[Steier *et al.*, 1987] D. Steier, J. E. Laird, A. Newell, P. S. Rosenbloom, et al. Varieties of learning in Soar: 1987. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*. Kluwer, 1987.

[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.

[Steier, 1990] D.M. Steier. Intelligent architectures for integration. In *Proceedings of the IEEE Conference on Systems Integration*, August 1990.

[Stobie *et al.*, 1993] I. Stobie, M. Tambe, and P. S. Rosenbloom. Flexible integration of path-planning capabilities. In W. J. Wolfe and W. H. Chun, editors, *Mobile Robots VII*, pages 52–61, 1993. Proceedings SPIE 1831.

[Tambe *et al.*, 1988] M. Tambe, D. Kalp, A. Gupta, C.L. Forgy, B.G. Milnes, and A. Newell. Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, July 1988.

[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.

[Unruh and Rosenbloom, 1989] A. Unruh and P. S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of IJCAI-89*, pages 681–687, Detroit, 1989. IJCAI.

[Ward, 1991] B. Ward. *ET-Soar: Toward an ITS for Theory-Based Representations*. PhD thesis, Carnegie Mellon University, 1991. Available as Technical Report CMU-CS-91-146.

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

[Whitehead and Russell, 1935] A. Whitehead and B. Russell. *Principia Mathematica*. The University Press, Cambridge, 1935.

[Wiesmeyer, 1991] M.D. Wiesmeyer. An operator-based model of rapid visual counting. In *Thirteenth Annual Conference of the Cognitive Science Society, Chicago*, 1991.

[Wiesmeyer, 1992] M.D. Wiesmeyer. *An Operator-Based Model of Covert Visual Attention*. PhD thesis, The University of Michigan, Ann Arbor, 1992.

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

[Rosenbloom and Newell, 1982] P. S. Rosenbloom and A. Newell. Learning by chunking: Summary of a task and a model. In *Proceedings of AAAI-82*, Menlo Park, CA, August 1982. American Association for Artificial Intelligence.

[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 and Newell, 1993] P. S. Rosenbloom and A. Newell. Symbolic architectures: Organization of intelligence. In T. A. Poggio and D. A. Glaser, editors, *Exploring Brain Functions: Models in Neuroscience*, pages 225–231. John Wiley and Sons, Chichester, England, 1993.

[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.*, 1987] P. S. Rosenbloom, J. E. Laird, and A. Newell. Knowledge-level learning in Soar. In *Proceedings of AAAI-87.* American Association for Artificial Intelligence, 1987.

[Rosenbloom *et al.*, 1988] P. S. Rosenbloom, J. E. Laird, and A. Newell. Meta-levels in Soar. In *Meta-Level Architectures and Reflection*, pages 227–240. North Holland, Amsterdam, 1988.

[Rosenbloom *et al.*, 1991] 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.

[Rosenbloom *et al.*, 1993] P. S. Rosenbloom, J. E. Laird, and A. Newell, editors. *The Soar Papers: Research on Integrated Intelligence.* MIT Press, Cambridge, MA, 1993.

[Rosenbloom *et al.*, 1994] P. S. Rosenbloom, W. L. Johnson, R. M. Jones, F. Koss, J. E. Laird, J. F. Lehman, R. Rubinoff, K. B. Schwamb, and M. Tambe. Intelligent automated agents for tactical air simulation: A progress report. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, pages 69–78, Orlando, FL, 1994.

[Rosenbloom, 1983] P. S. Rosenbloom. *The Chunking of Goal Hierarchies: A model of practice and stimulus-response compatibility.* PhD thesis, Carnegie-Mellon University, 1983.

[Ruiz and Newell, 1989] D. Ruiz and A. Newell. Tower-noticing triggers strategy-change in the Tower of Hanoi: A Soar model. In *Proceedings of the 11th Annual Conference of the Cognitive Science Society*, pages 522–529, Ann Arbor, MI, 1989.

[Rychener and Newell, 1977] M. D. Rychener and A. Newell. An instructable production system: Basic design issues. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems.* Academic Press, 1977.

[Scales, 1986] D. Scales. Efficient matching algorithms for the Soar/OPS5 production system. Technical Report KSL 86-47, Computer Science Department, Stanford University, June 1986.

[Simon *et al.*, 1991] T. Simon, A. Newell, and D. Klahr. A computational account of children's learning about number conservation. In D. Fisher, M. Pazzani, and P. Langley, editors, *Concept Formation: Knowledge and Experience in Unsupervised Learning.* Morgan Kaufmann, San Mateo, CA, 1991.

[Steier and Newell, 1988] D. M. Steier and A. Newell. Integrating multiple sources of knowledge into Designer-Soar, an automatic algorithm designer. In *Proceedings of AAAI-88.* Morgan Kaufmann, August 1988.

[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.

[Newell, 1955] A. Newell. The chess machine: An example of dealing with a complex task by adaptation. In *Proceedings of the 1955 Western Joint Computer Conference*, pages 101–108. Western Joint Computer Conference, March 1955. (RAND P-620).

[Newell, 1969] A. Newell. Heuristic programming: Ill-structured problems. In J. Aronofsky, editor, *Progress in Operations Research, III*, pages 360–414. Wiley, New York, 1969.

[Newell, 1972] A. Newell. A theoretical exploration of mechanisms for coding the stimulus. In A. Melton and E. Martin, editors, *Coding Processes in Human Memory*, pages 373–434. Winston and Sons, Washington, D. C., 1972.

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

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

[Newell, 1987] A. Newell. Unified theories of cognition: 1987 William James lectures. Available on videocassette from Harvard Psychology Department., 1987.

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

[Newell, 1992] A. Newell. Unified theories of cognition and the role of Soar. In J. Michon and A. Akyurek, editors, *Soar: A Cognitive Architecture in Perspective.* Kluwer Academic, Cambridge, Mass., 1992.

[Pearson *et al.*, 1993] D. J. Pearson, S. B. Huffman, M. B. Willis, J. E. Laird, and R. M. Jones. Intelligent multi-level control in a highly reactive domain. In *Proceedings of the International Conference on Intelligent Autonomous Systems*, 1993.

[Polk and Newell, 1988] T. A. Polk and A. Newell. Modeling human syllogistic reasoning in Soar. In *Proceedings of the 10th Annual Conference of the Cognitive Science Society*, pages 181–187, Montreal, 1988.

[Polk *et al.*, 1989] T.A. Polk, A. Newell, and R.L. Lewis. Toward a unified theory of immediate reasoning in Soar. unpublished, 1989.

[Polk, 1992] T. A. Polk. *Verbal Reasoning.* PhD thesis, Carnegie Mellon University, 1992. (Available as Carnegie Mellon University School of Computer Science Technical Report CMU-CS-92-178.).

[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 Laird, 1986] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI-86*, Philadelphia, PA, 1986. American Association for Artificial Intelligence.

[Milnes, 1992] B. G. Milnes. A specification of the Soar cognitive architecture in Z. Technical Report CMU-CS-92-169, Carnegie Mellon University School of Computer Science, 1992.

[Minton, 1990] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391, 1990.

[Mitchell *et al.*, 1986] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1, 1986.

[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.

[Modi and Westerberg, 1989] A.K. Modi and A.W. Westerberg. Integrating learning and problem solving within a chemical process designer. Presented at the Annual Meeting of the American Institute of Chemical Engineers, 1989.

[Musliner *et al.*, 1992] D. J. Musliner, E. H. Durfee, and K. G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 1992.

[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 and Simon, 1956] A. Newell and H. A Simon. The logic theory machine: A complex information processing system. *IRE Transactions on Information Theory*, IT-2:61–79, September 1956.

[Newell and Simon, 1961] A. Newell and H. A. Simon. GPS, a program that simulates human thought. In H. Billing, editor, *Lernende Automaten*, pages 109–124. Oldenbourg, Munich, 1961. (Reprinted in Feigenbaum, E. and Feldman, J. (Eds.), *Computers and Thought*, New York: McGraw-Hill, 1963).

[Newell and Simon, 1972] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, 1972.

[Newell and Steier, 1991] A. Newell and D. Steier. Intelligent control of external software systems. Technical Report EDRC 05-55-91, Engineering Design Research Center, Carnegie Mellon University, 1991.

[Newell *et al.*, 1957] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations of the logic theory machine: A case study in heuristics. In *Proceedings of the 1957 Western Joint Computer Conference*, pages 218–230. Western Joint Computer Conference, Western Joint Computer Conference, 1957. (Reprinted in Feigenbaum, E. and Feldman, J. (Eds.) Computers and Thought, New York: McGraw-Hill, 1963).

[Newell *et al.*, 1958] A. Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2:320–325, October 1958. (Reprinted in Feigenbaum, E. and Feldman, J. (Eds.) Computers and Thought, New York: McGraw-Hill, l963).

[Newell *et al.*, 1989] A. Newell, P. S. Rosenbloom, and J. E. Laird. Symbolic architectures for cognition. In M. I. Posner, editor, *Foundations of Cognitive Science*, chapter 3. Bradford Books/MIT Press, Cambridge, MA, 1989.

[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.*, 1984] J. E. Laird, P. S. Rosenbloom, and A. Newell. Towards chunking as a general learning mechanism. In *Proceedings of AAAI-84*. American Association for Artificial Intelligence, 1984.

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

[Laird *et al.*, 1986b] J. E. Laird, P. S. Rosenbloom, and A. Newell. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Kluwer Academic Publishers, Hingham, MA, 1986.

[Laird *et al.*, 1990a] 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.*, 1990b] J. E. Laird, M. Hucka, E. S. Yager, and C. M. Tuck. Correcting and extending domain knowledge using outside guidance. In *Proceedings of the Seventh International Conference on Machine Learning*, June 1990.

[Laird *et al.*, 1991] J. E. Laird, E. S. Yager, M. Hucka, and C. M. Tuck. Robo-Soar: An integration of external interaction, planning and learning using Soar. *Robotics and Autonomous Systems*, 8:113–129, 1991.

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

[Laird, 1986] J. E. Laird. *Soar User's Manual: Version 4.0.* Xerox Palo Alto Research Center, January 1986.

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

[Lehman *et al.*, 1991] J. Fain Lehman, R. Lewis, and A. Newell. Natural language comprehension in Soar: Spring 1991. *Carnegie Mellon University Technical Report*, 1991. CMU-CS-91-117.

[Lewis *et al.*, 1989] R.L. Lewis, A. Newell, and T.A. Polk. Toward a Soar theory of taking instructions for immediate reasoning tasks. In *Proceedings of the Annual Conference of the Cognitive Science Society*, August 1989.

[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.

[Lewis, 1993] R. L. Lewis. *An Architecturally-based Theory of Human Sentence Comprehension.* PhD thesis, Carnegie Mellon University, 1993.

[McDermott, 1982] J. McDermott. R1: A rule based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[Miller and Laird, 1991] C. S. Miller and J. E. Laird. A constraint-motivated model of concept formation. In *Proceedings of the Thirteenth Annual Meeting of the Cognitive Science Society*, pages 827–831, Hillsdale, NJ, 1991. Erlbaum.

[Miller, 1988] C. Miller. *Modeling Concept Acquisition in the context of a unified theory of cognition.* PhD thesis, Carnegie Mellon University, 1988.

[Forgy and McDermott, 1977] C. L. Forgy and J. McDermott. OPS, a domain-independent production system language. In *Proceedings Fifth International Joint Computer Conference*, Cambridge MA, 1977. MIT AI Laboratory.

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

[Forgy, 1982] C. L. Forgy. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–38, 1982.

[Gat, 1992] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 809–815, San Jose, CA, 1992. AAAI.

[Genesereth, 1983] M. Genesereth. An overview of meta-level architecture. In *Proceedings of AAAI-83*, Los Altos, CA, 1983. Kaufman.

[Golding *et al.*, 1987] A. Golding, P. S. Rosenbloom, and J. E. Laird. Learning general search control from outside guidance. In *Proceedings of IJCAI-87*, Milano, Italy, August 1987.

[Gupta *et al.*, 1988] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[Gupta, 1986] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie-Mellon University, 1986.

[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.

[Huffman and Laird, 1993] S. B. Huffman and J. E. Laird. Learning procedures from interactive natural language instructions. In P. Utgoff, editor, *Machine Learning: Proceedings of the Tenth International Conference (ML93)*, pages 143–150, Amherst, 1993.

[Huffman and Laird, 1994] S. B. Huffman and J. E. Laird. Learning from highly flexible tutorial instruction. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[John *et al.*, 1990] B. E. John, A. H. Vera, and A. Newell. Towards real-time GOMS. Technical Report CMU-CS-90-195, School of Computer Science, Carnegie Mellon University, 1990.

[Johnson *et al.*, 1992] T. R. Johnson, J. W. Smith, K. Johnson, N. Amra, and M. DeJongh. Diagrammatic reasoning of tabular data. Technical Report OSU-LKBMS-92-101, Laboratory for Knowledge-based Medical Systems, Ohio State University, 1992.

[Jones *et al.*, 1993] R. M. Jones, M. Tambe, J. E. Laird, and P. S. Rosenbloom. Intelligent automated agents for flight training simulators. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, pages 33–42, Orlando, FL, 1993.

[Korf, 1983] R. E. Korf. *Learning to Solve Problems by Searching for Macro-Operators.* PhD thesis, Carnegie-Mellon University, 1983. (Available as Carnegie-Mellon University Computer Science Tech. Rep. #83-138).

[Laird and Newell, 1983a] J. E. Laird and A. Newell. A universal weak method. Technical report, Computer Science Department, Carnegie-Mellon University, June 1983.

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

the goal. Thus, we rely more and more on knowledge search and the surprise was that the least-commitment, run-time control of behavior inherent in problem spaces is actually more important than the more traditional function of search. We are happy that on this second account, Allen (and Herb) were right (about the importance of problem spaces), but possibly for the wrong reason.

Acknowledgments

As should be clear from this paper, we owe a great debt to Allen Newell for his contributions to the development of Soar and our own careers. We also thank the Soar community for its vigor and support of Soar over the years.

References

[Anderson, 1983] J. R. Anderson. *The Architecture of Cognition.* Harvard University Press, Cambridge, MA, 1983.

[Bell and Newell, 1971] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.

[Brooks, 1991] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47, 1991.

[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.

[Davis, 1980] R. Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[DeJong and Mooney, 1986] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[Dietterich, 1986] T. G. Dietterich. Learning at the knowledge level. *Machine Learning*, 1:287–315, 1986.

[Doorenbos and Tambe, 1992] R. Doorenbos and A. Tambe, M.and Newell. Learning 10,000 chunks: What's it like out there? In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 830–836, Menlo Park, 1992. AAAI, AAAI Press.

[Doorenbos, 1993] R. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence.* AAAI, AAAI Press, 1993.

[Doorenbos, 1994] R. Doorenbos. Recent results on matching. In *Thirteenth Soar Workshop*, pages 62–72, Columbus, OH, 1994. Charts presented at the workshop.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[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.

this search can itself be controlled by any knowledge available to the agent (from memory or additional problem search); and by learning, the agent can convert problem search to memory search.

These distinctions are not unique to Soar, and go all the way back to the first AI systems developed in the '50s. What is unique to Soar is that it is structured so that all problems that Soar attempts can use either type of knowledge when it is available, preferring knowledge in memory when it is available. That is, in Soar, standard, non-problematic decision making is controlled directly by continual retrieval of preferences from production memory, and problem search only arises when memory search fails.

The biggest surprise in our investigation of Soar is that Soar is still with us (or possibly more correctly we are still with it) after twelve years. In AI, architectures are typically built, tested, used for one, maybe two systems, and then discarded so that the lessons learned can be incorporated in a new, better architecture that is built from scratch. Maybe Soar's longevity is just a testimony to our own stubbornness and perseverance. However, at the time Soar was created, we did not expect it to last this long. During the summer following the initial development and success of Soar, we (John Laird, Paul Rosenbloom and Allen Newell) decided to form a project for the continued investigation of general cognitive architectures. We needed a name, but Allen knew that the name of the project *could not* be Soar. Soar was just the name of our current architecture, and clearly we wanted the project to last longer than a transitory piece of software. We are happy that on this account, Allen was wrong.

One explanation for Soar's longevity may be rooted in a second surprise which concerns our understanding of the role of problem spaces in producing intelligent behavior. Originally Allen and Herb Simon conceived of problem spaces as an arena for search [Newell and Simon, 1972]. Allen's original statement of the Problem Space Hypothesis was that "all symbolic goal-oriented activity could be cast as search in a problem space" [Newell, 1980]. This statement became more and more awkward as many of our systems did less and less search, but still successfully solved problems using problem spaces. Instead of searching, they relied on the "situatedness" inherent to Soar. That is, each decision in Soar combines the available knowledge by retrieving preferences for action from production memory based on its sensing of the environment, its current goals, and its interpretation of the environment in light of

These problems aside, where are we on our quest for creating an architecture that supports general intelligence? We have come along way from the eight puzzle and hill climbing, but clearly we have a long way to go, with the need to generalize and extend many of the capabilities in Soar that are currently ad hoc and incomplete. For example

1. Some essential forms of learning, such as episodic (especially auto-biographical) and declarative (especially inductive) are still more complex and difficult, and less routine, than it seems they ought to be.
2. Learning is still fragile. Problems remain of overgeneralization, undergeneralization, utility, and working appropriately in the presence of a dynamic environment.
3. Ways of coping with multiple interacting goals are still rather ad hoc and not routine.
4. Spatial and temporal reasoning (and more generally, dealing with non-symbolic inputs and outputs) are still rather ad hoc and incomplete.
5. Soar is still radically incomplete with respect to being a full unified theory of cognition, and incorrect in a number of obvious ways (such as having an unbounded working memory).

Although there is a lot left to be done (thus guaranteeing that we will have no problem finding research projects for our graduate students for the foreseeable future), Soar has been a successful vehicle in our quest as evidenced by the range of systems that have been built with it. Our belief is that this success can be traced to the central idea in Soar, which is that there are two distinct sources of knowledge for controlling behavior. One source is the system's long-term memory. This memory is under the architecture's control and can be organized for efficient retrieval of knowledge relevant to the current situation. Thus, *memory search* need not be inherently combinatorial, and the associative retrieval of a production system is a useful way to organize long-term knowledge.

The second source of knowledge is the system's explicit inferences about the problem; that is, its search through an appropriate problem space. For novel problems, that search can *generate* new situations, i.e., new states, that the system has never previously considered. Thus, *problem search* can generate new knowledge and new options to consider, making it impossible for the architecture to avoid combinatorial search in all novel problems. However,

8.1 Implementation Level

Once it was clear that a complete reimplementation was called for, the next decision to be made was whether it should be in Lisp or some other language (in particular, C). Driven by concerns about portability and efficiency, the decision was eventually made to reimplement in C. In addition to the reimplementation, a formal specification of Soar 5 in Z was created [Milnes, 1992] to aid in constructing a correct implementation. The new implementation was designed not only for speed and maintainability, but also for scalability so that very large numbers of productions could be learned. To preserve the ability to compare systems in both Soar 5 and Soar 6, only a few very, relatively minor conceptual changes were made to the underlying architecture.

The resulting system has the following characteristics:

1. Soar 6 is 15-20 times faster than Soar 5 for medium size tasks (1000 productions).
2. Soar 6 learns over 1,000,000 productions without significant performance degradation for certain tasks [Doorenbos, 1994].
3. Soar 6 has run over 4,000,000 production firings on a single task (simulated aircraft control [Pearson *et al.*, 1993]) without significant performance degradation.

The performance improvements of Soar 6 have opened up new classes of tasks, such as real-time simulated aircraft control [Pearson *et al.*, 1993] and tactical behavior [Jones *et al.*, 1993; Rosenbloom *et al.*, 1994], and interactive natural language instruction [Huffman and Laird, 1993].

Conclusion

Our efforts over the last twelve years have been focussed on the scientific issues surrounding the development of a general cognitive architecture. This focus has been useful in the development and evolution of Soar in terms of functionality, but it has also not come without cost. For years we ignored software engineering, user interface and learnability issues that have made learning and using Soar sometimes more than a small challenge. In the reimplementation of Soar 6 we addressed many of the software engineering problems, but Soar is still difficult for novices to learn and use — an issue we continue to struggle with.

ning system. The real-time system consists of the three lower levels, while the reflective level allows unlimited planning. These levels can execute in parallel, and knowledge about activity (usually represented as preferences) is transferred from the planning/reflective level to the others through chunking. This makes Soar looks similar to multi-level agent architectures such as Atlantis [Gat, 1992] and Circa [Musliner *et al.*, 1992], but with some key differences (many of which are shared with Theo [Mitchell *et al.*, 1991]): Soar's levels are all based on a single underlying representation of knowledge (productions), its levels arise dynamically based on the needs of the problem, and its learning leads to a gradual transition from a reflective to a reactive system.

The range of tasks to which Soar 5 was applied included a number of robotic control systems: high-level control of a Puma arm using camera input for vision [Laird *et al.*, 1991], control of a Hero mobile robot [Laird and Rosenbloom, 1990], navigation in a 2D simulated mobile robot domain [Stobie *et al.*, 1993], control of simulated aircraft [Pearson *et al.*, 1993], and video game control [John *et al.*, 1990]. Another class of systems where built that interacted with other software systems [Newell and Steier, 1991] including databases, symbolic mathematics packages, chemical process simulators, drawing packages, tutorial environments [Ward, 1991], building-design tools [Steier, 1990], and physical-world simulators. A full summary of the domains tackled with Soar 5 can be found in the Introduction to [Rosenbloom *et al.*, 1993].

8 Soar 6 (1992)

Although Soar 5 broadened the types of domains that Soar could be applied to, it was not without cost. Soar 5 was created by modifying Soar 4, which in turn was based on the code for Soar 3 and Soar 2. There was still some of the original Lisp code for the RETE matcher from OPS5. The patchwork nature of the code made maintenance difficult. In addition, many of the algorithms and implementations developed for those earlier versions were extremely inefficient under the new architecture. Many programs ran a factor of 3 slower under Soar 5 than under Soar 4. Thus Soar 6 was generated as a complete reengineering and reimplementation of Soar 5.

Figure 7: Multiple Levels of Soar Architecture.

timing and semantic penetrability.

Another way of looking at Soar 5 as an architecture to support external interaction comes from looking at increasing time-scales of activities, as shown in Figure 7. At the fastest time-scale, fixed input and output modules interface Soar to its external environment. Input modules install data from sensors into working memory. Output modules detect motor commands in working memory and transfer them to the motor system.

The next level is the reactive level, at which productions match in parallel to the situation in working memory. This level provides for quick reflexive responses to changes in the environment, but does not allow for the integration of knowledge — a production is an isolated piece of knowledge.

The third level is the deliberative level, at which the functions of problem spaces are performed, such as selecting and applying operators. This level is less immediately responsive than the reactive level, as it is implemented by sequences of parallel production firings, plus a decision; however, it makes up for this by enabling two forms of knowledge integration in its action generation: first, through sequences of production firings, where the actions of one production are conditional on the actions of those that fired before it; and second through the combination of preferences in the decision procedure. The fact that the elaboration phase runs until quiescence means that there is an exhaustive access of all directly available knowledge in long-term memory. Thus, as new productions are added to long-term memory, the deliberative level automatically will use them when they are relevant. This is the base level of knowledge-intensive behavior — behavior where the system knows exactly what to do at each step.

The fourth, and final, level is the reflective level at which the functions of problem spaces can themselves be the object of problem solving in subgoals. This level is the least responsive timewise, but it also provides the most flexibility, by allowing arbitrary problem solving in producing its results. It can involve planning, reasoning by analogy, hierarchical decomposition, and other complex problem solving methods. It is also the source of new knowledge, because all results of subgoals lead to the creation of new productions that augment the other levels.

Taking these levels together, Soar 5 can be viewed as a combination real-time and plan-

7.3.4 Maintaining the coherence of problem spaces

In previous versions of Soar, the current state was always a "valid" state in the problem space. During operator application, the new state being constructed was not the current state. The system would "jump" from one valid state to the next. However, in Soar 5, once an operator is selected, the state changes in place. During application, the state may be in a state that is not valid within the problem space. For example, if the state consisted of a set of blocks, during application, a block might temporarily not have a location because its old location was deleted and a new location is being computed. However, when the operator is terminated, the state must be valid. That must be one of the post-conditions of the operator. The fact that the state can be temporarily invalid would be cause for concern if a decision were based on it; however all operator decisions are held off until operator termination, so that when a new operator is selected, it is for a valid state of the problem space.

In order to implement the new "termination" problem space function, a new preference was added called *reconsider*. The existence of this preference for a context slot (problem space, state, operator) signifies that a decision can be made for that slot. When a decision is made for the operator slot, it signifies that the current operator has terminated.

7.4 Implementation Level

Soar 5 was built on top of the existing implementation of Soar 4 in Common Lisp. Although it provided new functionality, the dynamic creation of justifications led to significant slowdowns, even though time was saved by eliminating most state copying.

7.5 Results

Soar 5 was successful in supported interaction with external environments. Input could be asynchronous, and the recognition nature of the production system allowed systems to be built that could be reactive to changes in their environment. In addition, it supported interruption, planning, hierarchical execution, and learning that converted planning into reactive execution [Laird and Rosenbloom, 1990]. However, problems — and controversies — still remain in understanding exactly how learning should occur in the presence of external interaction so that behavior after learning maintains appropriate properties with respect to

Figure 6: Processing Cycle of Soar 4 and Soar 5

matcher then detects when any of the conditions of the justification are no longer satisfied and retracts the result.

7.3.3 Overlapped application and selection

In previous versions of Soar, the basic cycle of progress was select a state, propose and select an operator, apply an operator and then select a state. This is shown in the top of Figure 6. In Soar 5, the process compacts so that following the selection of the initial state, operators are proposed and the current operator is selected, the operator is applied until termination, when a new operator is selected, and so on. Under this scheme, knowledge to select a future operator is applied in parallel with the application of the current operator.

preferences must have two different classes of persistence. The first class persists as long as the production matches from which the preferences were created are still satisfied, and the second persists independent of the satisfaction of the productions that created them.[2]

A key issue is how to determine which preferences should fall in which persistence class. Such decisions cannot be done solely by a manual labeling of a production because productions can be learned via chunking. Similarly, manual labeling of actions will not work, because even though the actions of a chunk are based on the actions of an existing production, the exact role the action plays — operator application or entailment — can be different in a subgoal, where the operator may participate in the application of a operator, and in the goal where the chunk applies, where the action might be an entailment that should retract when its enabling conditions are no longer satisfied.

It is difficult to determine the exact right way of doing this; however, for Soar 5, we adopted the strategy of using the problem space function of the action to determine its persistence. This is possible in Soar by examining the types of working memory elements tested in the condition of the production, and the type of object the preference is for. These factors determine whether or not the production fulfills the function of operator application. For example, if a preference is generated by a production that tests a state and an operator and modifies the state, then it is part of operator application and does not retract. This is an example where the problem space formulation plays a central role in the architecture.

In previous versions of Soar, the result of a subgoal would persist until the supergoal structure to which it was linked was removed from working memory. In Soar 5, some preferences must retract when the conditions of the production that created them no longer match (see above). However, the result of a subgoal should not be retracted when the conditions of the production that created it (i.e., the production in the subgoal) are retracted. Instead, its persistence must be equivalent to what it would have been if it had been created by the chunk learned from the subgoal (otherwise results would retract as soon as the subgoals that created them disappear). Therefore, a new structure must be built for each result; a *justification* that can be added to the matcher to represent the persistence of the result. The

[2]The complete details of the persistence scheme are available in the Soar 5.2 manual [Laird *et al.*, 1990a] and the Soar 5 specification [Milnes, 1992].

a production no longer directly changes working memory, but instead creates preferences for how working memory should be changed. This maintains the monotonic nature of production actions, thus avoiding action conflicts. The productions still only create structures, but the structures they create have a non-monotonic effect on working memory (through the decision procedure). If there is a conflict between preferences, this also can be represented in working memory as an impasse, but not one that necessarily prevents progress in problem solving, as does an impasse on a context slot. Instead such an impasse represents indecision as to the value of a particular structure. If progress (context decisions) can be made without resolving this indecision, that is fine.

7.3.2 Persistence

Although the preference scheme allows for non-monotonic changes to working memory, by changing a state in place instead of creating a new one from scratch, we were faced with issues of the consistency and persistence of structures in the state. In previous versions of Soar, a new state was created from scratch. If a part of the state was an entailment from other parts of the state, it would be computed anew for any new state. Thus, although there was duplication in the derivation of the entailment of a state, a new state would be consistent. In Soar 5, structures in the state could be based on data that was later modified by an operator. These structures would have to be removed or recomputed to retain the consistency of the state. In order to avoid requiring domain knowledge to explicitly remove these structures, the semantics of production was changed so that their actions would "retract" when their conditions no longer matched. Thus, a production that calculated an evaluation would retract its result when the aspects of the state it was based on changed (and possibly, independently add a new evaluation based on the updated information). Surprisingly, this "truth maintenance" calculation is provided at no additional computational cost by the RETE match algorithm employed by Soar.

However, retraction semantics cannot be adopted by all productions. Specifically, productions that participate in applying an operator must test existing portions of the state and assert preferences to change them. Such changes must persist beyond the match of the production's conditions, or the change would retract as soon as it was generated. Thus, to addition changes to the architecture, including implementation of an automatic persistence scheme, a modification to the elaboration/decision cycle, and the explicit signalling of operator termination via a new preference.

7.3.1 Destructive State Modification

In previous versions of Soar, an operator applied by creating a new state symbol, copying over much of the prior state, and the adding in those aspects of the state required by the application of the operator. This is essentially the situation variable model adopted in many logic-based problem solvers. Unfortunately, it requires a large amount of copying, especially for large states. Under the single state principle, the creation of completely new states is not possible, and instead, the current state is modified in place, a process called, *destructive state modification.*

In terms of frame axioms, the previous versions of Soar required axioms as to what stayed the same between states. In contrast, Soar 5 requires axioms as to what changes between states (like STRIPS, but more flexible in representation).

By adopting the single state principle and the associated destructive state modification approach to operator application, it was necessary to devise an approach to applying operators that was still consistent with encoding the actions of the operators in an associative long-term memory. We were still committed to having operators whose executions were composed at run time by relevant productions, as opposed to being limited to fixed declarative descriptions of the actions and post-conditions of the operator (as in STRIPS). On the positive side, productions as well as impasse-driven subgoals promised to provide a rich medium for representing conditional and time-dependent actions. On the negative side, all previous Soar systems assumed that all production firings were monotonic so as to avoid conflicts between actions. This was critical to avoid arbitrary action resolution schemes where some source of knowledge (a production) is masked. The only non-monotonic changes were made by the decision procedure — to select new problem spaces, states, or operators — where they could be directly influenced by the system's knowledge.

To support destructive changes to the state, we extended the preference scheme and decision procedure to apply to all working memory elements, not just the context slots. Thus,

7.2.4 Interaction

In previous versions of Soar, it was difficult to incorporate changes from the outside into the problem space model. Since all state transitions led to new states, conceptually, any input from the outside would have to lead to a new state. This was handled by not allowing asynchronous input; all input came in by explicit request during an operator application. In contrast, in Soar 5, it is easy to extend the problem space model so that the input from sensors dynamically changes the state. As the environment changes, the internal representation of those aspects of the environment sensed by the system also change. Under this model, the top problem space includes a representation of the external environment, although only a portion of the state is directly available to the system. Changes in that problem space can be effected by the agent, through actions on the environment and its own internal structures, as well as other processes in the environment.

7.2.5 Hierarchical Execution

In previous versions of Soar, a subgoal to implement an operator could perform arbitrary processing, but it would always terminate by generating a new state. It was not possible for the subgoal to incrementally modify the current state. Thus, it was not possible to have complex operators that required incremental interaction with an environment. In Soar 5, the implementation of a complex operator, such as "pick up the block" can be performed in a subgoal via more primitive operators, such as "move gripper", as shown in Figure 3. The state of the subgoal is the same as the state to which the "pick up the block" operator is being applied. The problem solving in the subgoal can incrementally perform the operator. This problem solving can be time-dependent, require feedback from the environment, or even involve planning.

7.3 Symbolic Architecture Level

To support the single state principle, and thus support interaction with external environments and eliminate extensive state copying, the architecture level was significantly changed. The most significant of these changes was *destructive state modification*. Adopting this led

7.2.2 Backtracking

In earlier versions of Soar, the ability to select previously generated states allowed the system to "backtrack" during problem solving within a single subgoal. Under the single state principle, such backtracking is no longer possible. Surprisingly, little backtracking of this sort actually ever occurred in Soar programs. The reason is that whenever there was a uncertainty in a decision, there would be an impasse and a look-ahead search across a series of subgoals would determine which was the best choice. This look-ahead search might generate multiple states, but these states were always the single current state in some goal in the context hierarchy. Figure 4 shows that all of the choice points during planning are initial states of evaluation goals. Thus, most backtracking was *across goals* and not within goals. The single state principle still allows this behavior, so that Soar can still search, as long as it is across goals using internal states.

7.2.3 Operator Termination

In previous versions of Soar, the termination of an operator was signaled by the selection of a new state. Thus, there was no need to explicitly signal that an operator had terminated. Also, all operator applications were under complete control of the system, so if it had the knowledge to apply an operator, then and only then was the operator completed. In contrast, Soar 5 has no state selection so that once an operator is selected, there must be some signal that the operator is complete so that a new operator can be selected. In addition, it is possible that completion of an operator requiring interaction with the external environment will be uncertain. There must be an explicit test that the post-conditions of the operator have been achieved. Merely wishing that a block is stacked on another does not make it come true, and conversely, it is possible that the stacking of a block will be carried out by another agent, even before the system has a chance to work on it. These factors combined to lead to the requirement for a new problem space function: termination.

to modify the external environment, and the environment has its own dynamics, possibly including other agents. The actions of the problem solver may require time to execute and their effects may be difficult to predict. The dynamics of the world implies that the problem solver cannot simply backtrack out of undersirable situations in an external worl, but must press ahead from its current state.

7.2 Problem Space Level

The characteristics of external environments listed above challenge not only Soar, but also the standard formulation of problem spaces as used by most of AI. This in turn has led to dissatisfaction by many with symbolic AI for external interaction. One response has been to abandon the problem space model for interaction with external environments, sometimes denying any internal representation of the problem [Brooks, 1991]. Often, when the problem space model is used, it is disassociated from execution and used only for planning activity, with the actual execution performed by separate "customized" modules [Fikes *et al.*, 1972; Gat, 1992]. The challenge for Soar 5 was to develop an approach which maintained a single formulation of problem spaces for both execution and planning.

Soar 4's formulation of operators and states was the source of its incompatibilities with external interaction. In Soar 1-4, every operator application created a new state, which was then usually selected to be the current state. All previously generated states were available within the problem space. For Soar 5, we adopted the *single state principle*, with the result being that there is a single state — the current state — available within a goal at any time. Adopting this principle has a number of ramifications as listed below.

7.2.1 State Selection

In previous versions of Soar, the decision to select the current state was one of the problem space functions. Under the single state principle, once the initial state is selected, there is no further state selection within a goal.

unchanged state information. A second, seemingly unrelated problem was that programs in Soar 4 had difficulty interacting with external environments. Almost all of the tasks encoded in Soar 4 were internal to the system itself. If it was solving blocks world problems, it solved them in an internal model of the blocks world. The crux of the problem was that there was no theory as to how Soar systems should interact with the world. There were some basic facilities from OPS5 for reading and writing text; however, the production system would stop running while waiting for the completion of the input (and chunking couldn't appropriately capture these interactions).

The remedy to both of these problems was to adopt a new symbol-level approach to operator application and state maintenance, called *destructive state modification*. This in turn led to a revision of the problem space computational model through the adoption of the *single state principle*: at any point in time there should be at most one state active for each goal (i.e., a stock of one state). Newell observed that this principle was based on both functional and psychological grounds. Psychologically, search strategies such as progressive deepening arise because of the trouble people have in maintaining more than one internal state (plus the external one that is available via perception). Functionally, it was a burden on Soar's matching capabilities to maintain so many states in working memory.

The Soar 5 manual provides the most complete description of Soar 5 [Laird *et al.*, 1990a].

7.1 Knowledge Level

At the knowledge level, the changes made to Soar 5 were meant to expand the types of tasks to which Soar could be applied, and in turn, expand the range of knowledge that the system could encode and use. In Soar 4, the problem solver had access to the complete state of a problem, and had complete control of changes to the current state. These changes could be made instantaneously, with complete certainty. Moreover, if the problem solver ever found itself in an undesirable state, it could always backtrack to some earlier state and continue from there.

In contrast, interaction with external environments violates many of these assumptions. In general, the problem solver can perceive only a limited portion of the current state of the problem and the data it receives may have errors. The problem solver has only limited ability originally every six months, and now every eight months so that by the spring of 1994 there have been a total of thirteen Soar workshops.

With the increase in the number of users, there was significant activity in creating new applications of Soar. On the AI side this included applications such as algorithm design (Designer-Soar and Cypress-Soar [Steier and Newell, 1988; Steier, 1987]), medical diagnosis [Washington and Rosenbloom, 1989], blood analysis [Johnson *et al.*, 1992], production line scheduling [Hsu *et al.*, 1989], chemical process modeling [Modi and Westerberg, 1989], natural language understanding [Lehman *et al.*, 1991], and intelligent tutoring [Ward, 1991].

1986 was also the start of one of the major research directions of Soar. It was during this time that Allen Newell proposed that many of the assumptions embedded in Soar were an appropriate basis for modeling human cognition. Newell's proposal was also based on the assumption that the time had come to consider *unified theories of cognition* (UTC), that is, theories that attempt to cover a broad range of psychological behavior. He proposed that the basis of such theories would be architectures, such as Soar or ACT* [Anderson, 1983]. The investigation of Soar as a UTC was spurred by Newell's presentation of the William James Lectures at Harvard in 1987 [Newell, 1987]. In preparation for these lectures, the Soar group set out to model a variety of human behavior in Soar. Newell's book, *Unified Theories of Cognition* (1990), captures his vision of a unified theory, and proposes Soar as a candidate. Work on Soar as a UTC has continued and has become one of the major thrusts of research within the Soar community [Lewis *et al.*, 1990; Newell, 1992]. Areas of research include immediate reasoning tasks [Polk *et al.*, 1989], syllogisms [Polk and Newell, 1988], verbal reasoning [Polk, 1992], number conservation [Simon *et al.*, 1991], problem solving [Ruiz and Newell, 1989], instruction taking [Lewis *et al.*, 1989; Huffman and Laird, 1993], visual attention [Wiesmeyer, 1991; Wiesmeyer, 1992], concept acquisition [Miller and Laird, 1991], and natural language understanding [Lehman *et al.*, 1991; Lewis, 1993].

7 Soar 5 (1989)

Although Soar 4 led to a greater use of Soar, it still had significant weaknesses. One was that for problem spaces with large state descriptions, the application of an operator, and the ensuing creation of a new state would require computationally expensive copying of all of the advantage of this approach is that there is transfer of subparts of some macro-operators to other macro-operators, thus decreasing the number of cases that must be learned [Laird *et al.*, 1986a].

The last demonstration of chunking in Soar 3 was within the context of R1-Soar, a re-implementation of part of the original R1 expert system for computer configuration [Rosenbloom *et al.*, 1985]. R1-Soar was the first knowledge-rich task encoded in Soar. In R1-Soar, the knowledge for configuring computers was encoded within a hierarchy of problem spaces. Through chunking, the system was able to reduce the time to configure computers, not only for later runs, but also for the initial run. The reason was that some of the configuration work was duplicated during a single run, and chunks learned during the initial part of the configuration transferred to later parts. One way to describe the action of chunking is that it compiled the deep general knowledge encoded in the hierarchy of problem spaces into more specialized, but efficient surface knowledge.

Since Soar 3, chunking has been an inherent part of Soar, with many systems using it for a variety of purposes. Some of the types of learning that have been demonstrated via chunking include strategy acquisition, macro-operator acquisition, learning from advice [Golding *et al.*, 1987; Laird *et al.*, 1990b], learning from instruction [Huffman and Laird, 1993; Huffman and Laird, 1994], learning from abstraction [Unruh and Rosenbloom, 1989], task acquisition [Yost and Newell, 1989], inductive learning [Rosenbloom and Aasman, 1990; Miller and Laird, 1991; Miller, 1988], constraint compilation [Newell, 1990], explanation-based learning [Rosenbloom and Laird, 1986], learning by analogy, and recovery and relearning [Laird, 1988]. The early work on these was summarized in Steier et al. (1987).

6 Soar 4 (1986)

The goal of Soar 4 was to create a version of Soar that could be released to users outside of the development group. This involved porting Soar to Common Lisp, fixing lots of bugs, improving the interface, and writing a manual [Laird, 1986]. Although Soar continued to be relatively hard to use and learn, the number of applications and users grew rapidly, so that by 1988 there were approximately 50 users of Soar 4. To maintain cohesion, especially since the project had become distributed across the country, workshops on Soar were started, the correctness of a result; for example, a rule that prefers the cheaper of two partial computer configurations could be used to drive the system towards the goal of a cheap full configuration, rather than explicitly comparing the costs of the full configurations as part of the goal test. In response to this, we introduced two additional preferences: require and prohibit. The semantics of require were that the object (usually an operator) *must* be selected for the goal to be achieved. If the object cannot be selected (because there is more than one required, or it is also prohibited), then there is an impasse. The semantics of prohibit were that the object *must not* be selected for the goal to be achieved. The traces for these productions were included in deriving chunks because they encoded goal completion knowledge and not just efficiency knowledge.

- **Match cost:**
  As more and more productions are learned, a serious issue is whether Soar's matcher slows down. This has been called the utility problem [Minton, 1990]. In response, the development of efficient, and bounded, matching strategies has been a significant path of research within Soar [Gupta, 1986; Gupta *et al.*, 1988; Tambe *et al.*, 1988; Tambe *et al.*, 1990]. Most recently, Bob Doorenbos has been studying systems that learn large numbers of productions (100,000) and has demonstrated for some tasks, there is no utility problem [Doorenbos and Tambe, 1992; Doorenbos, 1993].

5.4 Implementation Level

Little changed at the implementation level in moving from Soar 2 to Soar 3 other than porting it to additional versions of Lisp, thus allowing it to run on a broader set of machines.

5.5 Results

The major result of Soar 3 was the integration of chunking with Soar. The demonstration of transfer with chunking in Soar was done for simple toy tasks, such as the Eight Puzzle, Tower of Hanoi, and TicTacToe [Laird *et al.*, 1984]. We also demonstrated that chunking could learn macro-operators similar to Korf's work [Korf, 1983]. One interesting result was that instead of representing macro-operators as monolithic structures as in a macro-table, each macro-operator was composed of chunks that selected each substep of the macro. The tion firing within a subgoal were saved. We also separated the results into groups that were independent so that more than one chunk could be learned for a subgoal. The conditions of productions were then determined by tracing back from results, through the production traces, to those working memory elements that were connected to a supercontext. This increased the generality of the chunks by eliminating conditions that were not necessary for generating the results that became the actions of the chunk.

- **Condition Ordering:**

  The order in which the conditions of a production are matched can have an enormous impact on the cost of the match (as with other systems that perform conjunctive queries). In standard production systems, all productions are written by hand, so the programmer has the opportunity to his or her knowledge to provide an appropriate ordering on the conditions of the productions they write. With chunking, we were faced with productions that were created automatically. Initial implementations ordered the conditions of learned productions arbitrarily, and very quickly led to severe performance problems. In response, an automatic condition reorderer was developed to attempt to minimize matching time [Scales, 1986].

- **Incremental Chunking:**

  Chunks were originally built only upon subgoal termination, but there was actually no functional restriction that forced this approach. We modified chunking so that chunks were built for a result as soon as it was generated in a subgoal. This led to increased transfer because some chunks could fire immediately, eliminating the need for further duplicate problem solving within a subgoal.

- **Incorporation of Path Constraints:**

  One issue that arose with EBL-style chunking was whether all productions that contributed to producing a result should be included in the dependency analysis. The final decision was that search-control productions, those that created desirability preferences, would be not be included in the backtrace. These productions should affect the speed with which a solution is found, but not the *correctness* of the results. However, we also recognized that sometimes control knowledge might be used to influence

1. Problem space proposal and selection.
2. Initial state proposal and selection.
3. Operator proposal, selection and application.

Moreover, separate learning mechanisms are not required for learning control knowledge, operator creation and application knowledge, or problem formulation knowledge. However, knowledge must be encoded within the system as problem spaces that can generate the appropriate results, either from underlying domain knowledge, or from generation spaces constrained by external observations. The result of a subgoal can be success, failure, or some intermediate data structure. Chunking captures the processing leading to the result, independent of the semantic content of the result.

5.3 Symbolic Architecture Level

At the symbol level, the challenge is to create an architecture in which the processing of any subgoal can be captured by a production and conversely, anything represented in a production can be learned through chunking the problem solving of a subgoal. Over the years, there has been significant refinements of chunking and other aspects of the architecture to eliminate most of the cases where the processing in a subgoal can not be accurately and precisely represented within productions and vice versa.

Below is a chronology of some of the early developments in chunking.

- **Initial Implementation:**

  The first implementation of chunking in Soar kept track of all production firings during a subgoal, as well as all results produced during the subgoal. When the subgoal terminated, all working memory elements that were tested by productions that fired in the subgoal, and existed before the subgoal, became the basis for conditions. These working memory elements, together with the results (which became the actions), were variablized and then reordered to form the conditions and actions of chunks.

- **Backtracing:**

  Based on a talk by Tom Mitchell on goal-dependent learning, which would later become explanation-based generalization, we modified chunking so that traces of each producstrings of English) — then experience can determine which of these structures are actually useful and should be learned. If it is possible to distinguish what has been learned (and thus experienced) versus what could possibly occur, then this splits the implicational closure into a segment encoding knowledge (because it has been experienced) versus a segment that does not. Chunking over experiences moves knowledge from the latter segment to the former, thus performing learning at the knowledge level [Rosenbloom *et al.*, 1987; Rosenbloom and Aasman, 1990].

5.2 Problem Space Level

Within the problem space computational model, learning plays a communicative role. It is not one of the basic functions required for making progress, but instead moves knowledge into a problem space from the world and from other problem spaces (in subgoals). It thus enables direct performance of problem space functions that previously required consulting the world or other spaces. For example, if there is a subgoal to select an operator, the subgoal's problem will be to determine which operator is the best for the current situation. Once a determination has been made, a preference will be created to select the best operator. A chunk will be built that summarizes the processing in the subgoal and creates the appropriate preference. The chunk encodes search control knowledge which, in the future, can directly contribute to operator selection. In similar situations, a subgoal will not arise because the chunk will fire and create the preference, thus avoiding any impasse.

A central aspect of the design of chunking in Soar is that it does not interfere with the regular problem space functions. Instead, it is a background process that is invoked automatically whenever a subgoal result is produced, The problem solving has no direct control or sensing of chunking — learning occurs in parallel with problem solving. The intent is for Soar to learn unobtrusively, incrementally and continually on all tasks.

Although chunking is a fixed architectural mechanisms, the actual semantic content of what is learned can be as varied as the types of reasoning and problem solving that can be encoded in the problem spaces of subgoals. Also, because chunks arise from subgoals, and subgoals themselves can arise for any problem space function, chunks can encode any of the problem solving functions. Thus, chunks can be learned for the following:

1983; Laird *et al.*, 1986b]. This extended, task-independent chunking model was implemented as part of a new productions system architecture that was designed specifically to support it — XAPS3. During the summer and fall of 1983, we discussed how to incorporate chunking into Soar, and finally in January of 1984, we extended Soar 2 to include chunking [Laird *et al.*, 1984]. At the time, the modifications to the architecture were minor and thus did not lead to a new version of Soar. Soar 3, implemented during the summer of 1984, was a partial rewrite of Soar 2, refining the subgoaling scheme, eliminating decision productions and moving the decision procedure into the architecture (were it was more efficient). For this paper, we will take a revisionist approach and consider the major contribution of Soar 3 to be the addition of chunking, which is described in Laird, Rosenbloom and Newell (1986) .

5.1 Knowledge Level

At the knowledge level, learning is only relevant if it changes the knowledge available to the agent as it attempts to select actions to achieve its goals. Chunking, as it was originally formulated, cached the results of subgoal-based problem solving as productions. The productions, called chunks, summarized the problem solving of the subgoals, and in the future, the chunks would fire in situations that previously would have led to subgoals. Thus, chunking is a form of speed-up learning, moving knowledge from where deliberation in a subgoal is required, to productions, where it is directly available. Although this type of learning affects performance at the problem space and symbol levels, it has been argued that along with explanation-based learning (EBL) [Mitchell *et al.*, 1986; DeJong and Mooney, 1986], to which chunking is closely related [Rosenbloom and Laird, 1986], it does not produce learning at the knowledge level [Dietterich, 1986]. However, the argument is based on the mistaken notion that, at the knowledge level, a system "knows" everything that it can derive at the symbolic architecture level from its symbol structures and the processes that can operate on them (i.e., everything in the "implicational closure"). If, at the symbolic architecture level, the system can generate a wide variety of structures that the system doesn't necessarily know represent true knowledge about the world — for example, imagine that it can generate all possible representational structures within some language (just like the Monkeys in the British Museum algorithm can generate all possible

Figure 5: Structure of the Weak Methods

subcontext was removed from working memory as well as any structures that were not accessible to higher contexts.

4.4 Implementation Level

Soar 2 was implemented as a modification to OPS5 [Forgy, 1981], which was implemented in Lisp. The major changes centered on the conflict resolution scheme of Ops and the addition of the preference and decision schemes. There were no changes made to the Rete production matcher. Using the Rete matcher greatly improved the efficiency of matching productions in Soar.

4.5 Results

In Soar 2, the set of weak methods and tasks were expanded. Figure 5 shows the relationship among these methods. The lines indicate that a method was derived from an earlier method by adding more knowledge.

Also in Soar 2, programs were built to demonstrate the coverage of universal subgoaling. More important, the first large task, R1-Soar was built in Soar 2. R1-Soar was developed to demonstrate that Soar was sufficient for expert-level performance. We analyzed the knowledge in a key fragment of R1, the computer configuration expert system developed by John McDermott for Digital Equipment Corporation, and developed a system in Soar that had the same functionality. This will be discussed in more detail under Soar 3.

5 Soar 3 (1984)

In parallel with the development of Soar 1 and 2, Paul Rosenbloom and Allen Newell were creating computational theories of learning. The original goal was to model the ubiquitous power law of practice [Newell and Rosenbloom, 1981]. By the beginning of summer in 1982, they had built an initial task-specific model of procedural "chunking", where processing in a goal would be summarized by a set of three productions [Rosenbloom and Newell, 1982]. This model was implemented in the XAPS2 production system architecture; that is, the same architecture upon which Soar 1 was based. Over the next year the model was extended and made task independent, and became a central part of Rosenbloom's thesis [Rosenbloom, All of these impasses can be resolved by creating new preferences for the impassed decision. For example, if there is a tie among three operators, creating reject preferences for two of the operators will break the tie. These impasses can also be resolved by changing a decision higher in the context stack. For example, if there is a tie for the operator, the selection of a new state will eliminate the current preferences for the operator slot and lead to the creation of new preferences for operators relevant to the new state.

4.3.3 Subgoals

The symbol level implementation of automatic subgoaling was straight-forward given the scheme developed for Soar 1. There were four major changes. The first three were implemented in Soar 2, with the last being part of Soar 3.

1. The architecture automatically created new contexts when an impasse was encountered. Thus, instead of a production voting for a new goal, the architecture would automatically create a context whenever there was an impasse in the decision procedure.

2. Multiple contexts were represented in working memory at the same time. Thus, when a new context was created, the context that gave rise to it would stay in working memory. In Soar 1, there was only a single context, and when a new goal was suggested it would replace the current goal. When a subgoal terminated, the original context had to be reconstructed. In Soar 2, the original context was maintained and the subgoal had a pointer back to it. By maintaining the complete context stack, subgoals had much better access to the state of problem solving that led to the impasse, and returning to the original context no longer required any reconstruction.

3. The results of a subgoal were determined automatically by the graph structure in working memory. If a working memory element was created in the subgoal, but was linked through other working memory elements to a supercontext, it was a result and would be maintained after the subgoal terminated.

4. The architecture continually monitored the decisions for all context slots in working memory. Whenever a preference was created for a context slot, during the next decision phase, the decision procedure would redecide that slot. If that resolved an impasse, the

- Worst: The object mentioned should not be selected unless there are no other viable alternatives.
- Indifferent: The object mentioned can be selected at random in comparison to other indifferent objects.

In Soar 2, decision productions would interpret these preferences and translate them into a voting scheme that selected the appropriate object. In later versions of Soar, the processing of the decision productions was incorporated directly into the architecture and decision productions were eliminated. This was possible because the semantics of the preferences are fixed and do not change, even when learning was introduced.

Given the semantics of the preferences, four types of impasses could arise for each of the four types of context slots (goal, problem space, state, and operator). These corresponded to different types of incomplete or conflicting knowledge:

1. Tie: The control knowledge is incomplete as to which object is the appropriate selection. For example, if three operators are acceptable without any other preferences, there would be a tie. (Note: if there is knowledge that a choice can be made freely among the three operators, there would be indifferent preferences.) This impasse can be resolved by preferences that cause one choice to dominate (such as by rejecting the alternatives or preferring a single choice), or by making the tieing candidates indifferent.
2. Conflict: The control knowledge is conflicting. For example, if there are two acceptable operators, A and B, and there is one preference that A is better than B, and a second preference that B is better than A, then there is a conflict. This impasse can be resolved by rejecting one of the alternatives.
3. All-Rejects: The control knowledge is incomplete in providing a viable alternative. All acceptable candidates are also rejected. This impasse can be resolved by the creation of an acceptable preference for another candidate object.
4. No-change: No new candidates are suggested, so no new decision is made. This impasse can be resolved by the creation of an acceptable preference for another candidate object. This impasse typically arises when new objects must be generated, such as generating a new state when applying an operator.

result is known before the procedure is called. In Soar, a subgoal can create any type of result it deems necessary. For many impasses, preferences will be created as results to resolve the impasse. However, other structures can be created as results, which in turn trigger the creation of preferences in the supergoal that resolve the impasse. Similarly, it may be "expected" that a subgoal return as a result a preference that allows selecting between two alternative operators, whereas the subgoal may actually return results that reject both of the existing alternatives and generate a new operator, which may then be selected.

These properties of Soar's subgoaling allow the problem solving in the subgoal free range as to how the subgoal is achieved and the impasse is resolved. In actuality Soar's subgoals act more like open calls to the meta-level then restricted calls to subprocessing [Rosenbloom *et al.*, 1988].

4.3 Symbolic Architecture Level

4.3.1 The Basic Problem Solving Cycle

The basic problem solving cycle of elaboration-decision-application in Soar 1 was simplified in Soar 2 to be just elaboration-decision. Application was eliminated as a separate phase. The creation of a new state by an operator was performed during the elaboration phase following the selection of the operator.

4.3.2 Preferences and the Decision Procedure

The symbolic preference scheme developed for Soar 2 had the following types of preferences:

- Acceptable: The object mentioned is a candidate for selection.
- Reject: The object mentioned must not be selected.
- Better (Worse): The first (second) object mentioned should not be selected if the second (first) object mentioned is a viable candidate.
- Best: The object mentioned should be selected unless it is rejected or worse than some other object.

individually, that would be cast as a goal with a problem space that had operators for each of the conjuncts. When a conjunct was selected, it would then lead to the creation of a goal, unless it was so simple that it could be performed directly. The functions of generating and selecting between the conjuncts map directly onto generating and selecting the operators that represent the conjuncts. Thus, Soar replaces deliberate subgoals with operators. What is the advantage of Soar's approach? In addition to having a uniform approach to execution control and goal management, Soar provides a graceful path from a knowledge-lean system with lots of goals, to a knowledge-rich system with lots of rules. If knowledge is available to perform the operator/deliberate-goal directly, no goal has to be created. This becomes more important when learning is added so that an operator that originally requires problem solving as a goal becomes one which the agent is able to perform directly without any problem solving.

This combination of universal and automatic subgoaling leads to two hypotheses about the relationships among goals in an intelligent agent:

1. The relationships between goals and subgoals is that of lack of knowledge. A subgoal is created to obtain knowledge so that problem solving in the goal can continue.
2. The functions for creating and selecting goals are embedded within the architecture.

In addition to automatic and universal subgoaling, Soar's subgoaling has many unique properties.

1. Subgoals and goals are simultaneously active. Thus, if some changes are made in a supergoal, possibly through perception or through intermediate results of a subgoal, the problem solving in the supergoal can precede immediately without waiting for a termination signal from the subgoal.
2. The parameters to a subgoal are not prespecified. Unlike a procedure call, where the parameters to a procedure must be specified in advance, the exact set of parameters is determined dynamically by the needs of the problem solving in the subgoal. Thus, the supergoal does not need to know what information is going to be relevant to the subgoal.
3. The results of a subgoal are not prespecified. In a traditional procedure call, the type of

Figure 4: Soar 2 Search Control and Planning Subgoals.

for a single problem, many different types of subgoals and problem spaces may be generated and used.

How does Soar 2's approach to subgoals differ from AI systems that deliberately create goals? In Soar 1, and for many of the initial versions of Soar 2, goals could be deliberately created by productions suggesting new goals and having them be selected. In the final versions of Soar 2, this was eliminated so that all goals had to be generated by the architecture. It took us a long time to convince ourselves that the impasse-driven subgoaling mechanism in Soar 2 was sufficient by itself, for it eliminated deliberate goals. Where would such functionality now come from? The answer turned out to be simple. When the agent attempts to apply an operator, but is unable to do it directly, that is the same as a deliberate goal. Thus, if an agent wants to decompose a single goal into a set of conjuncts and attempt each one

Figure 3: Soar 2 Hierarchical Execution Subgoal.

is to get the blocks stacked in alphabetical order (A on top of B, which in turn is on top of C). The other problem space has operators that pickup, move, and put down a specific block. We assume that there is sufficient knowledge to select and apply of operators in this space (pickup, move, put down) directly.

At the top left of Figure 3, the initial state is shown, followed by the selection of MOVE-BLOCK(A,B). Once MOVE-BLOCK(A,B) is selected, it can be applied, but in this case, there is no directly available knowledge for performing the operator. It must be decomposed into simpler steps. Thus, there is an impasse. To resolve this impasse, the second problem space is selected, and its operators are applied to implement MOVE-BLOCK(A,B).

If we remove the assumption that there is sufficient search-control knowledge for the subgoal, we could get the trace shown in Figure 4. Here, the candidate operators are generated, but there is insufficient knowledge to select between them, leading to another impasse. In response to this impasse, the goal becomes to select the best operator. This is a control or meta-subgoal. The operators for the selected problem space evaluate and compare the task operators. In the example, an operator (*evaluate*) is selected to evaluate the operator PICK-UP block. The evaluation is computed in a subgoal. In general, any method for gaining more knowledge could be used, and in this case a limited lookahead is employed to determine how well PICK-UP contributes to the original task goal. This demonstrates how generated. That is, the subgoals have to be generated in situations that can be detected in a domain independent manner, no matter what knowledge is available for the task. For example, in GPS, the architecture could automatically detect when the preconditions of an operator did not match the current state and create a subgoal. In contrast, operator selection could never fail — the table of connections was always available to determine which operator should be selected. Similarly, every operator had a directly executable definition, so it was never problematic to implement an operator. Thus, GPS supported automatic subgoaling, but only for a single problem space function. Most problem space functions were never problematic, and functions were never open to new knowledge. As a result, GPS could only be applied to tasks with a single problem space that had simple operators and for which there existed a complete table of connections.

Soar expands automatic subgoaling to universal subgoaling because every problem space function is an open decision that is made while the task is being performed. The knowledge for these decisions, such as operator selection, is not pre-compiled into a table of connections before the task is attempted. Instead, the decisions are made based on the *preferences* retrieved during the elaboration phase. Ironically, opening up the decision procedure at run time to more knowledge also opens up the possibility that the knowledge will be incomplete or inconsistent and that more knowledge will be required before a decision can be made. We called the situations under which progress cannot be made because of incomplete or inconsistent knowledge, *impasses*.[1] In Soar 2, the architecture automatically created a subgoal whenever an impasse arose. In later versions of Soar, the architecture would also automatically terminate the subgoal when the impasse was resolved.

Figure 3 shows a graphical trace of Soar as it encounters impasses in applying operators, leading to hierarchical execution. The task is to stack blocks, where multiple problem spaces are used to control the problem solving. The top problem space has a single operator for moving blocks. It has two parameters, the first being the block being moved, and the second being the location of its destination (either the top of another block or the table). For a given state, this operator may have many instantiations. For this specific example, the goal

[1]They were originally called difficulties, but we then adopted the word "impasses" from Brown and VanLehn's work (1980). Recently it became clear that VanLehn meant dead-ends in problem solving, as opposed to architectural inabilities to make progress.

in one space with the control of an operator selection in another.

4.2 Problem Space Level

The major thrust for change at the problem space level was the desire to introduce subgoals into Soar. We knew when writing Soar 1 that subgoals were necessary, but we delayed introducing them until we understood the basics of integrating production systems and problem spaces.

The guiding principles behind introducing subgoals are quite similar to the ones we used for developing the universal weak method: Soar should be able to generate any and all types of goals. That is, an agent should be able to create goals for achieving operator preconditions, as in GPS and STRIPS. It should also be able to create goals for performing operators. It should even be able to create goals for deciding which operator to select, so-called *meta-goals*. Thus, the agent should be able to create all types of goals. We called this property *universal subgoaling*.

A related but separable property is the way in which goals are created and terminated. We posited, in parallel with actually building the agent, that it is possible to create an agent that generates its goals automatically when they are required by problem solving. An agent that creates goals automatically does not have to encode knowledge about when to generate goals, but instead bases the creation of goals on an inability to make progress on its task. We called this property *automatic subgoaling*. GPS had automatic subgoaling, but for only a limited class of goals.

Universal subgoaling and automatic subgoaling are obviously desirable properties, but how is it possible to build an agent that incorporates them? For universal subgoaling, what is the space of all possible goals? For automatic subgoaling, how can we insure that goals are generated whenever they are required? The answers to both of these questions came from using the problem space computational model (PSCM) as a framework for problem solving. The PSCM defines a set of functions — generation, selection, and application — from which all behavior is constituted. These functions in turn determine the types of subgoals that can be generated by the agent.

For the agent to support automatic subgoaling, the goals have to be architecturally

4 Soar 2 (1983)

The big conceptual advance in going from Soar 1 to Soar 2 was adding subgoaling. Once this was in place, Soar's fundamental problem solving structure was set. The next ten years were spent refining this structure, and combining it with learning abilities (in Soar 3) and an ability to interact with external worlds (Soar 5). Soar 2 is described in detail in Laird (1984).

4.1 Knowledge Level

The advances of Soar 2 were mostly at the problem space and symbolic architecture levels. However, Soar 2 did expand the range of control knowledge that could be encoded for a task in two ways. It expanded the expressiveness of knowledge for selecting context objects as well as allowing for the expression of general meta-knowledge.

The Soar 1 decision scheme restricted the expression of knowledge about making decisions to simple votes for or against proposed alternatives. Many types of knowledge are not expressible in this language, such as partial orderings where it is known that one candidate should always be preferred over another. Although knowledge can be added in Soar 1 to vote for a choice, say A, and vote against another choice, say B, in an attempt to prefer A over B, there is no guarantee that other knowledge will not contribute enough votes for B so that it is selected. Thus, the voting scheme might not always maintain the underlying semantics behind why a vote might be made (or more accurately, the actual semantics of a vote were unclear). To remedy this problem, the representation of selection knowledge was changed so that it corresponded more closely to the semantics of preference and goal achievement.

A second restriction in Soar 1 was that all of the knowledge was *directly* related to the generation or selection of objects for the task. It was not possible to encode knowledge about general (*meta-level*) principles for generating operators or determining which operator was best, such as by making analogies with similar cases, or by performing a look-ahead search. Although Soar 1 allowed multiple goals and problem spaces, it lacked the key reflective ability of allowing the processing in one goal/problem-space to examine and modify the processing in another. Thus, for example, it was not possible to link the results of a lookahead search

Task	ExS	ADS	HS	MEA	BrFS	DFS	SHC	SAHC	BFS	MBFS	A*
Eight Puzzle	+	+	+	+	+	+	+	+	+	+	+
Tower of Hanoi	+	+	+		+	+					
Missionaries and Cannibals	+	+	+	+	+	+	+	+	+	+	+
Water Jug	+	+	+		+	+					
Picnic Problem I	+										
Picnic Problem II	+	+	+	+	+	+	+	+	+	+	+
Picnic Problem III	+	+	+	+	+	+	+	+	+	+	+
Syllogisms	+			+							
Wason Verification	+		+								
Three Wizards	+		+								
Root Finding I	+	+	+		+	+	+		+		
Root Finding II	+										

Figure 2: Soar 1 Methods versus Tasks

decisions when there was no other knowledge available. These productions, together with the architecture, provided the essence of the universal weak method. Additional control knowledge was then encoded as elaboration and decision productions. These productions encoded additional knowledge about the task, such as when one state or operator was more desirable than another. We called these productions *method increments* because they could be freely added in combination for a given task and together they determined the method performed on a task. For example, the method increments for simple hill climbing were:

- If the current state is not acceptable or has an evaluation worse than the ancestor state, vote for the ancestor state.
- If the current state is acceptable and has an evaluation better than the ancestor state, vote for the current state.

Not all methods arise for all tasks. For example, many of the tasks do not demonstrate means-ends analysis. This is because there is no available means-ends knowledge for the task. Similarly, knowledge about comparing intermediate states can be formulated for only a subset of the tasks (Eight Puzzle, Missionaries and Cannibals, the Picnic Problem, and Root Finding).

during elaboration of possible problem spaces. Decision productions cast votes that lead to the selection of a current problem space, followed in turn by the generation and selection of an initial state. Once the state is selected, operators are suggested during elaboration, followed by the selection of one based on votes by decision productions. This selection can be based on means-ends knowledge or other heuristics that are encoded in long-term memory. Following the selection of the operator, the application phase is entered and a new state is created and added to the stock. At this point, knowledge can be used to select between the various states in the stock. Once one is selected, operators are suggested and the search continues. If a state is ever reached that achieves the goal, elaboration productions can recognize this fact, augmenting the goal appropriately, and trigger decision productions that can select a new goal to work on.

3.4 Implementation Level

Soar 1 was implemented in XAPS2[Rosenbloom and Newell, 1987], a parallel production system implemented in Lisp. XAPS2 also supported activation-based matching, and an early task-specific version of chunking, but these features were not used in Soar 1.

3.5 Results

The major success of Soar 1 was its ability to support a Universal Weak Method for a variety of simple puzzles and tasks. The UWM was demonstrated by encoding nine different tasks in Soar 1, and then demonstrating that as knowledge was added about the tasks, the behavior of the agent changed and was consistent with well-known weak methods. Figure 2 summarizes these results. The methods demonstrated include exhaustive search (ExS), avoiding duplicate states (ADS), heuristic search (HS), means-ends analysis (MEA), breadth-first search (BrFS), depth-first search (DFS), simple hill climbing (SHC), steepest-ascent hill climbing (SAHC), best-first search (BFS), modified best-first search (MBFS), and A*.

The tasks were encoded as productions that generated the appropriate goals, problem spaces, initial states, and operators. The knowledge encoded in these productions was necessary to even attempt the problem and did not include any control knowledge. In addition to these tasks productions, there was a set of productions for default control — making

Figure 1: Processing Cycle of Soar 1

cation productions. In contrast to STRIPS-like operators, this approach allows for a continuum of declarative representation of operators. It is possible to have productions that interpret a general declarative representation of the operator; for example, operators can be generated with explicit declarative lists of preconditions, additions and deletions. It is also possible to have ad hoc representations for operators in working memory, in which case the vast majority of the knowledge about performing the operator is represented as productions that are sensitive to the name of the selected operator.

The basic processing cycle consists of elaboration, followed by decision as shown in Figure 1. If there is no operator selected, the cycle repeats with elaboration. Once an operator is selected, an application phase occurs so that a new state is generated. Then elaboration begins again. Elaboration provides the agent with the ability to perform limited, monotonic inference, and consists of firing all matched productions in parallel. Elaboration continues as long as new productions are matched. Although this could be unbounded in theory, in practice, elaborations last only 2–3 production cycles.

With this architecture, the selections of the goal, problem space, state, and operators are open at each step of the problem solving to the agent's long-term knowledge. The typical trace of problem solving starts with the initial goal being selected, followed by the creation also have attributes and values, providing a semantic network representation for objects and their substructures.

These working memory structures support the problem space functions of generation, selection, and application by explicitly representing the generated candidates and the current selections. But how are the problem space functions performed? In Soar 1 we supported the different functions by functionally distinct types of productions.

1. Generation corresponds to the addition of new candidate objects to the available stock. Productions match against the current context and can create new objects for the stock. For example, a means-ends production can test the current state and goal, and retrieve into working memory an operator that reduces the difference between the state and the goal. Productions that perform generation are called *elaboration* productions, and they fire in parallel because they only add objects to working memory. In addition to creating new objects, elaboration productions can also augment the objects in the current context with additional information, such as computing an evaluation of a state.

2. Selection corresponds to the replacement of an object in the current context slot. Replacement is guided by *decision* productions which cast votes for or against objects in working memory. Thus, search-control knowledge is encoded as decision productions. The votes for the context slots are totaled starting with the goal, then problem space, state, and operator. If the votes suggest a change to the context, the new object is installed and the remaining slots are emptied. For example, if the goal changes, the current problem space and all the candidate problem spaces are possibly no longer appropriate. The same is true when a new state is selected — the existing operator and the candidate operators may not be appropriate. Therefore, in Soar 1, all of the unprocessed slots and proposed objects are emptied, to be rebuilt with appropriate candidate and selected objects through future elaboration and selection. Ties are broken arbitrarily if they arise.

3. Operator application corresponds to the creation of a new state. Application is performed by *application* productions which are sensitive to the current context state and operator. Thus, knowledge about the execution of the domain is encoded as appli-

include using activation schemes as in ACT* [Anderson, 1983] or meta-rules — additional rule bases which match against the competing rules and pick the best one [Davis, 1980; Genesereth, 1983]. However, all of these schemes identify the state of the problem solving as the complete working memory, and thus do not support multiple goals, multiple problem spaces, or the various search methods that require multiple states.

The list of problem space functions suggests another approach to integration. In our earlier list, there are two basic functions: generation and selection. These two functions are applied uniformly to the four types of problem space objects: goals, problem spaces, states, and operators. The generation function is a recall from long-term memory of candidate objects. Selection then uses additional knowledge to decide on the current goal, problem space, state, or operator. Thus, the agent must be able to represent candidates, as well as the current selections. There is one additional function, which is that of applying operators to states to generate new states.

In Soar 1, we took what now seems to be the obvious approach of supporting generation and selection by having a special data structure in working memory called that *current context* that has slots to hold symbols representing the current selections for the goal, problem space, state and operator. Thus, these object became "first-class" objects that would be explicitly generated and selected. In addition, there was a stock of generated candidate operators, states, problem spaces, and goals. Below is an example of the stock and current context for Soar 1.

Current Context:	Goal	Problem Space	State	Operator
	$goal_{10}$	$problemspace_{3}$	$state_{107}$	$operator_{6}$
Stock:	$goal_{102}$	$problemspace_{14}$	$state_{34}$	$operator_{74}$
	$goal_{30}$	$problemspace_{23}$	$state_{702}$	$operator_{56}$
	$goal_{231}$	$problemspace_{1}$	$state_{19}$	$operator_{202}$
	...	...	...	...

Each object in working memory could have additional attributes and values, which could

This formulation of problem spaces is extremely flexible, allowing knowledge to be used to generate and select any problem space objects. It is possible to encode the operator selection methods of GPS and STRIPS, as well as state selection methods, such as depth-first search, best-first search, or even alpha-beta and A*. However, This flexibility comes at the cost of requiring that knowledge be available for all of these functions for every problem the system attempts. This requirement was partially ameliorated by providing default knowledge to control for those decisions in which no domain-specific knowledge was available.

3.3 Symbolic Architecture Level

Given the prior formulation of the problem space level, how is it realized in a symbolic architecture? Our initial inclinations for building a problem space architecture were to avoid production systems, thinking that they would only complicate the project, and thus we developed a Lisp based system called the Task Experimenter (TEX). TEX provided us with immense experience in problem spaces and methods, but it was obvious that Lisp did not provide the appropriate control structure for incrementally adding more control knowledge. To achieve the desired flexibility we would need to build a second representation of knowledge on top of Lisp, possibly similar to production systems.

After abandoning Lisp, we needed to identify the appropriate role for productions within a problem space. The standard view at the time, and one that seems to still be ubiquitous in AI and Cognitive Science, is that productions are to be mapped onto operators in problem spaces. Thus, the selection and firing of a production is equivalent to taking a step in problem solving. Under this view, the problem state is the complete working memory, the operators are productions stored in long-term memory, and the control of the problem solving corresponds to *conflict resolution*: finding the right production to fire for the current situation.

Unfortunately, this integration of problem spaces and production systems greatly restricts the types of methods that could be used for problem solving. In most approaches, such as OPS5, the selection of the next production/operator is fixed by the syntax of the productions and the recency of the working memory elements it matches. This provides a depth-first style to problem solving and it cannot be influenced by additional knowledge. Other approaches goal that is most appropriate for the current situation. This includes detecting success or failure for the current goal.

2. **Problem space generation and selection.** The agent can generate problem spaces that are appropriate for a given goal, and select the one most appropriate. It should also be possible to change problem spaces during work on a goal if, for example, the current problem space is exhausted without achieving the goal.

3. **State generation and selection.**

   The agent can generate the initial state of the problem, thereby partially instantiating the goal within the current problem space (the description of the desired state is often left on the goal itself). It should also be possible to freely select from previously generated states during the search, so that the agent can use methods that require selections of prior states, such as best-first search or depth-first search.

4. **Operator generation and selection.**

   The agent can generate the operators of the problem spaces that are appropriate for the current state (or all of the operators in the space) and use knowledge to select the one most appropriate for achieving the goal.

5. **Operator application.**

   The agent creates new states through operator application. These new states then become candidates for state selection. Although this function is a state generation function, we distinguish it from state generation, because it is based on the current state and operator.

   Under this approach, the operators create completely new states. Progress is made in the problem space by jumping from one state to the next, possibly backing up to a state generated earlier; i.e., one that is available from the stock of previously generated states.

Our formulation assumed that the agent is serial at the problem space level, in that there is only a single current goal, problem space, state, and operator at any time. For Soar 1, there was also an unlimited stock of states that could be generated during problem solving and the current state was selected from that stock.

GPS or STRIPS. Their second weakness was that they could support only a single problem space. Thus, they could not use different problem spaces, that is different sets of operators and different representations, for different problems. GPS could be "programmed" with different tasks, but it could encode only a single problem space at a time. Similarly, these system could not switch problem spaces, nor could they use specialized problem spaces for subproblems.

If problem spaces are to be a distinct computational level, it is necessary to precisely define the processing that occurs within problem spaces and the types of objects that are manipulated. The objects include goals, problem spaces, states, and operators. The exact structure of these objects are unspecified at the problem space level, and their semantics are defined in terms of the functions that apply to them. During the construction of the first Soar implementation, it was unclear what functions were required at the problem space level. We were driven by a combination of concerns, some arising from behavior we wished the agent to exhibit (the UWM) and some from what had been learned about search and problem spaces in AI. The original set of functions proposed by Newell (1980), included:

1. Decide on success (the state is a desired state).
2. Decide on failure or suspension (the goal will not be achieved on this attempt).
3. Select a state from those directly available (if the current state is to be abandoned).
4. Select an operator to apply to the state.
5. Decide to save the new state for future use.

One of the breakthroughs in Soar 1 was to move to a more uniform list that arises from the acts of generation and selection. Some of the original functions are subsumed by more general functions (goal success and failure become selection of new goals), while others were eliminated and assumed to happen automatically (saving a new state for future use). The Soar 1 scheme assumes that there are two basic functions of generation and selection that can be applied to any problem space object (goal, problem space, state, and operator), plus there is operator application.

1. **Goal generation and selection.**

   The agent can generate new goals for itself, and can also use knowledge to select the

1. Different methods are used for different tasks.
2. Different methods are used on the same task.
3. The methods arise from whatever knowledge is available at the time, without explicit programming.

In sum, the idea was that the agent would behave as if it had every weak method, and that the method that was most appropriate to the current task would be applied automatically. With little or no knowledge about the task, the method would resemble the weakest of the weak methods, exhaustive search. As more knowledge was added, the behavior of the agent would change, and the method would evolve to more powerful methods. This required an architecture in which knowledge can be added incrementally, and there is no precompiled set of methods, but instead the methods emerge from the available knowledge. An agent that behaved in this way was said to have a *Universal Weak Method* (UWM).

3.2 Problem Space Level

Given a desire to support the knowledge level and a universal weak method, it was critical that our formulation had a flexible control scheme that allowed many different methods to be specificed, as well as allowed a variety of knowledge to be used in controlling the problem solving. Based on our experience with the IPS project, we realized that trying to achieve a knowledge-level agent without an intermediate organizational framework would be futile. This led Newell to consider problem spaces as a possible intermediate level of computation [Newell, 1980], and in turn, led us to use problem spaces as a separate computational level within Soar 1.

Many previous AI systems solved problems by casting them as search within a problem space. LT, GPS, and STRIPS [Fikes and Nilsson, 1971] are canonical examples. However they all had two basic weaknesses. Their first weakness was that they all used a single method to control all problem solving, independent of the task. (In fact, one of the contributions of GPS and STRIPS was that they demonstrated the usefulness of means-ends analysis.) This restricted the types of knowledge they could encode. For example, it was not possible to encode goal-independent operator selection heuristics, or state evaluation knowledge in duction systems, but still had a variety of scaling and efficiency problems, particularly when rules were being learned automatically.

3 Soar 1 (1982)

Soar 1 arose out of two intertwined goals. The first was to create an architecture that supported problem spaces where production systems were used as the underlying representation of knowledge. The second was to create an architecture that could support many different weak methods. Both goals had existed in some form since the early seventies when the importance of weak methods, problem spaces, and production systems were identified. However, their integration was to wait for at least a decade and the development of Soar 1. Soar 1 is described in Laird and Newell (1983a) and summarized in Laird and Newell (1983b) .

3.1 Knowledge Level

An analysis of architecture at the knowledge levels starts with considering the goals, physical body (for actions and perception), and knowledge that are supported by the architecture. As mentioned earlier, a knowledge-level agent will act in a way that is consistent with its goals and knowledge. This abstracts away from the internal structures and processes of the agent, which are exactly what must be determined in constructing Soar 1 as a production system. However, by viewing the system at the knowledge level, we can confront issues concerning the generality and acquisition of the system's knowledge, independent of the underlying structure.

Soar 1 was based on the observation that humans — our best example of knowledge-level agents — can use many different methods for solving problems. The key question was whether getting an agent to use such a range of methods required adding explicit knowledge about the methods — as, for example, a library of methods — or whether just adding task knowledge by itself can be sufficient to result in behavior that follows these methods. If task knowledge alone is sufficient, the agent approaches a knowledge-level agent.

Our approach to this problem was to identify the following capabilities we wanted our agent to support.

failure was that there was no overall framework for organizing tasks or control knowledge.

The concept of a production system as a model of memory was carried over verbatim to Soar, as was the RETE match algorithm. However, their direct use as a control structure was not.

2.5 Review

Newell and Simon's work prior to Soar provided strong direction at all four levels of descriptions, but still left much remaining to be done:

1. **Knowledge Level**

   The idea of the knowledge level was proposed, but it was not yet clear how to apply it constructively to the development and analysis of intelligent systems.

2. **Problem Space Level**

   The basic concept of the problem space developed from LT, through GPS, and on to the Problem Space and Weak Methods Hypotheses. So it was well defined prior to the development of Soar. However there was little agreement as to the details of the structure of the objects involved (goals, problem spaces, states, operators), the control mechanisms for them (the weak methods), or the representation of knowledge. Also, most systems that were created using the problem space paradigm were one of a kind, with only a single problem space, and only limited types of goals and subgoals.

3. **Symbolic Architecture Level**

   Symbolic representations and architectures, as developed in LT and GPS, are foundational concepts in Soar. In addition, automatic subgoaling (as existed in a limited manner in GPS), and the OPS production system languages [Forgy and McDermott, 1977] provided initial models of subgoaling and memory for Soar. However, the former doesn't specify how all of the other types of subgoals needed by an intelligent system are to be created, while the latter requires significant adaptation for it to support the problem space computational level.

4. **Implementation Level**

   The RETE match algorithm provided an efficient algorithm for pattern matching in prospecific knowledge available, were not just a random collection, but instead a family, which he termed "weak methods" [Newell, 1969]. (From here on in we'll refer to this as the Weak Methods Hypothesis, though Newell did not explicitly so name it). The weak methods were weak, not because they failed to tightly constrain the search, but because they made weak demands on the knowledge about the task required to apply the method. Many different tasks could use hill climbing. All that is required is the ability to compare neighboring states and select the best. Thus, the weak methods are exactly those methods that get used when there is not "strong" knowledge available to solve a problem directly. They are the methods of last resort, but they are also the methods that provide robustness in the face of novelty. Thus, an understanding of the weak methods may provide an understanding of important classes of control.

Both the Problem Space and Weak Methods Hypotheses were foundational in the development of Soar.

2.4 Production Systems

In the late sixties, Newell and Simon began to recognize that the control structure for GPS was too inflexible to capture of all of the variety of human behavior. Specifically, the table of connections underlying means-ends analysis was only a part of the knowledge that the subjects were using. They would also use knowledge that seemed specific to the situation, and not necessarily goal-directed. This led Newell and Simon to consider production systems as the underlying representation of knowledge for intelligent systems. Newell experimented with a variety of production system languages [Newell, 1972; Newell and Simon, 1972; Newell, 1973] in the late sixties and early seventies. In 1975, he formed a group to investigate building large production systems, which evolved into the Instructable Production System (IPS) project [Rychener and Newell, 1977]. During the lifetime of this project, the OPS family of languages was developed — with OPS5 eventually becoming a de facto standard for much of the expert systems community [Forgy, 1981] — along with the RETE match algorithm [Forgy, 1982]. Although IPS led to the creation of the OPS languages, the RETE match algorithm, and indirectly the creation of R1 [McDermott, 1982], it failed at its main mission which was to create large, instructable production systems. One diagnosis of its techniques, in which an architecture automatically generates subgoals based on an inability to make progress. All versions of Soar, except for the very first, depend heavily on automatic subgoaling.

2.3 Problem Spaces and Weak Methods

In both LT and GPS, a problem was represented as an initial state and a set of desired states to be achieved by applying operators. Newell and Simon recognized this and proposed that a general way to formulate tasks was in terms of a problem space. The problem space is the set of states and the set of operators in which a problem can be attempted. The complete set of states can either be pre-enumerated, or can be generated by applying operators to existing states. Problem spaces provide a framework for organizing knowledge in terms of operators, states, and control knowledge.

A given problem space can be used for many different problems. For each problem there would be different initial state and desired states. Likewise, a given problem can be attempted in different problem spaces, where different operators are possible and different states could be generated. The difficulty of a problem, as formulated in a particular problem space, is determined by the difficulty of transforming the initial state into one of the desired states, which in turn is determined by factors such as the number of operators available at each point of the search (the branching factor), the number of operators that must be applied to achieve a desired state (the depth), and the knowledge available to control the problem solving.

As work in AI progressed during the sixties, Newell was struck by the fact that many systems — and not just LT and GPS — were based on search in problem spaces. This observation led to the Problem Space Hypothesis, that all symbolic goal-oriented behavior takes place in problem spaces [Newell, 1980]. This hypothesis promised to provide a uniform structure for casting all problem solving. The observation also led to an attempt to characterize the types of methods being used to control search in problem spaces. In analyzing various AI systems of the era, Newell noticed that many of them shared the same methods, such as means-ends analysis, generate and test, hypothesize and match, and hill climbing. He hypothesized that these types of methods, which were used when there was little task-operators to apply to a state to transform it step by step until a proof was constructed. Although the term *problem space* would emerge years later, this was the first program to formulate a task in terms of a problem space. The selection of operators was guided by *heuristics*, and the complete process was called heuristic search. Heuristic search introduces a framework for problem solving (operators applied to states to achieve a goal) controlled by knowledge.

2.2 GPS

Following the success of LT and some related work on chess [Newell *et al.*, 1958], Newell, Shaw, and Simon attempted to generalize the concepts of heuristic search and symbol systems so that they could construct a single program that could solve many different problems. They were guided in their work by comparisons between LT's behavior and the protocols of humans attempting to solve similar problems. The humans used a more goal-directed approach, where operators were selected based on how well they reduced the difference between the current state and the goal. This method, called *means-ends analysis* was ubiquitous in the human protocols. Another observation was that humans created their own goals to help them break the problem up into simpler problems. They did not restrict themselves to just the goals of the task, but instead would create goals to achieve situations in which they could apply operators that could not apply in the current situation, a method called *operator subgoaling*. The system based on this insight was called the General Problem Solver (GPS), because it was not limited to working in just a single domain [Newell and Simon, 1961].

As with LT, GPS also contained two key innovations that would later show up in Soar.

1. **Symbolic Architecture:**

   Although means-ends analysis and operator subgoaling are powerful methods for problem solving — and are used in Soar applications — the most important lesson to be taken from GPS is the separation of the fixed underlying control structure of the program from the operator and control knowledge about the task: i.e., the separation of architecture from knowledge. Thus, GPS was the first real symbolic architecture.

2. **Automatic Subgoaling:**

   Operator subgoaling was the first example of a general class of automatic subgoaling be matched with a time complexity independent of the number of productions. Does there exist an implementation that supports this assumption? Thus, this level is an important level both practically and theoretically.

2 Pre-Soar

Although Soar has evolved significantly over the last twelve years, its central intellectual core still can be traced to a range of research done earlier by Allen Newell and Herbert Simon. In this section we review the key developments that set the stage for Soar before launching into the evolution of Soar.

2.1 LT

In 1954, Newell, Shaw, and Simon considered the possibility of creating a program that could solve problems that required complex thought processes. Newell initially worked on a chess machine [Newell, 1955] and then after considering geometry as a domain, settled with Shaw and Simon on trying to construct a program that could prove some of the theorems in Whitehead and Russell's *Principia Mathematica*. On August 9, 1956, the Logic Theorist (LT) created the first mechanical proof of a theorem [Newell and Simon, 1956; Newell *et al.*, 1957].

LT contained two innovations that would later show up in Soar.

1. **Symbolic Representations:**
   All of the knowledge in LT was represented by symbol structures and all of the cognitive processing occurred by symbol manipulation. This was in sharp contrast to the numeric processing that was (and often still is) standard in computer programming.

2. **Heuristic Search:**
   To build the proofs, LT created an initial data structure containing the initial axioms and theorem to be proved. It then had discrete transformations that it could apply to data structures to create new theorems by combining, modifying or decomposing existing axioms and theorems. The data structure was called the *state* and the transformations were called *operators*. A problem was solved by selecting the appropriate

the limitations on space and time imposed by the real world imply that this ideal can never be completely achieved for sufficiently broad and complex combinations of knowledge and goals. Here we will trace the expansion of types of goals and knowledge accessible to Soar through the sequence of versions. For the reader who just wishes to get a flavor of "what Soar can do," this is the level on which to concentrate; however the emphasis of this paper will be on the evolution of the other levels.

The problem-space level characterizes the structure of problem solving and reasoning in an intelligent agent. It is intended to provide a physically realizable approximation of the knowledge level. In general, it is concerned with the characterization of operators, states, and problem spaces, plus the relationships between goals and subgoals. It sits between the knowledge level, which abstracts away from all internal processing considerations, and the symbolic architecture level which includes details of control flow and memories. One hypothesis underlying the research on Soar is that the problem-space level is a real computational level with its own types of processing and media [Newell *et al.*, 1991]. In Soar, the problem space level determines how tasks are formulated. In many ways it is the level at which Soar is most firmly entrenched and where it is the most unique (sic). Major changes could be — and are — made to the lower levels, and as long as they continued to support the problem-space level, Soar would still be Soar. Of course, evolution does also occur at this level as our understanding of problem spaces increases, particularly as driven by a continuous expansion of the range of tasks that we attempt to formulate in problem spaces.

The symbolic architecture level provides the basic control structure, memory organization, and processing structure to support the problem-space level. Just as the task of the problem-space level is to support the knowledge level, a major task throughout the development of the Soar architecture has been to derive a symbolic architecture level that can support the required flexibility of the problem space level.

The implementation level is the underlying technology that provides the symbolic architecture level. For the most part, the details of the implementation are irrelevant to the Soar architecture. The implementation does show through though in terms of the efficiency, boundedness, and correctness of executing the various processes of the symbolic architecture level. For example, at the symbolic architecture level, Soar assumes that productions can

4. Public distribution, via robustification and documentation.
5. External interaction, via destructive state modification.
6. Speed, portability, and maintainability, via formal specification and reimplementation in C.

To improve the conceptual coherence of the presentation, we will occasionally "rationalize" history by moving the discussion of some capabilities forward and/or backward in time (i.e., to an earlier or later version). For example, chunking was first developed in Soar 2, was the major focus of Soar 3, and a continued locus of development in Soar 4 and (to a lesser extent) in subsequent versions. For conceptual coherence, we will focus on chunking during the discussion of Soar 3.

Within each major version of Soar, the presentation will be organized around four levels of description: the knowledge level, the problem space level, the symbolic architecture level, and the implementation level. The notion of levels of description first shows up in computer science in Newell's work with Gordon Bell on computer structure [Bell and Newell, 1971]. The levels for computer structure start with electronice devices and move up through electrical circuits, logic circuits, register transfer systems and ultimately provide program level-systems. Our analysis of Soar has the symbolic architecture level which corresponds to the programming level, but the implementation level covers important details below the symbol level that can not be directly mapped on to the register transfer level. The knowledge level and problem space level are higher levels of description, both of which were articulated recently by Newell.

The knowledge level is the most abstract level used to characterize the behavior of an intelligent agent. It was originally proposed by Newell [Newell, 1982] as an analysis tool. At the knowledge level, an intelligent agent is described only in terms of its knowledge, goals, and body (perceptions and actions). The agent is considered to be a knowledge-level system if it behaves according to the *Principle of Rationality:* the actions it intends are those that its knowledge indicates will achieve its goals. One of the central goals of the Soar project is to create agents that provide a good approximation to the knowledge-level ideal of rational behavior across a wide range of knowledge and goals [Rosenbloom *et al.*, 1991]; however,

1 Introduction

In the Spring of 1976, while trying to decide on which computer science graduate school to attend, we (John Laird and Paul Rosenbloom) independently visited the Computer Science Department of Carnegie Mellon University (CMU) in Pittsburgh. We were both interested in artificial intelligence (AI) as our field of study, so we naturally met with Allen Newell. Newell was fully immersed in production systems at the time, and was in the process of starting the Instructable Production System project. His boundless enthusiasm was infectious, and we decided independently — we did not meet each other until the first day of graduate school — to attend CMU and work with Newell. That was undoubtably the best decision of our professional careers.

In deciding to work with Newell, we had been captivated by the concept of creating architectures in which AI systems could be built. Not necessarily hardware, but *cognitive architecture* in the sense of the the fixed structures underlying intelligence [Newell *et al.*, 1989; Rosenbloom and Newell, 1993]. We saw this as a way of studying the properties of intelligence in general, not just specific algorithms for specific problems, but architectures that could support the variety of behaviors so characteristic of humans.

Over the next six years, there were many stops and starts in our research. For example, during our first year, Newell was on sabbatical and both of us worked on other research projects. Also both of us left Pittsburgh at different times in the middle of graduate school to take a year off. However, by 1982, the first version of Soar was up and running. During the next twelve years, Soar evolved through six different major versions, and by 1994, over 100 researchers were using Soar worldwide.

This paper is the story of the evolution of the Soar architecture, structured according to its major versions, 1–6. Roughly, the major contributions of these six versions can be characterized as follows:

1. Combining flexible search and knowledge, via problems spaces and productions.
2. Universal automatic subgoaling and reflection, via impasse detection and subgoal generation.
3. Learning, via chunking.

The Evolution of the Soar Cognitive Architecture

John E. Laird
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
laird@umich.edu

Paul S. Rosenbloom
Information Sciences Institute &
Computer Science Department
University of Southern California

June 30, 1994

Abstract

The origins of the Soar architecture can be traced back to the seminal research of Allen Newell and Herbert Simon on symbol systems, heuristic search, goals, problem spaces, and production systems. Since its official inception in 1982, Soar has evolved through six major releases, as both an AI architecture and as the basis for a unified theory of cognition. This paper traces this evolutionary path, starting with Soar's intellectual roots, and then proceeding through the stages defined by the six major system releases. Each stage is characterized with respect to a hierarchy of four levels of analysis: the knowledge level, the problem space level, the symbolic architecture level, and the implementation level.