Separating Skills from Preference: Using Learning to Program by Reward

Daniel Shapiro DGS@STANFORD.EDU
Pat Langley LANGLEY@ISLE.ORG
Institute for the Study of Learning and Expertise, 2164 Staunton Court, Palo Alto, CA 94306 USA

Abstract

Developers of artificial agents commonly assume that we can only specify agent behavior via the expensive process of implementing new skills. This paper offers an alternative expressed by the separation hypothesis: that behavioral differences among individuals can be captured as distinct preferences over the same set of skills. We test this hypothesis in a simulated automotive domain by using reinforcement learning to induce vehicle control policies, given a structured set of driving skills that contains options and a user-supplied reward function. We show that qualitatively distinct reward functions produce agents with qualitatively distinct behavior over the same set of skills. This leads to a new development metaphor that we call 'programming by reward'.

1. Motivation and Background

In many domains, humans exhibit complex physical behaviors that let them accomplish sophisticated tasks. Researchers have explored two main approaches to learning such behaviors, each associated with a different class of representational formalisms. One paradigm encodes control knowledge as rules or similar structures (e.g., Laird & Rosenbloom, 1990; Sammut, 1996) that state conditions under which to execute actions. An alternative framework instead specifies some function that maps state-action pairs onto a numeric utility (e.g., Watkins & Dayan, 1992), which is then used to select among actions.

Both approaches have repeatedly demonstrated their ability to learn useful control policies across a broad range of domains, yet each lends itself to expressing different aspects of intelligent behavior. This idea is best illustrated by work on game playing, where developers use rules or other logical constraints to specify which moves are legal but invoke numeric evaluation functions to select among them. We claim that a similar division of labor will prove useful in developing policies for reactive control, including learning such policies from agent experience.

We formalize this intuition by stating the *separation hypothesis*:

> We can effectively construct physical agents by encoding legal behavior in a set of logical skills and separately specifying a set of preferences over those skills cast as value functions.

This framework seems especially appropriate when one desires a number of distinct agents that exhibit a great variety of behaviors in the same domain. Furthermore, we claim that such agents can automatically learn these behaviors from feedback signals, meaning the developer only needs to implement these reward signals given a base of shared skills. This approach to agent design – which we call *programming by reward* – should prove useful in constructing synthetic agents for interactive entertainment, personalized services, and many other tasks.

In the following pages, we report one instance of this general framework, which we have cast in an architecture for physical agents called ICARUS. We begin by describing the architecture's logical formalism for encoding hierarchical skills, taking examples from the task of driving an automobile. We then turn to the value functions that ICARUS uses to select among applicable skills and its algorithm for using delayed rewards to update these functions. After this, we present experimental studies designed to test our hypothesis that providing such a system with different rewards can produce distinctive yet viable policies. Finally, we examine some other approaches to learning complex skills and suggest directions for additional research on this topic.

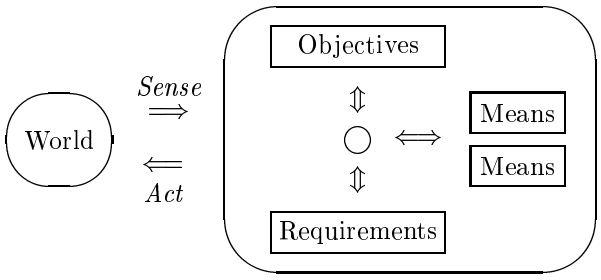

Figure 1. The structure of ICARUS plans.

2. The ICARUS Language

ICARUS is 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 support policy learning in a computationally efficient way. We responded to these goals by supplying ICARUS with powerful representations that map cleanly into the Markov Decision Process model, which provides a conceptual framework for developing learning algorithms. In particular, we cast ICARUS as a reactive computing language.

2.1 A Reactive Formalism

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 repetition provides adaptive response; it lets an agent retrieve a relevant action even if the world changes from one interpreter cycle to the next.

ICARUS is an extremely reactive language, as its interpreter's view of the world can change from one recognizable state to any other in exactly one time step. The architecture shares the logical orientation of teleoreactive trees (Nilsson, 1994) and universal plans (Schoppers, 1987), but adds vocabulary for expressing hierarchical intent, as well as tools for problem decomposition found in more general-purpose languages. For example, it supports function calls, Prolog-like parameter passing, pattern matching on facts, optional parameters, and recursion.

An ICARUS program contains up to three elements: an objective, a set of requirements, and a set of alternate means, 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 it is false, the interpreter examines the :requires field to determine if the preconditions for action have been met. If so, 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, since the objectives and requirements contain no options. To support this choice, the interpreter associates a value estimate with each plan and learns to select the plan with the largest expected reward.

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( ) ]
```

The architecture also supports several unusual features. It allows the execution of a process to be a goal and it embeds a sequence primitive within a reactive interpreter (where reaction within a sequential plan is more common). Moreover, it supports control over plan expansion; ICARUS can commit to a subplan before investigating it, or it can investigate subplans and choose among the actions returned. Shapiro (2001) provides a more complete description of the language.

2.2 An ICARUS Plan for Driving

Table 1 presents an excerpt from an ICARUS plan for freeway driving. The top-level routine, Drive, contains an ordered set of objectives implemented as further subplans. The first clause defines a reaction to an impending collision. The second specifies a plan for reacting to trouble ahead, defined as a car traveling 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 without changing speed or lane. The third clause defines a goal-driven subplan (not shown) for bringing the agent to its target speed. The fourth defines options for reacting to a faster car behind; it lets the agent move over or ignore the vehicle and cruise. The final clause leads the agent to Cruise in its current lane and at its current speed.

ICARUS processes the Drive program repetitively, starting with its first clause on every execution cycle. It performs a depth-first, left to right walk of the calling tree, using a three-valued semantics in which every statement in the language evaluates to True, False, or an action. 'True' means the statement was true in the world, 'False' means the plan did not apply, and

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-to-slow-down(?tti, ?rd, ?rt)
     move-right(?art)
     move-left(?art) ]
```

a returned action identifies code for controlling actuators that addresses the objectives of the plan. Since a :means clause can produce multiple actions, the interpreter selects and returns the best one. It passes the action returned from Drive to an external execution system, which applies it in the world. Thus, the purpose of an ICARUS program is to find action.

This repetitive evaluation process lets the system return an action from an entirely different portion of Drive on each successive iteration. For example, the agent might slam on the brakes on cycle 1, but change lanes on cycle 2 to avoid the (still) slower car in front, after the first clause returns True. Assuming the new lane is clear (the first two clauses return True), the agent might speed up in service of Get-to-target-speed on cycles 3 to 5, and then select the fifth clause, Cruise, until some other obstacle appears.

Whenever a plan offers a choice (e.g., in the :means field of Avoid-trouble-ahead), the agent needs a method for selecting the right option to pursue. ICARUS provides this capability by associating a value estimate with each 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 value using a linear function of current observations. For example, Avoid-trouble-ahead binds several parameters solely for the purpose of value estimation; the data are not required to execute any of the routines in its :means field.

This approach lets a plan's value depend upon its context. For example, the future of 'decelerate' is very different if the car in front is close or far. However, we should not force ICARUS programmers to specify all of the information required to estimate value when writing individual functions, so the system inherits context-setting parameters down the calling tree. Thus, Avoid-trouble-ahead measures the distance to the car in front, and ICARUS passes that parameter to the decelerate action several levels deeper in the calling tree. The programmer writes code in the usual fashion, without concern for these implicit data.

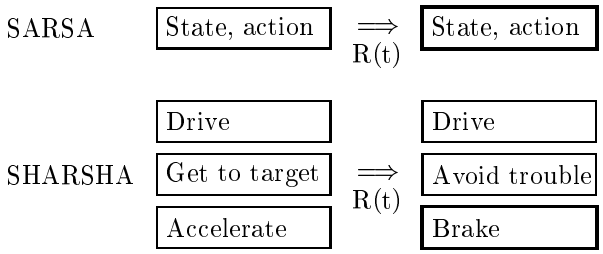

Figure 2. A comparison of SARSA and SHARSHA.

2.3 The SHARSHA Algorithm

SHARSHA is a reinforcement learning method mated to ICARUS plans. It is a model-free, on-line technique that updates its control policy by exploring a single, unbounded trajectory of states and actions. SHARSHA (for State Hierarchy, Action, Reward, State Hierarchy, Action) adds hierarchical intent to the well-known SARSA algorithm (for State, Action, Reward, State, Action).

SARSA operates on state-action pairs, learning an estimate for the value of taking a given action in a given state by sampling its future trajectory. SARSA repeats four steps: (1) select and apply an action in the current state; (2) measure the in-period reward; (3) observe the subsequent state and commit to an action in that state; and (4) 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 et al. (2000) have shown that SARSA converges to the optimal policy and correct values for the future discounted reward stream under a common set of Markov assumptions.

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 course of action), as depicted in Figure 2. For example, at time 1 the ICARUS agent accelerates to reach its target speed in order to drive, while at time 2 it brakes in order to avoid trouble as part of the same driving skill. Our method employs the SARSA inner loop with slight modifications: where SARSA observes the current state, we observe the calling hierarchy, and where SARSA updates the current state, we update 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 cycle. This primitive carries the best estimate because it utilizes the more informed picture of world state built while evaluating the ICARUS program.

Our implementation of SHARSHA includes several additional features. It employs eligibility lists to speed learning, it normalizes sensor values at run time (since the update rule can otherwise diverge), it supports multiple exploration policies, and it employs linear approximations for value functions in place of tabular forms. SHARSHA learns the coefficients of these linear mappings from delayed reward. We have proven SHARSHA's convergence properties under a common set of Markov assumptions (Shapiro, 2001).

3. An Experiment with Programming by Reward

Now that we have reviewed the ICARUS architecture, we can utilize it to experimentally evaluate the separation hypothesis. Here we focus on the task of freeway driving, in which human drivers exhibit considerable variation. Again, we hold that one can effectively capture this variability in synthetic agents by decomposing behavior into logical skills, which are shared across agents, and value-coded preferences, in which they differ. Moreover, such agents can learn these value functions from delayed reward, using their shared skills as background knowledge.

We evaluate the separation hypothesis in our test domain by conducting experiments with programming by reward. We start with an intuitively reasonable and constrained set of skills for the domain (thus distinguishing our approach from traditional reinforcement learning). Next, we define a set of distinct reward functions and use them as the target of learning. If this produces diverse behavior over the same skill set, we have support for the hypothesis. However, if we must permute the skills to alter behavior, the hypothesis is partially disconfirmed. The ideal result in the driving domain would be to show coverage over common driver types, including the ability to mimic extreme observed behaviors like aggressive driving. The remainder of the section reports on experiments of this form that utilize the ICARUS skills outlined in Tables 1 and 2 to constrain behavior.

3.1 The Driving Environment

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. Each car has a target velocity drawn from a normal distribution with $\mu = 60$ mph 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, the distance to the car ahead center and behind center, and whether it can change lanes to the left or right without hitting another vehicle. There are six primitive actions: speed up by two mph, slow down by two, cruise at current speed, change lane to the left, change lane to the right, and emergency brake.

3.2 Agent-held Reward Functions

In order to test the model of programming by reward, we defined a set of qualitatively different reward functions. All of them are linear in their feature values, and Table 3 associates their features with mnemonic names. The airport driver is motivated solely by the desire to reach the airport on time; it becomes less happy as its velocity deviates from target speed. The safe driver wants to avoid collisions; its reward function penalizes small times to impact with cars in front and cars behind. The shorter the time to impact, the larger the penalty, with times greater than 100 seconds having no reward. The goldfish driver has an imaginary fishbowl as luggage, and does not want maneuvers to upend the fish. Alternatively, we can view the goldfish driver as a bit queasy; its reward function penalizes all forms of maneuver. The reckless teenager is out for thrills; it garners reward for near misses and cares about maintaining its cruising speed. The crowd lover and the crowd hater desire the expected things; their reward increases (or decreases) with the number of surrounding cars. ICARUS calculates the reward once every execution cycle, and the learning system seeks to acquire the greatest reward stream over time.

3.3 A Profile of Learned Behavior

We used each of the above reward functions to develop agent personalities by employing them as the target of policy learning. We conducted ten 32,000-iteration training runs for each driver and averaged results over the final 20,000 iterations of each run. In all cases, we initialized the driver's velocity to a random number between zero and its target speed (62 mph), and all of

Table 3. Agent-held reward functions in terms of impact time ahead (a) and behind (b), deviation from target speed (t), slowing down (d), speeding up (u), changing lanes (l), and nearby cars (c).

	airport driver	safe driver	goldfish driver	reckless teenager	crowd lover	crowd hater
(a)		+		−		
(b)		+		−		
(t)	−			−		
(d)			−			
(u)			−			
(l)			−			
(c)					+	−

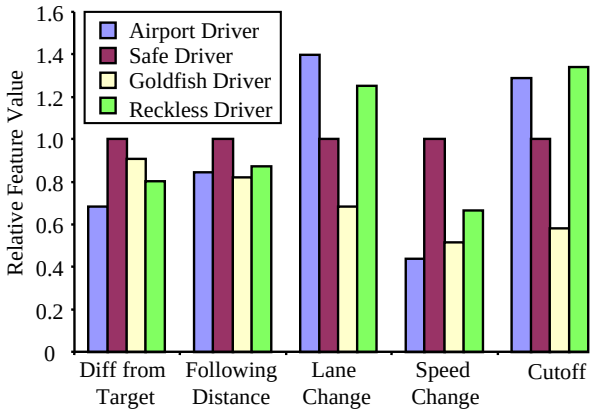

Figure 3. Learned behaviors in low-density traffic.

its value-estimation functions to zero. Figure 3 focuses on behavioral measures, using the safe driver's score as the unit quantity. We analyze the maximum and minimum values in each category.

The first measure is the absolute value of the agent's difference from its target speed. The fact that the airport driver has the lowest score is not surprising, since its reward function directly penalizes nonzero values. However, the safe driver shows the highest difference from target speed. It is not motivated (positively or negatively) by this quantity, but apparently it readily adjusts its velocity to avoid potential collisions (i.e., short times to impact).

The safe driver also shows the largest following distance. This makes intuitive sense, since safe drivers know that tailgating produces potential collisions. (If the car in front slows down, the safe driver inherits a significant penalty.) In contrast, the goldfish driver has the shortest following distance, by a small margin. We explain this observation by a cruise control effect: drivers who resist velocity changes will tend to creep up on the car in front. Both the airport and goldfish drivers contain this bias, but note that none of the agents assign direct value to following distance in their reward functions.

The airport driver displays the highest number of lane changes. This makes sense because it must maneuver to maintain its target speed. The goldfish driver shows the fewest, as it is centrally motivated not to make such changes. The speed change results are similar: the airport driver is directly biased against deviating from target speed, while the safe driver freely adjusts its speed to avoid potential impacts.

Finally, the goldfish driver performs the fewest cutoff actions, defined as a lane change in front of a faster vehicle, as it is motivated to avoid all maneuvers. In contrast, the reckless driver actively seeks near collisions, as they contribute positive terms to its reward.

Note that the driving program prevents the reckless teenager from simply colliding with the car in front; its opportunity to learn is confined to an allowable realm. Said differently, the agent's skills ensure reasonable behavior. Its reward function is irrelevant whenever the behavior is determined, and relevant only when choice is allowed. This design frees us to construct reward functions in an unconstrained way.

3.4 Learned Lane Preferences

Figure 4 illustrates an emergent property of programming by reward. We plot the agent's occupancy in the different freeway lanes, and note that the crowd lover evolves a slight preference for the middle lane, while the crowd hater generates a strong preference for the right hand lane. These preferences were never encoded in the reward functions, although the results make sense. A car in the center lane can encounter up to six adjacent vehicles (good for a crowd lover), while one in the right or left lane can have at most four neighbors. It seems clear that the crowd hater will avoid the center lane, but not why it prefers the right lane to the left. The fixed control policies of the other cars do act to sort vehicles into lanes by speed. Perhaps there is a smaller difference between the average speed in the right and center lanes than between left and center. If so, the crowd hater will encounter fewer cars per unit time if it gravitates to the right.

3.5 Driver Behavior Across Two Domains

It is clear that we can generate distinct behavior via programming by reward, but we would also like to know if that behavior is in some sense robust to environmental change. We investigated this question by

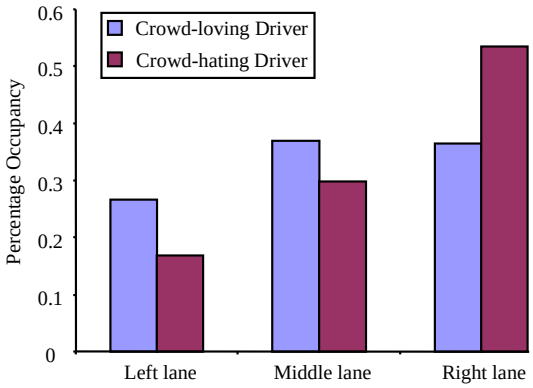

Figure 4. Lane preferences learned by two different drivers.

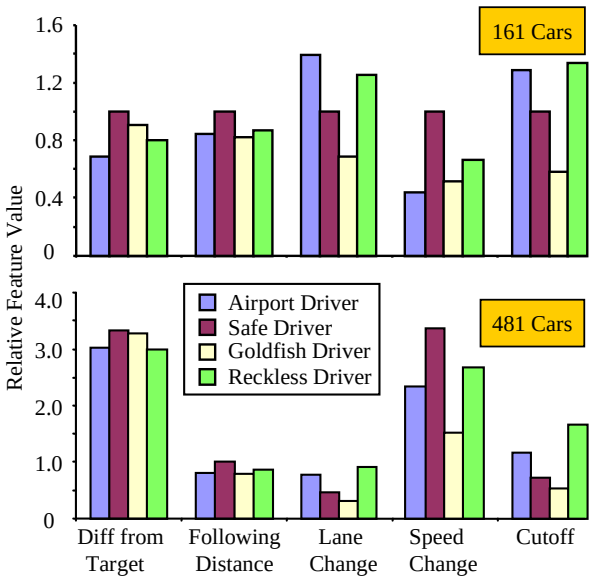

Figure 5. Learned behavior in two driving domains.

training the same agents in a high-density vs. a low-density traffic scenario. Figure 5 provides the results. Here, we take the performance of the safe driver in low-density traffic as the unit quantity, so that we can compare behavior both within and across domains.

Our first observation is that the absolute magnitudes of the metrics differ as we move between scenarios. It is generally harder to maintain target speed in high-density traffic; following distances shrink, it becomes more difficult to change lanes, and agents must adjust their speed more often in order to respond to other traffic. These changes are largely forced upon the agents by increased traffic density.

A more striking observation is that the behavioral profiles are beautifully preserved across domains. Although the number of instances of any given behavior changes, the shapes of the curves are virtually identical in low and high-density traffic. There are only two shifts in relative order, for the maximum number of lane changes and minimum number of speed changes. This constancy of behavior provides evidence that programming by reward shapes agent behavior in a predictable way, and that it can be used in a development model where agents are trained in a test domain and deployed in an application environment.

3.6 Searching the Space of Reward Functions

Now that we have examined the relation between a reward function and the behavior it generates, it is worth asking the inverse question. Can we generate a specific, predefined behavior via programming by reward? We pursued this question by attempting to duplicate (in a qualitative sense) the behavior of a colleague who is a particularly aggressive driver. We did this by searching across the space of possible reward functions, seeking to minimize the agent's following distance while simultaneously maximizing the number of cutoff maneuvers. This was an informal process, where we hand tuned the coefficients of a linear reward function that measured time to impact ahead/behind, distance ahead/behind, cutoff events, own speed, and the relative speed of cars in adjacent lanes.

The results were both positive and negative. On the positive side, we succeeded in generating a ten-fold increase in the number of cutoff maneuvers performed by the 'road rage driver' relative to the reckless teenager, as shown in Figure 6. However, we could do so only by introducing a slight modification to the shared driving skill; we gave both drivers the option to change lanes in the absence of a slower car in front or a faster car behind. The original skill lacked the flexibility to support the desired (extremist) behavior.

This experiment also generated an interesting strategic lesson for programming by reward. We discovered that it was far more successful to penalize the road rage driver as it was being passed by other cars, rather than to reward the agent when it cut off other vehicles. The reason is that it there are more opportunities to learn from persistent conditions than momentary events.

4. Related Work on Control Learning

Earlier we contrasted our framework for control learning with other approaches, but the previous work on this topic and its differences from our own deserves a more detailed discussion. Here we consider four distinct paradigms for learning control policies from experience that have appeared in the literature.

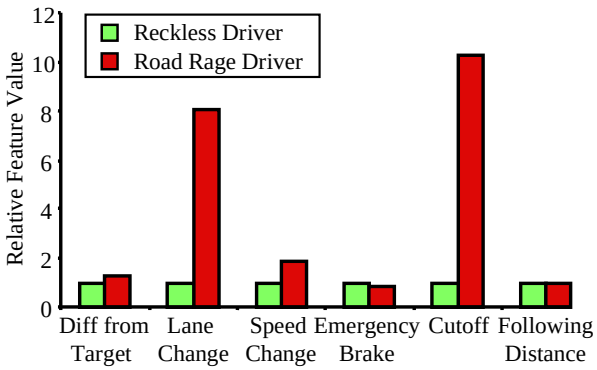

Figure 6. Using a reward function to generate road rage.

One body of research focuses on architectures for intelligent agents, with two well-known examples being SOAR (Laird & Rosenbloom, 1990) and PRODIGY (Minton, 1990). These systems represent knowledge about legal actions as production rules or logical operators, which they utilize during problem solving and execution. Because this knowledge predicts the effects of operators, they can learn from the results of problem solving, rather than relying, as does ICARUS, on feedback from the environment. Both architectures learn preferences over actions, states, and goals, but they encode these as logical rules, in contrast with ICARUS' use of utility functions. The ACT-R architecture (Anderson, 1993) associates strengths with learned production rules based on their success in achieving goals, but these specify a scalar value rather than a numeric function of environmental features.

An alternative framework learns control policies from observations of another agent's behavior by transforming traces into supervised training cases. Such behavioral cloning typically generates knowledge in the form of decision trees or logical rules (e.g., Sammut, 1996; Urbancic & Bratko, 1994), though other encodings are possible (Anderson, Draper, & Peterson, 2000). Unlike ICARUS, these systems typically acquire control knowledge from scratch, but one could utilize high-level plans to parse a behavioral trace and thus constrain the cloning process. More broadly, the separation hypothesis suggests that behavioral cloning can proceed in two steps, where the first learns the structure of shared skills from observations, and the second learns numeric value functions that characterize individual behavior. Some work on adaptive interfaces can induce such functions from user choices.

A larger body of research emphasizes learning policies from delayed external rewards. Within this framework, some methods represent control knowledge as logical rules that state the conditions under which particular actions are desirable (e.g., Grefenstette, Ramsey, & Schultz, 1990). Others achieve the same effect with different formalisms like multilayer neural networks (e.g., Moriarty & Langley, 1998). In this paradigm, learning involves a search through the space of policies, using genetic or other methods, guided by the rewards that alternative candidates receive from the environment. The search process typically starts from scratch, but, clearly, it could be aided by starting from skills that specify legal actions. However, this framework does not lend itself to a division between legal skills and preferences stated as utility functions.

An alternative approach to learning from delayed rewards encodes policies as utility functions, an idea that plays a central role in ICARUS. These functions are typically stored in a table that associates an estimated value with each state-action pair, but some work instead uses approximations. This mapping is learned through methods like Q learning (Watkins & Dayan, 1992), which update the estimated value of a state-action pair based on the discounted reward that it produces. Most research on estimating value functions in this manner emphasizes learning from scratch, though some work on hierarchical reinforcement learning (e.g., Andre & Russell, 2000; Dietterich, 2000; Parr & Russell, 1998; Sutton et al., 1998) provides the learner with background knowledge. Our approach fits comfortably within this framework, but the notion of programming by reward distinguishes it from these efforts.

In summary, our approach to representing, using and learning control policies has many common features with other work on this topic. However, ICARUS differs from previous systems in its clear separation of control knowledge into logical skills and numeric utility functions, which we claim supports considerable variety in agent behavior while keeping it within domain constraints. This division in turn lets us program agents by reward to exhibit quite different behaviors.

5. Conclusions

Our experiments have shown that we can produce qualitatively distinct agents via programming by reward. That is, we can construct one set of skills, define individual agents by encoding suitable reward functions, and train those agents by letting them learn from experience. The reward functions are easy to construct and their content is unconstrained.

The experimental results provide evidence in support of the separation hypothesis in the context of our traffic domain. If it holds more generally, then skills and value-encoded preferences may be decoupled sufficiently to enable programming by reward in practical

applications. If so, then we can create entire families of agents in these domains without writing new skills. This is important because skill development is time consuming and difficult work.

Although this paper emphasized the use of reward functions in a programming metaphor, we also designed a reward function to accomplish a specific objective. This required some search, but more direct methods are possible. In particular, we have shown elsewhere (Shapiro, 2001) that one can always align an agent's reward function with human utility, such that the agent will do the best job possible for that person as a byproduct of learning to maximize its own reward. This is an open area for future research.

In summary, the ICARUS architecture and the methodology of programming by reward appear to provide an efficient means of encoding a diverse range of desired behaviors. The approach merits an in-depth examination in a variety of domains, including conversational agents, characters in computer games, and household robots whose personalities are tailored to their owners.

Acknowledgements

This research was carried out at the DaimlerChrysler Research and Technology Center. We thank Simon Handley, David Moriarty, and Mark Pendrith for developing the traffic simulator, and Ross Shachter for his many contributions to the overall framework.

References

Anderson, C., Draper, B., & Peterson, D. (2000). Behavioral cloning of student pilots with modular neural networks. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 25–32). Stanford: Morgan Kaufmann.

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

Andre, D., & Russell, S. J. (2001). Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems 13* (pp. 1019–1025). Cambridge, MA: MIT Press.

Dietterich, T. G. (2000). State abstraction in MAXQ hierarchical reinforcement learning. *Advances in Neural Information Processing Systems 12* (pp. 994–1000). Cambridge, MA: MIT Press.

Grefenstette, J. J., Ramsey, C. L., & Schultz, A. C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning*, *5*, 355–381.

Laird, J. E., & Rosenbloom, P. S. (1990). Integrating execution, planning, and learning in soar for external environments. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 1022–1029). Boston, MA: AAAI Press.

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

Moriarty, D., & Langley, P. (1998). Learning cooperative lane selection strategies for highways. *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (pp. 684–691). AAAI Press.

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). Cambirdge, MA: MIT Press.

Sammut, C. (1996). Automatic construction of reactive control systems using symbolic machine learning. *Knowledge Engineering Review*, *11*, 27–42.

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.

Shapiro, D. (2001). *Value-driven agents*. Doctoral dissertation, Department of Management Science and Engineering, Stanford University, Stanford, CA.

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.

Singh, S., Jaakola, T., Littman, M. L., & Szepesvari, C. (2000). Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*, *38*, 287–308.

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). Madison, WI: Morgan Kaufmann.

Urbancic, T., & Bratko, I. (1994). Reconstructing human skill with machine learning. *Proceedings of the Eleventh European Conference on Artificial Intelligence* (pp. 498–502). Amsterdam: John Wiley.

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