# A Value-Driven Architecture for Intelligent Behavior

**Pat Langley, Daniel Shapiro,**
**Meg Aycinena**, and **Michael Siliski**
Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA

## Abstract

In this paper, we describe ICARUS, an integrated architecture for intelligent agents that diverges from earlier efforts. The framework supports long-term memories for concepts and skills, and it includes mechanisms for recognizing concepts, calculating internal reward, nominating and selecting skills, executing them in a reactive manner, repairing skills' conditions when they fail, and abandoning skills when they promise poor returns. We illustrate these processes with examples from the domain of highway driving, and we relate ICARUS' assumptions to principles of architectural design and to previous research in this important area.

## 1. Introduction and Background

Research on agent architectures pursues a central goal of artificial intelligence and cognitive science: the creation and understanding of synthetic agents that support the same capabilities as humans. Such architectures aim for breadth of coverage across many domains, and they offer an account of intelligence at the systems level, rather than focusing on component methods designed for specialized tasks. They run counter to the increasing fragmentation of these fields, in that they provide integrated frameworks for producing complex behavior in a general, domain-independent manner.

An agent architecture – sometimes called a *cognitive architecture* – specifies the infrastructure for an intelligent system that remains constant across different domains and knowledge bases. This infrastructure includes a commitment to formalisms for representing knowledge, memories for storing this domain content, processes that utilize the knowledge, and learning mechanisms to acquire or revise it. An agent architecture can interpret different knowledge bases, just as a computer architecture can run different programs.

In this paper, we report on the latest version of ICARUS, an agent architecture that extends our previous work in this area (Shapiro & Langley, 1999, 2002; Shapiro et al., 2001). We begin by describing five design principles that have guided our development of the architecture. After this, we describe ICARUS' long-term and short-term memories, including their formalisms for encoding knowledge. Next we examine the framework's mechanisms for operating on these memory structures, focusing on performance rather than learning, which we have discussed in earlier papers. In closing, we consider the intellectual influences on ICARUS and outline our plans for extending its capabilities.

## 2. Design Principles for ICARUS

The past 30 years have seen extensive research on architectures for intelligent agents and considerable progress in this area. However, we believe that existing architectures downplay important facets of intelligent behavior that deserve increased attention. Our research on ICARUS attempts to respond to these needs and has been guided by a number of design principles:

1. *Primacy of categorization over problem solving.* Because most architectures focused initially on multi-step problem solving and planning, they emphasize the generation of solutions to problems or the execution of actions. However, categorization is a central aspect of intelligent behavior that, in humans, takes place rapidly and unconsciously. This suggests that categorization should occur at a more basic level of the architecture, with problem solving and execution relying on it, rather than the reverse.
2. *Primacy of execution over problem solving.* The early focus on problem solving and planning, which involve mental operations, led to alternative frameworks that instead emphasized reactive execution of physical behaviors. Humans can both generate plans and act on them, but the inability of most animals to form abstract plans suggests that execution is more basic, with problem solving building on this capacity.
3. *Internal origins of tasks and intentions.* Many architectures for intelligent agents assume that top-level tasks and goals are provided by the programmer, whereas most reactive frameworks lack even this taskability. Humans can respond to external requests, but they can also operate autonomously, generating their own tasks and intentions. This suggests the need for

architectural mechanisms that support autonomous generation and abandonment of high-level tasks.

4. *Value-driven nature of behavior.* Initial designs for most agent architectures relied almost entirely on symbolic processing, using numbers for strength or recency in only limited ways. Research on reinforcement learning incorporates notions of expected and received reward, but does not link them to cognitive structures like concepts or plans. However, affect plays a central role in human experience and behavior, which indicates the need for a more value-driven approach to cognition, perception, and action.
5. *Internal origins of agent reward.* Methods for reinforcement learning emphasize the role of reward in shaping behavior, but they invariably assume this reward comes from the external environment. However, reward in humans and animals is influenced by their perceptions of the world and their cognitive structures. This suggests that we recast the calculation of reward as a process internal to the agent, which in turn requires architectural support.

Taken together, these constraints have led to an architecture that, although it incorporates many ideas from earlier research, differs from them in important respects. We attempt to highlight these differences as we describe Icarus in the sections that follow.

## 3. Memories and Representations

An integrated architecture should make some commitment to its representation of knowledge and the memories in which that knowledge resides. In this section we describe Icarus' memories for long-term knowledge and short-term beliefs, along with the general forms taken by their contents.

### 3.1 Long-Term Conceptual Memory

Icarus incorporates a long-term memory for concepts that encodes its knowledge of familiar situations. This includes descriptions of categories for isolated objects, like cars and trucks, but also physical relations among objects, such as one vehicle being ahead and to the right of another. These concepts provide Icarus' vocabulary for describing its experience of the world.

Each concept has a name and zero or more arguments, and Icarus supports two distinct kinds of concepts. Boolean concepts are either true or false, and correspond to the traditional notion of a logical category. For instance, long-term memory might include a Boolean concept that covers situations in which there are cars next to the driver in both adjacent lanes. In contrast, numeric concepts take on quantitative values that correspond to attributes of objects or situations. Thus, a knowledge base might include a numeric concept that refers to the average distance to cars ahead of and behind the driver.

The architecture supports both primitive and nonprimitive concepts of both types. Primitive concepts correspond to the output of sensors that can directly perceive various aspects of the external environment. Table 1 refers to two primitive Boolean concepts from the driving domain, `car` and `lane`, and four primitive numeric concepts: `#xdistance`, `#yfront`, `#yback`, and `#speed`. Nonprimitive Booleans are defined as conjunctions of other Boolean concepts, numeric concepts, and logical predicates like `>`, as illustrated by `ahead-of` and `coming-from-behind`. Similarly, numeric concepts like `#distance-ahead` are defined in terms of other numeric concepts, plus optional Boolean concepts and logical predicates. A numeric concept definition also includes an arithmetic function to compute its associated quantity from numeric constituents.

Taken together, these definitions implicitly organize Icarus categories into a conceptual hierarchy. This hierarchy is similar in spirit to those in earlier models of memory like Epam (Feigenbaum, 1963), Unimem (Lebowitz, 1987), and Cobweb (Fisher, 1987), as well as frameworks like description logics (Nardi & Brachman, 2002). The actual form is a lattice, with primitive concepts occurring at the top, concepts defined in terms of them immediately below, and more complex concepts at lower levels. Structurally, this lattice bears a close resemblance to the Rete networks (Forgy, 1982) used for matching in production-system architectures, an analogy to which we will return later.

Each Boolean concept also has an associated function that specifies the reward the agent receives when that concept is true and that provides an analogy for utility in humans, in that it motivates all agent choices within the architecture. The function is described by two fields, one (`:reward`) referring to the numeric concepts that influence the reward and another (`:weights`) that indicates the weight on each numeric concept. The architecture assumes these are combined in a linear fashion to compute the reward obtained when that concept is true.

### 3.2 Long-Term Skill Memory

To complement its conceptual memory, Icarus incorporates a long-term skill memory that encodes knowledge about ways to act and achieve goals. This contains specifications for skills that are applicable in certain situations and that produce desired effects. Skills provide Icarus with a repertoire of behaviors that let it influence the environmental situations in which it finds itself.

Each skill has a name, zero or more arguments, and six fields. The `:objective` field specifies a conjunction of known concepts that, taken together, encode the desired situation the skill is intended to achieve. Each skill also includes a `:start` field, again cast as a conjunction of known concepts, which specifies the situation that must hold to initiate the skill, and a `:requires` field, which must hold throughout the skill's execution. For example, Table 2 shows the skill `pass`, which has the objective of getting `?car1` ahead of `?car2` and in the same lane, can start only if `?car1` is behind `?car2` in the same lane, and it also requires that `?car2` remain in this lane during the passing activity.

In addition, each Icarus skill includes another field that specifies how to decompose that skill into subskills.

An `:ordered` field indicates the order in which the agent should consider these component skills. For example, `pass` directs the agent to consider `speed-and-change`, `overtake`, and `change-lanes`, in that sequence, and to select an action reactively from the first subskill that applies. In contrast, an `:unordered` field identifies a choice among subskills. For instance, the table's decomposition for `speed-and-change` involves the subskills `speed-up-faster-than` and `change-lanes`, from which the system picks the best, regardless of order.

More accurately, Icarus specifies one or more 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 objective. However, they can differ in their components and in their requirements. For example, the skill `change-lanes` has two such decompositions, one for moving to the left and another to the right.

In a primitive skill, the `:ordered` field specifies a single opaque action. For the driving domain, such actions might correspond to turning the wheel or changing pressure on the pedal by a given amount. Thus, a primitive skill plays the same role as a Strips operator in a traditional planning system, with the `:start` field serving as the preconditions and the `:objective` field specifying the effects of execution.

Each skill decomposition also includes an expected value function that takes a form similar to the rewards associated with concepts. This encodes the discounted reward that the agent expects to receive if it executes the skill with this decomposition. As with concept rewards, this function is specified in two parts, a `:value` field that indicates the numeric concepts involved and a `:weights` field that states the weights on quantities returned by these concepts. The expected reward for a skill decomposition is a linear function of the numeric descriptors matched by that skill. For example, the value for `pass` might depend on the numeric concepts `#distance-ahead` and `#speed` of another car, which can vary from moment to moment.

### 3.3 Short-Term Memories

Icarus' long-term memories encode stable knowledge that changes only slowly in response to its accumulated experience. However, to generate behavior, the architecture requires short-term stores that change more rapidly. These should make contact with long-term concepts and skills, but they must also represent temporary beliefs about the agent's environment and desires.

One such memory is Icarus' *perceptual buffer*, which contains instances of primitive Boolean and numeric concepts that correspond to the output of sensors. For example, this short-lived memory might contain the literal `(#speed car-007 20.3)`, which specifies the speed of `car-007` as perceived on the current time step. This literal is an instance of the primitive `#speed` concept because it refers to specific parameters rather than to generalized variables.

In contrast, Icarus' *short-term conceptual memory* contains instances of concepts that are defined in long-term concept memory. These literals encode specific beliefs about the environment that the agent can infer from those present in its perceptual buffer. For instance, this memory might contain the instance `(faster-than self car-007)`, which depends on the `#speed` instance mentioned above. In addition, each instance of a Boolean concept includes a numeric reward computed from the reward function associated with that concept and the literals matched in its `:reward` field.

Finally, Icarus includes a *short-term skill memory* that contains instances of skills the agent intends to execute. Each of these literals specifies the skill's name and its concrete arguments. For example, this memory might contain the skill instance `(pass self car-007 lane-a)`, which denotes that the driver has an explicit intention to execute the `pass` skill with these arguments when possible. In addition, each skill instance includes the expected discounted reward if executed, which is computed from the expected reward function associated with that skill and literals matched in its `:value` field. The agent uses this number to make choices among skills and among alternatives within skills.

## 4. Interpreting and Utilizing Knowledge

Like most architectures for intelligent agents, Icarus operates in distinct cycles. Every cycle, the system updates its perceptual buffer, determines which concepts are matched, and calculates reward based on these concepts. The architecture then selects a skill and executes it, producing changes in the environment that influence decisions on the next cycle. In this section, we discuss each of these processes in turn.

### 4.1 Categorization and Belief Update

On each cycle, Icarus refreshes the contents of its perceptual buffer by applying preattentive sensors to every object within a given distance of the agent. This produces a set of primitive concept instances which are deposited into the short-term store and which are then sent to the top nodes of the lattice for conceptual long-term memory. Each instance of a primitive concept is stored with the long-term node for that concept, along with the bindings of variables for that instance.

When the categorization module stores a new instance $I$ at the node for concept $C$, it accesses each defined concept $D$ that includes $C$ in its definition, then checks to see whether the addition of $I$ makes any new instances of $D$ possible. To this end, it accesses the instances of all other concepts that appear in $D$'s definition and considers whether their variables bind consistently with those for $I$. If so, then the module adds a new instance to the $D$ node for each such consistent binding and recursively ships each one to the nodes for concepts that include $D$ in their definitions.

A similar process occurs when perceptual updating removes a Boolean concept instance from the perceptual buffer or changes the quantity associated with a numeric

instance. Whenever a primitive instance is removed, the module deletes the instances of all defined concepts to which that literal contributes and recursively removes instances of all concepts that depend on it indirectly. Changes to a numeric concept instance can lead to either removal or addition of Boolean concepts in which it occurs, depending on whether the change makes predicates like `(< ?distance 20)` true or false. In general, the categorization module plays the same role for Icarus as a truth maintenance module in some logical inference systems (e.g., Doyle, 1979).

Earlier we mentioned that the concept lattice is similar in form to the Rete networks that are used in many production-system architectures. The concept recognition process just described uses effectively the same mechanism to support efficient matching. The main difference is that every node in the hierarchy corresponds to a concept that has some meaning to the agent, rather than simply existing to support the match process. A typical Rete network utilizes binary trees, whereas Icarus instead has N-ary trees, with the branching factor determined by the number of literals mentioned in each concept definition.

### 4.2 Calculation of Reward

As we noted earlier, Boolean concepts in long-term memory have associated reward functions, which are combined into a global utility metric that informs Icarus' decisions. To determine this quantity, the architecture considers every instance of a Boolean concept in short-term memory and computes the dot product of its numeric attribute values and their associated weights. For example, suppose the concept `ahead-of` in Table 1 has two numeric concepts in its `:reward` field that match the instances `(#distance-ahead self car-006 20)` and `(#speed ?car-006 25)`. If we suppose further that this Boolean concept specifies the parameters 4.7 and 1.6 in its `:weights` field, then the reward contributed by the concept instance `(ahead-of self car-006)` would be $4.7 \times 20 + 1.6 \times 25 = 134$.

After calculating the reward for each matched Boolean instance, Icarus sums their contributions to produce the overall reward for the current cycle. This corresponds to the reward provided in traditional approaches to reinforcement learning, but our framework does not view it as coming from outside the agent. Rather, reward is an internal response to what the agent perceives in its environment, so that if a concept is unmatched, for whatever reason, it has no impact. This opens the way for selective attention to influence the reward signal, though we have not implemented this idea in our current Icarus agents.

In principle, every matched Boolean concept can contribute to the agent's overall reward. However, because an Icarus programmer may not want to specify reward functions for every concept in long-term memory, the architecture assumes zero as the default reward when none is given. Thus, only a few concepts may influence the calculation in practice. Also, note that distinct instances of the same concept make separate contributions to the overall reward. For example, if a driver dislikes being close to other cars, then the reward (which can be negative) will decrease linearly with the number of nearby cars. One can imagine other combination schemes that produce different effects, but simple summation seems a reasonable starting point.

### 4.3 Nomination and Selection of Skills

Recall that Icarus includes a short-term skill memory that contains a set of skill instances the agent considers worth executing. Most architectures for intelligent behavior assume the agent is given some top-level goals to pursue, but this does not explain their source. In contrast, Icarus includes a mechanism for nominating skills that should be added to short-term skill memory and thus considered for execution.

On each cycle, the nomination process accesses all skills in long-term memory that refer to concepts appearing in short-term memory. More precisely, for each short-term concept instance, the module finds every skill that includes the analogous concept in its own `:start` or `:requires` fields. Moreover, it considers different instances of each such skill, based on the variables matched during retrieval. This strategy produces skill instances that are potentially relevant to the current situation in short-term memory.

The nomination process selects at most one of these skills to add to short-term skill memory, which it does on the basis of value calculations. In particular, the system computes the expected reward for executing each skill in the current situation, identifies the most promising one, and compares its estimate against the expected reward for skills already in short-term memory. To aid this decision, the architecture maintains a running discounted average $R_A$ of the overall reward it has received on past cycles. If the expected reward for the highest-scoring skill instance is higher than $R_A$, then the nomination process adds the instance to short-term skill memory. Otherwise, there is no reason to expect it will produce better results than currently exist, so it adds nothing to the set of active intentions.

Once it has completed the nomination process, Icarus selects which skill to execute on the current cycle. To this end, it computes the expected value for each skill instance in the active set, again basing this calculation on the value function stored with each skill and the instantiated numeric concepts against which that skill matches. On each cycle, the architecture simply selects the skill instance that scores the highest on this criterion, whether it has just been added or has been active for some time. This decision involves deep evaluation of the skill, including examination of actions suggested by its subskills, to which we will turn shortly.

Taken together, the nomination and selection processes correspond roughly to the conflict-resolution stage in production-system architectures like OPS (Forgy, 1982) and ACT-R (Anderson, 1993). Both introduce a sequential bottleneck which focuses cognitive atten-

tion on one knowledge structure that seems most appropriate for the current situation. However, Icarus adapts this idea to skills that may have complex internal structures, rather than to smaller, independent condition-action rules.

### 4.4 Execution of Skills

Once Icarus has selected a skill instance to apply on the current cycle, it invokes an execution module on that instance. An attempt to execute a skill instance returns either True, False, or a primitive action, which is then applied in the environment. Because a skill may be defined in terms of other skills, this process is recursive, with a skill returning the same result as its selected subskill produces.

An executed skill instance returns True if its instantiated `:objective` field matches the current state of conceptual short-term memory. If this happens at the top level, the system takes no action, since it is already in the desired situation. However, neither does it remove the instance from short-term skill memory, since its purpose may involve a maintenance activity (e.g., staying far enough from the car ahead) that may require a response in the future.

On the other hand, an executed skill returns False if its requirements do not match the current state of conceptual short-term memory. Recall that a skill's definition may include multiple decompositions, and all of their `:requires` fields must fail to match for this to occur. In this case, Icarus invokes the repair module discussed in the next section in an attempt to alter the environment so the skill's requirements are met.

If one or more skill decompositions has satisfied requirements, then the system must select which one to execute. This process differs from nomination, which examines only the expected value function associated with the nominated skills. Instead, the module recursively considers all ways to execute the selected skill instance, expanding each subskill in turn until reaching the lowest level. The system calculates the expected value of each primitive skill instance that has an unmatched objective and matched requirements, and it returns the instantiated action associated with the highest-scoring skill.

However, the architecture treats a skill expansion differently depending on whether its components are 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 action. If they are ordered, it instead treats the list as a reactive program that considers each subskill in turn. If the first one does not apply, then the enclosing skill fails as a whole. If it produces an action, the system returns that action for possible execution. However, if the first subskill returns true, this means its objective has been met in the world, so the system considers the second subskill, and so forth. The process continues in this manner, terminating either in success, failure, or by selecting an action that furthers the skill's objective.

### 4.5 Repair of Skill Conditions

As just noted, Icarus can decide to execute a skill instance with unsatisfied requirements, in that the concept instances needed to match those requirements are not present in conceptual short-term memory. In this case, the architecture invokes a repair module that attempts to change the environment into a state that meets these requirements. This process involves two steps, one that selects an unmatched concept to repair and another that selects a skill which, if executed, should ultimately produce a situation that satisfies the missing concept.

When determining which concept instance to repair, Icarus considers all the ways in which the current skill has failed, including concepts mentioned in the `:start` and `:requires` fields within its subskills, then selects the one that, if corrected, would produce the highest expected reward. When determining how to remedy this omission, the repair module accesses every skill that refers to the failed requirement in its `:objective` field. For instance, suppose a driving agent has decided to execute the skill `(overtake self car-007)`, but that its requirement `(in-different-lane self car-007)` does not hold. If the agent is currently in the middle lane (`lane-b`), then it could repair the problem by executing either `(change-lanes self lane-a)` or `(change-lanes self lane-c)`, which correspond to passing on the left or right, respectively.

As this example illustrates, the repair process may need to choose among alternative skill instances that address a given requirement. To this end, Icarus uses the same mechanism as for nomination, which involves computing the expected value for each candidate skill instance and selecting the repair skill $R$ with the highest score. This selection takes an extra cycle, so attempted execution of $R$ must wait until the next time step. If $R$ in turn has failed requirements, Icarus repeats the process, selecting another skill that should correct this problem. In summary, skill repair is a cognitive activity that occurs at a higher level than skill execution, which we assume is automatized.

The backward-chaining repair process is closely related to techniques for generating subgoals in means-ends analysis (Newell, Shaw, & Simon, 1960) and planning systems. Moreover, the focus on repair is similar to ideas for impasse-driven problem solving in Soar (Laird, Newell, & Rosenbloom, 1987) and Sierra (VanLehn, 1990). The key difference is that Icarus invokes its repair process with respect to physical skills rather than mental reasoning. Thus, this architectural feature remains consistent with the principle that execution has primacy over problem solving, despite its resemblance to traditional planning methods.

### 4.6 Abandonment of Skills

Another common assumption in agent architectures is that, once an agent decides to pursue some goal, it continues indefinitely. However, humans often decide to abandon their goal-directed activities when they do not produce the expected results, and we maintain that

Icarus should also include this facility. We want the system to exhibit persistence with respect to nominated tasks, but it should give up on them when they seem unachievable or unlikely to generate the reward executed originally. For example, if the agent begins passing another car which then speeds up substantially, it may be reasonable to abandon the attempt.

To produce such behavior, Icarus relies on the discounted average of past rewards that we described for the nomination process. On each cycle, the architecture updates this running average and compares it to the expected reward computed for each skill instance present in the short-term skill memory. If the expected value of executing a skill is substantially lower than the average reward (currently 20 percent of this amount), then the system removes the instance from short-term memory.

Because Icarus uses the same average reward here as for nomination, it is unlikely the system will abandon a skill just after it has been nominated. Typically, some time will pass before a skill's expected value drops significantly from its level when the architecture first added it to short-term memory. This leads Icarus to exhibit a certain persistence, though not the uncritical kind represented by more traditional goal-driven systems.

## 5. Intellectual Precursors

Despite its novel features, Icarus draws on many ideas that have a long history in artificial intelligence and cognitive science. The most important intellectual influence comes from the cognitive architecture movement, which aims to develop integrated frameworks that support general intelligent behavior. Since Newell's (1973) call for more systems-level research in these fields, a number of research groups have developed a variety of such architectures, two of the best known being Soar (Laird et al., 1987) and ACT-R (Anderson, 1993). Other efforts at integration have focused on architectures for robotic control (e.g., Bonasso et al., 1997), many of which combine methods for planning and execution. Icarus incorporates concepts from both traditions in a framework that supports physical agents with cognitive abilities.

Many cognitive architectures have been cast as *production systems* (Neches, Langley, & Klahr, 1987), which encode long-term knowledge as a set of condition-action rules that match against and modify the contents of short-term memory. Our design for Icarus incorporates central ideas from this framework, including a reliance on pattern matching, with skill fields being compared to short-term memory and the conceptual hierarchy utilizing a Rete network to support bottom-up recognition. This latter process is also closely related to methods for truth maintenance (e.g., Doyle, 1979), which often store justifications for beliefs so they can be removed when the environment changes.

Icarus also borrows from a distinct tradition of reactive control (e.g., Schoppers, 1987; Nilsson, 1994), which emphasizes sensor-driven execution to changing situations in uncertain environments. Similar concerns dominate research on reinforcement learning, from which we have adapted methods for estimating value functions from delayed reward (e.g., Watkins & Dayan, 1992). This paradigm has focused mainly on stimulus-response systems that associate value functions with situation-action pairs. Sun et al. (2001) incorporate reinforcement learning in their Clarion architecture, and some researchers (e.g., Parr & Russell, 1998) have examined variants that take advantage of hierarchical background knowledge, but our work is the first to embed hierarchical reinforcement learning in an agent architecture. A related influence comes from decision theory (Howard, 1968), which addresses value-driven decision making in uncertain circumstances. Icarus relies centrally on the decision-theoretic notion of alternative actions that produce outcomes with different expected values.

Finally, the agent architecture we have described in the previous pages retains many ideas from earlier versions of Icarus. Langley et al. (1989, 1991) report early designs for the architecture, which even then focused on reactive agents for physical environments. Early editions of Icarus also included distinct but connected long-term memories for concepts and plans, a module for executing complex motor skills, and a method for generating new tasks. A more recent version (Langley, 1997) emphasized selective attention during skill execution. The third incarnation introduced reactive execution of hierarchically organized skills (Shapiro & Langley, 1999) and methods for learning from delayed reward (Shapiro et al., 2001). The current Icarus incorporates ideas from each of its predecessors in a unified way. This includes our algorithm for estimating the linear value functions associated with hierarchical skills (Shapiro & Langley, 2002), which operates with the new skill representation and the internal reward signal that comes from matched concepts.

## 6. Directions for Future Research

Although the latest version of Icarus constitutes a significant advance over its predecessors, the architecture still lacks many capabilities that we would expect in a general intelligent agent. One such omission relates to our framework's emphasis on execution over problem solving and planning, which are important in their own rights. The backward-chaining mechanism for skill repair incorporates one key idea from work in this area, but planning also involves projecting the effects of future activities on the environment. For example, an Icarus agent should be able to imagine the experience of driving along different routes. The framework should also support mental execution of skills for nonreactive tasks like multi-column subtraction, where cognitive processing is sufficient to achieve the objective.

Another limitation of the current architecture is its restriction to executing one skill on each time step. Future versions should support the execution of skills in parallel, but place resource constraints on this ability. This will require an expanded formalism for skills that specifies the resources they consume on each cycle. We will also need to generalize Icarus' current method for skill selection to take expected resource consumption into ac-

count. We envision a decision-theoretic treatment that trades costs against benefits, similar to that in ACT-R but incorporating multiple resource dimensions. An important special case involves perceiving the environment, which currently happens automatically through preattentive processes. A more realistic scheme would handle some perception through explicit sensing actions that require resources and thus must be invoked selectively.

Our description of Icarus has emphasized the hierarchical nature of long-term skill memory, but, as it stands, the architecture offers no account of this hierarchy's acquisition. We have made some progress toward inducing hierarchical skills from behavior traces (Ichise, Shapiro, & Langley, 2002), but we should also develop methods that construct them from the agent's own execution efforts. One promising idea involves caching the results of each successful skill repair into a new higher-level skill that includes the repaired and repairing skills as components. This approach is similar in spirit to methods for chunking in Soar (Laird et al., 1987), knowledge composition in early versions of ACT (Anderson, 1983), and macro-operator formation (e.g., Iba, 1989). However, previous work along these lines has focused on structures that invoke actions in a fixed order, whereas cached Icarus skills would retain their reactive nature.

As noted earlier, Icarus incorporates ideas from reinforcement learning to revise the value functions associated with executed skills based on discounted reward. The architecture diverges from traditional treatments by embedding reward calculations within the agent, rather than in the environment. This raises intriguing questions about the origin of such rewards, which decision theory has not addressed, and suggests the radical notion that the reward functions associated with Boolean concepts might themselves be modified. In this scenario, the agent might start with only a few innate reinforcers, whereas most concepts have neutral affect. Over time, many of these concepts would become learned reinforcers, capable of rewarding behavior even when the innate concepts are unmatched. We envision a learning mechanism similar to the one that revises expected value functions, but applied to concepts that occur in executed skills, rather than to the skills themselves.

As we have seen, the current Icarus can recognize familiar situations with its conceptual hierarchy, but future versions should also recognize and interpret the behavior of other agents using its skill hierarchy. For example, when another car comes up quickly from behind and changes lanes, it should infer that the other agent is passing. To this end, we should incorporate ideas from the literature on plan understanding, which addresses similar problems. However, because the agent must infer what skills another agent is using before they have been completed, we cannot rely on the deductive matching scheme used for concept recognition. Rather, we will require something closer to abduction (e.g., Ng & Mooney, 1992), which uses general knowledge to explain a limited number of observations in a more flexible manner.

Finally, like most other agent architectures, Icarus lacks any episodic memory to store its own previous experience. Knowledge about concept instances that were once true and skills that it once executed would support important abilities, such as answering questions about past events. Upon reflection, episodic memory seems closely related to short-term memory, in that it deals with specific instances of general concepts and skills. We might encode such memories as variants on short-term literals that include time markers to indicate when they entered and left the short-term stores. Such traces should also include average statistics about the rewards generated by concept instances and those achieved when executing instantiated skills. The mechanisms responsible for retrieval from episodic memory, and the status of retrieved traces once they are deposited in short-term memory, remain open issues for future research.

## 7. Concluding Remarks

In this paper, we have described Icarus, a novel architecture for intelligent physical agents. Our approach embodies five design principles – that categorization has primacy over problem solving and execution, execution has primacy over problem solving, top-level intentions originate within the agent, intelligent behavior is inherently value driven, and reward is calculated internally rather than originating in the environment. These ideas distinguish our theoretical framework from most earlier architectures for cognition.

Icarus includes a long-term memory for concepts, which are defined as logical conjunctions of other concepts, and another memory for skills, which are defined in terms of concepts and component skills. The concept hierarchy supports a recognition process, which deposits concept instances in short-term memory, and reward calculation, which associates an affective score with each instance. The architecture also includes distinct modules for nominating, selecting, executing, repairing, and abandoning skill instances, each of which draws on the expected value functions associated with the generalized versions of those skills in long-term memory.

Despite the recent extensions we have described, Icarus remains an immature architecture relative to frameworks like Soar and ACT-R. However, we believe that its value-driven approach to intelligent behavior, along with its other distinctive features, will support functionalities that are difficult to achieve in these more traditional approaches. We hope to demonstrate these abilities in our future work on Icarus agents for driving and other physical domains.

## Acknowledgements

This research was funded in part by Grant NCC-2-1220 from NASA Ames Research Center through the Intelligent Systems Program. We thank Stephanie Sage and David Nicholas for extended discussions that led to many of the ideas in this paper.

## References


Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

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

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.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, *12*, 231–272.

Feigenbaum, E. A. (1963). The simulation of verbal learning behavior. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill.

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, *2*, 139–172.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*, 17–37.

Howard, R. A. (1968). The foundations of decision analysis. *IEEE Transactions on Systems, Science, and Cybernetics*, *4*, 211–219.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Ichise, R., Shapiro, D. G., & Langley, P. (2002). Learning hierarchical skills from observation. *Proceedings of the Fifth International Conference on Discovery Science* (pp. 247–258). Lubeck, Germany: Springer.

Kaelbling, L. P. (1993). Hierarchical learning in stochastic domains: Preliminary results. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 167–173). Amherst, MA.

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

Langley, P. (1997). Learning to sense selectively in physical domains. *Proceedings of the First International Conference on Autonomous Agents* (pp. 217–226). Marina del Rey, CA: ACM Press.

Langley, P., McKusick, K. B., Allen, J. A., Iba, W. F., & Thompson, K. (1991). A design for the Icarus architecture. *SIGART Bulletin*, *2*, 104–109.

Langley, P., Thompson, K., Iba, W. F., Gennari, J., & Allen, J. A. (1989). *An integrated cognitive architecture for autonomous agents* (Technical Report 89–28). Irvine: University of California, Department of Information & Computer Science.

Lebowitz, M. (1987). Experiments with incremental concept formation: Unimem. *Machine Learning*, *2*, 103–138.

Minton, S. N. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, *42*, 363–391.

Nardi, D., & Brachman, R. J. (2002). An introduction to description logics. In F. Baader et al. (Eds.), *Description logic handbook*. Cambridge: Cambridge University Press.

Neches, R., Langley, P., & Klahr, D. (1987). Learning, development, and production systems. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development*. Cambridge, MA: MIT Press.

Newell, A. (1973). You can't play 20 questions with nature and win: Projective comments on the papers of this symposium. In W. G. Chase (Ed.), *Visual information processing* New York: Academic Press.

Newell, A., Shaw, J. C., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Information Processing: Proceedings of the International Conference on Information Processing* (pp. 256–264). UNESCO House, Paris.

Ng, H. T., & Mooney, R. J. (1992). Abductive plan recognition and diagnosis: A comprehensive empirical investigation. *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning* (pp. 499–508). San Mateo, CA: Morgan Kaufmann.

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

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems 10* (pp. 1043–1049). Cambridge, MA: MIT Press.

Schoppers, M. (1987). Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039–1046). Milan, Italy: Morgan Kaufmann.

Shapiro, D., & Langley, P. (1999). Controlling physical agents through reactive logic programming. *Proceedings of the Third International Conference on Autonomous Agents* (pp. 386–387). Seattle: ACM Press.

Shapiro, D., & Langley, P. (2002). Separating skills from preference: Using learning to program by reward. *Proceedings of the Nineteenth International Conference on Machine Learning* (pp. 570–577). Sydney: 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.

Sun, R., Merrill, E., & Peterson, T. (2001). From implicit skills to explicit knowledge: A bottom-up model of skill learning. *Cognitive Science*, *25*, 203–244.

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

Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Table 1: Icarus concepts for the highway driving domain, omitting the `:reward` and `:weights` fields used to compute reward. Variables start with the symbol `?` and numeric concepts are marked with the symbol `#`.

```
(in-lane (?car ?lane)
    (lane ?lane ?left-line ?right-line)(car ?car)
    (#xdistance ?car ?left-line ?dleft)
    (#xdistance ?car ?right-line ?dright)
    (< ?dleft 0) (> ?dright 0))

(ahead-of (?car1 ?car2)
    (car ?car1)(car ?car2)
    (#yback ?car1 ?back1)
    (#yfront ?car2 ?front2)
    (> ?back1 ?front2))

(overlaps (?car1 ?car2)
    (car ?car1)(car ?car2)(#yfront ?car1 ?front1)
    (#yback ?car1 ?back1)(#yfront ?car2 ?front2)
    (> ?front1 ?front2)(> ?front2 ?back1))

(#distance-ahead (?car1 ?car2 ?diff)
    (ahead-of ?car1 ?car2)
    (#yback ?car1 ?back1)
    (#yfront ?car2 ?front2)
    (*bind ?diff (- ?back1 ?front2)))

(faster-than (?car1 ?car2)
    (#speed ?car1 ?s1)(#speed ?car2 ?s2)
    (> ?s1 (+ ?s2 5)))

(in-same-lane (?car1 ?car2)
    (car ?car1)(car ?car2)
    (in-lane ?car1 ?lane)
    (in-lane ?car2 ?lane))

(in-different-lane (?car1 ?car2)
    (car ?car1)(car ?car2)
    (in-lane ?car1 ?lane)
    (not (in-lane ?car2 ?lane)))

(overlaps-and-adjacent (?car1 ?car2)
    (car ?car1)(car ?car2)
    (in-lane ?car1 ?lane1)
    (in-lane ?car2 ?lane2)
    (adjacent ?lane1 ?lane2)
    (overlaps ?car1 ?car2))

(coming-from-behind (?car1 ?car2)
    (car ?car1)(car ?car2)
    (in-lane ?car1 ?lane1)
    (in-lane ?car2 ?lane2)
    (adjacent ?lane1 ?lane2)
    (faster-than ?car1 ?car2)
    (ahead-of ?car2 ?car1))

(adjacent (?lane1 ?lane2)
    (lane ?lane1 ?shared-line ?right-line)
    (lane ?lane2 ?left-line ?shared-line))

(adjacent (?lane1 ?lane2)
    (lane ?lane1 ?left-line ?shared-line)
    (lane ?lane2 ?shared-line ?right-line))

(clear-for (?lane ?car)
    (lane ?lane ?left-line ?right-line)
    (not (overlaps-and-adjacent ?car ?other))
    (not (coming-from-behind ?car ?other))
    (not (coming-from-behind ?other ?car)))
```

Table 2: An Icarus skill for one car passing another and the subskills it relies upon, omitting the `:value` and `:weights` fields used to compute expected values.

```
(pass (?car1 ?car2 ?lane)
 :start      (ahead-of ?car2 ?car1)
             (in-same-lane ?car1 ?car2)
 :objective  (ahead-of ?car1 ?car2)
             (in-same-lane ?car1 ?car2)
 :requires   (in-lane ?car2 ?lane)
             (adjacent ?lane ?to)
 :ordered    (speed-and-change ?car1 ?car2 ?lane ?to)
             (overtake ?car1 ?car2 ?lane)
             (change-lanes ?car1 ?to ?lane))

(speed-and-change (?car1 ?car2 ?from ?to)
 :start      (ahead-of ?car2 ?car1)
             (in-same-lane ?car1 ?car2)
 :objective  (much-faster-than ?car1 ?car2)
             (in-different-lane ?car1 ?car2)
 :requires   (in-lane ?car2 ?from)
             (adjacent ?from ?to)
 :unordered  (speed-up-faster-than ?car1 ?car2)
             (change-lanes ?car1 ?from ?to))

(speed-up-faster-than (?car1 ?car2)
 :start      (faster-than ?car2 ?car1)
 :objective  (faster-than ?car1 ?car2)
 :requires   ( )
 :ordered    (*accelerate))

(change-lanes (?car ?from ?to)
 :start      (in-lane ?car ?from)
 :objective  (in-lane ?car ?to)
 :requires   (lane ?from ?shared-line ?right-line)
             (lane ?to ?left-line ?shared-line)
             (clear-for ?to ?car)
 :ordered    (*shift-left))

(change-lanes (?car ?from ?to)
 :start      (in-lane ?car ?from)
 :objective  (in-lane ?car ?to)
 :requires   (lane ?from ?left-line ?shared-line)
             (lane ?to ?shared-line ?right-line)
             (clear-for ?to ?car)
 :ordered    ((*shift-right)))

(overtake (?car1 ?car2 ?lane)
 :start      (behind ?car1 ?car2)
             (in-different-lane ?car1 ?car2)
 :objective  (ahead-of ?car1 ?car2)
 :requires   (in-lane ?car2 ?lane)
             (faster-than ?x ?y)
 :ordered    (*wait))
```