Chapter 3

Methodology

A key contribution of this work is the idea that diversity should be evaluated as a *result* rather than an initial condition of multi-robot experiments. In earlier investigations researchers configured homogeneous and heterogeneous teams *a priori*, then evaluated their performance [FM97, GM97, Par94]. The latter approach is useful when investigating the impact of diversity on performance, but it does not provide for the study of diversity as an emergent property of agents interacting with their environment. Defining heterogeneity as an independent rather than dependent variable enables the examination of diversity from an ecological point of view. We can now ask questions like "how does the number of agents impact diversity?" or "how does reward impact diversity in learning teams." These issues are the core of this work.

Along with the opportunity for new kinds of research, the paradigm of diversity as a measured outcome presents new challenges. First, a quantitative measure of diversity is necessary. Second, a methodology for employing the metric in the experimental exploration of diversity in multi-robot systems must be developed. The issue of a quantitative metric is dealt with in Chapter 5. The purpose of this chapter is to introduce a methodology for investigating diversity in multi-robot systems and to explain how it was applied experimentally.

3.1 Overview

Principled research in any field requires adherence to a methodological framework. Over the last decade the Mobile Robot Laboratory at Georgia Tech has evolved and refined a successful approach to behavior-based robot design and implementation. Key components of the method are the use of simulation for experimentation and behavioral prototyping along with verification of the results on mobile robots.

The framework was extended significantly in this research. First, a formal view of multi-robot task is adopted; multi-robot tasks are classified according to how performance is measured in them. This enables a principled description and exploration of the multi-robot task space. Second, a classification of reward functions is employed. This classification defines an experimental space for investigating the impact of reward on multi-robot systems. To support these experiments, motor schema control and reinforcement learning are integrated using a new object-oriented system for behavioral specification. Finally, new evaluation metrics necessary for the measurement of diversity in multi-robot teams were developed and employed in the analysis of experimental data.

The design and implementation of multi-robot systems and their experimental evaluation is carried out in the following steps:

1. **Task and performance metric specification:** This step defines performance, one of the dependent variables of experimentation.

2. **Behavioral design:** In this phase, a library of behaviors are developed for solving the task. Both hand-coded and learning systems are built using the behavioral components.

3. **Reinforcement function specification:** A goal of the research is to explore how different reinforcement functions impact performance and diversity in learning systems. In each task domain, several reinforcement functions are employed, with primary focus on the comparison of local and global rewards.

4. **Simulation:** The behaviors and learning systems are prototyped and tested in simulation.

5. **Implementation on mobile robots:** Performance of the simulated system is validated on mobile robots. If inconsistencies are discovered the simulation environment is refined to more closely approximate mobile robot performance.

6. **Data collection:** Multiple runs (thousands, usually) are conducted in simulation, and when possible, on mobile robots. The experimental space is explored by varying the independent variables (e.g. number of robots and/or the reward function).

7. **Analysis:** The data are analyzed using the performance metric the measures of diversity presented in Chapter 5.

The remainder of this chapter describes the methodology in more detail and provides a high-level synopsis of the thesis experiments.

3.2 Task and performance specification

The research was conducted using the same methodology in each of three multi-robot task domains. Within each domain, quantitative performance data were gathered to determine the utility of various reinforcement functions and their impact on the robots' societal structure.

In systems using reinforcement learning the best possible reward function is the performance metric itself. Following this philosophy, the task for a robot team is to maximize the performance metric. The tasks investigated in this work and their performance metrics include

- **multi-foraging:** earlier multi-robot research defined the foraging task for a robot as "find all attractor objects and deliver them to a location (or homebase)" [BBC$^+$95, ABN93, Mat92]. In multi-foraging, however, the task is extended to include coded attractors and bases. Individual attractors must be delivered to a base of the same color: red attractors to red bases, blue attractors to blue bases and so on. Do agents diversify by specializing in the collection of one type of attractor or the other? Performance in this task is defined as the number of attractors collected and properly delivered in a set time period.

- **robotic soccer:** the task is to propel a ball by bumping and kicking to a goal. The problem is complicated by an opposing team trying to do the same thing in the opposite direction. The game is interesting for multiagent robotics research because it is familiar, reasonable for implementation on mobile robots, and offers opportunities for specialization. Do learning robotic agents specialize as humans do

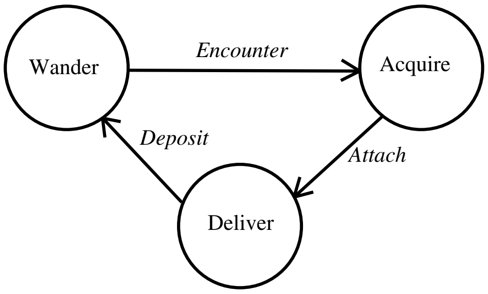

Figure 3.1: The forage FSA.

(goalie, forward, etc.)? The task has been explored by several researchers [KAK+97, NMH96] and has been the focus of international competitions in 1997 and 1998. Performance in soccer is defined as the difference in score at the end of a game.

- **cooperative movement:** The task is for a robot team to negotiate cluttered terrain using the most efficient formation behavior (or none at all). Is there an advantage to traveling in a group? Earlier research by the author and Ron Arkin has shown that some types of formation have an advantage over the others [BA95b]. The primary performance metric in this task is time to navigate across a specified distance.

The relationship between task and performance is covered in depth in Chapter 4.

3.3 Behavioral design

A schema-based reactive control system is used for robot programming. In this approach the agent is provided several pre-programmed behavioral assemblages that correspond to steps in achieving the task. As an example, for a foraging robot we might develop *wander*, *acquire* and *deliver* behaviors for steps in the task [AM94]. Binary perceptual features (also referred to as perceptual triggers) are used to sequence the robot through the behaviors to complete the task. Selection of the appropriate behavior(s) given the situation may be hand-coded or discovered by the robot through reinforcement learning. In this work, both hand-coded and learning systems are developed and evaluated side-by-side.

To ensure a fair comparison between the various hand-coded and learning systems, a fixed repertoire of behaviors are developed for each task domain. In each case, the repertoire is suitable for building behaviorally homogeneous teams as well as heterogeneous strategies. Hand-coded agents proceed deliberately from behavior to behavior as they accomplish the task while the learning agents must discover which behavior to activate when.

3.3.1 Example: foraging

To illustrate how behaviors are developed and coded, consider the programming of a foraging robot. In this example, a robot can be in one of three behavioral states: *wander*, *acquire* and *deliver*. The robot begins in the *wander* state. If there are no attractors within the robot's field of view, the robot remains in *wander* until one is encountered. When an attractor is encountered, a transition to the *acquire* state is triggered. While in the *acquire* state, the robot moves towards the attractor and when it is sufficiently close, grasps it. The last state, *deliver*, is triggered when the robot grasps the attractor. While in the *deliver* state the robot carries the attractor back to home base. Upon reaching home base, the robot deposits the attractor and reverts back to the *wander* state. An FSA summarizing this behavioral configuration is presented in Figure 3.1. The behavioral states are composed of more primitive behaviors (motor schemas) as follows:

- *wander:* move randomly about the environment in search of attractors. Upon encountering an attractor, the pre-programmed agents automatically transition to an appropriate *acquire* behavior. Learning systems, in contrast, discover an appropriate follow-on behavior on their own. Motor schemas active in the *wander* assemblage are
  - **noise:** high gain, moderate persistence to cover a wide area of the environment.
  - **avoid_obstacles:** gain sufficiently high to avoid collisions.
  - **avoid_robots:** high gain to encourage the robots to distribute about the search area.
- *acquire:* move towards the closest visible attractor. When close enough to grasp the attractor, hand-coded agents close their gripper and transition to the *deliver* behavior. Learning agents must learn which follow-on behavior to activate. Motor schemas activated in this assemblage include
  - **noise:** low gain, to deal with local minima endemic to potential field approaches.
  - **avoid_obstacles:** gain sufficiently high to avoid collisions.
  - **avoid_robots:** gain sufficiently high to avoid collisions.
  - **move_to_attractor:** high gain to move the agent to the attractor.
- *deliver:* move towards the delivery area. When close enough to deposit the attractor in the delivery area, hand-coded agents open their gripper and transition to the *wander* assemblage.

- **noise:** low gain, to deal with local minima endemic to potential fields approaches.
- **avoid_obstacles:** gain sufficiently high to avoid collisions.
- **avoid_robots:** gain sufficiently high to avoid collisions.
- **move_to_homebase:** high gain to move the agent to the delivery area.

In this simplified example the robots select from only three behavioral assemblages, in actual foraging experiments the robots are provided six (see Chapter 6). The next section shows how these behaviors can be implemented in Clay, the behavioral coding used in this research.

3.3.2 Configuring behavior with Clay

Robot behaviors are implemented using Clay, a library of primitive behaviors and coordination mechanisms coded in Java [Bal97a]. Clay is a component of JavaBots, a new system for simulation and control of multi-robot teams. Clay is the first behavioral configuration tool that integrates motor schema-based control and reinforcement learning. The name "Clay" was chosen for its connotations of reconfigurability and ease of use. Robots utilizing Clay benefit from the real-time performance of motor schemas in continuous and dynamic environments and 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 is similar to earlier approaches integrating reinforcement learning and behavior-based control [MC92, Lin93, Mat94], 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 [AB97]. 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 [SSR98]. 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. In previous research Georgia Tech's Mobile Robot Laboratory developed a system called *MissionLab* to support the design and test of sequenced behaviors on robots and in simulation [MCA95]. *MissionLab* includes a set of tools for recursively expressing sequenced behaviors. Like *Missionlab*, Clay provides for recursive expression of behavior, but it adds learning coordination operators and an object-oriented syntax.

The basic building block in Clay is a **node**. There are two important phases in a node's life: intialization and runtime. Nodes have only two methods, corresponding to these phases: the constructor, used for initialization; and `Value()`, called repeatedly at runtime. The object-oriented approach provides for a direct expression of schema instantiation and the embedding of perceptual schemas in motor schemas. The embedding is specified at initialization time using the node's constructor. Here is an example of how one node is embedded in another:

```
PS_OBS = new Obstacles(abstract_robot);
MS_AVOID_OBSTACLES  = new Avoid(2.0, 1.0, PS_OBS);
```

In this case, a perceptual schema for detecting obstacles (`PS_OBS`) is embedded in an `Avoid` node. The resulting motor process `MS_AVOID_OBSTACLES`, will draw the robot away from the perceived obstacles. The `PS` and `MS` prefixes are used to help readers and programmers distinguish between perceptual and motor schemas in the code.

Note that the embedding provides for code re-use. We could, for instance, avoid robots instead of obstacles by embedding a `PS_ROBOTS` versus `PS_OBS` in the `Avoid` node. It is also possible to re-use instantiated nodes by embedding them in several other nodes. In this next example, `PS_OBS` is imbedded in an `MS_AVOID_OBSTACLE` node and a `MS_SWIRL_OBSTACLE` node:

```
PS_OBS = new Obstacles(abstract_robot);
MS_AVOID_OBSTACLES  = new Avoid(2.0, 1.0, PS_OBS);
MS_SWIRL_OBSTACLES  = new Swirl(2.0, 1.0, PS_OBS, heading);
```

Nodes are combined by embedding them in a blending node. `StaticWeightedSum` is an example blending node class. This type node is used for the "sum and normalize" step of schema and assemblage combining. It takes an array of nodes and an array of weights as input at configuration time. At runtime, it multiplies the output of each embedded node by the associated weight or gain, then sums them. The following statements generate a new node, `AS_AVOID_N_SWIRL`, that is the average of its two embedded nodes:

```
AS_AVOID_N_SWIRL = new StaticWeightedSum();
AS_AVOID_N_SWIRL.embedded[0]    = MS_AVOID_OBSTACLES;
AS_AVOID_N_SWIRL.weights[0]     = 0.5;
AS_AVOID_N_SWIRL.embedded[1]    = MS_SWIRL_OBSTACLES;
AS_AVOID_N_SWIRL.weights[1]     = 0.5;
```

Figures 3.2 through 3.4 show how foraging behaviors may be coded in Clay. In this example a sequential (non-learning) system is configured. First, perceptual schemas and the motor schemas they are embedded within are declared (Figure 3.2). Next (Figure 3.3) the behaviors are grouped into assemblages using a weighted-sum operator. Finally, a high-level behavioral sequence is defined by coding it as an FSA (Figure 3.4; also shown graphically in Figure 3.1). Transitions between the behavioral states are triggered when appropriate perceptual features become true.

The code in Figures 3.2 through 3.4 specifies a multi-level hierarchical behavioral system. The hierarchy is presented graphically in Figure 3.5. The lowest level perceptual features are declared first (e.g. `PS_HOMEBASE`), then embedded in motor schemas (e.g. `MS_MOVE_TO_HOMEBASE`) that are in turn embedded in behavioral assemblages (e.g. `AS_DELIVER`). At the assemblage level, behaviors are combined cooperatively using a weighted sum operator. At the next level up, assemblages are selected in sequence using an FSA. The highest level behavior, `FORAGE`, is at the top of a four-layer recursive tree. If a designer was interested in building a more complicated agent with foraging as one of its several capabilities, `FORAGE` could be included as just another assemblage for integration at the next level up.

Control systems coded in Clay follow a perception/action cycle where each cycle is referred to as a timestep or movement cycle. For each cycle, computation begins at the top (e.g. `FORAGE`) and continues recursively downward through the configured assemblages and schemas. A potential difficulty for hierarchically specified behavioral systems is that as a behavioral configuration grows more complex, run time computational demands increase dramatically. Clay avoids the problem by only executing activated assemblages and schemas. Computational demands are also reduced when the designer re-uses schemas in a configuration (as `PS_ATTRACTORS` is re-used above). A synchronization technique ensures a schema's output is only computed once per movement cycle.

Reinforcement learning can be incorporated into a behavioral configuration using an additional coordination operator, `CoordinateLearner`. `CoordinateLearner` is

```
//=== Perceptual schemas
        // type of object in gripper (-1 if nothing)
        PS_IN_GRIPPER = new InGripper(abstract_robot);

        // sonar readings
        PS_OBS = new Obstacles(abstract_robot);

        // other robots
        PS_ROBOTS = new Teammates(abstract_robot);

        // location of the homebase
        PS_HOMEBASE = new GlobalToEgo(abstract_robot, PS_HOMEBASE_GLOBAL);

        // the list of visible attractors
        PS_ATTRACTORS = new VisualObjects(0,abstract_robot);

        // the closest attractor
        PS_CLOSEST_ATTRACTOR = new Closest(PS_ATTRACTORS);

//=== Motor schemas
        // sphere of influence 1.5m, safety radius 0.1m, objects to avoid
        MS_AVOID_OBSTACLES = new Avoid(1.5, RADIUS + 0.1, PS_OBS);

        // sphere of influence 2.5m, safety radius 0.1m, objects to avoid
        MS_AVOID_ROBOTS = new Avoid(2.5, RADIUS + 0.1, PS_ROBOTS);

        // generate a new random direction every five seconds
        MS_NOISE_VECTOR = new Noise(5,seed);

        // attraction decreases linearly inside 0.4m constant beyond that
        MS_MOVE_TO_HOMEBASE = new LinearAttraction(0.4, 0.0, PS_HOMEBASE);

        // attraction decreases linearly inside 0.4m constant beyond that
        MS_MOVE_TO_ATTRACTOR = new LinearAttraction(0.4, 0.0, PS_CLOSEST_ATTRACTOR);
```

Figure 3.2: Clay source code for the declaration of perceptual and motor schemas employed in the foraging task. For clarity, a `PS` prefix is used in the declaration of a perceptual schema, while `MS` is used for motor schemas. Note how the syntax supports the embedding of perceptual schemas in motor schema declarations. For example, the `PS_HOMEBASE` perception is embedded in the `MS_MOVE_TO_HOMEBASE` motor schema.

```
//=== Wander assemblage
       AS_WANDER = new StaticWeightedSum();
       AS_WANDER.weights[0]  = 1.0;
       AS_WANDER.embedded[0] = MS_NOISE_VECTOR;
       AS_WANDER.weights[1]  = 0.8;
       AS_WANDER.embedded[1] = MS_AVOID_OBSTACLES;
       AS_WANDER.weights[2]  = 0.8;
       AS_WANDER.embedded[2] = MS_AVOID_ROBOTS;

//=== Acquire assemblage
       AS_ACQUIRE = new StaticWeightedSum();
       AS_ACQUIRE.weights[0]  = 0.2;
       AS_ACQUIRE.embedded[0] = MS_NOISE_VECTOR;
       AS_ACQUIRE.weights[1]  = 0.8;
       AS_ACQUIRE.embedded[1] = MS_AVOID_OBSTACLES;
       AS_ACQUIRE.weights[2]  = 0.8;
       AS_ACQUIRE.embedded[2] = MS_AVOID_ROBOTS;
       AS_ACQUIRE.weights[3]  = 1.0;
       AS_ACQUIRE.embedded[3] = MS_MOVE_TO_ATTRACTOR;

//=== Deliver assemblage
       AS_DELIVER = new StaticWeightedSum();
       AS_DELIVER.weights[0]  = 0.2;
       AS_DELIVER.embedded[0] = MS_NOISE_VECTOR;
       AS_DELIVER.weights[1]  = 0.8;
       AS_DELIVER.embedded[1] = MS_AVOID_OBSTACLES;
       AS_DELIVER.weights[2]  = 0.8;
       AS_DELIVER.embedded[2] = MS_AVOID_ROBOTS;
       AS_DELIVER.weights[3]  = 1.0;
       AS_DELIVER.embedded[3] = MS_MOVE_TO_HOMEBASE;
```

Figure 3.3: Source code for the declaration of behavioral assemblages used in foraging. The `AS` prefix indicates the declaration of a behavioral assemblage. Motor schemas are cooperatively combined using the `StaticWeightedSum` operator. Note how the syntax supports reuse of motor schemas (e.g. `MS_AVOID_OBSTACLES`) in several assemblages.

```
//=== Perceptual Features
      // true if an attractor is visible
      PF_ATTRACTOR_VISIBLE = new NonZero(PS_CLOSEST_ATTRACTOR);

      // true if an attractor has been grasped
      PF_ATTRACTOR_IN_GRIPPER = new Equal(0, PS_IN_GRIPPER);

      // true if close enough to homebase to deposit the attractor
      PF_CLOSE_TO_HOMEBASE = new Close(PS_HOMEBASE);

//=== Behavioral state machine
      STATE_MACHINE = new FSA(); // declare the state machine
      STATE_MACHINE.state = 0;        // set initial state to 0 (wander)

      // if an attractor is visible, transition to state 1 (acquire)
      STATE_MACHINE.triggers[0][0] = PF_ATTRACTOR_VISIBLE;
      STATE_MACHINE.follow_on[0][0]= 1;

      // if an attractor in gripper, transition to state 2 (deliver)
      STATE_MACHINE.triggers[1][0] = PF_ATTRACTOR_IN_GRIPPER;
      STATE_MACHINE.follow_on[1][0]= 2;

      // if at homebase, go back to state 0 (wander)
      STATE_MACHINE.triggers[2][0] = PF_CLOSE_TO_HOMEBASE;
      STATE_MACHINE.follow_on[2][0]= 2;

//=== Select assemblage based on behavioral state
      FORAGE = new Select(STATE_MACHINE); // declare selector

      // activate appropriate behavior based on state
      FORAGE.embedded[0]    = AS_WANDER;
      FORAGE.embedded[1]    = AS_ACQUIRE;
      FORAGE.embedded[2]    = AS_DELIVER;
```

Figure 3.4: A behavioral sequence is configured using the `FSA` operator in Clay. A `PF` prefix indicates the declared object is a perceptual feature. Transitions between behavioral assemblages are triggered by perceptual features.

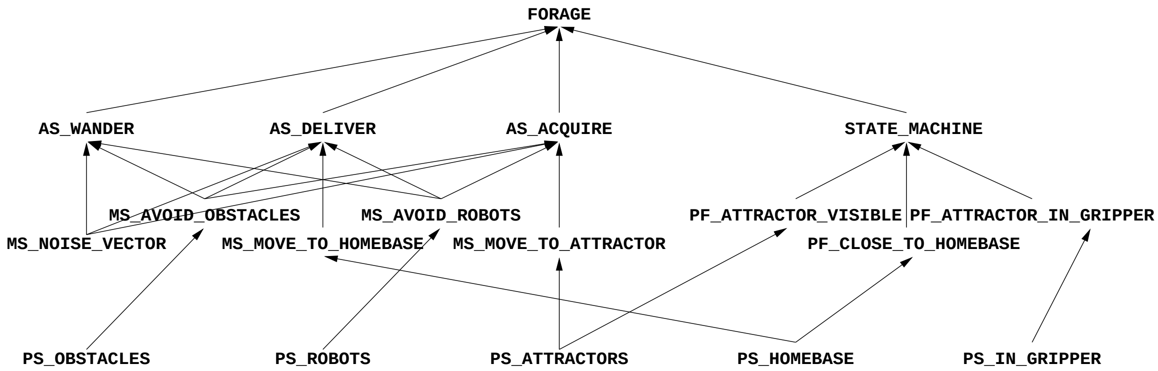

Figure 3.5: A graphical representation of the hierarchically configured `FORAGE` behavior. Prefixes are used to indicate perceptual schemas (`PS`), motor schemas (`MS`), behavioral assemblages (`AS`) and perceptual features (`PF`).

"plug compatible" with the `FSA` operator (see the bottom of Figure 3.4 for an example use of `FSA`). `CoordinateLearner` is able to learn to select appropriate assemblages rather than following a fixed sequence. At configuration time, an instantiation of `CoordinateLearner` is provided an embedded reward schema that it uses for learning over time. Any learning module that conforms to Clay's Application Programmer's Interface (API) can be integrated into a behavioral configuration. Q-learning was used in this research.

3.4 Reinforcement function specification

The next step in learning multi-robot team development is the specification of appropriate reinforcement (or reward) functions. One focus of this research is to find out if and how the choice of reward function impacts performance and diversity in different tasks. In the experiments described in later chapters, performance is first evaluated in learning teams using a global function. Global functions are those where the entire team of robots is provided the same reward signal at once. Performance with the global function serves as a baseline for comparison with teams using other functions. In addition to the global function robot teams were also trained using a local reward function. Learning agents using a local reinforcement function are rewarded for their individual performance rather than the overall performance of the team. Even though local rewards are targeted to the individual robot, they may, in some cases require global information for implementation. For more information on reward functions see the classification of reinforcement functions in Chapter 4.

The reward functions used experimentally are listed below (they are also described in corresponding chapters later in the dissertation). Each reward function is defined at timestep t based on events occurring at timestep $t-1$ as follows:

- **For multi-robot foraging** three functions were explored; performance-based global, performance-based local and heuristic functions:

$$
\begin{aligned}
R_{\text{global}}(t) &= \begin{cases} 1 & \text{if any agent delivered an attractor at time } t-1 \\ -1 & \text{otherwise} \end{cases} \\
R_{\text{local}}(t) &= \begin{cases} 1 & \text{if the agent delivered an attractor at time } t-1 \\ -1 & \text{otherwise} \end{cases} \\
R_{\text{shaped}}(t) &= R_{\text{event}}(t) + R_{\text{intruder}}(t) + R_{\text{progress}}(t)
\end{aligned}
$$

  The R_{event}, R_{intruder} and R_{progress} components of R_{shaped} encapsulate separate heuristic components of the overall reward. See Chapter 6 for more information on this reward function.

- **For robot soccer** three performance-based functions were examined:

$$
\begin{aligned}
R_{\text{global}}(t) &= \begin{cases} 1 & \text{if the team scored at } t-1, \\ -1 & \text{if the opponent score at } t-1, \\ 0 & \text{otherwise.} \end{cases} \\
R_{\text{local}}(t) &= \begin{cases} 1 & \text{if the agent was closest to the ball} \\ & \text{when its team scores,} \\ -1 & \text{if the agent was closest to the ball} \\ & \text{when the opposing team scores,} \\ 0 & \text{otherwise.} \end{cases} \\
R_{\text{touch}}(t) &= \begin{cases} \gamma^{t_{\text{touch}}} & \text{when the team scores,} \\ -\gamma^{t_{\text{touch}}} & \text{when the opponent scores,} \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
$$

  Even though implementation of R_{local} requires global information the reward is referred to as local because the reinforcement based on the individual's performance. In R_{touch} the variable t_{touch} is time in milliseconds since the agent last touched the ball. The parameter γ is set to values between 0 and 1. When $\gamma = 1$ R_{touch} is equivalent to R_{global}. As γ is reduced towards 0, the reward becomes increasingly agent-centered.

- **For cooperative movement** two reward functions were tested:

$$
\begin{aligned}
R_{\text{global}} &= -\text{elapsed time for all robots to cross the field} \\
R_{\text{local}} &= -\text{elapsed time for the individual robot to cross the field}
\end{aligned}
$$

These functions reward agents for moving across the field as quickly as possible. In this task only a single reward is provided at the end of the run.

3.5 Simulation

Many autonomous robot designers use simulation as a tool to speed behavioral development. Once behaviors work well in simulation they are moved to mobile robots for further debugging and verification. Simulation plays a valuable role in experimentation. In the time it takes to complete a few runs on mobile robots, simulation systems can complete thousands, perhaps millions of multiagent trials. Simulation is also important for use in learning systems where many thousands of trials may be required for an agent to learn a behavior or strategy. It would be impossible to conduct such learning on a mobile robot in a timely manner. Finally, simulation enables the exploration of a larger experimental space than would otherwise be possible.

The JavaBots system is utilized for simulation and mobile robot experimentation [Bal98, Bal]. Behaviors coded in JavaBots may be run in simulation, and without modification, on Nomadic Technologies' Nomad 150 mobile robots (the ISR Pebbles robot is also supported). The bulk of statistical results in this work were gathered by running robot behaviors in thousands of simulation trials.

In the foraging and cooperative movement investigations, each robot is a kinematically holonomic vehicle (a simulated Nomad 150) controlled by a behavioral system coded in Clay. Simulated motor and sensor capabilities are based on performance of the physical robots. The robots can detect hazards with sonar out to a range of nine meters. Attractors can be detected visually out to three meters across a 90 degree field of view. In soccer experiments, dimensions and dynamics are based on RoboCup F-180 class robots. Speed and turning rates mimic the performance of mobile robots built for RoboCup competition [Sto98].

The external environment perceived by robot control systems in simulation is described in a file read by the simulation application at run time. Multiple robots may be distributed about a "playing field" along with obstacles, opponent robots and attractor objects. Even though some control systems utilize a pseudo-random number generator (e.g. `NOISE`), determinism is supported with a `seed` statement in the description file syntax. Time is measured in simulated seconds. Since reactive control systems are very fast, several thousand control cycles are completed each second. Simulations proceed faster than real time with each control cycle fixed at 200 milliseconds (simulation time).

3.6 Implementation on mobile robots

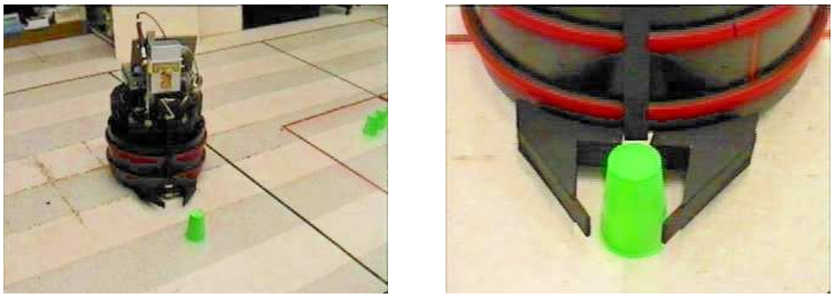

Figure 3.6: Nomad 150 robotic platform. Close-up view of the passive gripper (right).

Simulated behavior should be validated with experiments on mobile robots. When discrepancies are found between behavior in simulation and on real robots, the simulator must be revised to reflect real world performance (such discrepancies are most often due to inaccurate sensor or actuator modeling). The multi-robot foraging behaviors developed in this research were prototyped in simulation and verified on Nomadic Technologies' Nomad 150 robots. Since the behaviors are implemented in JavaBots, they can run in simulation and on hardware without revision.

Learning systems are developed and evaluated on mobile robots in the following steps:

1. initialize control system on mobile robot(s) with a random policy;
2. evaluate performance of the initial policy;
3. transfer policy to simulation system;
4. train in simulation;
5. transfer policy back to mobile robot(s);
6. evaluate performance on mobile robot(s) after learning.

These steps are covered in more detail on the chapter concerning multi-robot foraging (Chapter 6).

3.6.1 Hardware platform

The Nomad 150 is a three-wheeled commercially available kinematically holonomic vehicle (Figure 3.6, left). Nomad 150s are equipped with a separately steerable turret, 16 ultrasonic range sensors and a ring of rubber bump sensors. The robots were modified at Georgia Tech to add real-time vision and grippers.

Each robot's vision system is able to segment video images into blobs according to color 30 times per second. The location of objects detected as blobs in the image are determined in JavaBots using a lookup table that maps image coordinates to real-world locations. The lookup table is computed before experimental runs by moving a robot to known positions while it tracks a brightly painted object at the origin. The robot's field of view was expanded to approximately 160 degrees by mounting a security door lens (peep-hole) to the robot's video camera. Unfortunately, only the central 90 degrees are usable because objects at the periphery are quite small in the image. The two types of attractor object used in foraging experiments are painted different colors (fluorescent red and green) to enable the robots to tell them apart. Example images generated by the vision system are presented in Figure 3.7.

The robots were equipped with hobby servo-actuated grippers that enable them to grasp and lift attractors. The active grippers work well, but frequently require time-consuming repair. For later experiments the active grippers were replaced with passive devices (Figure 3.6, right). The passive gripper is designed so that once a robot has "captured" an attractor object it will remain under the robot's control as long as the robot does not move backwards. Robots then drop attractors by moving in reverse.

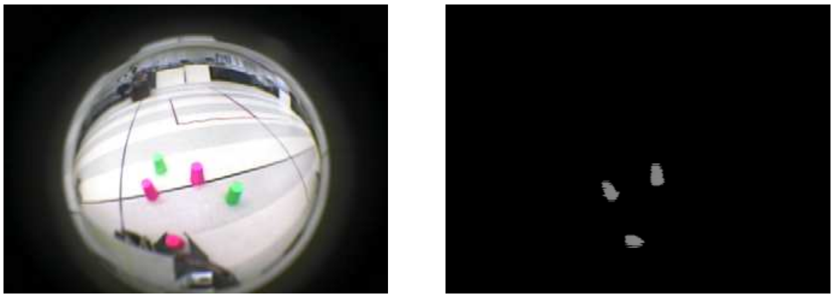

Figure 3.7: Mobile robot's eye view of several attractor objects (left). After processing to segment out red objects (right).

3.7 Data collection and analysis

Within each task domain an experimental space was explored by varying one or more independent variables and evaluating the resulting systems. Statistical significance was ensured by simulation in multiple "worlds" initialized with distinct random number seeds. Within each task the question of how the choice of reward impacts multi-robot teams was explored by varying the reward function. Additionally, the number of robots was varied from 1 to 8 for simulation experiments in robot foraging, 1 and 2 agents were used mobile robot experiments. The dependent variables in these experiments are

- **performance:** How well does the multiagent system perform in the task?
- **learning rate:** How quickly does the team converge to stable behavior?
- **diversity:** Once the system has converged to stable behavior, what is the degree of diversity in the team?

Performance is defined separately for each task (e.g. score difference in soccer). Since learning agents don't always perform well initially, it is also important to consider their learning rate. Some systems require a long time to converge to optimal or near-optimal solutions while others converge to sufficient, but sub-optimal solutions quickly. So, in addition to measuring against the performance metric, the time required for a team to converge to stable behavior is recorded as well.

The experimental space is summarized below:

- **in multi-robot foraging**
  - **independent variables:**
    - strategy: 3 hand-coded strategies and 3 learning strategies.
    - number of robots: 1 to 8 for each strategy.
  - **worlds:** 5 worlds for each combination of independent variable.
  - **runs in each world:** 100 in hand-coded systems, 300 in learning.
  - **total trials:** 48,000 in simulation, 22 on mobile robots.
- **in robot soccer**
  - **independent variables:**
    - reward function: 3, including R_{local}, R_{global} and R_{touch}.
    - γ for R_{touch} was varied from 0.1 to 1.0 in steps of 0.1.

- **worlds:** 10 worlds for each combination of independent variable configuration.
- **runs in each world:** 100 for R_{local} and R_{global}, 200 for each value of γ in R_{touch} runs.
- **simulation trials:** 24,000 in simulation.

• **in cooperative movement**

- **independent variables:**
    * strategy: 3 hand-coded and 2 learning strategies.
- **worlds:** 8 worlds for each strategy independent variable configuration.
- **runs in each world:** 100.
- **total trials:** 4,000 in simulation.

3.8 Discussion and summary

An important contribution of this work is the idea that diversity should be measured as an experimental outcome rather than specified as an initial condition. The view of diversity as a dependent variable enables us to investigate new questions about multi-robot systems, including one focus of this work: the impact of reward structure on diversity in learning multi-robot systems.

The investigation of these questions lead to the development of a new methodology for multi-robot experimentation Steps in the methodology include

1. **task and performance metric specification,**
2. **reinforcement function specification,**
3. **behavioral design,**
4. **simulation,**
5. **implementation on mobile robots,**
6. **data collection and**
7. **analysis.**

Several components of this framework are new. First, a formal view of multi-robot task is adopted; this enables a principled description and exploration of the multi-robot task space. Second, a classification of reward functions is employed. This classification defines an experimental space for investigating the impact of reward on multi-robot systems. Also, to support these experiments, motor schema control and reinforcement learning are integrated using a new object-oriented system for behavioral specification. Finally, new evaluation metrics necessary for the measurement of diversity in multi-robot teams were developed and employed in the analysis of experimental data.