Integrating RL and Behavior-based Control for Soccer*

Tucker Balch

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

1 Introduction and Background

This paper describes Clay, an evolutionary architecture for autonomous robots integrating motor schema-based control and reinforcement learning. Robots utilizing this system 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. Clay was used by Georgia Tech in the configuration of a soccer team for the RoboCup-97 simulator competition [Kitano *et al.*, 1997]. A simple robot soccer strategy is used to to illustrate the utility of the system.

Motor schemas are the reactive component of Arkin's Autonomous Robot Architecture (AuRA) [Arkin and Balch, 1997]. 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.

Individual motor schemas, or primitive behaviors, express separate goals or constraints for a task. 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, case-based reasoning or genetic algorithms. 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 (e.g. [Connolly and Grupen, 1993]), 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 [Arkin and Balch, 1997]; resorting to high-level planning; repulsion from previously visited locales [Balch and Arkin, 1993]; continuous adaptation [Clark *et al.*, 1992]; and other learning strategies. Schema-based robot control has been demonstrated to provide robust navigation in complex and dynamic worlds.

1.1 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* [Arkin and MacKenzie, 1994].

*Proc. 1997 IJCAI Workshop on RoboCup, Nagoya, Japan

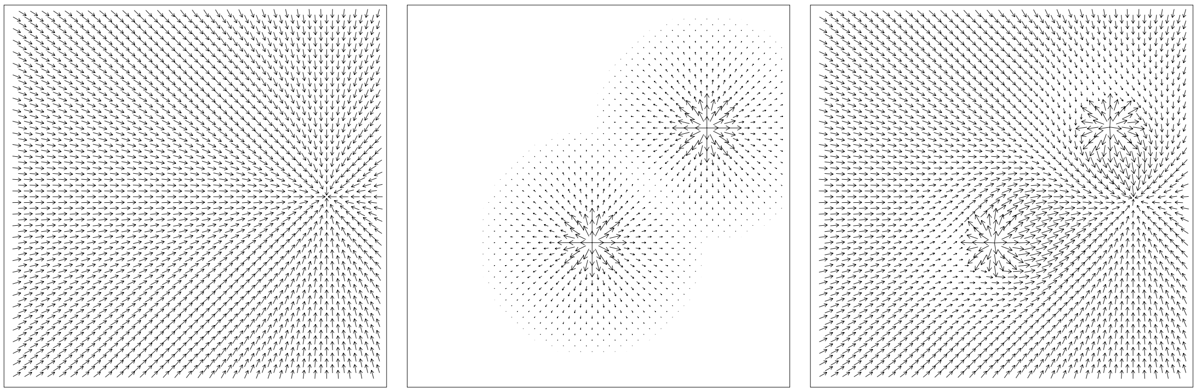


Figure 1: Motor schema example. The diagram on the left shows a vector field corresponding to a **move-to-goal** schema, pulling the robot to a location on the right. The center diagram shows an **avoid-obstacles** field, repelling the robot from two sensed obstacles. On the right, the two schemas are 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. Similarly schemas are used for soccer-playing strategies.

As an example use of temporal sequencing, consider the strategy for a robot soccer player. The strategy outlined here was used in initial research; the actual system used by Georgia Tech in the RoboCup-97 competition is more complex. 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.

In this example behavioral system 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 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. The following sections describe how learning can be introduced, so that the agents don't necessarily follow a fixed sequence.

2 Expressing Behaviors in Clay

Behavioral expression in Clay is fully recursive: there is no limit to the number of levels in a behavioral hierarchy. Clay's primitive, the motor schema, provides a rich repertoire for behavioral design [Arkin and Balch, 1997]. Motor schemas take full advantage of continuous sensor values and can generate an infinite range of actuator output; most other approaches integrating reinforcement learning and behavior-based control only select from a discrete list of actions. Experiments with Clay have so far only explored learning at one level, but the designer is free to introduce learning at any level in the behavioral hierarchy.

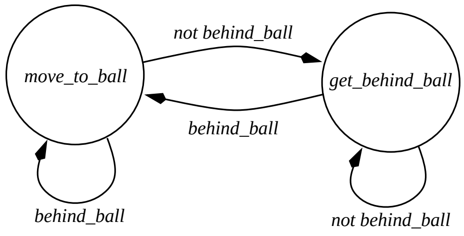
Clay provides for recursive expression of behavior and adds learning coordination operators and an object-oriented syntax. The object-oriented approach provides for a direct expression of schema instantiation and the "embedding" of perceptual schemas in motor schemas. For example, the statement:

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

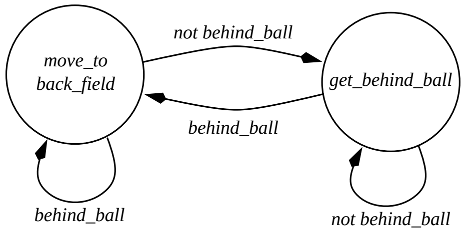
creates a new instance of the **MoveTo** motor schema using the embedded **ball_direction** perceptual schema. The resulting motor process "**move_to_ball**," will draw the robot towards the perceived ball location.

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."



Control Team Forward



Control Team Goalie

Figure 2: An example soccer team's strategy viewed as FSAs. This strategy is used as a control in experiments using Clay. The strategies employed by Georgia Tech in RoboCup-97 are more complex.

- **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

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: A simple soccer team strategy viewed as lookup 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.

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 *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 following declaration of *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,
                  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 in the goalie configuration as follows:

```
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 potential difficulty for 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.

3 Adding Reinforcement Learning

The initial implementation of Clay utilizes a form of Q-learning as a coordination operator. 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. 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 [Watkins and Dayan, 1992] has developed an algorithm for determining $Q(s, a)$ that converges to optimal under certain conditions.

Q-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.

References

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

[Arkin and MacKenzie, 1994] 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.

[Balch and Arkin, 1993] 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.

[Clark *et al.*, 1992] 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.

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

[Kitano *et al.*, 1997] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proc. Autonomous Agents 97*. ACM, 1997. Marina Del Rey, California.

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