Behavior for Mobile Robots

Bhaskar Mookerji
(updated from Chris Batten’s IAP 2007 Talk)
Maslab IAP Robotics Course
January 6, 2010
What is so hard about designing a mobile robot controller?

Sensors are far from perfect
Camera white balance = bad colors
Ultrasound reflections
Infrared sensors can be noisy
... and many more!

Actuators are far from perfect
Motor velocity changes over time
Wheels and gears slip
Servos get stuck
... and many more!
Even if the world was perfect, the sheer complexity of a robot can be daunting.
Don’t just code a control system, **design** a control system!

Just as you must carefully **design** your robot chassis you must carefully **design** your robot control system

- How will you debug and test your robot?
- What are the performance requirements?
- Can you easily improve aspects of your robot?
- Can you easily integrate new functionality?
An example of how not to design your robot control system

```c
void moveForward( int time ) {
    while ( t < time ) {
        // Drive forward a bit
        --------------------------------------
        --------------------------------------
    }
}
```
An example of how not to design your robot control system

```c
void moveForward( int time ) {
    while ( t < time ) {
        // Drive forward a bit
        --------------------------------------
        --------------------------------------
        // Check ir sensor and stop if necessary
        --------------------------------------
        --------------------------------------
    }
}
```
An example of how not to design your robot control system

```c
void moveForward( int time ) {
    while ( t < time ) {
        // Drive forward a bit
        --------------------------------------
        --------------------------------------
        // Check ir sensor and stop if necessary
        --------------------------------------
        --------------------------------------
        // Rotate if there is an obstacle
        --------------------------------------
        --------------------------------------
    }
}
```
An example of how not to design your robot control system

```c
void moveForward( int time ) {
    while ( t < time ) {
        // Drive forward a bit
        // Check ir sensor and stop if necessary
        // Rotate if there is an obstacle
        // Need to find some balls
        // Somehow pick up a ball
        // What if there is more than one ball?
    }
}
```
An example of how not to design your robot control system

```c
void moveForward( int time ) {
    while ( t < time ) {
        // Drive forward a bit
        // Check ir sensor and stop if necessary
        // Rotate if there is an obstacle
        // Need to find some balls
        // Somehow pick up a ball
        // What if there is more than one ball?
        ...
        // Need to find some goals
        // What if there are no goals visible?
        // Drop off some balls
        // Find more balls I guess
        // Make sure to ignore balls in goal
        // Try to go somewhere new
    }
}
```
An example of how not to design your robot control system

```cpp
void moveForward( int time ) {
  while ( t < time ) {
    // Drive forward a bit
    --------------------------------------
    // Check ir sensor and stop if necessary
    // Rotate if there is an obstacle
    // Need to find some balls
    // Somehow pick up a ball
    // What if there is more than one ball?
    // Need to find some goals
    // What if there are no goals visible?
    // Drop off some balls
    // Find more balls I guess
    // Make sure to ignore balls in goal
    // Try to go somewhere new
    // Find more balls I guess
    // Make sure to ignore balls in goal
    // Try to go somewhere new
  }
}
```
Basic primitive of a control system is a behavior

Behaviors should be well-defined, self-contained, and independently testable

- Turn right 90°
- Go forward until reach obstacle
- Capture a ball
- Explore playing field
Key objective is to compose behaviors so as to achieve the desired goal
Outline

• High-level control system paradigms
  – Model-Plan-Act Approach
  – Emergent Approach
  – Finite State Machine Approach

• Low-level control loops (Tomorrow)
  – PID controllers for motor velocity
  – PID controllers for robot drive system

• Examples from past years
Model-Plan-Act Approach

1. Use sensor data to create model of the world
2. Use model to form a sequence of behaviors which will achieve the desired goal
3. Execute the plan (compose behaviors)
Exploring the playing field to create a model of the world

Red dot is the mobile robot while the blue line is the mousehole
Exploring the playing field to create a model of the world

Robot uses sensors to create local map of the world and identify unexplored areas
Exploring the playing field to create a model of the world

Robot moves to midpoint of unexplored boundary
Exploring the playing field to create a model of the world

Robot performs a second sensor scan and must align the new data with the global map
Exploring the playing field to create a model of the world

Robot continues to explore the playing field
Exploring the playing field to create a model of the world

Robot must recognize when it starts to see areas which it has already explored
Finding a path to the mousehole using the convex cell algorithm

Given the global map, the goal is to find the mousehole.
Finding a path to the mousehole using the convex cell algorithm

Transform world into configuration space by convolving robot with all obstacles
Finding a path to the mousehole using the convex cell algorithm

Decompose world into convex cells
Trajectory within any cell is free of obstacles
Finding a path to the mousehole using the convex cell algorithm

Connect cell edge midpoints and centroids to get graph of all possible paths
Finding a path to the mousehole using the convex cell algorithm

Use an algorithm (such as the A* algorithm) to find shortest path to goal
Finding a path to the mousehole using the convex cell algorithm

The choice of cell decomposition can greatly influence results
Finding a path to the mousehole using the Voronoi cell algorithm

Create a Voronoi partitioning - paths are equidistant from obstacles
Finding a path to the mousehole using the Voronoi cell algorithm

Treat Voronoi paths as “highways”
Maximally avoids obstacles
Example using Voronoi path planning in real world office environment

http://www.cs.columbia.edu/~pblaer/projects/path_planner
Advantages and disadvantages of the model-plan-act approach

- **Advantages**
  - Global knowledge in the model enables optimization
  - Can make provable guarantees about the plan

- **Disadvantages**
  - Must implement all functional units before any testing
  - Computationally intensive
  - Requires very good sensor data for accurate models
  - Models are inherently an approximation
  - Works poorly in dynamic environments
Emergent Approach

Living creatures like honey bees are able to explore their surroundings and locate a target (honey)

Is this bee using the model-plan-act approach?

Used with permission, © William Connolley
http://wnconnolley.ork.uk
Emergent Approach

Living creatures like honey bees are able to explore their surroundings and locate a target (honey)

Probably not! Most likely bees layer simple reactive behaviors to create a complex emergent behavior

Used with permission, © William Connolley
http://wnconnolley.ork.uk
Emergent Approach

Should we design our robots so they act less like robots and more like honey bees?
As in biological systems, the emergent approach uses simple behaviors to directly couple sensors and actuators. Higher level behaviors are layered on top of lower level behaviors.
To illustrate the emergent approach we will consider a simple mobile robot.
Layering simple behaviors can create much more complex emergent behavior

Cruise behavior simply moves robot forward
Layering simple behaviors can create much more complex emergent behavior.

Left motor speed inversely proportional to left IR range
Right motor speed inversely proportional to right IR range
If both IR < threshold stop and turn right 120 degrees
Layering simple behaviors can create much more complex emergent behavior.

Escape behavior stops motors, backs up a few inches, and turns right 90 degrees.
Layering simple behaviors can create much more complex emergent behavior.

The track ball behavior adjusts the motor differential to steer the robot towards the ball.
Layering simple behaviors can create much more complex emergent behavior

Hold ball behavior simply closes ball gate when ball switch is depressed.
Layering simple behaviors can create much more complex emergent behavior.

The track goal behavior opens the ball gate and adjusts the motor differential to steer the robot towards the goal.
Layering simple behaviors can create much more complex emergent behavior.

**Arbitration Techniques**
- Fixed priority
- Round-robin
- Random
- Merge messages
- Vote
Bsim robot simulator illustrates emergent approach

http://www.behaviorbasedprogramming.com
Controller architecture for collection simulation

From "Robot Programming: A Practical Guide to Behavior Based Robotics", Joseph Jones
Advantages and disadvantages of the behavioral approach

• Advantages
  – Incremental development is very natural
  – Modularity makes experimentation easier
  – Cleanly handles dynamic environments

• Disadvantages
  – Difficult to judge what robot will actually do
  – No performance or completeness guarantees
  – Debugging can be very difficult
Model-plan-act fuses sensor data, while emergent fuses behaviors

Model-Plan-Act
Lots of internal state
Lots of preliminary planning
Fixed plan of behaviors

Emergent
Very little internal state
No preliminary planning
Layered behaviors
Finite State Machines offer another alternative for combining behaviors

FSMs have some preliminary planning and some state. Some transitions between behaviors are decided statically while others are decided dynamically.

- **Fwd (dist)**: Fwd behavior moves robot straight forward a given distance
- **TurnR (deg)**: TurnR behavior turns robot to the right a given number of degrees
Finite State Machines offer another alternative for combining behaviors.

Each state is just a specific behavior instance - link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Each state is just a specific behavior instance - link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Each state is just a specific behavior instance - link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Each state is just a specific behavior instance - link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Since the Maslab playing field is unknown, open loop control systems have no hope of success!
Finite State Machines offer another alternative for combining behaviors

Closed loop finite state machines use sensor data as feedback to make state transitions
Finite State Machines offer another alternative for combining behaviors

Closed loop finite state machines use sensor data as feedback to make state transitions
Finite State Machines offer another alternative for combining behaviors.

Closed loop finite state machines use sensor data as feedback to make state transitions.
Finite State Machines offer another alternative for combining behaviors.

Closed loop finite state machines use sensor data as feedback to make state transitions.
Finite State Machines offer another alternative for combining behaviors

Closed loop finite state machines use sensor data as feedback to make state transitions
Implementing a Finite State Machine in Java

```
switch (state) {
    case States.Fwd_1:
        moveFoward(1);
        if (distanceToObstacle() < 2)
            state = TurnR_45;
        break;
    case States.TurnR_45:
        turnRight(45);
        if (distanceToObstacle() >= 2)
            state = Fwd_1;
        break;
}
```
Implementing a FSM in Java

- Implement behaviors as parameterized functions
- Each case statement includes behavior instance and state transition
- Use enums for state variables

```java
switch ( state ) {
    case States.Fwd_1 :
        moveForward(1);
        if ( distanceToObstacle() < 2 )
            state = TurnR_45;
        break;

    case States.TurnR_45 :
        turnRight(45);
        if ( distanceToObstacle() >= 2 )
            state = Fwd_1;
        break;
}
```
Finite State Machines offer another alternative for combining behaviors.

Can also fold closed loop feedback into the behaviors themselves.
Simple finite state machine to locate red balls

Wander (20sec) → Scan 360 → Found Ball → TurnR

No Balls → Scanning 360 → Lost Ball → Align Ball

Ball < 1ft → Fwd (1ft) → Stop

Ball > 1ft → Align Ball

Does this FSM work?
Simple finite state machine to locate red balls

Does this FSM work?
Simple finite state machine to locate red balls

Wander (20sec)  |  Scan 360  |  TurnR

No Balls  |  Found Ball

Lost Ball  |  Ball < 1ft

Ball > 1ft  |  Stop

Align Ball  |  Fwd (1ft)

Does this FSM work?
Simple finite state machine to locate red balls

Scan 360

Wander (20 sec)

Found Ball

TurnR

Align Ball

Lost Ball

Fwd (1ft)

Ball < 1ft

Ball > 1ft

No Balls

Does this FSM work?
Simple finite state machine to locate red balls

- **Wander (20 sec)**
- **Scan 360**
  - No Balls
  - Lost Ball
- **Align Ball**
  - Ball < 1 ft
  - Ball > 1 ft
- **Fwd (1 ft)**
  - Obstacle < 2 ft
- **Stop**
  - Found Ball
  - TurnR
  - Ball < 1 ft
To debug a FSM control system verify behaviors and state transitions

What if robot has trouble correctly approaching the ball?
To debug a FSM control system verify behaviors and state transitions

Independently verify Align Ball and Fwd behaviors
Improve FSM control system by replacing a state with a better implementation

Could replace random wander with one which is biased towards unexplored regions
Improve FSM control system by replacing a state with a better implementation

What about integrating camera code into wander behavior so robot is always looking for red balls?

- Image processing is time consuming so might not check for obstacles until too late
- Not checking camera when rotating
- Wander behavior begins to become monolithic

```plaintext
ball = false
turn both motors on
while ( !timeout and !ball )
capture and process image
  if ( red ball ) ball = true
read IR sensor
  if ( IR < thresh )
    stop motors
    rotate 90 degrees
    turn both motors on
  endif
endwhile
```
Multi-threaded finite state machine control systems

Controller FSM

Drive Motors

Obstacle Sensors Thread

Image Compute Thread

Short IR + Bump

Camera
Multi-threaded finite state machine control systems

Controller FSM

Short IR + Bump

Obstacle Sensors Thread

Image Compute Thread

Camera

Drive Motors
Multi-threaded finite state machine control systems

Controller FSM

Short IR + Bump

Obstacle Sensors Thread

Camera

Image Compute Thread

Stalk Sensors

Sensor Stalk Thread

Stalk Servo

Drive Motors
Multi-threaded finite state machine control systems

- Controller FSM
- Obstacle Sensors Thread
- Image Compute Thread
- Sensor Stalk Thread
- Mapping Thread
- Drive Motors
- Short IR + Bump
- Camera
- Stalk Sensors

Stalk Servo
Finite state machines can combine the model-plan-act and emergent approaches and are a good starting point for your Maslab robotic control system.
Outline

• High-level control system paradigms
  – Model-Plan-Act Approach
  – Behavioral Approach
  – Finite State Machine Approach

• Low-level control loops
  – PID controller for motor velocity
  – PID controller for robot drive system

• Examples from past years
Team 15 in 2005 used a map-plan-act approach (well at least in spirit)

Multiple runs around a mini-playing field

Odometry data from exploration round of contest
Team 10 in 2003 used odometry so Bob could retrace his steps and return home.
Team 4 in 2005 used an emergent approach with four layered behaviors

- **Boredom**: If image doesn’t change then move randomly
- **ScoreGoals**: If image contains a goal the drive straight for it
- **ChaseBalls**: If image contains a ball then drive towards ball
- **Wander**: Turn away from walls or move to large open areas
Team 12 in 2004 learned the hard way how hard building a controller can be!
Take Away Points

• You cannot just hack together a robot controller, you must **design** a robot controller

• Design simple, module behaviors and then decide how to compose these behaviors to achieve the desired task

• Simple **finite state machines** make a solid starting point for your Maslab control systems

• Integrating **feedback** into your control system “closes the loop” and is essential for creating robust robots