Using Background Knowledge to Speed Reinforcement Learning in Physical Agents

Daniel Shapiro
Department of Management Science and Engineering
Stanford University
Stanford, CA 94305
dgs@stanford.edu

Pat Langley
Institute for the Study of Learning and Expertise
2164 Staunton Court
Palo Alto, CA 94306
langley@isle.org

Ross Shachter
Department of Management Science and Engineering
Stanford University
Stanford, CA 94305
shachter@stanford.edu

ABSTRACT
This paper describes Icarus, an agent architecture that embeds a hierarchical reinforcement learning algorithm within a language for specifying agent behavior. An Icarus program expresses an approximately correct theory about how to behave with options at varying levels of detail, while the Icarus agent determines the best options by learning from experience. We describe Icarus and its learning algorithm, then report on two experiments in a vehicle control domain. The first examines the benefit of new distinctions about state, whereas the second explores the impact of added plan structure. We show that background knowledge increases learning rate and asymptotic performance, and decreases plan size by three orders of magnitude, relative to the typical formulation of the learning problem in our test domain.

Categories and Subject Descriptors
1.2.4 **Knowledge representation formalisms and methods.**
1.2.6 **Learning.**

General Terms
Algorithms, performance, design, experimentation, languages.

Keywords
Adaptation and learning, agent architectures, action selection and planning, hierarchical reinforcement learning.

1. INTRODUCTION
Artificial agents are technological artifacts that perform tasks for people. They sense their world and relate perceptions to their tasks in order to identify, and then apply, an appropriate response. In general, we want to build agents that operate in uncontrolled environments yet perform increasingly important and complex functions outside of human supervision. This requires sophisticated representations for encoding domain knowledge as well as a practical capacity to learn action policies from experience.

While current agent architectures supply methods of encoding domain knowledge, current policy learning techniques have no means to adequately exploit this information. For example, reinforcement learning has focused on the use of 'flat' situation-action maps that present all feasible options to the agent in all known situations. Recent work in hierarchical reinforcement learning improves on this situation by learning in the context of structured plans, but the emphasis is still on algorithm development as opposed to practical agent design.

This paper provides a general method of merging policy learning with behavior specification. In particular, we describe an agent architecture called Icarus that embeds an algorithm for hierarchical reinforcement learning within a programming language for composing agent-held action plans. Icarus supports a novel development process in which the programmer writes an approximately correct plan that includes options at varying levels of detail, the agent isolates the best options by learning from experience, and the user supplies an appropriate reward function. Since the user can create many distinct agents by defining different targets of optimization, while the programmer encodes a single set of domain skills, we say that Icarus supports "programming by reward".

Icarus has the potential to impact the fields of computational learning and agent architectures because it combines ideas from both areas. In particular, the ability to learn should increase agent autonomy, while the ability to incorporate domain knowledge should increase the efficiency of learning. We report on two experiments in an automotive control domain that compare our work to a standard reinforcement learning design, and we show that background knowledge increases learning rate and asymptotic performance while profoundly decreasing plan size.

We introduce the Icarus language in Section 2 and its embedded learning algorithm in Section 3. After this, Section 4 motivates our experimental work and introduces our automotive test domain, while Sections 5 and 6 discuss our experiments to measure the impact of abstraction quality on learning and plan structure on learning. We review relevant work in Section 7 and offer concluding remarks in Section 8.

2. THE ICARUS LANGUAGE
Icarus provides a language for specifying the behavior of artificial agents that learn. Its structure is dually motivated by the desire to build practical agent applications and the desire to supply a behavioral guarantee based on a convergent learning algorithm. In particular, the need to solve complex problems argues for the use of powerful representations, while the desire for convergent learning suggests a simpler format that offers a clear mapping into the Markov decision process (MDP) model. MDPs provide a conceptual framework for

AGENTS'01, May 28-June 1, 2001, Montréal, Quebec, Canada.

developing algorithms, and mathematical properties useful for convergence proofs. We resolve this tension by casting Icarus as a reactive computing language.

Reactive languages are tools for specifying highly contingent agent behavior. They supply a representation for expressing plans, together with an interpreter for evaluating plans that employs a repetitive sense-think-act loop. This iterative interpretation provides adaptive response; it lets an agent retrieve a relevant action even if the world changes from one cycle of the interpreter to the next.

Reactive languages offer a spectrum of vocabularies for expressing plans. This includes combinational logic [1], directed graphs [8], prioritized procedures [3], ordered production rules [14], and goal structures with preconditions [17]. Reactive languages also support different degrees of adaptive response. Some embed reaction in an overall schema for sequential behavior, while extremely reactive languages make no commitment to control flow (because their interpreters let the world change from one state to any other recognized by the plan in exactly one time step). This format is very similar in spirit to an MDP, since both employ an iterated situation-response loop and both allow arbitrary transitions with no memory of past state.

Icarus is an instance of an extremely reactive language. It shares the logical orientation of teleoreactive trees [14] and universal plans [17], but adds vocabulary for expressing hierarchical intent, as well as tools for problem decomposition found in more general-purpose programming languages. For example, Icarus supports function call, parameter passing, Prolog-like variable binding, pattern matching on facts, conditional control flow, and recursion.

An Icarus plan contains up to three elements: an objective, a set of requirements (or preconditions), and a set of alternate means (or methods for achieving objectives), as illustrated in Figure 1. Each of these can be instantiated by further Icarus plans, creating a logical hierarchy that terminates with calls to primitive actions or sensors. Icarus evaluates these fields in a situation-dependent order, beginning with the objective field. If the objective is already true in the world, evaluation succeeds and nothing further needs to be done. If the objective is false, the interpreter examines the requirements field to determine if the preconditions for action have been met. If the objective is false and the requirements are true, evaluation progresses to the means field, which contains alternate methods (primitive actions or subplans) for accomplishing the objective. The means field is the locus of all value-based choice in Icarus, since the objectives and requirements contain no options. Icarus learns to select the action or subplan that promises the largest expected reward. Shapiro [20] provides a more complete description of the Icarus language.

Table 1 illustrates the top-level elements of an Icarus plan for freeway driving. It contains an ordered set of objectives implemented as further subplans. Icarus repetitively processes this plan, starting with its first statement every execution cycle. The interpreter employs a three-valued semantics, where every statement in the language evaluates to one of True, False, or an Action. 'True' means the statement was true in the world, 'False' means the plan did not apply, and an 'Action' return identifies a piece of code for controlling actuators that addresses the objectives of the plan.

The first clause in Table 1 defines a reaction to an impending collision. If this context applies, Icarus returns the emergency-brake action for application in the world. If emergency braking is not required, evaluation proceeds to the second clause, which specifies a plan for reacting to trouble ahead, defined as a car travelling slower than the agent in the agent's own lane. This subplan contains options, as shown in Table 2. Here, the agent can move one lane to the left, move right, slow down, or cruise at its current speed and lane, but the plan does not include the option to speed up. Icarus makes a selection based on the long-term expected reward of each alternative.

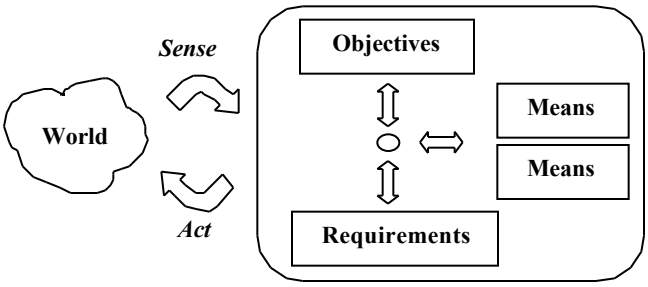

Figure 1. The structure of an Icarus plan.

If there is no imminent collision or trouble in front, Icarus examines the third clause of Table 1, which invokes a goal-driven subplan for bringing the agent to its target speed. This subplan causes the agent to speed up if it is traveling too slow or slow down if it is moving too fast, but it evaluates to 'True' if the agent is currently traveling at its target speed. (Note that the fields in an Icarus plan contain default values: False for the :objective, True for :requires, and False for the :means field.)

If the first three clauses in Table 1 are True, Icarus examines the fourth clause, a subplan for reacting to a faster car behind. This subplan (not shown) also contains options; it lets the agent move over or simply ignore the vehicle behind and cruise. Finally, if there is no cause to emergency brake, no trouble ahead, the agent is at its target speed, and there is no trouble behind, the fifth clause always returns an action. This causes the agent to cruise in its current lane at its current speed.

Table 1. The top level of an Icarus freeway-driving plan.

```
(drive ()
  :objective
   ((*not* (emergency-brake))
    (*not* (avoid-trouble-ahead))
    (get-to-target-speed)
    (*not* (avoid-trouble-behind))
    (cruise)))
```

Since Icarus plans contain choice points, the interpreter needs a method of selecting the right option to pursue. In particular, we would like to know the total benefit (as opposed to the immediate return) for making a given choice on the current time step, so that the agent can maximize its prospective future reward. Icarus provides this capability by associating a value estimate with each Icarus plan. This number represents the expected future discounted reward stream for choosing a primitive action or subplan on the current execution cycle and following the policy (being learned) thereafter. Icarus computes this expected value using a linear function of current observations. For example, avoid-trouble-ahead (Table 2) defines several parameters solely for the purpose of value estimation; the data are not required to execute any of the routines in its :means field.

The estimation architecture addresses an interesting tension in information needs. On one hand, the value of a plan clearly

Table 2. An Icarus plan with alternate subplans.

```
(avoid-trouble-ahead ()
   :requires
    ((bind ?c {car-ahead-center })
    {> {velocity } {velocity ?c }}
    (bind ?tti  {time-to-impact })
    (bind ?rd  {distance-ahead })
    (bind ?rt {- {target-speed } {velocity }})
    (bind ?art {abs ?rt }))
  :means
   ((safe-cruise ?tti ?rd ?art)
    (safe-slow-down ?tti ?rd ?rt)
    (move-left ?art)
    (move-right ?art)))
```

depends upon its context; the future of 'decelerate' is very different if the car in front is close or far. On the other hand, the cardinal rule of good programming is "hide information". We should not force Icarus programmers to define subplans with a suite of value-laden parameters that are irrelevant to performing the task at hand. Our solution is to inherit context-setting parameters down the calling tree. Thus, avoid-trouble-ahead measures the distance to the car in front, and Icarus *implicitly* passes that parameter to the decelerate action several levels deeper in the calling tree. The programmer writes Icarus code in the usual fashion, without concern for this implicit data.

3. THE SHARSHA ALGORITHM

Icarus contains an algorithm for learning the best decision to make at each choice point in the plan. In general terms, this is the problem of optimal control; we want choose actions that maximize an objective function by influencing the trajectory of a dynamic system. In reinforcement learning, the actions are constrained to come from a conditional plan, the system evolves stochastically, and the objective is to maximize (or improve) an expected reward stream, most often a future discounted sum of in-period rewards. Reinforcement learning typically treats the system dynamics as unknown. Moreover, 'model-free' methods never even attempt to determine an explicit probabilistic mapping between states. Instead, they learn a direct map from action to the expected reward stream. This approach requires search over the space of possible sequences of control actions. Many reinforcement learning algorithms can discover an optimal policy via a search of this kind, given special problem structure. Kaelbling, Littman, and Moore [9] provide an excellent survey of such techniques, including both policy-space and value iteration methods.

SHARSHA is a reinforcement learning method mated to Icarus plans. It is a model-free, on-line technique that determines an optimal control policy by exploring a single, infinitely long trajectory of states and actions. SHARSHA (for State Hierarchy, Action, Reward, State Hierarchy, Action) adds a sense of hierarchical intent to an earlier method called SARSA (for State, Action, Reward, State, Action).

The well-known SARSA algorithm operates on state-action pairs. It learns an estimate for the value of taking a given action in a given state by sampling its future trajectory. SARSA repeats the following steps:

- Select and apply an action in the current state;
- Measure the in-period reward;
- Observe the subsequent state and commit to an action in that state; and
- Update the estimate for the starting state-action pair, using its current value, the current reward, and the estimate associated with the destination pair.

In other words, SARSA bootstraps; it updates value estimates with other estimates, grounding the process in a real reward signal. Singh, Jaakola, Littman, and Szepesvari [21] have recently shown that SARSA converges to the optimal policy and the correct values for the future discounted reward stream. The proof imposed common Markov assumptions, required an exact (tabular) representation of the true reward function, and allowed a range of action selection policies that guaranteed sufficient exploration of apparently sub-optimal choices.

SHARSHA adapts SARSA to plans with a hierarchical model of intent. In particular, it operates on *stacks* of state-action pairs, where each pair corresponds to an Icarus function (encoding a plan to pursue a course of action in a given situation), as depicted in Figure 2. For example, at time 1 an Icarus agent might accelerate to reach its target speed in order to drive, while at time 2 it might brake in order to avoid trouble as part of the same driving skill. SHARSHA employs the SARSA inner loop with slight modifications: where SARSA observes the current state, SHARSHA observes the calling hierarchy, and where SARSA updates the current state, SHARSHA updates the estimates for each function in the calling stack. The second difference is that SHARSHA's update operator inputs the current estimate, the reward signal, and the estimate associated with the *primitive* action on the next execution cycle. In principle, this primitive carries the best estimate because it is based on the most informed picture of world state built while descending the tree.

We have proven SHARSHA's convergence properties elsewhere [19]. Our implementation adds practical features that go beyond the proof, such as eligibility lists to speed learning and linear value approximation functions to increase the method's generality in place of tabular forms. This version of SHARSHA learns the coefficients of these linear mappings from delayed reward.

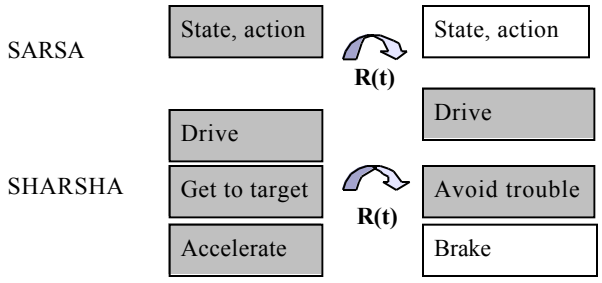

Figure 2. A comparison of Sarsa and Sharsha.

4. THE FREEWAY DRIVING DOMAIN

While an Icarus agent will learn an optimal policy given appropriate assumptions about the domain, we need empirical studies to establish the system's behavior in practice. In particular, Icarus supplies a novel capacity to incorporate domain knowledge into learning, and we would like to know its effect relative to a standard reinforcement learning design (like SARSA).

We used a freeway driving domain to conduct empirical tests. This environment consists of a simulator (written in C) together with an agent program (written in Icarus) that pilots

one of several hundred simulated cars. The cars live on an endless loop freeway that contains three lanes, but no entrances or exits. The traffic density is light. Each car has a target velocity drawn from a normal distribution with $\mu = 60$ and $\sigma = 8$ mph. With the exception of the one "smart" car that is capable of learning, every vehicle in the simulation determines its maneuvers by one of two fixed situation-action maps. All of them will change lanes to maintain their target speed, but roughly half will also move over to let a faster car pass.

The Icarus program controlling the smart car can sense its own target speed (fixed at 62 mph), the presence and relative velocity of six surrounding cars (ahead left, behind left, ahead right, behind right, ahead center and behind center), the distance to the car ahead center and behind center, and whether it is possible to change lanes to the left or right without colliding with another vehicle. There are six primitive actions: speed up by two mph, slow down by two, cruise at the current speed, change lane to the left, change lane to the right, and emergency brake. The last action instantaneously slows the car to two mph below the speed of any car in front, or to zero mph, whichever is greater.

We used the same reward function for all experiments in this paper. It is a piecewise linear function that mediates between the desires to maintain safety and target speed:

$$R(t) = \min(0, 10\ \text{TimeToImpactAhead} - 1000) + \min(0, 10\ \text{TimeToImpactBehind} - 1000) + 10\ |\text{TargetSpeed} - \text{Velocity}| \quad (1)$$

The freeway driving domain has several motivating features. First, it is familiar, which means that we can bring our intuitions to bear in generating and evaluating results. The task is also complex enough to support multiple control strategies. This makes it interesting from the perspective of computational learning. Next, we express personality in our driving, and this research develops a method of evolving such stylized controllers. Freeway driving is also a physical domain of the kind addressed by the Icarus architecture, and its complexity is appropriate for a research study. Finally, driving is relevant to people's lives (perhaps too relevant for some).

5. EFFECTS OF STATE ABSTRACTION

Our first experiment focused on the use of hierarchical reactive skills to specify action on the basis of partial state information. This capability is absent or unrecognized in most reinforcement learning designs, which treat a full diagnosis of world state as an implicitly available input. Our intuition (following good decision-theoretic principle) is that an increased appreciation of state will produce higher quality decisions. However, more distinctions imply a larger state space, making the optimal policy harder to find. This suggests a tension between learning rate and eventual performance.

We examined this topic by controlling the number of questions the agent can ask about its environment before it acts. To do so, we defined an Icarus plan that determines if the agent is above or below its own target speed and if the surrounding six cars are present or absent. After each observation, the plan offers the agent the option to slow down, speed up, cruise, or change lane to the left or right (if those actions are feasible). We define *cognitive resource*, w, as the probability of asking the next question vs. acting on the basis of the partial information at hand. Thus, if w = 1, the agent asks all available questions before acting, but if w = 0, it acts on no information at all. (While we could have compared plans of different depths, viewing w as a probability provides a better model of the tendency to think vs. act.)

Given this background, we can restate our intuition as a formal hypothesis: as the agent's cognitive resource grows, its asymptotic performance will increase but its learning rate will fall. That is, new distinctions about state will improve eventual performance, but raise the number of trials needed to reach that level.

We plot the average reward obtained by an Icarus agent in the freeway domain in Figure 3a, given the above plan, and low, medium, or high cognitive resource levels. Each curve represents the average of ten trials of 32,000 iterations apiece, smoothed within a moving window of 200 iterations. (Note that this is a log plot.) Since each run explores a very different scenario, the result is still quite noisy, so for clarity we plot the convex hull.

The relative shape of these curves has clear significance. As expected, the learning rate decreases with increasing cognitive resource and the asymptotic performance grows. This effect is also visible in the less processed data of Figure 3b, which shows the first 10,000 elements of the data in Figure 3a without the convex hull operation. This pattern repeats when we run the same experiment for a much larger plan that lets the agent ask up to 14 questions about the presence or absence of adjacent cars, as well as their relative distance and velocity. We conclude that the hypothesized effect is robust.

Given that agent performance clearly depends upon the quantity of its distinctions about world state, it also makes sense to examine the impact of distinction quality. Our intuition is that high-quality distinctions should improve performance faster than lower quality ones, meaning that there is an interaction between plan quality and cognitive resource.

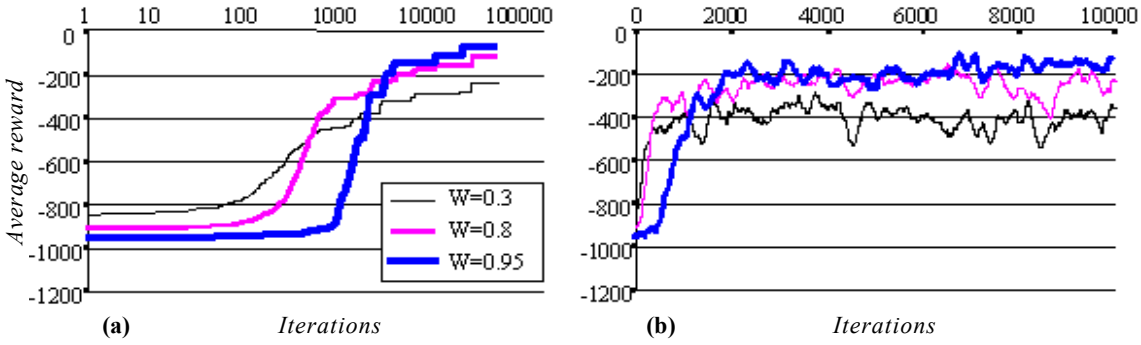

Figure 3. Learning rate as a function of cognitive resource, w, as shown in a log plot of the convex hull of average reward (a), and by the first 10,000 elements of average reward alone (b).

In order to examine this question, we constructed two Icarus agents, one aligned and the other misaligned relative to the performance metric in equation (1). These agents possess the same distinctions but they ask questions about the world in the exact opposite order, as shown in Figure 4. The 'aligned' agent determines if it is above (or below) its own target speed, and then tests for the presence or absence of the surrounding six cars in the order shown. After each question, it can choose to slow down, speed up, change lanes left or right, or cruise, as before. The 'unaligned' agent has the same options, but it asks about cars in adjacent lanes, then in its own lane, and finally about its velocity relative to its own target speed. Since the reward function concerns in-lane parameters, the unaligned agent should operate at a severe disadvantage.

Given this context, we predicted that asymptotic performance would increase more rapidly with cognitive resource for the aligned agent than for the unaligned one. We tested this hypothesis by training the aligned and unaligned agents at various cognitive resource levels and measuring average reward. Figure 5a gives the results of this interaction experiment. Once again, cognitive resource (denoted by w) is the probability of asking the next question, so 1−w is the probability of acting on current information alone. Each data point reports the asymptotic performance of an agent operating with resource w, measured as the average reward for the final 150,000 iterations of a 250,000-iteration learning run.

The data show that new information about state clearly increases asymptotic performance. Both agents learn policies that increase reward more or less monotonically with increasing cognitive resource. In addition, the agent with the most useful distinctions about state learns to perform at least as well as the agent with bad abstractions for any fixed resource level. Figure 5 shows the improvement rate: the aligned agent extracts most of its value by asking a relatively small number of questions, while the unaligned agent performs poorly until it accesses the end of its list. The curious graph shapes occur because the agents are almost equivalent at the two extremes of cognitive resource. At w = 0.3, both agents act with their eyes largely closed, while at w = 1, both ask all available questions.

This effect is repeatable for very large plans, as shown in Figure 5b. Here, we plot the same interaction curve but let the aligned and unaligned agents measure the presence or absence of cars in all six positions, their relative velocity, and relative distance (if the car is in the agent's own lane). This generates a very large plan with ~40,000 state-action pairs, as opposed to 1,200 for the plan that produced Figure 5a. Once again, performance increases monotonically with cognitive resource.

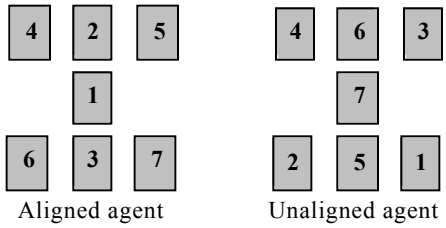

Figure 4. The order of examination for neighboring cars.

In summary, our experiments on the effects of state abstraction show that agent performance improves with increasing knowledge about state, while learning rates simultaneously decrease. However, there is an interaction between cognitive resource and abstraction quality: performance increases faster with limited amounts of cognitive resource if the agent possesses well-tailored distinctions.

6. EFFECTS OF PLAN STRUCTURE

Our second empirical investigation examines Icarus' ability to exploit domain knowledge by encoding it into the structure of a plan. In particular, the language lets a programmer compose an approximate plan that focuses learning by restricting the agent's options and ordering its concerns. An agent of this kind should learn faster than a typical reinforcement learning system, which lacks access to such constraints. At the same time, reliance on background knowledge can be a two-edged sword because it eliminates feasible options. A misleading approximate plan should degrade performance.

In order to measure the impact of plan structure on learning, we compared the behavior of a knowledgeable agent that pursues a hand-coded, structured plan for driving against an unconstrained agent that employs a 'flat' plan. (We have implemented both as Icarus programs.)

The flat plan corresponds to the normal reinforcement learning approach in that it presents the agent with every feasible option in every possible situation. It determines the presence or absence of the cars in the adjacent lanes. If any are present, it determines their velocity relative to the agent (faster/slower) and if the car is in the agent's lane, it determines the relative distance (near/far). After gathering all these data, the agent chooses among five actions in each of the resulting world states: accelerate by two, decelerate by two, cruise, change lane left, or change lane right. We prefaced the plan with an emergency-braking check, and we eliminated any infeasible action that would produce a collision before it was applied.

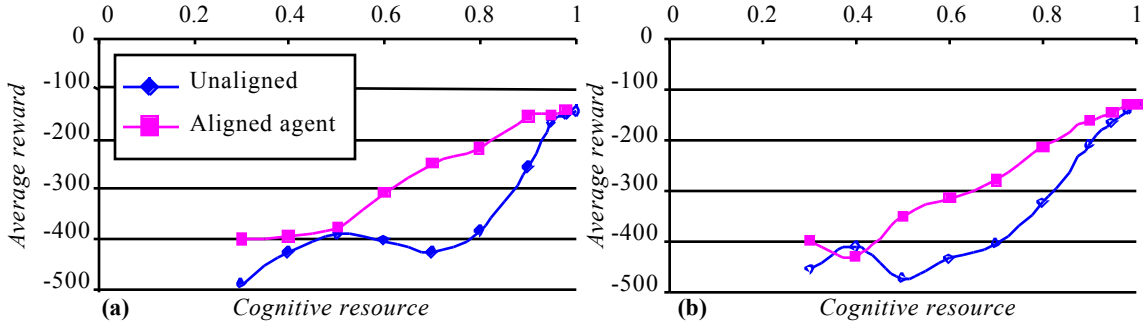

Figure 5. The interaction between cognitive resource and abstraction quality, as shown by average reward for aligned and unaligned agents in (a) a plan with 1200 state-action pairs, and (b) a plan with 40,000 pairs.

In contrast, the structured plan embeds domain knowledge into a hand-coded Icarus skill by hierarchically decomposing the knowledgeable agent's concerns and restricting its options. The top-level routine (Table 1) contains five reactive contexts, and the agent considers them in order on every execution cycle. This plan instructs the agent to worry about the need to slam on the brakes, followed by the need to react to trouble in front (a slower moving car in the agent's own lane), reaching its own target speed, and then dealing with trouble behind. If none of these situations apply, the agent cruises at its current speed in its current lane.

This Icarus skill lacks top-level options. However, the subplan for dealing with trouble in front (Table 2) contains alternatives (as does the plan for trouble behind): the agent can move left, move right, cruise, or slow down, but it cannot accelerate. Icarus learns a policy over these restricted options. Note that the combination of hierarchy and subplan order increases the scope of the agent's reactions. Thus, the responses within avoid-trouble-ahead hold across many world states, whether or not the agent is at its target speed and/or there is trouble behind.

Given this background, we hypothesized that the structured plan would produce a higher learning rate than the flat plan because the added domain knowledge reduces the search space. However, the knowledgeable agent should obtain the same or a lower level of asymptotic performance because its plan eliminates some options.

Figure 6a compares the behaviors of the structured plan and the flat design. We used the same reward function to measure performance (equation 1), and the same learning algorithm, sensor tests, and underlying actions. We collected ten runs of 32,000 iterations for each agent, and randomized the starting conditions by resetting the agent's velocity to a random number between zero and its desired target speed (62 mph) between each run. We calculated a smoothed average of the received reward and plot the convex hull, as before. Note that we use a 200-iteration window to smooth the data from the flat plan, but a 50-iteration window for the structured plan. This asymmetry preserves the leading edge of the knowledgeable agent's learning curve, which is exceptionally fast.

The results are somewhat surprising. As expected, the knowledgeable agent learns faster than the unconstrained agent, but by approximately two orders of magnitude. This improvement flows from the use of domain knowledge in two ways. First, the knowledgeable agent learns over a restricted set of options whenever it experiences a choice (e.g., in responding to a slower car in front). Second, the added background knowledge can remove all choice, for example, by forcing the agent towards its target speed in the absence of obstacles. The data reflects both effects, and both capture the intended benefit of incorporating domain knowledge.

The other surprise in Figure 6a is that the knowledgeable agent appears to perform *better* than the unconstrained agent even after 32,000 iterations of learning. This asymptotic performance result disconfirms our hypothesis and our intuition, since the agent with the superset of options should do at least as well as the agent with a restricted set. This effect is also visible in Figure 6b, which shows the first 10,000 elements of the data in Figure 6a, with a uniform 200-iteration smoothing window and without the convex hull operation. Once again, the structured plan shows dominant performance.

In order to check this result, we extended the duration of the experiment from 32,000 to 250,000 iterations. As it was only practical to train one instance of each agent type for that length of time, we computed two data points representing the asymptote of learning as the average over the last 150,000 iterations of each 250,000 iteration learning run. We show these points in Figure 6b. The result supports our previous conclusion: the knowledgeable agent learns faster *and* performs better than the unconstrained agent even after 250,000 iterations of learning. Moreover, because the structured plan asks fewer questions and provides fewer actions, we note that it involves less work per iteration than the standard, flat design.

This leverage is largely due to reduced plan size. When we counted the number of state-action pairs in each design (somewhat after we generated the learning curves), we discovered a three order of magnitude difference. While the standard (flat) statement of the reinforcement learning problem for vehicle control has over 20,000 primitive alternatives, the hand-coded, structured plan contains exactly 24 state-action pairs. Since our learning algorithm (and reinforcement learning in general) searches the space of possible action sequences, it is not surprising that the unconstrained agent failed to identify the better policy in any reasonable period of time.

In summary, the ability to encode domain knowledge simultaneously improves learning rate, increases performance, and decreases plan size. The fact that we obtain a two order of magnitude increase in learning rate and a three order of magnitude decrease in plan size suggests that our method can qualitatively expand the scope of complex learning applications. This power comes from Icarus' ability to encode domain knowledge into the statement of learning problems, and from embedding a reinforcement learning algorithm into a general-purpose hierarchical language for agent design.

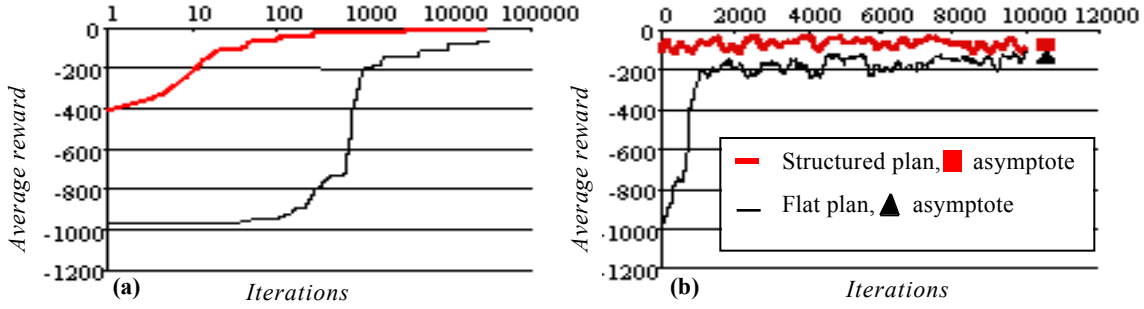

Figure 6. The impact of domain knowledge in learning, as shown (a) in a log plot of the convex hull of average reward, and (b) by the first 10,000 elements of average reward alone, with asymptotes at 250,000 iterations.

7. RELATED WORK

Research on agent architectures often has an interdisciplinary feel, and our work on Icarus is no exception, as it draws on ideas from three distinct literatures: reactive languages, reinforcement learning, and agent design. We discuss related work from each of these perspectives in turn.

Every agent architecture provides a method of specifying behavior, and many offer special-purpose languages for the task. We characterize this design space by the degree of reactivity the languages are intended to support and by their emphasis on practical application. Icarus contributes to the most reactive and most application-oriented end of this spectrum. It is *extremely* reactive, like universal plans [17], teleoreactive trees [14], and subsumption architectures [3], since all four systems return a relevant action even if the world shifts to any other recognized state between execution cycles. Yet, Icarus adds an element of expressivity to these designs. It offers a logical and symbolic vocabulary to subsumption, and we have shown [18] that it adds hierarchically defined actions to universal plans, explicit goals and preconditions to teleoreactive trees, and a novel construct to both systems: in Icarus, the objective of one plan can be to engage in another.

If we look at reactive languages through the lens of application, Icarus emerges as one of the few approaches that emphasize the practicalities of composing software. PRS [8] shares this emphasis and provides a larger array of primitives, while RAPS [7] inherits all of Lisp. However, Icarus is the only extremely reactive design with such features. For comparison, universal plans natively interpret modal logic formulae, teleoreactive trees employ an ordered list of production rules, and subsumption architectures present a format similar to a wiring diagram. Finally, Icarus offers a value-based learning mechanism that is absent in other reactive languages. This lets Icarus learn to select the best options, while other methods rely on predetermined (or random) choice.

Work in hierarchical reinforcement learning shares our interest in learning within the context of a structured plan. For example, Dayan and Hinton [4] impose a hierarchy of learning modules, while Kaelbling [10] structures the solution by forcing the learned policy to pass through an ordered sequence of goals. Sutton, Precup, and Singh [22] define macro-actions that fix behavior over a region of the state space and learn an optimal policy that switches among these options. Parr and Russell [15] define a hierarchy of non-deterministic finite-state machines and employ learning to extract an optimal deterministic controller. Dietterich [6] notes that these methods require a full description of system state, and shows that his MAXQ algorithm [5] will converge under appropriate state abstractions. MAXQ itself represents learning problems as an unordered tree of subtasks with a simultaneous value decomposition. Icarus adds a general mechanism for incorporating domain knowledge to these designs in the form of a reactive programming language. In addition, it bases convergent learning on a reactive execution model [19] while other convergent algorithms lace together temporally extended actions that resemble subroutines. This distinction is quite important in physical application domains, where uncontrollable events readily interrupt agent intentions.

Several other efforts have shown that domain knowledge improves performance relative to flat Q-learning. If we take a rough estimate from their published charts, Dietterich, Parr, and Sutton all show that learning rate improves by a factor of one to ten, while unconstrained Q-learning eventually obtains superior performance. Dietterich [5] notes an exception where MaxQ outperforms Q-learning at the end of the feasible training period. These results are similar to our own, and support our claim that added domain knowledge speeds learning.

Taken together, Icarus and SHARSHA offer a general-purpose architecture for constructing agents that learn. Another such architecture, Soar [11][13], has been used extensively to build deliberative problem solving agents, although it has also been applied to reactive execution systems. Soar provides a layered production-rule architecture and a means of creating subgoals to resolve impasses, as in the case where multiple operators simultaneously apply. Learning typically focuses on the desire to speed up future problem solving in response to success or failure. Soar makes it quite convenient to express the results of learning as new control rules that determine which operators to select or states to expand. In contrast, Icarus focuses on learning in the physical world, and it learns the value of existing options instead of new plan structure. The specific mechanism inputs a numeric vs. a qualitative signal and generates a numeric vs. a symbolic result. Thus, the two systems employ complementary approaches. We hope to use ideas from Soar to address structure learning in Icarus.

Prodigy [23] is an architecture for integrating planning and learning. Like Soar, it has been applied to execution systems, but the main emphasis has been on problem solving. An early version of Prodigy [12] applied explanation-based learning to develop new rules that govern which states to expand or operators to select. More recent work adds learning methods with mutually interpretable knowledge structures, such as analogical reasoning and learning by experimentation. These methods support the goals of improving planner efficiency, improving plan quality, and developing domain knowledge for use in planning, and they operate by learning structure. In contrast, Icarus learns the value of existing options, which makes its relation to Prodigy similar to its relation with Soar.

ACT-R [2] is an agent architecture dedicated to the proposition that cognitive skills are realized by production rules. As a result, much of the work focuses on duplicating trace data from human experiments. ACT-R offers production rules with an explicit concept of goals, a separate declarative memory for facts, and a method of efficiently finding high-quality productions relevant to the goal and situation at hand. Icarus shares many of these elements. Like Soar and Prodigy, ACT-R uses learning to acquire new skills (here, via reasoning by analogy), and Icarus can benefit from that technology. Icarus also shares ACT-R's ability to determine the long-term value of employing specific actions, although the systems use the concept of value in different ways. In particular, ACT-R is descriptive, while Icarus follows a normative design. Thus, we adopt a decision-theoretic framework with a maximizing model of choice, while ACT-R employs a satisficing method.

8. CONCLUDING REMARKS

Icarus contributes to the state of the art in agent design and several component technologies. The language for expressing agent behavior extends highly reactive designs, adding methods for encoding hierarchical intent to universal plans and teleoreactive trees, and encapsulating reactive processing in a programming tool. In addition, SHARSHA contributes to research in hierarchical reinforcement learning. It supports convergent policy learning within hierarchical reactive plans,

while other convergent methods rely on more constrained representations and a non-interruptible execution model.

More broadly, the combination of Icarus and SHARSHA offers a general method of incorporating domain knowledge in reinforcement learning. It lets programmers encode an approximate model of behavior, and relies on the agent to find the best options by learning from experience. Empirical evidence shows that this technique increases learning rate and performance, while reducing plan size, relative to the standard expression of such problems. The improvements are substantial and suggest that our approach offers a qualitative increase in the scope and efficacy of learning applications.

Our future work will explore several novel aspects of the Icarus architecture. We are currently investigating the model of programming by reward by using Icarus to evolve agent personalities in response to a user-supplied reward function. This work raises the interesting conjecture that behavioral differences can result from distinct preference structures acting on the same set of skills. Next, we plan to test our claim that Icarus renders complex learning problems feasible by applying the system in complex domains that involve multiple, dissimilar skills. We also hope to demonstrate an analytic property of the architecture [20] that guarantees certain Icarus agents will maximize human utility as a consequence of learning to maximize their own reward.

9. ACKNOWLEDGEMENTS

We thank the DaimlerChrysler Research and Technology Center for funding, and David Moriarty, Mark Pendrith and Simon Handley for developing our traffic simulator.

10. REFERENCES

[1] Agre, P. (1988). *The dynamic structure of everyday life.* Tech Report AI-TR-1085, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

[2] Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

[3] Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation, 2, 1*.

[4] Dayan, P., & Hinton, G. (1993). Feudal reinforcement learning. *Advances in Neural Information Processing Systems, 5* (pp 271-278). San Francisco: Morgan Kaufmann.

[5] Dietterich, T.G. (1998). The MAXQ method for hierarchical reinforcement learning. *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. 118-126). Morgan Kaufmann.

[6] Dietterich, T.G. (2000). State abstraction in MAXQ hierarchical reinforcement learning. *Advances in Neural Information Processing Systems, 12*. MIT Press.

[7] Firby, J. (1989). *Adaptive execution in complex dynamic worlds*. PhD Thesis, Department of Computer Science, Yale University, New Haven, CT.

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

[9] Kaelbling, L. P., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research, 4,* 237-285.

[10] Kaelbling, L. P. (1993). Hierarchical learning in stochastic domains: Preliminary results. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 167-173). Morgan Kaufmann.

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

[12] Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach*. Boston, MA: Kluwer Academic Publishers,.

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

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

[15] Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems, 10* (pp. 1043-1049). MIT Press.

[16] Rosenbloom, P. S., Laird, J.E., Newell, A., & McCarl, R. (1991). A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence, 47,* 289-325.

[17] Schoppers, M. (1987). Universal Plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039-1046). Morgan Kaufmann.

[18] 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.

[19] Shapiro, D., & Shachter, R. (2000). *Convergent reinforcement learning algorithms for hierarchical reactive plans*. Unpublished manuscript. Department of Management Science and Engineering, Stanford University, Stanford, CA.

[20] Shapiro, D. (2000). *Value-driven agents*. PhD thesis, Department of Management Science and Engineering, Stanford University, Stanford, CA.

[21] Singh, S., Jaakola, T., Littman, M., & Szepesvari, C. (in press). Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*.

[22] Sutton, R. S., Precup, D., & Singh, S. (1998). Intra-option learning about temporally abstract actions. *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. 556-564). Morgan Kaufmann.

[23] Veloso, M., Carbonell, J., Perez, M., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence, 7,* 81-120.