Learning Hierarchical Skills from Observation

Ryutaro ICHISE,[1,2] Daniel SHAPIRO,[1] and Pat LANGLEY[1]

[1] Center for the Study of Language and Information
Stanford University, Stanford CA 94305-4115, USA
[2] National Institute of Informatics, Tokyo 101-8430, Japan
{ichise,dgs,langley}@csli.stanford.edu

Abstract. This paper addresses the problem of learning control skills from observation. In particular, we show how to infer a hierarchical, reactive program that reproduces and explains the observed actions of other agents, specifically the elements that are shared across multiple individuals. We infer these programs using a three-stage process that learns flat unordered rules, combines these rules into a classification hierarchy, and finally translates this structure into a hierarchical reactive program. The resulting program is concise and easy to understand, making it possible to view program induction as a practical technique for knowledge acquisition.

1 Introduction

Physical agents like humans not only execute complex skills but also improve their ability over time. The past decade has seen considerable progress on computational methods for learning such skills and control policies from experience. Much of this research has focused on learning through trial and error exploration, but some has addressed learning by observing behavior of another agent on the task. In particular, research on *behavioral cloning* (e.g., Sammut, 1996) has shown the ability to learn reactive skills through observation on challenging control problems like flying a plane and driving an automobile.

Although such methods can produce policies that predict accurately the desirable control actions, they ignore the fact that complex human skills are often have a hierarchical organization. This structure make the skills both more understandable and more transferable to other tasks. In this paper, we present a new approach to learning reactive skills from observation that addresses the issue of inferring their hierarchical structure. We start by specifying the learning task, including the training data and target representation, then present a method for learning hierarchical skills. After this, we report an experimental evaluation of our method that examines the accuracy of the learned program and its similarity against the program that generated the training cases. In closing, we discuss related work and directions for future research on this topic.

2 The Task of Learning Hierarchical Skills

We can state the basic problem of learning control policies in terms of inputs and outputs:

– Given: a trace of agent behavior containing feature-action pairs
– Find: a program that generates the same actions when presented with the same features.

Research on behavioral cloning (e.g., Anderson et al., 2000; Sammut, 1996) has already addressed this task, having developed methods that learn reactive skills from observation that are both accurate and comprehensible. However, complex skills can often be decomposed naturally into subproblems, and here we focus on capturing this hierarchical structure in an effort to produce even more concise and understandable policies.

As an additional constraint, we adopt the hypothesis that differences in individual behavior (for common tasks) are due to the action of distinct preferences over the same set of skills. In other words, we all know how to drive, but our preferences distinguish safe from reckless drivers. This assumption simplifies the task of program acquisition because it implies that we should learn a nondeterministic mapping from the observed situation to a feasible set of actions, instead of aiming for a deterministic characterization of a single agent's behavior. The resulting program will represent fewer distinctions, and thus will be easier to understand.

2.1 Nature of the Training Data

We assume that the learner observes traces of another agent's behavior as it executes skills on some control task. As in earlier work on learning skills from observation, these traces consist of a sequence of environmental situations and the action(s) the agent carried out in each case. Because we are concerned with learning reactive skills, we ignore the order in which situations occur and transform them into an unordered set of training cases, one for each situation.

Traditional work in behavioral cloning turns an observational trace into training cases for supervised learning, treating each possible action as a class value. Because we are concerned with learning flexible skills, we instead find sets of actions that occur in the same environmental situation, then generate training cases that treat each observed action set as a class value. This lets us employ standard methods for supervised induction to partition situations into reactive but nondeterministic control policies.

2.2 Nature of the Learned Skills

We assume that learned skills are stated in Icarus (Shapiro, 2001), a hierarchical reactive language for specifying the behavior of physical that encodes contingent mappings from situations to actions. Like other languages of this kind

(Brooks, 1986; Firby, 1989; Georgeff et al., 1985), Icarus interprets programs in a repetitive sense-think-act loop that lets an agent retrieve a relevant action even if the world changes from one cycle of the interpreter to the next. Icarus 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, Icarus supports function call, Prolog-like parameter passing, pattern matching on facts, and recursion.

An Icarus plan contains up to three elements: an objective, a set of requirements (or preconditions), and a set of alternate means for accomplishing the objective. 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 so, evaluation progresses to the means field, which contains alternate methods (subplans or primitive actions) for accomplishing the objective. The means field is the locus of all value-based choice in Icarus. The system learns to select the alternative that promises the largest expected reward.

2.3 A Sample Plan for Driving

Table 1 presents an Icarus plan for freeway driving. The top-level routine, Drive, contains an ordered set of objectives implemented as further subplans. Icarus repetitively evaluates this program, starting with its first clause every execution cycle. As the interpreter encounters sensor tests it selectively descends the calling tree, ultimately locating one or more relevant actions. As the calling stack unwinds, Icarus returns the best action found in each subplan. The action returned from the top-level plan is passed to an external execution system, which applies it in the world. Thus, the purpose of an Icarus program is to find action.

For example, the first clause in Drive shown in Table 1 defines a reaction to an impending collision. If this context applies, Icarus returns the Slam-on-brakes action for application in the world. However, if Emergency-brake is not required, evaluation proceeds to the second clause, which encodes a reaction to trouble ahead, defined as a car traveling slower than the agent in the agent's own lane. This subplan contains multiple options. It lets the agent move one lane to the left, move right, slow down, or cruise at its current speed. Icarus makes a selection based on the long-term expected reward of each alternative.

At each successive iteration, Icarus can return an action from an entirely different portion of Drive. For example, the agent might slam on the brakes on cycle 1, and speed up in service of Get-to-target-speed (a goal-driven plan) on cycle 2. However, if Emergency-brake and Avoid-trouble-ahead do not apply, and the agent is already at its target speed, Icarus might return the Change-right action in service of Avoid-trouble-behind on cycle 3.

Table 1. The Icarus program for freeway driving.

```
Drive()
   :objective
    [*not*(Emergency-brake())
     *not*(Avoid-trouble-ahead())
     Get-to-target-speed()
     *not*(Avoid-trouble-behind())
     Cruise()]
Emergency-brake()
   :requires [Time-to-impact() <=  2]
   :means    [Slam-on-brakes()]
Avoid-trouble-ahead ()
   :requires
    [?c = Car-ahead-center()
     Velocity() > Velocity(?c)]
   :means
    [Safe-cruise()
     Safe-slow-down()
     Safe-change-left()
     Safe-change-right()]
Get-to-target-speed()
   :objective
    [Near(Velocity(), Target-speed())]
   :means
    [Adjust-speed-if-lane-clear()
     Adjust-speed-if-car-in-front()]
Avoid-trouble-behind()
   :requires ;;faster car behind
    [?c = Car-behind-center()
     Velocity(?c) > Velocity()]
   :means
    [Safe-cruise()
     Safe-change-right()]
Safe-cruise()
   :requires [Time-to-impact() > 2]
   :means [Cruise()]
Safe-slow-down()
   :requires [Time-to-impact(-2) > 2]
   :means [Slow-down()]
Safe-speed-up()
   :requires [Time-to-impact(2) > 2]
   :means [Speed-up()]
Safe-change-left()
   :requires [Clear-left()]
   :means [Change-left()]
Safe-change-right()
   :requires [Clear-right()]
   :means [Change-right()]
Adjust-speed-if-lane-clear()
   :requires [*not*(Car-ahead-center())]
   :means
    [Slow-down-if-too-fast()
     Speed-up-if-too-slow()]
Adjust-speed-if-car-in-front()
   :requires
    [Car-ahead-center()
     *not*(Slow-down-if-too-fast())]
   :means
    [Speed-up-if-too-slow()
     Safe-cruise()
     Safe-slow-down()]
Slow-down-if-too-fast()
   :requires [Velocity() > Target-speed()]
   :means [Safe-slow-down()]
Speed-up-if-too-slow()
   :requires
    [Velocity() <= Target-speed()]
   :means [Safe-speed-up()]
Slam-on-brakes()
    :action [match-speed-ahead()]
```

The remainder of the program follows a similar logic as the interpreter considers each clause of Drive in turn. If a clause returns True, the system advances to the next term. If it returns False, Drive would exit with False as its value. However, Icarus supports a third option: a clause can return an action, which becomes the return value of the enclosing plan. For example, Avoid-trouble-behind might return Change-right, which would become the return value of Drive (the final clause, Cruise, would not be evaluated that cycle).

3 A Method for Learning Hierarchical Skills

Now that we have defined our task, we can describe our method for learning hierarchical skills from behavior traces. The approach involves three distinct stages. The first induces unordered flat rules using a standard supervised learning technique that induces If-Then rules, each of which predicts an action sets for a class of situations. To this end, we employ CN2 (Clark & Boswell, 1991) to generate a set of unordered production rules that determine the target class

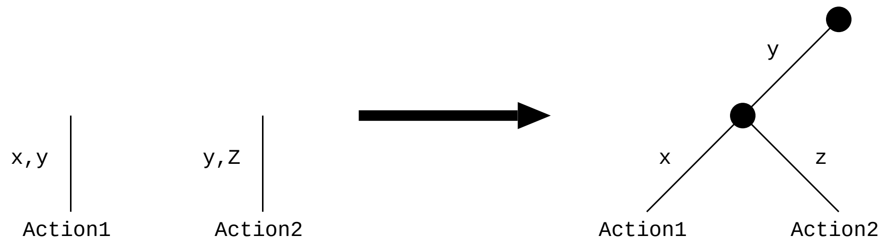

Fig. 1. Operator for promoting conditions.

from attribute values. The second stage creates a classification hierarchy by combining tests that appear in multiple rules. When viewed as an action generator, this structure resembles a hierarchical program. The final stage transforms the classification tree into an Icarus program, taking advantage of Icarus' unique, three-valued logic. In this section, we discuss the second and third stages.

3.1 Constructing Hierarchies

The second stage of our approach to program induction generates a classification hierarchy. Our method operates by promoting conditions that appear in multiple rules. Consider the two rules:

- If x and y Then $Action1$
- If y and z Then $Action2$

Since the condition y appears in the both rules, we can promote it by creating a more abstract rule that tests the common precondition, using a technique borrowed from work on grammar induction (e.g., Langley & Stromsten, 2000). We illustrate this transformation in Figure 1. Here, the labels on arcs denote conditional tests, and the leaf nodes denote actions. The black circles indicate choice points, where one (or more) of the subsequent tests apply. These structures are interpreted from the top downwards. For example, the right side of Figure 1 classifies the current situation first by testing y, and then, if y holds, by testing x and z (in parallel) to determine which action or actions apply. This results in a more efficient classification process; y is only tested once and, if it does not hold, there is no reason to test x or z. This structure is similar to the decision trees output by C4.5, but more general in that it allows non-exclusive choice.

In addition to promoting conditions, we can promote actions within a classification hierarchy. Figure 2 provides a simple example, where the Action2 occurs at all leaf nodes within a given subtree. If the system is guaranteed to reach at least one of the leaf nodes[3] we can associate Action2 with the root node of the subtree. We represent such nodes with a hollow circle. This simplification applies even if the leaf nodes are at an arbitrary depth beneath the root of the subtree.

[3] Here we mean that the tests in the subtree form a collectively exhaustive set.

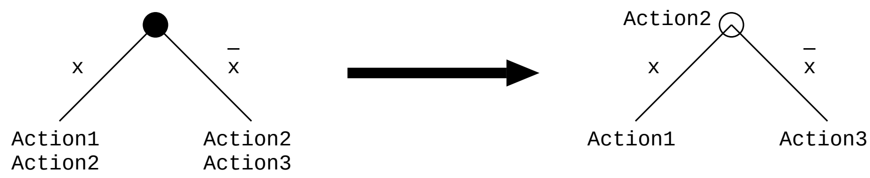

Fig. 2. Operator for promoting actions.

Condition promotion transforms the flat rules learned by CN2 into a classification hierarchy. However, since there are many possible ways to combine rules by promoting conditions, we have an opportunity to shape the final classification hierarchy by defining rule-selection heuristics. (Note that this degree of control would not be available if we had used decision trees instead instead of rules.) The key idea is to merge rules with similar actions. In particular, we identify three heuristics that seem to produce well-structured trees and understandable programs (after the final transformation of our learning process):

1. Select rules with same action or same set of actions.
2. Select rules with subset relations among the actions.
3. Select rules with the same conditions.

Our algorithm considers these heuristics in priority order. If two rules select the same action class, they are the highest priority candidates for condition promotion. The operation will only be successful, of course, if the rules share conditions. If more than two rules select the same action class, the ones that share the largest number of conditions will be combined. The second heuristic applies if no two rules select the same action class. In this case, the algorithm looks for rules whose action sets bear a subset relation to one another, such as "Speed-up" and "Speed-up, Change-right". If a single rule enters into many such pairings, the system takes the ones with the *smallest* number of actions on the theory that these rules express the most cohesive intent. Ties are broken by a similarity metric that maximizes the number of shared and thus promotable conditions. Finally, if no action sets bear subset relations, the system picks rules that share the largest number of conditions. The combination of any two rules yields a subtree with shared conditions on its top-level arc, and these conditions can enter into further promotion operations.[4] The remaining conditions cannot be merged with any other rules. This process of rule selection and combination continues to exhaustion, merging top-level conditions to build multi-layered subtrees.

Some simple examples may help to clarify this algorithm. Consider the following three rules (whose abbreviations defined in Table 2):

[4] For the purpose of the rule-selection heuristics, the action set of a subtree is the union of the action sets in its leaf nodes, while the most similar subtrees share the largest number top-level conditions.

Table 2. Notation used in example rules and hierarchies.

actions		conditions	
abbreviation	meaning	abbreviation	meaning
CRU	Cruise	CAC	Car Ahead Center
SLO	Slow Down	CBC	Car Behind Center
SPE	Speed Up	CLR	Clear Right
MAT	Match Speed Ahead	CLL	Clear Left
CHR	Change Right	TTIA	Time To Impact Ahead
CHL	Change Left	TTIB	Time To Impact Behind
		VEL	Velocity

IF $TTIA < 52.18$
AND $TTIA > 1.82$
AND $CLR = True$
AND $CLL = False$
THEN Action =
CHR, CHL, CRU, SLO

IF $TTIA < 52.18$
AND $TTIA > 1.82$
AND $CLR = True$
AND $CLL = False$
THEN Action =
CHR, CRU, SLO

IF $TTIA < 52.18$
AND $TTIA > 1.82$
AND $CLR = False$
AND $CLL = False$
THEN Action =
CRU, SLO

Although no two rules select identical actions sets, all three action sets bear subset relations. In this case, the algorithm will select the last two rules because their action sets are the smallest, and promote three conditions to obtain a new shared structure. Two of those conditions can be combined with the rule for CHR,CHL,CRU,SLO, yielding a three level subtree representing all three rules.

When the process of condition promotion terminates, we add a top-level node to represent the choice among subtrees. Then, we simplify the structure using the action promotion rule shown in Figure 2. When we apply this algorithm to the entire set of observational data, we obtain the classification tree in Figure 3.

3.2 Constructing the Icarus Program

We translate this classification structure into an Icarus program by the use of a few simple rules. We performed this transformation by hand, although the process could be automated. The key idea is to recognize that Figure 3 represents a mutually exclusive and collectively exhaustive classification hierarchy, and thus the branches can be ordered without loss of generality.

We number the branches in Figure 3 in increasing order from left to right, and note that conditions which let a branch return ANY action must not hold in a later branch. When we take these branches in the order 4,5,3,1,2, we can immediately eliminate $TTIA > 1.82$ from branch 5, and $TTIA > 52.18$ from branches 3 and 1. Now consider branch 3. Without the clause $TTIA > 52.18$, we can discard the strongest requirement on VEL and eliminate one of the leaf nodes. This yields a simplified rule:

```
CAC=T and 56.5>VEL => CRU SLO SPE
```

Now compare the revised expression for branch 3 with the rule in branch 2:

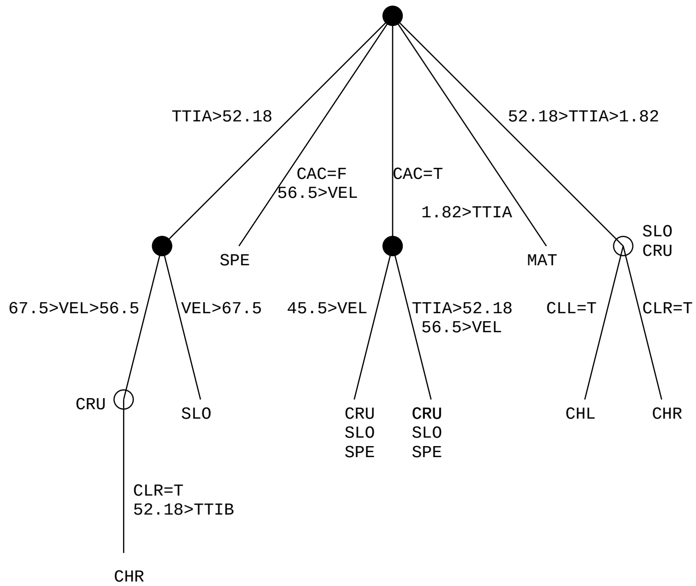

Fig. 3. The classification tree obtained by our approach.

```
CAC = N and 56.5>VEL => SPE
```

Here, the predicate CAC is irrelevant to SPE, and we can merge both rules into a single branch representing the statements:

```
56.5>VEL => SPE
56.5>VEL and CAC=T => CRU SLO
```

We use this expression to simplify branch 1, since VEL must be greater than 56.5 if execution reaches that point. This leaves a single choice point on VEL in branch 1, which we can simplify by ordering the subtrees from right to left. This results in the ordered rules:

```
VEL > 67.5 => SLO
CRU or  (CLR = T and 52.18 > TTIB) => CHR
```

Taken together, these changes produce the Icarus program in Table 3. At this point, our method halts, having induced a hierarchical control policy from observational traces.

4 Experimental Evaluation

Although our approach to learning hierarchical skills seems quite plausible, we wanted to evaluate its behavior experimentally. We chose the driving task for this purpose, both because we already had considerable experience with the domain and because we had developed manually a hierarchical driving policy.

Table 3. The ICARUS program induced by our method.

```
Drive ()                                  R3 ()
    :requires [NOT(R1)                        :requires [VEL < 56.5]
               NOT(R2)                        :means [SPE R31]
               NOT(R3)
               NOT(R4)]                   R31 ()
R1 ()                                         :requires [CAC = True]
    :requires [TTIA < 1.82]                   :means [SLO CRU]
    :means [MAT]
                                          R4 ()
R2 ()                                         :requires [NOT(R41)]
    :requires [TTIA < 52.18]                  :means [CRU R42]
    :means [SLO CRU R21 R22]
                                          R41 ()
R21 ()                                        :requires [VEL > 67.5]
    :requires [CLL = True]                    :means [SLO]
    :means [CHL]
                                          R42 ()
R22 ()                                        :requires [CLR = True
    :requires [CLR = True]                               TTIB < 52.18]
    :means [CHR]                              :means [CHR]
```

4.1 Data Generation

We utilized the ICARUS program in Table 1 to generate observational traces for use in training and testing. We instrumented the program and recorded situation-action pairs, using all situation features, not just those tested en route to action. Since our goal was to recover the structure of a shared driving skill, we wanted traces from multiple drivers whose preferences would collectively span all options encoded in the Drive plan. Instead of creating these agents, we took the simpler approach of directly exercising every control path in the ICARUS program. This produced a list of situation-action tuples that included every possible action response.

Specifically, we enumerated five values of in-lane separation (both to the car ahead and behind), five values of velocity for each of the three in-lane cars, and the status of the adjacent lane (whether it was clear or not clear). We chose the particular distance and velocity numbers to produce True and False values for the relevant predicates in the driving program (e.g., time to impact ahead, velocity relative to target speed). This procedure also created multiple occurrences of many situation-action tuples (i.e., the mapping from distance and velocity onto time to impact was many-one).

The resulting data had nine attributes. Four of these were Boolean, representing the presence or absence of a car in front/back, and whether the lanes to the right or left of the agent are clear. The rest were numerical attributes. Two of these represented time to impact with the car ahead or behind, two encoded relative velocity ahead or behind, and the last measured the agent's own velocity.

Our formulation of the driving task assumes six primitive actions. We preprocessed the data to identify sets of these actions that occurred under the same situation. We obtained ten such sets, each containing one to four primitive actions. These classes define a mutually exclusive and collectively exhaustive set of responses for our induced program. The resulting data set included 4600 such situation and action-set tuples.

4.2 Experimental Results

We evaluated our learning method in several ways. First, we measured the accuracy of the learned program by employing a standard cross-validation technique to determine how much of the original behavior we were able to recover. In addition, we examined the conciseness of the hierarchical Icarus program induced by our method relative the flat rules produced by CN2. Finally, we evaluated the structure of the learned Icarus program in a more subjective sense, by comparing it against the original Icarus program that generated the data.

We measured the accuracy of the learned program by conducting a 10-fold cross validation. The results showed that for each training set, our method induced a program that had 100% accuracy on the corresponding test set. Moreover, even though the rules induced by CN2 were slightly different across the training runs, the resulting classification hierarchies were identical to the tree in Figure 3. So, our heuristics for rule combination regularized the structure.

We also compared the number of conditions that must be evaluated to select action in the flat rules and the Icarus program. This provides a measure of the computational efficiency of the two representations. The flat rules required an average of 7361 evaluations to process the training data, while the learned Icarus program employed 2216. Thus, the hierarchical representation requires only 30% of the effort.

When we compare the learned Icarus program in Table 3 with the original program in Table 1 several interesting features emerge. First, the learned program is simpler. It employs 10 Icarus functions, whereas the original program required 14. This was quite surprising, especially since the original code was written by an expert Icarus programmer. Next, the learned program captures a good deal of the natural structure of the driving task. This is evidenced by the fact that the top level routines call roughly the same number of functions, and half of those implement identical reactions. To be specific, R1 in Table 3 correspond to Emergency-brake in Table 1, while R2 represents Avoid-trouble-ahead using a simpler gating condition. Similarly, R4 captures all of the behavior of Avoid-trouble-behind, although it adds the Slow-down operation found in Get-to-target-speed. R3 represents the remainder of Get-to-target-speed, absent the Slow-down action. The system repackaged these responses in a slightly more efficient way. The only feature truly missing from the learned program is the idea that maintaining target speed is an objective of the original plan. This reflects a limitation of our current induction technique.

5 Related Work on Control Learning

We have already mentioned in passing some related work on learning control policies, but the previous research on this topic deserves more detailed discussion. The largest body of work focuses on learning from delayed external rewards. Some methods (e.g., Moriarty et al., 1999) carry out direct search through the space of policies, whereas others (e.g., Kaelbling et al., 1996) estimate value functions for state-action pairs. Research in both paradigms emphasizes exploration and learning from trial and error, whereas our approach addresses learning from observed behaviors of another agent. However, the nondeterministic policies acquired in this fashion can be used to constrain and speed learning from delayed reward, as we have shown elsewhere (Shapiro et al., 2001).

Another framework learns control policies from observed behaviors, but draws heavily on domain knowledge to interpret these traces. This paradigm includes some, but not all, approaches to explanation-based learning (e.g., Segre, 1987), learning apprentices (e.g., Mitchell et al., 1985), and programming by demonstration (e.g., Cypher, 1993). The method we have reported for learning from observation relies on less background knowledge than these techniques, and also acquires reactive policies, which are not typically addressed by these paradigms.

Our approach is most closely related to a third framework, known as *behavioral cloning*, that also observes another agent's behavior, transforms traces into supervised training cases, and induces reactive policies. This approach typically casts learned knowledge as decision trees or logical rules (e.g., Sammut, 1996; Urbancic & Bratko, 1994), but other encodings are possible (Anderson et al., 2000; Pomerleau, 1991). In fact, our method's first stage takes exactly this approach, but the second stage borrows ideas from work on grammar induction (e.g., Langley & Stromsten, 2000) to develop simpler and more structured representations of its learned skills.

6 Concluding Remarks

This paper has shown that it is possible to learn from a trace of an agent's behavior an accurate and well-structured program that is easy for a person to understand. Our approach extends behavioral cloning techniques, and our results illustrate that such methods can produce simpler control programs hierarchical structure with no loss in predictive accuracy. Moreover, its emphasis on learning the shared components holds promise for increased generality of the resulting programs.

Our technique employed several heuristics for learning hierarchical structures that provided a substantial source of inductive power. In particular, the attempt to combine rules for similar action sets tended to group rules by purpose, while the operation of promoting conditions tended to isolate special cases. Both techniques led to simpler control programs and, presumably, more understandable encodings of reactive policies.

We hope to develop these ideas further in future work. For example, we will address the problem of inferring Icarus objective clauses, which is equivalent to learning teleological structure from observed behavior. We also plan to conduct experiments in other problem domains, starting with traces obtained from simulations and/or human behavior. Finally, we intend to automate the process of transforming classification hierarchies into Icarus programs. This will let us search for criteria that generate the most understandable or aesthetic skills.

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.

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

Clark, P., Boswell, R. (1991). Rule induction with CN2: Some recent improvements. *Proceedings of the European Working Session on Learning : Machine Learning*, LNAI, 482, 151–163.

Cypher, A. (Ed.). (1993). *Watch what I do: Programming by demonstration*. Cambridge, MA: MIT Press.

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

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

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

Langley, P., & Stromsten, S. (2000). Learning context-free grammars with a simplicity bias. *Proceedings of the Eleventh European Conference on Machine Learning* (pp. 220–228). Barcelona: Springer-Verlag.

Mitchell, T. M., Mahadevan, S., & Steinberg, L. (1985). Leap: A learning apprentice for VLSI design. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, (pp. 573-580). Los Angeles, CA: Morgan Kaufmann.

Moriarty, D. E., Schultz, A. C., & Grefenstette, J. J., (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, *11*, 241–276.

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

Pomerleau, D. (1991). Rapidly adapting artificial neural networks for autonomous navigation. *Advances in Neural Information Processing Systems 3* (pp. 429–435). San Francisco: Morgan Kaufmann.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

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.

Segre, A. (1987). A learning apprentice system for mechanical assembly. *Proceedings of the Third IEEE Conference on AI for Applications* (pp. 112–117).

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.

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

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.