Clay: Integrating Motor Schemas and Reinforcement Learning

Tucker Balch

Mobile Robot Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
tucker@cc.gatech.edu

GIT-CC-97-11

Abstract

Clay is an evolutionary architecture for autonomous robots that integrates motor schema-based control and reinforcement learning. Robots utilizing Clay benefit from the real-time performance of motor schemas in continuous and dynamic environments while taking advantage of adaptive reinforcement learning. Clay coordinates assemblages (groups of motor schemas) using embedded reinforcement learning modules. The coordination modules activate specific assemblages based on the presently perceived situation. Learning occurs as the robot selects assemblages and samples a reinforcement signal over time. Experiments in a robot soccer simulation illustrate the performance and utility of the system.

1 Background and Related Work

1.1 Motor Schemas

Motor schemas are an important example of behavior-based robot control. The motor schema paradigm is the central method in use at the Georgia Tech Mobile Robot Laboratory, and is the platform for this research.

Motor schemas are the reactive component of Arkin's Autonomous Robot Architecture (AuRA) [2]. AuRA's design integrates deliberative planning at a top level with behavior-based motor control at the bottom. The lower levels, concerned with executing the reactive behaviors are incorporated in this research.

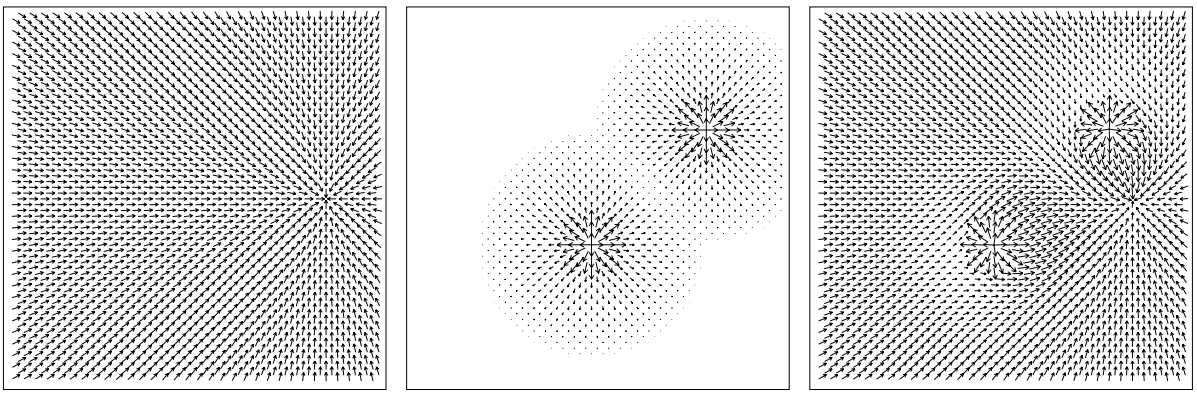


Figure 1: Motor schema example. The diagram on the left shows a vector field corresponding to a **move-to-goal** schema, pulling the robot to a location on the right. The center diagram shows an **avoid-obstacles** field, repelling the robot from two sensed obstacles. On the right, the two schemas are summed, resulting in a complete behavior for reaching the goal. It is important to note that the entire field is never computed, only the vectors for the robot's current location.

Individual motor schemas, or primitive behaviors, express separate goals or constraints for a task [1]. As an example, important schemas for a navigational task would include **avoid_obstacles** and **move_to_goal**. Since schemas are independent, they can run concurrently, providing parallelism and speed. Sensor input is processed by perceptual schemas embedded in the motor behaviors. Perceptual processing is minimal and provides just the information pertinent to the motor schema. For instance, a **find_obstacles** perceptual schema which provides a list of sensed obstacles is embedded in the **avoid_obstacles** motor schema.

The concurrently running motor schemas are integrated as follows: First, each produces a vector indicating the direction the robot should move to satisfy that schema's goal or constraint. The magnitude of the vector indicates the importance of achieving it. It is not so critical, for instance, to avoid an obstacle if it is distant, but crucial if close by. The magnitude of the **avoid_obstacle** vector is correspondingly small for distant obstacles and large for close ones. The importance of motor schemas relative to each other is indicated by a gain value for each one. The gain is usually set by a human designer, but may also be determined through automatic means, including on-line learning [5], case-based reasoning [17] or genetic algorithms [16]. Each motor vector is multiplied by the associated gain value and the results are summed and normalized. The resultant vector is sent to the robot hardware for execution. An example of this process is illustrated in Figure 1.

The approach bears a strong resemblance to potential field methods [6, 9, 8], but with an important difference: the entire field is never computed, only the robot's reaction to its current perception of the world at its present location. In the example figure an entire field is shown, but this is only for visualization purposes. Problems with local minima, maxima, and cyclic behavior which are endemic to many potential fields strategies are handled by several methods including: the injection of noise into the system [1]; resorting to high-level planning; repulsion from previously visited locales [4]; continuous adaptation [5]; and other learning strategies [16, 17]. Schema-based robot control has been demonstrated to provide robust navigation in complex and dynamic worlds.

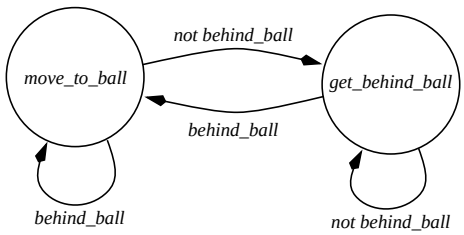
1.2 Temporal Sequencing

As illustrated above for navigation, motor schemas may be grouped to form more complex, emergent behaviors. Groups of behaviors are referred to as *behavioral assemblages*. One way behavioral assemblages may be used in solving complex tasks is to develop an assemblage for each sub-task and to execute the assemblages in an appropriate sequence. The steps in the sequence are separate *behavioral states*. Perceptual events that cause transitions from one behavioral state to another are called *perceptual triggers*. The resulting task-solving strategy can be represented as a Finite State Automaton (FSA). The technique is referred to as *temporal sequencing* [3].

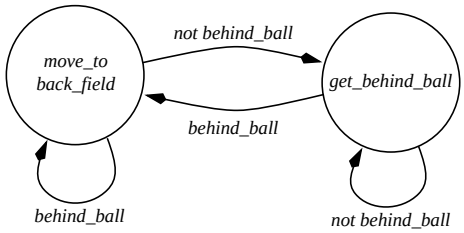
As an example use of temporal sequencing, consider the strategy for a robot soccer player. The salient issue for now is that points are scored by bumping the ball across the opponent's goal. Robots must avoid bumping the ball in the wrong direction, lest they score against their own team. A reasonable approach is for the robot to first ensure it is behind the ball, then move towards it to bump it towards the opponent's goal. Alternately, a goalie robot may remain in the backfield to block an opponent's scoring attempt.

A robot can be in one of three behavioral states: *move_to_ball*, *get_behind_ball*, and *move_to_backfield*. The robot is initialized in the *get_behind_ball* state. If it detects that it is behind the ball it immediately transitions to the *move_to_ball* or *move_to_backfield* state, depending on whether it is serving as a "forward" or "goalie." The transition occurs on the trigger *behind_ball*. The robot will remain in the new state until triggered again by *not behind_ball*.

At the highest level, the soccer strategy is an assemblage represented as a finite state automaton (FSA) consisting of two states. FSAs illustrating the forward and goalie strategies are shown in Figure 2. The robot's policy may be equivalently viewed as a look-up table (Figure 3). This paper will focus on the look-up table representation as it is also useful for viewing the policies of learning robots. Note, however, that the FSA is potentially a more powerful representation since it implicitly provides state while look-up tables do not.



Control Team Forward



Control Team Goalie

Figure 2: A soccer team's strategy viewed as FSAs. This strategy is used as the control in experiments using Clay.

1.3 Learning in Behavior-based Systems

Even though behavior-based approaches are robust for many tasks and environments, they are not necessarily adaptive. We now consider some of the ways learning can be integrated into a behavior-based system.

Reinforcement learning offers a powerful set of techniques that allow a robot to learn a task without requiring its designer to fully specify how it should be carried out. If the task is feasible, and feedback regarding how well the agent is doing is provided, several reinforcement learning techniques are *guaranteed* to converge (within an arbitrary ϵ) to the *optimal* solution [20, 19]. The guarantees are tempered by rather strong conditions for convergence; Q-learning for example, requires all actions to be repeatedly sampled in all states. The reader is referred to Kaelbling [7] for an excellent survey of reinforcement learning in robotics.

Reinforcement learning methods fall into two broad groups: model-based and model-free. Model-free systems like Q-learning are computationally simple, but require many experience steps to converge. Model-based systems like Dyna [18] seek to reduce the cost of experience in the real-world (as in risk of damage

perceptual feature	assemblage *mtb*	*gbb*	*mtb f*
not behind_ball	0	1	0
behind_ball	1	0	0

Control Team Forward

perceptual feature	assemblage *mtb*	*gbb*	*mtb f*
not behind_ball	0	1	0
behind_ball	0	0	1

Control Team Goalie

Figure 3: The control team's strategy viewed as look-up tables. The 1 in each row indicates the behavioral assemblage selected by the robot for the perceived situation indicated on the left. The abbreviations for the assemblages are introduced in the text.

to the robot) by using experience to model interaction with the world, then developing a policy based on the model. Most learning research in the behavior-based robotics community to date has focused on model-free methods. The initial implementation of Clay utilizes a form of Q-learning as a coordination operator, but there is no reason other types of learning should not be added. In fact, future research will investigate the utility of alternate learning operators.

Q-learning is a type of reinforcement-learning in which the value of taking each possible action in each situation is represented as a utility function, $Q(s,a)$. Where s the state or situation and a is a possible action[1]. If the function is properly computed, an agent can act optimally simply by looking up the best-valued action for any situation. The problem is to find the $Q(s,a)$ that provides an optimal policy. Watkins [20] has developed an algorithm for determining $Q(s,a)$ that converges to optimal. He prefers to represent $Q(s,a)$ as a table, $Q[s,a]$, and asserts in [20] that the algorithm is not guaranteed to converge otherwise. The following scheme updates Q-values as the agent interacts with the environment and receives rewards:

$$Q[s_t,a_t] \;=\; (1-\alpha)\underbrace{Q[s_t,a_t]}_{\text{old value}} + \alpha\underbrace{\left(R(s_{t+1},a_{t+1}) + \gamma \max_u Q[s_{t+1},u]\right)}_{\text{improved estimate}} \quad (1)$$

This update is applied each time action a_t is selected in state s_t. The first

[1]It is important to distinguish between s and *behavioral state*. Behavioral state is another way of referring to which behavioral assemblage (group of motor schemas) is active. Conversely, s refers to the *environmental state* or situation the robot is in. For the purposes of discussing Q-learning, we will assume the robot's sensors pass the world state on to the agent unmodified.

term is the old Q-value, while the second is an improved estimate based on an actual reward R and the estimated value of the subsequent state. α is a learning rate that indicates how much "trust" should be given the improved estimate. γ is the rate at which future rewards are discounted.

Q-learning then, is one way for a robot to learn appropriate sequences of action to attain a goal. Mahadevan and Connell [13] have applied it in a slightly different manner: to learn the component behaviors within a pre-defined sequence. The particular task they investigate is for a robot to find, then push a box across a room. They pre-define three behavioral states **F, P** and **U** for find-box, push-box and unwedge-box respectively; they also define conditions under which the robot transitions from one state to another. Separate reinforcement functions and tables of Q-values apply for each state. The state vector s is composed of local sonar occupancy information, infra-red bump sensors, and a "stuck" sensor. The possible actions are: go forward, turn left, turn hard left, turn right, and turn hard right. Since the state space is rather large, Mahadevan sought ways to reduce it, including weighted Hamming distance and statistical clustering to group similar states. Using this approach, their robot, OBELIX was able to learn to perform better than hand-coded behaviors for box-pushing.

In research at Carnegie Mellon University [10], Lin developed a method for Q-learning to be applied hierarchically, so that complex tasks are learned at several levels. He argues that by by decomposing the task into sub-tasks, and learning first at the sub-task level, then the task level, the overall rate of learning is increased compared to monolithic learners.

Similarities between Lin's decomposition and temporal-sequencing for assemblages of motor schemas (Section **??**) are readily apparent. Lin's sub-tasks or elementary skills correspond to behavioral assemblages, while a high-level skill is a sequence of assemblages. Learning at the high-level is equivalent to learning the state-transitions of an FSA (as in Figure 2) and learning the elementary skills corresponds to tuning individual states or behavioral assemblages.

The reinforcement learning approaches outlined so far use a centralized scheme for learning when particular sub-behaviors should be activated. Maes and Brooks [12] propose a alternative, distributed mechanism. In their approach, each behavior learns for itself when it ought to be applied. They predefine a set of behaviors and a set of binary perceptual conditions. Each behavior learns when it should be "on" or "off" based the perceptual conditions. Positive and negative feedback are provided to guide the learning. The approach was demonstrated on a learning robotic hexapod.

Mataric's research has focused on developing specialized reinforcement functions for social learning [14, 15]. The overall reinforcement, $R(t)$, for each robot is composed of separate components, D, O and V. D indicates progress towards the agent's present goal. O provides a reinforcement if the present action is a repetition of another agent's behavior. V is a measure of vicarious reinforcement; it follows the reinforcement provided to other agents. She tested her approach in a foraging task with a group of three robots. Preliminary results

indicate that performance is best when the reinforcement function includes all three components. In fact the robots' behavior did not converge otherwise.

1.4 What's New about Clay

Clay is similar to the approaches outlined above in that it integrates reinforcement learning and behavior-based control, but it differs in several important aspects: First, behavioral expression in Clay is fully recursive: there is no limit to the number of levels in a behavioral hierarchy. Second, Clay's primitive, the motor schema, provides a rich repertoire for behavioral design [2]. Motor schemas take full advantage of continuous sensor values and can generate an infinite range of actuator output; most of the other approaches only select from a discrete list of actions. Finally, while experiments with Clay have so far only explored learning at one level, the designer is free to introduce learning at any level in the behavioral hierarchy.

Georgia Tech's Mobile Robot Laboratory has developed *MissionLab* to support the design and test of sequenced behaviors on robots and in simulation [11]. *MissionLab* includes a set of tools for recursively expressing sequenced behaviors. Behaviors can be designed graphically, using a the cfgedit tool, or textually in the Configuration Network Language (CNL).

Clay draws from CNL but extends it in several important ways. Like CNL, Clay provides for recursive expression of behavior, but it adds learning coordination operators and an object-oriented syntax. The object-oriented approach provides for a direct expression of schema instantiation and the "embedding" of perceptual schemas in motor schemas. For example, the statement:

```
move_to_ball      = new MotorSchemaMoveTo(ball_direction);
```

creates a new instance of the **MoveTo** motor schema using the embedded **ball_direction** perceptual schema. The resulting motor process "**move_to_ball**," will draw the robot towards the perceived ball location.

Finally, since Clay is coded in Java, it's portable. Clay has been demonstrated on the Solaris, Irix (SGI), Windows 95, and Linux operating systems.

2 Expressing Behaviors in Clay

Before moving to a discussion of how Clay integrates motor schemas and learning, it is helpful to show how a basic motor schema-based robotic control system is specified. We begin by outlining the available primitives and coordination operators. The following generic motor schemas are available in Clay:

- **MoveTo**: generates a vector with constant magnitude directly towards a perceptual goal.
- **LinearAttraction**: generates a vector directly towards a perceptual goal with magnitude increasing linearly (up to a maximum of 1.0) with distance from the goal.

- **LinearRepulsion**: generates a vector directly away from a perceived object. The magnitude decreases linearly with distance from the object, falling to zero at the limit of the object' "sphere of influence."
- **Dodge**: generates a "swirling" field around a perceived obstacle. The robot is nudged around it rather than directly repulsed from it.

When instantiated with appropriate embedded perceptual schemas, the generic motor schemas become specific. For instance **MoveTo** serves as **move_to_ball** when instantiated with a ball-finding perceptual schema, or **move_to_goal** when instantiated with a goal-finder. Among others, Clay includes the following perceptual schemas germane to soccer:

- **EgoBall**: provides a vector towards the soccer ball, based on sensor values.
- **DefendedGoal**: returns a vector towards the defended goal.
- **BehindBall**: returns a 1 if the robot is behind the ball, 0 otherwise. This perceptual feature is used to trigger transitions between behavioral states.

Schemas may be combined and coordinated with these operators:

- **CoordinateSum**: multiplies each constituent motor schema or assemblage output by an associated gain value, then sums the results.
- **CoordinateSelection**: selects one motor schema or assemblage for output based on an embedded discrete selector process.

Now we revisit the goalie soccer strategy outlined above to show how perceptual and motor schemas are composed and coordinated in Clay. Recall that the agents are to be provided three behavioral assemblages: *move_to_ball*, *get_behind_ball* and *move_to_backfield*. The assemblages and their primitive components are configured when the robot is initialized. Here is code specifying the it move_to_ball assemblage:

```
ball_direction    = new PerceptSchemaEgoBall(m);
move_to_ball      = new MotorSchemaMoveTo(ball_direction);
```

When it is initialized, the control system configuration routine is passed a handle, `m`, for access to the robot's sensor and actuator interface. The handle is in turn passed to primitive perceptual schemas so they can access the sensor hardware. The first line of code above instantiates a new perceptual schema, **ball_direction**, which provides a vector from the robot to the sensed location of the ball. The second line embeds **ball_direction** in **move_to_ball**, an instantiation of the **MoveTo** motor schema.

Motor and perceptual schemas, once instantiated, are easily reused. This is illustrated in the declaration *move_to_backfield*. *move_to_backfield* is a weighted combination of **move_to_ball** and a new schema, **stay_near_goal**:

```
defended_goal     = new PerceptSchemaDefendedGoal(m);
stay_near_goal    = new MotorSchemaLinearAttraction(1.0, 0.25, defended_goal);

move_to_backfield_schemas[0] = move_to_ball;
move_to_backfield_gains[0]   = 0.5;
move_to_backfield_schemas[1] = stay_near_goal;
move_to_backfield_gains[1]   = 1.5;

move_to_backfield = new CoordinateSum(move_to_backfield_schemas,
                                      move_to_backfield_gains);
```

The first two lines declare **stay_near_goal** as a linear attraction motor schema configured to move the robot towards its defended goal. The parameters 1.0 and 0.25 specify ranges at which the schema's magnitude is maximized and minimized, respectively. Next, the schemas comprising *move_to_backfield* and their gains are specified: **move_to_ball** and **stay_near_goal** are assigned gains of 0.5 and 1.5 respectively. Finally, the component schemas are coordinated by gain multiplication and summation. The *get_behind_ball* is declared similarly.

Once the primitive behaviors have been combined as assemblages, they are coordinated as follows. In the goalie configuration:

```
behind_ball       = new PerceptFeatureBehindBall(m);

assemblages[0]    = get_behind_ball;
assemblages[1]    = move_to_backfield;

top_level         = new CoordinateSelection(assemblages, behind_ball);
```

top_level is the output of a selection operator which chooses between *get_behind_ball* and *move_to_backfield* depending on whether the robot is behind the ball. `assemblages[0]`, or *get_behind_ball*, is selected when `behindball == 0`. `assemblages[1]`, or *move_to_backfield* is selected when `behindball == 1`.

This completes the specification of a goalie robot's behavior. If a designer were interested in building a more complicated agent with soccer as one of several capabilities *top_level* could be included as just another assemblage for integration at the next level up. A graphic, hierarchical representation of the system is illustrated in Figure 4.

A potential difficulty for other hierarchically specified behavioral systems is that as a behavioral configuration grows more complex, run time computational demands can explode exponentially. Clay avoids the problem by only executing currently activated assemblages and schemas. Computational demands are also be reduced when the designer re-uses schemas in a configuration (as *move_to_ball* is re-used above). Synchronization techniques ensure a schema's output is re-calculated only once per movement cycle.

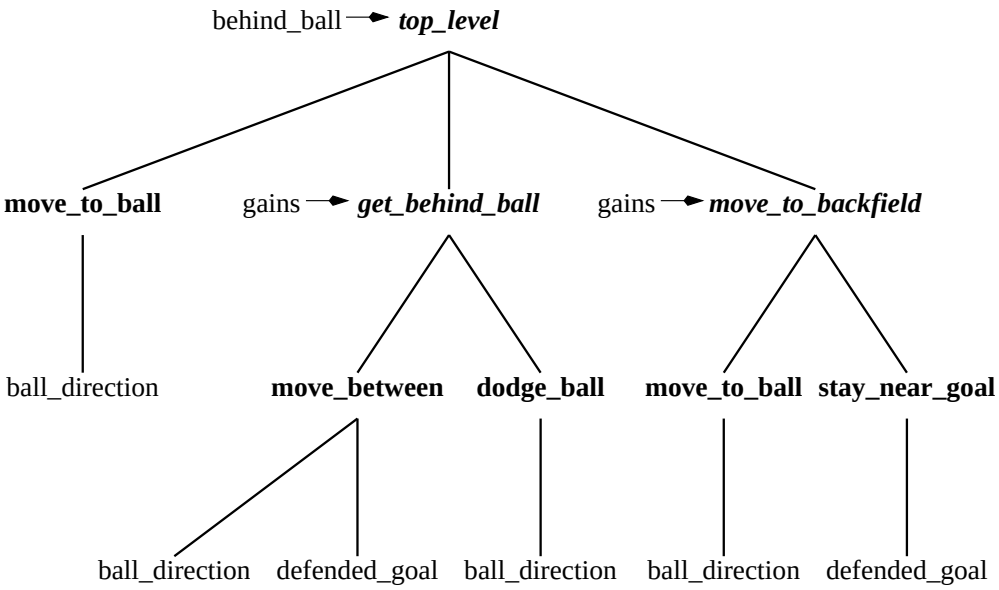


Figure 4: A hierarchical view of the behavioral configuration for soccer robots discussed in the text. Perceptual schemas are shown in plain text. Motor schemas are in **bold**. Assemblages are shown in *italic*.

3 Integrating Reinforcement Learning

Learning is integrated by the addition of a new coordination operator, **CoordinateLearner**. **CoordinateLearner** is "plug compatible" with **CoordinateSelection** but it learns which subordinate assemblage to activate. At configuration time, an instantiation of **CoordinateLearner** is provided an embedded "reward schema" that it uses for learning over time. Here is how a Q-learning soccer robot might be configured:

```
learner          = new LearnerQ(states, actions, alpha, gamma,
                                randomrate, randomdecay);
reward           = new RewardOnScore(m);

assemblages[0]   = move_to_ball;
assemblages[1]   = get_behind_ball;
assemblages[2]   = move_to_backfield;

top_level        = new CoordinateLearner(assemblages, behind_ball,
                                         reward, learner);
```

First, a Q-learning module is instantiated (the parameters aren't important for this discussion). Next a reward schema is instantiated. **RewardOnScore**, is one of several potentially useful reward functions for soccer. It returns 1

when the robot's team just scored, -1 when the opponents score and 0 otherwise. The next few lines specify which assemblages are to be selected from. Finally, *top_level* is declared with a learning coordination operator. An important advantage of the declaration syntax is the ease with which alternate learning techniques and reward functions may be substituted.

After configuration, the coordination runs as follows: At each movement step the reward schema is queried as to the current reinforcement signal. Next, the perceptual feature **behind_ball** is accessed to determine the agent's perceived state. Finally, the learning module is queried with the state and provided the reinforcement signal. The learning module updates its Q-values accordingly and returns an integer indicating which of the assemblage to activate.

4 Conclusion

Clay is a new robot architecture offering the real time performance of motor schemas and the adaptive capabilities of reinforcement learning. Unlike earlier architectures integrating behavior-based control and reinforcement learning, Clay provides for continuous-valued sensing and action at the motor control level. Discrete learning takes place at a higher assemblage selection level.

Clay provides a fully recursive object-oriented syntax for expressing behaviors. The syntax corresponds closely to intuitive notions of schema instantiation and embedding. It also provides for reuse of identical schemas in multiple assemblages.

The expressive power of Clay was illustrated through the design of a learning soccer robot control system.

References

[1] R.C. Arkin. Motor schema based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, 1989.

[2] R.C. Arkin and T.R. Balch. Aura: principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence*, in press, 1997.

[3] R.C. Arkin and D.C. MacKenzie. Temporal coordination of perceptual algorithms for mobile robot navigation. *IEEE Transactions on Robotics and Automation*, 10(3):276–286, 1994.

[4] T. Balch and R.C. Arkin. Avoiding the past: a simple but effective strategy for reactive navigation. In *IEEE Conference on Robotics and Automation*, pages 678–685. IEEE, May 1993. Atlanta, Georgia.

[5] R.J. Clark, R.C. Arkin, and A. Ram. Learning momentum: On-line performance enhancement for reactive systems. In *IEEE Conf. on Robotics and Automation*, pages 111–116. IEEE, May 1992. Nice, France.

[6] C. Connolly and R. Grupen. On the applications of harmonic functions to robotics. *Journal of Robotic Systems*, 10(7):931–936, 1993.

[7] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[8] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Proceedings 1985 IEEE Conference on Robotics and Automation*, page 500, St. Louis, 1985.

[9] B. Krogh. A generalized potential field approach to obstacle avoidance control. Sme - ri technical paper, 1984. MS84-484.

[10] Long-Ji Lin. Hierachical learning of robot skills by reinforcement. In *International Conference on Neural Networks*, 1993.

[11] D.C. MacKenzie, J.M. Cameron, and R.C. Arkin. Specification and execution of multiagent missions. 1995. To appear IROS-1995.

[12] P. Maes and R. Brooks. Learning to coordinate behaviors. In *Proceedings, Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 796–802. AAAI, 1990.

[13] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, pages 311–365, 1992.

[14] M. Mataric. Designing emergent behaviors: From local interactions to collective intelligence. In *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats 2*, pages 432–441, 1992.

[15] M. Mataric. Learning to behave socially. In *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, 1994.

[16] M. Pearce, R.C. Arkin, and A. Ram. The learning of reactive control parameters through genetic alg orithms. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 130–137. IEEE/RSJ, 1992. Raleigh, NC.

[17] A. Ram and J.C. Santamaría. Multistrategy learning in reactive control. *Informatica*, 17(4):347–369, 1993.

[18] Richard S. Sutton. DYNA, an Integrated Architecture for Learning, Planning and Reacting. In *Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures*, March 1991.

[19] J. Tsitsiklis and B Van Roy. An analysis of temporal-difference learning with function approximation. Technical report, M.I.T. Laboratory for Infomation and Decision Systems, 1996. Available at http://donald-duck.mit.edu/lids.

[20] Christopher J. C. H. Watkins and Peter Dayan. Technical note: Q learning. *Machine Learning*, 8:279–292, 1992.