent to *trace memory* which is then processed by *chunking*. The output of chunking is new productions which are added to recognition memory.

3.2.1 Integration of Learning.

The integration of chunking within the symbol level architecture has some interesting properties.

1. Learning is autonomous and not under direct control of domain knowledge. Thus, learning does not interfere with the other activities, it just happens. In constrast, there is no simple command that can be used to remember a declarative fact, but instead, Soar must engage in some indirect activity to deliberately learn. The system must somehow create a situation in which an impasse arises and the fact that is to be recalled becomes the result of the ensuing subgoal [Rosenbloom *et al.*, 1991b].

2. Different types of learning arise not from different learning mechanisms, but instead through different tasks and different methods for generating knowledge within problem spaces [Rosenbloom and Aasman, 1990].

3. Chunking as a learning mechanism can not be improved through the addition of knowledge. However, it will incorporate new knowledge for future problem solving. In particular, the quality of future learning behaviors can be improved by acquiring knowledge that alters what happens in subgoals, and thus alters what is learned.

3.2.2 Integration of Perception and Action.

The perceptual and motor systems consist of independent modules—one for each input and/or output channel—that can run asynchronously with respect to each other and to the remainder of the architecture. Perceptual modules deliver data into working memory whenever it is available. Motor modules accept commands from working memory and execute them (their progress can be monitored through sensors that deliver feedback into working memory).

Soar does not introduce any additional modules or control structures for processing perception or controlling the motor systems but instead relies on its three levels of control: productions, operators, and subgoals. At the lowest level, productions can encode incoming data in parallel. Similarly, productions can decode an operator into a series of commands to the motor system. If more deliberate parsing is required, operators or even subgoals may be involved. This is also true on the action side where the system may dynamically decompose a complex operator (such as "put the cup in the basket") into a sequence of finer-grain operators ("find the cup", "pick up the cup", "find the basket", "drop the cup"), and finally into motor commands that are executed in either parallel or sequentially (move the arm joint to 45 degrees and open the gripper).

One issue of recent concern is the potential of "choking" the system if too much data arrives too fast. This has not been a problem to date when using sonar sensors or slow vision systems, but could be a problem with real-time vision. We intend to avoid these problems by introducing an attentional mechanism that was first developed for simulated vision and has been used to model human visual attention [Wiesmeyer and Laird, 1990]. A key component of this is determining the appropriate type of features that are delivered by perception.

4 Architecture Characterization

This analysis of Soar is by necessity brief, and is intended to complement the earlier analysis that appeared in Rosenbloom *et al.* (1991a).

1. **Generality:**
   Soar has been applied to a wide variety of tasks including simple puzzles and games [Laird *et al.*, 1987]; more complex cognitive tasks, such as computer configuration [Rosenbloom *et al.*, 1985], medical diagnosis [Washington and Rosenbloom, 1989], medical analysis, natural language understanding [Lehman *et al.*, forthcoming], production scheduling [Hsu *et al.*, 1989], and algorithm discovery [Steier, 1987]; robotic control tasks, such as the control of hand-eye and mobile-hand systems [Laird *et al.*, 1991; Laird and Rosenbloom, 1990]; and cognitive modeling tasks [Lewis *et al.*, 1990; Newell, 1990]. These tasks can be characterized as fundamentally being search-based, knowledge-based, learning, or robotic tasks [Rosenbloom *et al.*, 1991a]. Soar is less well suited for purely algorithmic tasks (such as computing account receivable for a business), heavily numeric tasks, and tasks that require detailed analysis of large data sets.

2. **Versatility:**
   Soar was designed to support a "universal weak method" which responds to a task and the system's knowledge of the task with the appropriate method [Laird and Newell, 1983]. This originally covered a range of weak search methods, and more recently has been demonstrated to cover a range of planning behaviors [Laird and Rosenbloom, 1990; Rosenbloom *et al.*, 1990]. Similarly, Soar incorporates "universal subgoaling" which automatically generates all of the types of goals necessary for reasoning in a problem space [Laird, 1984]. Soar's versatility in learning has been described earlier in the section on integration of learning.

3. **Rationality:**
   Soar bases its decisions on a parallel and exhaustive retrieval of knowledge from its long-term recognition memory; the productions in this memory test the current situation and goals and suggest relevant actions. This comprises the knowledge that is *immediately available* for decision making. However, it does not cover all of the system's knowledge that may be relevant. Some of this additional knowledge is *indirectly available*—that is, it can be made accessible, but only through problem solving in one or more other problem spaces—while the remainder may just be inaccessible for this purpose (but accessible for other purposes). For example, the system may be able to simulate its own behavior internally, but the results of such a simulation are not directly available and can only be produce through significant problem solving.

   The support of both immediate and indirect knowledge can lead to two sources of irrationality through the failure to properly reflect knowledge embodied by the system. The first source of irrationality is that, since the indirectly available knowledge can only be accessed if an impasse arises—that is, because the immediately available knowledge is incomplete or inconsistent—there is the potential for a decision to be made based on only the immediately available knowledge that would not be made in the presence of the indirectly available knowledge. This irrationality can be ameliorated in those cases where the system suspects that its immediately available knowledge is incorrect—possibly because of errors it makes on a task. If sufficient time is available to ruminate further, the system can deliberate create an impasse in which it can access the indirectly available knowledge (which in turn can allow it to correct its immediately available knowledge via learning) [Laird, 1988]. However, in sufficiently time-constrained situations, irrationality may very well occur. The second source of irrationality is that some of the extant knowledge may simply be irretrievable for the current situation; for example, its retrieval might depend on the external world's being in a particular configuration that is distinct from the present one.

4. **Ability to add new knowledge:**
   All Soar's long-term knowledge is encoded as productions and new knowledge is added by writing new productions. Problem spaces provide a framework for organizing knowledge that has many of the desirable qualities of structured programming for building expert systems [Yost and Newell, 1989].

5. **Ability to learn:**
   Soar learns new productions through chunking. Chunking improves performance at the problem-space level by replacing problem solving in a subgoal with a production. Chunking gets its generality because of its integration with Soar's subgoaling scheme; it can learn whatever can be produced in a subgoal. Different types of learning arise from different types of processing in a subgoal. For example, if the problem solving in a subgoal

is deductive, the learning will be deductive. If the problem solving is inductive, the learning will be inductive. The types of results produced by a subgoal can lead to learning either from success or failure.

Learning does not modify the symbol-level components of Soar except to add productions to production memory. Thus, Soar does not improve its matcher, decision procedure, or learning mechanism. These are fixed architectural mechanisms.

6. **Taskability:**
Preliminary work has been done in using Soar's developing natural language capability [Lehman *et al.*, forthcoming] to direct the system's activities. This has been demonstrated for both simple robotic commands and instruction taking for psychological tasks [Lewis *et al.*, 1990]. Since natural language is general, it must be *operationalized* to affect system performance. Chunking over the operationalization process allows the link between language constructs and the task domain to be learned, improving performance on future tasks. The ability to integrate knowledge from diverse problem spaces can allow for a strong interaction between language processing, perception, and action.

7. **Scalability:**
The largest task Soar has been applied to is a reimplementation of Neomycin [Washington and Rosenbloom, 1989] that has over 4000 productions. This is a bit of an anomaly because a single Neomycin rule was automatically expanded to multiple Soar rules. A better example is NL-Soar, the natural language system in Soar that has approximately 900 productions and learns an additional 400 productions through chunking. One issue that could affect scalability is the impact that new productions have on the speed of the production system matcher. Initial results suggest this problem can be solved by combining restrictions on the expressiveness of the production language with parallelism [Tambe *et al.*, 1990; Tambe and Rosenbloom, 1990]. A second issue is the size of memory required to hold productions. This issue has not been addressed except to note that similar production system architectures (such as OPS5 and OPS83) have been used to create systems with 10,000 productions.

   There may be additional limits to scalability, such as an inability to organize large bodies of knowledge within a problem space framework; however it is difficult to predict them at this time.

8. **Reactivity:**
By encoding long-term knowledge in a recognition-driven fashion—as a production system—Soar is able to quickly respond to changes in its environment. When a sensor detects a change, it is relayed to working memory. If the change is irrelevant given the current context (the environment and the system's internal state and goals), no productions will fire and the change will be ignored. If the change in the environment requires a small modulation in the current activity, productions will fire to modify the current action. If the system is currently planning and the change signals that the planning is irrelevant, or that an action must be taken immediately, once again, productions will fire to suggest the appropriate action. This behavior has been demonstrated in robotic domains [Laird and Rosenbloom, 1990].

9. **Efficiency:**
Soar's reactivity is limited by the cycle time for firing productions which empirically is approximately 25 msec. on an Explorer II+. This is sufficient for control of a Hero mobile robot, where computation time is dominated by communication costs with the robot. However, it is sufficiently slow to interfere with the development of knowledge-based systems that require significant internal problem solving. The current time bottlenecks are the production match, the maintenance of data structures for determining the persistence of working memory elements, and chunking. A new implementation of Soar is currently being designed to improve its efficiency.

   There is no guaranteed bound on the response time for Soar, although we are working towards bounding the time required by various architectural components, beginning with bounding the computation required to match productions. To date, the best time bound we have been able to achieve on match (per production), without sacrificing too much in expressiveness, is linear in the number of conditions [Tambe *et al.*, 1990].

10. **Psychological or neuroscientific validity:**
Soar has been used to model many different psychological phenomena such as immediate reasoning tasks, visual attention, and conversation development [Lewis *et al.*, 1990; Newell, 1990]. We have also begun to lay out the relationship of Soar to the neural level [Newell, 1990; Rosenbloom, 1989], and to implement a version of Soar based on neural networks [Cho *et al.*, 1991].

5 Knowledge Integration

Sharing knowledge at the symbol level is not really an issue. The real issue is how both short-trem and long-term knowledge are shared at the problem space level.

Soar provides sharing of its short-term knowledge in its working memory. All productions continually match against the perceptual input and the states and operators of active problem spaces. This sharing is important for reactivity; for example, ongoing planning has continual access to perception and can be aborted if the external environment changes drastically.

All long-term knowledge is context-dependent, but can vary widely as to which contexts it is appropriate in. Thus, there is no *a priori* modularization of long-term knowledge. For example, a single operator may be shared by many problem spaces, or a bit of control knowledge may be independent of a goal (never step on a tack). However, since the behavior is structured in terms of problem spaces, it is natural for most knowledge to be sensitive to the current goal or problem space. Within a problem space, long-term knowledge and intermediate data structures can be specialized so that the reasoning in that problem space is efficient. Thus, problem spaces can provide the advantages of modularity of knowledge when necessary. For example, there are separate problem spaces for syntactic and semantic knowledge in natural language understanding. Although the problem spaces provide modularity, Soar can combine the knowledge from different problem spaces during problem solving through its impasse-driven subgoaling. Chunking can capture the processing in different problem spaces into productions that combine the different types of knowledge (and also thus move the knowledge between spaces).

Although problem spaces support integration, they do not guarantee it. When a problem space is used within a subgoal, the representation of information in working memory must be in a form that the operators of the problem space

can make use of. Thus, the operators must be able to interpret the data that is shared with other problem spaces. Information in working memory might have to be explicitly translated from one representation to another at the start of a subgoal, and then results of the subgoal might have to be translated back. Our hope is that Soar can take advantage of the efficiencies to be gained by supporting multiple representations of the same knowledge, without paying a high price in mapping between these representations. Although this translation may be initially complex and require its own subgoals, through chunking, Soar should be able to compile the translation into productions that perform the mapping directly, with little cost.

The initial modularity of long-term knowledge, combined with the ability to combine it together leads to the following evolution of a system. It starts out with very general problem spaces that cover the different types of knowledge required for the task. This knowledge is not specialized to its different uses, and may require significant processing to operationalize for a new task. With experience, chunking combines knowledge from different problem spaces, creating efficient specialized knowledge for the tasks the system has had experience in.

6 Control Integration

As in knowledge sharing, the control between components at the symbol level is fixed. The flexibility and generality of the system comes from the way in which these symbol level components interact to support the control of processing within and across problem spaces. Control is mediated by the contents of working memory, which itself is shared, and thus makes it possible for a global determination of control. All decisions for problem space activity are mediated by the knowledge encoded in productions. As stated earlier, changes can occur at any time at any level of the goal hierarchy if the consensus of the knowledge is to make a change.

7 Comparison to Other Systems

Soar is comparable to other symbolic architectures that emphasize the integration of problem solving, planning, and learning within an operator-based paradigm such as STRIPS [Fikes *et al.*, 1972], PRODIGY [Minton *et al.*, 1989] and Theo [Mitchell *et al.*, 1991].

8 Conclusion

Soar's quest for integration is based on the assumption that intelligent behavior can be be cast as attempting to achieve goals within a problem space. Thus, a single uniform framework can be adopted for all tasks and subtasks. The Soar architecture supports problem space activities through adopting uniform approaches to the basic functions of symbol processing including long-term memory, short-term memory, deliberation, interaction with external environments, and acquisition of knowledge.

References

[Brown and VanLehn, 1980] J. S. Brown and K. VanLehn. Repair Theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.

[Cho *et al.*, 1991] B. Cho, C. P. Dolan, and P. S. Rosenbloom. Neuro-Soar: A neural-network architecture for goal-oriented behavior. In preparation, 1991.

[Ernst and Newell, 1969] G. W. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, New York, 1969.

[Fikes *et al.*, 1972] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Forgy, 1981] C. L. Forgy. OPS5 user's manual. Technical report, Computer Science Department, Carnegie-Mellon University, July 1981.

[Hsu *et al.*, 1989] W. Hsu, M. Prietula, and D. Steier. Merl-Soar: Scheduling within a general architecture for intelligence. In *Proceedings of the Third International Conference on Expert Systems and the Leading Edge Production and Operations Management*, May 1989.

[Laird and Newell, 1983] J. E. Laird and A. Newell. A universal weak method: Summary of results. In *Proceedings of IJCAI-83*, Los Altos, CA, 1983. Kaufman.

[Laird and Rosenbloom, 1990] J. E. Laird and P. S. Rosenbloom. Integrating execution, planning, and learning in Soar for external environments. In *Proceedings of AAAI-90*, July 1990.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3):1–64, 1987.

[Laird *et al.*, 1990] J. E. Laird, C. B. Congdon, E. Altmann, and K. Swedlow. Soar user's manual: Version 5.2. Technical Report CSE-TR-72-90, Electrical Engineering and Computer Science Department, The University of Michigan, October 1990. Also available from The Soar Group, School of Computer Science, Carnegie Mellon University, as technical report CMU-CS-90-179.

[Laird *et al.*, 1991] J. E. Laird, M. Hucka, E. S. Yager, and C. M. Tuck. Robo-Soar: An integration of external interaction, planning and learning using Soar. *Robotics and Autonomous Systems*, 1991. In press.

[Laird, 1984] J. E. Laird. *Universal Subgoaling.* PhD thesis, Carnegie-Mellon University, 1984.

[Laird, 1988] J. E. Laird. Recovery from incorrect knowledge in Soar. In *Proceedings of the AAAI-88*, August 1988.

[Lehman *et al.*, forthcoming] J. Fain Lehman, R. Lewis, and A. Newell. Natural language comprehension in Soar. *Carnegie Mellon University Technical Report*, forthcoming.

[Lewis *et al.*, 1990] R. L. Lewis, S. B. Huffman, B. E. John, J. E. Laird, J. F. Lehman, A. Newell, P. S. Rosenbloom, T. Simon, and S. G. Tessler. Soar as a unified theory of cognition: Spring 1990. In *Proceedings of the 12th Annual Conference of the Cognitive Science Society*, pages 1035–1042, Cambridge, MA, 1990.

[Minton *et al.*, 1989] S. Minton, J. G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artficial Intelligence*, 40(1–3):163–118, 1989.

[Mitchell *et al.*, 1991] T. M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzionoi, M. Ringuette, and J. Schlimmer. Theo: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1991. In press.

[Newell and Rosenbloom, 1981] A. Newell and P. Rosenbloom. Mechanisms of skill acquisition and the law of practice. In J. R. Anderson, editor, *Learning and Cognition*. Erlbaum, Hillsdale, NJ, 1981.

[Newell *et al.*, 1991] A. Newell, G. R. Yost, J. E. Laird, P. S. Rosenbloom, and E. Altmann. Formulating the problem space computational model. In R. F. Rashid, editor, *Carnegie Mellon Computer Science: A 25 Year Commemorative*. ACM Press/Addison-Wesley, 1991. In press.

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

[Newell, 1982] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[Newell, 1990] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[Rosenbloom and Aasman, 1990] P. S. Rosenbloom and J. Aasman. Knowledge level and inductive uses of chunking (EBL). In *Proceedings of AAAI-90*, pages 821–827, Boston, 1990. AAAI, MIT Press.

[Rosenbloom and Newell, 1987]
P. S. Rosenbloom and A. Newell. Learning by chunking: A production-system model of practice. In *Production System Models of Learning and Development*, pages 221–286. Bradford Books/MIT Press, Cambridge, MA, 1987.

[Rosenbloom *et al.*, 1985] 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 Transactions on Pattern Analysis and Machine Intelligence*, 7(5):561–569, 1985.

[Rosenbloom *et al.*, 1990] P. S. Rosenbloom, S. Lee, and A. Unruh. Responding to impasses in memory-driven behavior: A framework for planning. 1990.

[Rosenbloom *et al.*, 1991a] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 1991. In press.

[Rosenbloom *et al.*, 1991b] 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, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1991. In press.

[Rosenbloom, 1989] P. S. Rosenbloom. A symbolic goal-oriented perspective on connectionism and Soar. In *Connectionism in Perspective*, pages 245–263. Elsevier (North-Holland), Amsterdam, 1989.

[Rychener and Newell, 1978] M. D. Rychener and A. Newell. An instructable production system: Basic design issues. In *Pattern-Directed Inference Systems*. Academic Press, New York, 1978.

[Rychener, 1983] M. D. Rychener. The instructable production system: A retrospective analysis. In *Machine Learning: An Artificial Intelligence Approach*. Tioga, Palo Alto, CA, 1983.

[Steier, 1987] D. M. Steier. Cypress-Soar: A case study in search and learning in algorithm design. In *Proceedings of IJCAI-87*, Milano, Italy, August 1987. Morgan Kaufmann.

[Tambe and Rosenbloom, 1990] M. Tambe and P. S. Rosenbloom. A framework for investigating production system formulations with polynomially bounded match. In *Proceedings of AAAI-90*, pages 693–700, Boston, 1990. AAAI, MIT Press.

[Tambe *et al.*, 1990] M. Tambe, A. Newell, and P. S. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5(4):299–348, 1990.

[Washington and Rosenbloom, 1989] R. Washington and P. S. Rosenbloom. Applying problem solving and learning to diagnosis. Computer Science Department, Stanford University., 1989.

[Wiesmeyer and Laird, 1990] M.D. Wiesmeyer and J.E. Laird. A computer model of visual attention. In *Proceedings of the 12th Annual Conference of the Cognitive Science Society, Cambridge, MA*, 1990.

[Yost and Newell, 1989] G. Yost and A. Newell. A problem space approach to expert system specification. In *Proceedings of IJCAI-89*, 1989.