Hierarchical Skills and Cognitive Architectures

Pat Langley (langley@csli.stanford.edu)
Kirstin Cummings (kirstinc@ccrma.stanford.edu)
Daniel Shapiro (dgs@stanford.edu)
Computational Learning Laboratory, CSLI
Stanford University, Stanford, CA 94305

Abstract

In this paper, we examine approaches to representing and utilizing hierarchical skills within the context of a cognitive architecture. We review responses to this issue by three established frameworks – ACT-R, Soar, and PRODIGY – then present an alternative we have developed within ICARUS, another candidate architecture. Unlike most earlier systems, ICARUS lets skills refer directly to their subskills and communicate within a single recognize-act cycle. This assumption has implications for the number of cycles required to complete complex tasks. We illustrate our approach with the domain of multi-column subtraction, then discuss related methods and directions for future work in this area.

Introduction and Overview

Human skills are organized in a hierarchical fashion. There seems to be general agreement with this claim, as it is consistent not only with experimental findings about the execution and acquisition of skills, but also with introspection about our everyday behavior. Upon request, most people can describe their complex skills at successive levels of aggregation, whether these involve how they drive to work each day, how they cook a meal, or how they write a technical article.

What remains an open question is how we should model such skill hierarchies in computational terms. Alternative approaches to modeling cognition encode the notion of hierarchy in distinct ways that have different implications for performance and learning. The most interesting positions are those which are embedded in theories of the human cognitive architecture, such as Soar (Laird et al., 1987), ACT-R (Anderson, 1993), and EPIC (Kieras & Meyer, 1997). These frameworks make strong commitments to both the mental representations of knowledge and to the processes that operate on them.

In the pages that follow, we consider the challenge of modeling hierarchical skills within a unified theory of cognition. We begin with a brief review of three such architectures and their responses to this issue, then turn to ICARUS, an alternative framework that approaches hierarchical skills from a different perspective. The key issue involves whether one can traverse levels of a hierarchy within a single cognitive cycle. Our illustrative example comes from a familiar cognitive skill that has a hierarchical organization – multi-column subtraction – but we also consider other models that incorporate multi-level skills. We conclude by discussing related work and our plans to extend the architecture's capabilities.

Previous Research on Hierarchical Skills

A cognitive architecture (Newell, 1990) specifies the infrastructure for an intelligent system, indicating those aspects of a cognitive agent that remain unchanged over time and across different application domains. Many proposals for the human cognitive architecture take the form of a production system, which stores long-term knowledge as a set of condition-action rules, encodes short-term elements as declarative list structures, and relies on a recognize-act cycle that matches against, and executes rules that alter, the contents of short-term memory. Soar, ACT-R, and EPIC are all examples of the production-system paradigm, although other architectural frameworks are also possible.

Despite the general agreement that cognitive skills are organized hierarchically, there exist different ways to implement this basic idea. Within a production-system architecture, the most natural scheme involves communication between skills and their subskills through the addition of elements to short-term memory. For instance, ACT-R achieves this effect using production rules that match against a generalized goal structure in their condition sides, such as the desire to prove that two triangles are congruent, and, upon firing, create new subgoals, such as the desire to prove that two lines have the same length. This approach requires the explicit addition of goal elements to short-term memory, since this is the only mechanism through which other production rules can be accessed.

Soar takes a somewhat different approach to encoding hierarchical skills that relies on multiple problem spaces. For instance, Jones and Laird (1997) report a detailed model of flying an aircraft in combat training scenarios. This system organizes its capabilities in a hierarchical manner, with each node implemented as a Soar problem space with associated states, operators, and goals. To invoke a lower-level problem space, the system adds a new element to short-term memory that refers to that space, and it must take similar action in order to exit. The details differ from those in ACT, but the passing of messages through short-term memory is similar.

The PRODIGY architecture (Minton et al., 1989) produces cognitive behavior in yet another manner. The system represents knowledge about actions as STRIPS-like operators, and the central module utilizes means-ends analysis to decompose problems into subproblems. This gives PRODIGY the ability to generate hierarchical

structures dynamically for each problem it encounters, and it can use control rules to select among candidate operators, states, and goals that determine the decompositions. However, these hierarchical structures do not remain in memory after a problem has been solved; instead, the system stores its experience in generalized control rules that let it reconstruct them for new problems. Again, this requires adding explicit elements to short-term memory that serve as mediators.

Thus, despite their diversity, these three frameworks share a basic assumption about how communication occurs among skills and subskills. This tenet has implications for the cognitive behavior of systems cast within them, including whether intermediate results are available, the time required to execute complex skills, and the effects of learning on hierarchical structure. We will see shortly that another response to this issue is possible.

Hierarchical Skills in ICARUS

ICARUS (Choi et al., in press) is a cognitive architecture that shares some central features with its predecessors. For instance, it relies on symbolic list structures to encode information, and it makes a clear distinction between long-term and short-term memories, with the former storing generic knowledge and the latter containing specific beliefs. Moreover, ICARUS assumes a recognize-act cycle, which relies on matching patterns in long-term memory against elements in the short-term store, to determine behavior over time. The framework also comes with a programming formalism for building knowledge-based systems.

However, ICARUS has other characteristics that distinguish it from earlier frameworks in significant ways. One such feature is its commitment to embodied cognition, which requires that all symbols in ICARUS programs be grounded in sensori-motor primitives.[1] A second key assumption is that each long-term memory structure may have not only a symbolic description but also an associated numeric function that computes its value as a function of sensory attributes. A third novel feature is that ICARUS contains distinct long-term memories for skills, which store its knowledge about action, and for concepts, which specify its knowledge about states and relations. Yet another contribution lies in the strong correspondence between long-term and short-term memory, specifically that every element in the latter must be an instance of some structure in the former.

A fifth important assumption, which is our focus here, is that hierarchical structures play a central role in cognition. Although production systems and related architectures allow hierarchies, many do not encourage them, and we maintain that ICARUS supports them in a sense that is stronger and more plausible psychologically than do frameworks like ACT and Soar. To understand this claim, we must first examine the architecture's representation for skills and the relations among them.

[1]Some recent architectural variants, like ACT-R/PM and EPIC-Soar, incorporate sensori-motor modules, but these were grafted onto systems based on theories of human problem solving, whereas ICARUS included them from the outset.

ICARUS skills bear a strong resemblance to production rules, but they have an even closer kinship to the operators in PRODIGY and STRIPS. Each skill has a name, arguments, and some optional fields. These include:

- a `:start` field, which encodes the situation that must hold to initiate the skill;
- a `:requires` field, which must be satisfied throughout the skill's execution across multiple cycles; and
- an `:effects` field, which specifies the desired situation the skill is intended to achieve.

For example, Table 1 shows a complete set of ICARUS skills for the domain of multi-column subtraction, including *borrow*, which has the objective of getting *?digit1* to be greater than *?digit2*. This skill can start only if *?digit2* is greater than *?digit1* and if a third element, *?digit3*, is nonzero. Moreover, its execution requires that *?digit1* be above *?digit2*, that *?digit3* be in the top row, and that *?digit3* be left of *?digit1*.

In addition, each ICARUS skill includes another field that specifies how to decompose it into subskills or actions. This may be either:

- an `:ordered` field, which indicates the agent should consider the components in a specific order;
- an `:unordered` field, which identifies a choice among skill components; or
- an `:actions` field, in which a primitive skill specifies one or more actions that are directly executable.

For example, *borrow* states that one should invoke *decrement* on one digit and call *add-ten* on another, in that order. These are both primitive skills that play the same role as STRIPS operators in a traditional planning system, with their `:start` fields serving as preconditions and their `:effects` fields specifying the desired results of execution.

The table also clarifies that ICARUS can specify multiple ways to decompose each skill in this manner, much as a Prolog program can include more than one Horn clause with the same head. Different decompositions of a given skill must have the same name, number of arguments, and effects. However, they can differ in their start conditions, requirements, and subskills. The skill *borrow* has two such expansions, one for borrowing from nonzero elements and another for borrowing across zero, which involves a recursive call to itself.

Each skill decomposition may also include a numeric function that encodes the utility expected if it executes this decomposition. This function is specified by a `:percepts` field that matches against the values of objects' attributes and a `:value` field that states an arithmetic combination of these quantities. The expected utility for a skill decomposition is a linear function of the numeric descriptors matched by that skill. Such functions are not required when the available skills specify deterministic behavior, as do those in the table, but we have used them in other domains and we mention them here for completeness.

We should note that ICARUS also organizes its long-term conceptual memory in a hierarchy, with higher-level concepts being defined in terms of lower-level ones, which

Table 1: ICARUS skills for multi-column subtraction.

```
(subtract ()
 :percepts ((digit ?digit))
 :requires ((top-row ?digit) (right-col ?digit))
 :unordered ((process ?digit))

(subtract ()
 :requires ((processed ?digit1) (top-row ?digit2)
            (left-of ?digit2 ?digit1))
 :unordered ((process ?digit2))

(process (?digit)
 :ordered  ((borrow ?digit)
            (find-difference ?digit))
 :effects  ((processed ?digit)))

(borrow (?digit1)
 :percepts ((digit ?digit2) (digit ?digit3))
 :start    ((greater ?digit2 ?digit1)
            (nonzero ?digit3))
 :requires ((above ?digit1 ?digit2)
            (top-row ?digit3)
            (left-of ?digit3 ?digit1))
 :ordered  ((decrement ?digit3)
            (add-ten ?digit1))
 :effects  ((greater ?digit1 ?digit2)))

(borrow (?digit1)
 :percepts ((digit ?digit2) (digit ?digit3))
 :start    ((greater ?digit2 ?digit1)
            (zero ?digit3))
 :requires ((above ?digit1 ?digit2)
            (top-row ?digit3)
            (left-of ?digit3 ?digit1))
 :ordered  ((borrow ?digit3))
 :effects  ((greater ?digit1 ?digit2)))

(decrement (?digit)
 :percepts ((digit ?digit val ?val))
 :start    ((nonzero ?digit))
 :actions  ((*cross-out ?digit)
            (*decrement ?digit ?val))
 :effects  ((crossed-out ?digit)))

(add-ten (?digit1)
 :percepts ((digit ?digit1 val ?val1)
            (digit ?digit2) (digit ?digit3))
 :start    ((above ?digit1 ?digit2)
            (top-row ?digit3)
            (left-of ?digit3 ?digit1)
            (crossed-out ?digit3))
 :actions  ((*add-ten ?digit1 ?val1))
 :effects  ((greater ?digit1 ?digit2)))

(find-difference (?digit1)
 :percepts ((digit ?digit1 col ?col val ?val1)
            (digit ?digit2 col ?col val ?val2))
 :start    ((above ?digit1 ?digit2)
            (greater-or-equal ?digit1 ?digit2))
 :actions  ((*add-difference ?col ?val1 ?val2))
 :effects  ((processed ?digit1)))
```

ultimately connect to perceptual elements. The literals that appear in the `:start`, `:requires`, and `:effects` fields must be defined concepts, thus linking the skill and conceptual memories in an interleaved manner. The architecture also incorporates three short-term memories. These include a perceptual buffer, updated on each cycle, that contains descriptions of perceived objects, a conceptual short-term memory that contains inferences derived from the perceptual buffer, and an intention memory that contains instances of skills the system intends to execute. The elements in these memories are simple literals but, because their predicates correspond to hierarchical structures in long-term memory, they encode substantial information about the agent's beliefs and intentions.

Distinctive Aspects of ICARUS Hierarchies

From the preceding discussion, it should be clear that ICARUS utilizes a more structured representation of knowledge than traditional cognitive architectures, but the implications of this structure depend directly on the processes that operate over them. Together, they enable cognitive processing that exhibits important differences from that in older frameworks.

One such characteristic involves the ability of ICARUS' skills to reference subskills by their names, rather than through the indirect references used in Soar, ACT, and PRODIGY. For example, the *borrow* skill in Table 1 calls directly on *decrement* and *add-ten*. ICARUS' approach has some aspects of subroutine calls in procedural programming languages but, when combined with multiple expansions for each skill (such as two variants for *borrow*), effectively embeds these calls within an AND/OR search. This makes our formalism a close relative of logic programming languages like Prolog, which uses a very similar syntax to support logical reasoning. But like other cognitive architectures, ICARUS is concerned with agents that exist over time, so it situates these computations within a recognize-act cycle.

As a result, ICARUS retains the overall flavor of a production system but gains the ability to invoke subskills directly, rather than through the creation of short-term memory elements. This lets it execute complex skills in a top-down manner without having to descend through the hierarchy one step at a time. ICARUS can take advantage of this hierarchical organization without requiring the generation of explicit intermediate goal structures that are needed by production systems.

Recall that ICARUS includes a short-term memory that contains skill instances the agent considers worth executing. On each cycle, for each skill instance, the architecture retrieves all decompositions of the general skill and checks whether they are applicable. A skill is applicable if, for its current variable bindings, its `:effects` field does not match, the `:requires` field matches, and, if the system has not yet started executing it, the `:start` field matches the current situation. Moreover, at least one of its subskills must also be applicable. Since this test is recursive, a skill is only applicable if there exists at least one acceptable path downward to executable actions.

For each such path, the architecture computes the expected value and selects the candidate with the highest utility for execution. For a given path, it uses the value function stored with each skill and the numeric values matched in that skill's `:percepts` field to calculate the expected value at each level, summing the results along the path to compute the overall score. For instance, for the path *((subtract), (process digit11), (borrow digit11), (decrement digit21))*, the system would sum the expected value for all four levels to determine the utility of decrementing. This means that the same action can

have different values on a given cycle depending on which higher-level skill invokes it, providing a natural way for the hierarchy to incorporate the effect of context.

The architecture treats a skill expansion differently depending on whether its components appear in an `:unordered` set or an `:ordered` list. If they are unordered, the module considers each of the subskills and selects the one that yields the highest scoring subpath. If they are ordered, it instead treats the list as a reactive program that considers each subskill in reverse order. If the final subskill is applicable, then it expands further only down paths that include that subskill. Otherwise, it considers the penultimate skill, the one before that, and so forth. The intuition is that the subskills are ordered because later ones are closer to the parent skill's objective, and thus should be preferred when applicable.

We should clarify that ICARUS' consideration of alternative paths through its skill hierarchy does not involve generative planning. On each cycle, the architecture finds the best pathway through a set of flexible but constrained structures. The process is much closer to the execution of a hierarchical task network (e.g., Myers, 1999) than to the construction of a plan from primitive operators. Such computation can be done efficiently within a single recognize-act cycle, at least for well-structured skill hierarchies. One can craft ICARUS programs that are expensive to evaluate, but the same holds for production systems with complex conditions on their rules.

An Illustrative Example

We can best clarify ICARUS' operation with an example that uses the skills from Table 1 on the subtraction problem $305 - 147$. The system interacts with a simulated environment that, on each cycle, deposits perceptual elements such as *(digit digit11 col 1 row 1 val 5)* and *(digit digit12 col 1 row 2 val 7)* into the perceptual buffer. A conceptual recognition process draws inferences from these elements, such as *(greater digit12 digit11)* and *(above digit11 digit12)*, which it adds to conceptual short-term memory.

The model also begins with the single top-level intention *(subtract)*, which focuses cognitive behavior on the skills in the table even if others are present in long-term memory. On the first cycle, the system would consider both expansions of *subtract*, selecting the first one and binding *?digit* to *digit11*, the object in the top row and right column. This skill instance would in turn invoke *(process digit11)*, which would consider its subskills *find-difference* and *borrow*. Only the latter skill has its `:start` and `:requires` fields met, specifically by its second expansion, which handles situations that require borrowing from zero.

This skill instance, *(borrow digit11)*, then invokes itself recursively, producing the call *(borrow digit21)*, where the argument denotes the digit 0 in the top row and second column. Because the digit to its left is nonzero, this instantiation can only utilize the first expansion of *borrow*, which in turn calls on *(decrement digit31)* in its `:ordered` field, since its preconditions are satisfied, but those for *add-ten*, which occurs later in this ordering, are not. Because *decrement* is a primitive skill, it becomes the terminal node for an acceptable path through the skill hierarchy. Also, because this is the only such path ICARUS finds, it executes the instantiated actions *(*cross-out digit31)* and *(*decrement digit31 3)*.

These actions alters the environment and lead to another execution cycle. This time ICARUS again finds a single acceptable path that shares all but the last skill instance with that from the first round. The difference is that *digit31* has been crossed out, making *(decrement digit31)* inapplicable but enabling the skill instance *(add-ten digit21)*. Again this is the only acceptable path through the hierarchy, so ICARUS executes the action associated with this primitive skill, thus altering the simulated display so that *digit21*'s value is increased by ten.

On the third cycle, the architecture again finds only one path, in this case *((subtract), (process digit11), (borrow digit11), (decrement digit21))*, since the top number in the second column is no longer zero and can be safely decremented. This action enables execution of the path *((subtract), (process digit11), (borrow digit11), (add-ten digit11))* on the fourth cycle, after which (on the fifth cycle) the model selects the path *((subtract), (process digit11), (find-difference digit11))*, which writes the two digits' difference in the rightmost column.

This altered situation leads ICARUS to add the inference *(processed digit11)*, which on the sixth cycle causes it to select the second expansion of *subtract*; this invokes the skill instance *(process digit21)* on the revised top digit in the second row. Because this digit has already been incremented by ten, it is greater than the one below it, so the skill instance *(find-difference digit21)* is now applicable. Execution of this path produces an answer in the second column, which leads on the next cycle to processing of the third column and to an answer there as well. No paths are satisfied on additional cycles, so the system idles thereafter, waiting for new developments.

General Implications

The sample trace above does not illustrate all of ICARUS' capabilities, since it ignores details about the conceptual inference process and it does not utilize value functions. However, it should make clear that ICARUS operates over its skill hierarchy in a different manner than frameworks like Soar and ACT-R. They can model behavior on complex tasks in two distinct ways. One scheme assumes hierarchical rules or problem spaces, which require additional cycles for stepping downward through each level of the hierarchy. Another assumes that learning has produced compiled rules that eliminate the hierarchy and thus the need for intermediate goal structures. Such compilation methods have been used to explain both the power law of learning and reduced access to subgoals (e.g., Neves & Anderson, 1981).

However, it seems unlikely that the hierarchical structure of skills disappears entirely with practice, and ICARUS offers an account that avoids the above dichotomy. An architecture that traverses levels in a skill hierarchy within a single recognize-act cycle also predicts that intermediate structures will be inacessible, and the

power law follows from the construction of the hierarchy itself, which we discuss later. This account also makes different predictions than traditional schemes about the number of cycles, and thus the time, required to accomplish tasks with hierarchical and recursive structures.

Again, we can use multi-column subtraction to illustrate this point. A standard production-system model for this domain, like that described by Langley and Ohlsson (1984), finds the difference between two numbers in a column when the top one is larger but otherwise adds a goal to process or borrow from the adjacent column. Analysis reveals that such a model will take $5 \cdot b + 2 \cdot n$ cycles to complete a problem with b columns that require borrowing and n columns that do not. In contrast, the ICARUS model in Table 1 requires $3 \cdot b + 2 \cdot n$ cycles on the same problems. The two frameworks both indicate that solution time will increase with the number of borrow columns, but they predict quite different slopes.

Experiments with human subjects should reveal which alternative offers a better explanation of skilled behavior in this arena. We have not yet carried out such studies, but to test our framework's generality, we have developed ICARUS models for behavior in other domains that appear to have hierarchical organizations. One involves the well-known Tower of Hanoi puzzle, which can be solved using a hierarchical strategy. Our model for behavior on this task includes three primitive skills for lifting, lowering, and moving a disk sideways, along with one high-level skill for moving a disk to a target peg that, in two expansions, calls itself recursively. However, the Tower of Hanoi is like subtraction in that the environment changes only when the agent takes some action.

To ensure that ICARUS can also support behavior in more dynamic domains, we have developed two additional models. One involves hierarchical skills for balancing an upright pole by moving its lower end back and forth. This system includes two high-level skills with knowledge about the order in which the four primitive skills should be invoked. As described elsewhere (Choi et al., in press), we have also constructed a system that drives a vehicle and delivers packages in a simulated in-city environment. This model includes 46 skills that are organized in a hierarchy five levels deep. The high-level skills let the agent drive straight in lanes, get into rightmost lanes, slow for intersections, drive through intersections, turn at intersections, and make U turns. These terminate in actions for changing speed and altering the wheel angle. Another 13 skills support the delivery of packages to target addresses.

We have not attempted to compare the details of these systems' operations to human behaviors on the same tasks. Nor have we attempted to show that ICARUS produces more robust behavior than programs cast in earlier frameworks like Soar and ACT-R. Rather, our goal has been to demonstrate the same broad functionality as we observe in humans, including their apparent organization of complex skills into hierarchies. We have also aimed to show that ICARUS constitutes a viable point in the space of cognitive architectures, which we believe has been too thinly explored to date.

Related Efforts and Future Research

Although ICARUS incorporates a number of features that distinguish it from typical cognitive architectures, some related ideas have appeared elsewhere under different guises. For instance, our framework has much in common with the 'reactive planning' movement, which often utilizes hierarchical procedures that combine cognition, perception, and action in physical domains.[2] Examples include PRS (Georgeoff et al., 1985), teleoreactive programs (Nilsson, 1994), and the 3T robotic architecture (Bonasso et al., 1997), and some case-based planners (e.g., Hammond, 1993) embody similar notions.

However, within this paradigm, only Freed's (1998) APEX has been proposed as a candidate architecture for human cognition. This framework shares ICARUS' commitment to hierarchical skills, but it has a more procedural flavor and it does not incorporate a separate conceptual memory or enforce a correspondence between long-term and short-term structures. Another kindred spirit is Albus and Meystel's (2001) RCS architecture, which organizes knowledge hierarchically and makes a clear distinction between logical structures and value judgments. ICARUS and RCS share many common features, but they also retain many differences due to their origins in cognitive modeling and control theory, respectively.

We should clarify that, as a cognitive architecture, ICARUS still requires some development. The framework lacks many processing assumptions that would make it more plausible as a general theory of human behavior. For instance, it lacks any limits on perceptual bandwidth that require attention, which arises even on routine tasks like subtraction. We intend to model such behavior by introducing an action that focuses the agent's attention on an object and deposits its description in the perceptual buffer, with ICARUS being able to apply this action to only one visible object per cycle. This should produce longer subtraction runs that require additional steps for attentional events, much as in Anderson, Matessa, and Lebiere's (1997) ACT-R model of visual attention.

We should also extend our models for subtraction and other domains to handle lower levels of behavior more faithfully. The skills in Table 1 terminate with actions for decrementing, adding ten, and finding a difference, but these can be further decomposed into subskills for writing specific digits and even drawing individual lines. Our claim to model embodied cognition would be stronger if we extended the skill hierarchy downward in this fashion. We should also extend the hierarchy upward to model choices about which problem to tackle when many are present on the page. Such an expanded system would model subtraction behavior in a more complete way than do most accounts.

However, a more important omission concerns the origin of ICARUS' hierarchical skills. To handle this, we hypothesize a mechanism that is related to chunking in Soar and production composition in earlier versions of ACT. Although humans prefer to use routine behaviors when possible, they can, within limits, combine

[2] Our model of subtraction skills also has similarities to VanLehn's (1990) hierarchical treatment of this domain.

knowledge elements when needed to solve novel problems. Means-ends analysis has been implicated in such situations, so we plan to incorporate this method into future versions of the architecture. The generalized results of means-ends problem solving would be cached as a new skill. However, unlike chunking and composition, which produce flat rules that eliminate structure, ICARUS would store a new hierarchical skill that refers to the original ones as components. This method should lead to the construction of skill hierarchies in a gradual, bottom-up manner as an agent learns about a domain.

Concluding Remarks

In closing, we should review the main claims we have made about hierarchical skills and their treatment within various cognitive architectures. There seems to be general agreement that skills are organized in some hierarchical fashion, but most existing models implement the invocation of subskills through goal structures that are deposited in short-term memory. In contrast, ICARUS produces hierarchical behavior by letting skills communicate directly with their subskills, as in procedural languages and logic programming formalisms.

We illustrated this idea by presenting an ICARUS model for multi-column subtraction and tracing its behavior on a specific problem. We saw that this system takes the same basic steps as a production-system model, but that steps involved in traversing the skill hierarchy occur within a single recognize-act cycle rather than across successive cycles. This theoretical difference leads to different predictions about the time required to execute complex skills. Future research should test these predictions and extend ICARUS to incorporate mechanisms for attention and construction of skill hierarchies.

Acknowledgements

This research was funded in part by Grant IIS-0335353 from the National Science Foundation, by Grant NCC 2-1220 from NASA Ames Research Center, and by Grant HR0011-04-1-0008 from Rome Labs. Discussions with Stephanie Sage, David Nicholas, and Seth Rogers contributed to many of the ideas presented in this paper.

References

Albus, J. S., & Meystel, A. M. (2001). *Engineering of mind: An introduction to the science of intelligent systems*. New York: John Wiley.

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R. Matessa, M., & Lebiere, C. (1997). ACT-R: A theory of higher level cognition and its application to visual attention. *Human-Computer Interaction*, *12*, 439–462.

Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D., & Slack, M. (1997). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, *9*, 237–256.

Choi, D., Kaufman, M., Langley, P., Nejati, N., & Shapiro, D. (in press). An architecture for persistent reactive behavior. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*. New York: ACM Press.

Freed, M. (1998). Managing multiple tasks in complex, dynamic environments. *Proceedings of the National Conference on Artificial Intelligence* (pp. 921–927). Madison, WI: AAAI Press.

Georgeff, M., Lansky, A., & Bessiere, P. (1985). A procedural logic. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*. Los Angeles: Morgan Kaufmann.

Hammond, K. (1993). Toward a theory of agency. In S. Minton (Ed.) *Machine learning methods for planning*. San Francisco: Morgan Kaufmann.

Jones, R. M., & Laird, J. E. (1997). Constraints on the design of a high-level model of cognition. *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society* (pp. 358–363). Stanford, CA: Lawrence Erlbaum.

Kieras, D., & Meyer, D. E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, *12*, 391–438.

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

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D., Etzioni, O., & Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Myers, K. L. (1999). CPEF: A continuous planning and execution framework. *AI Magazine*, *20*, 63–70.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Nilsson, N. (1994). Teleoreactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

Neves, D. M., & Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition*. Hillsdale, NJ: Lawrence Erlbaum.

Langley, P., & Ohlsson, S. (1984). Automated cognitive modeling. *Proceedings of the Fourth National Conference on Artificial Intelligence* (pp. 193–197). Austin, TX: Morgan Kaufmann.

Shapiro, D., Langley, P., & Shachter, R. (2001). Using background knowledge to speed reinforcement learning in physical agents. *Proceedings of the Fifth International Conference on Autonomous Agents* (pp. 254–261). Montreal: ACM Press.

VanLehn, K. (1990). *Mind bugs: The origins of procedural misconceptions*. Cambridge, MA: MIT Press.