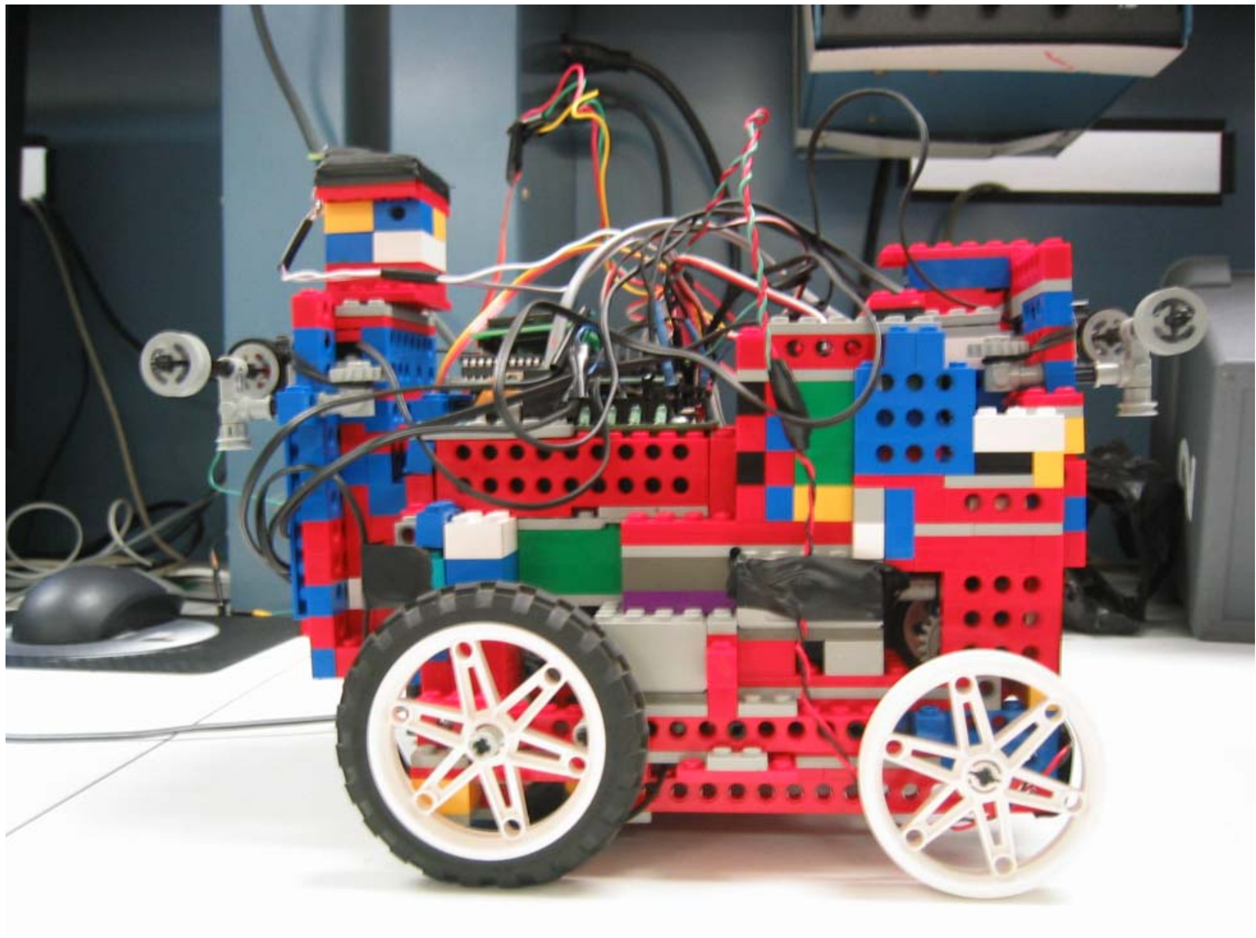
A Multithreaded Rule-Based Architecture for Massive Robot Control Systems

Chris Morgan
University of Southern California
Viterbi School of Engineering
CSCI 445: Introduction to Robotics

RoboCup Team Members:
Brent Nash
Lino Manansala
Carlos Espinoza
Siu Yu Kwok
Gautam Dandavate



Overview – Rule-Based Architectures

Rule-based architectures are very common in modern robotics because of their short development time and modularity. Based on a hierarchy of rules which determine the behavior of the robot, these architectures allow for the development of individual behaviors that can then be combined to perform complex tasks. Rule-based architectures can also be very robust when the behaviors are treated as discrete modules and are very loosely coupled with each other. One disadvantage of a rule-based architecture is the lack of run-time flexibility for rule priorities and conditions.

Example – A Rule-Based Multithreaded Controller for Striker Control

This paper will be focused on the example of a soccer-playing striker robot which uses a variant of a rule-based control architecture as its controller. The controller was implemented in Interactive C on the Handyboard microprocessor. The implementation is multithreaded, using modified semaphores for thread communication and signaling. Due to threading limitations of the microprocessor, the architecture is a blend of parallel layers and sequential execution. This interesting mix will be discussed further later in the paper.

The requirements for the striker robot are simple:

1. Find the soccer ball
2. Avoid obstacles (static & dynamic)
3. Maneuver towards the opposing goal
4. Shoot the ball into the opposing goal

Based on these requirements, an architecture was created using the following layers:

1. WANDER
2. AVOID
3. ACQUIRE_BALL
4. ATTACK
5. SHOOT

A diagram of these layers and their interaction can be seen in Figure 1.

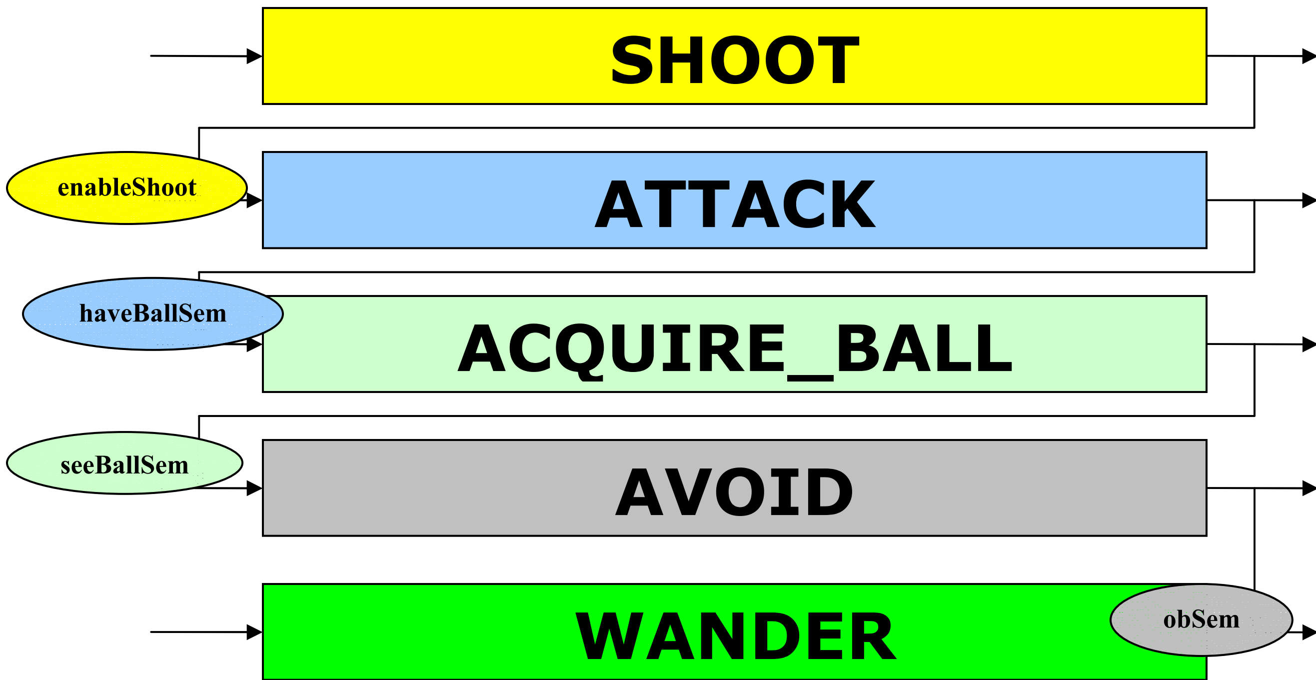


Figure 1: Architecture Overview

WANDER

The WANDER layer runs constantly, and takes control whenever the robot cannot see the ball and is not avoiding obstacles. The wandering algorithm is relatively simple because the range of the camera is large enough to cover the entire field from nearly every position. The striker spins in a circle until a timer times out, then randomly travels in a different direction for a short time and resumes spinning. This process can be overridden by any of the other processes using any semaphore (see Listing 1).

Listing 1: Wandering layer

```
void wander()
{
    int i = 0;

    while(1)
      {
        // turn in a circle while I don't see the ball and I'm not overidden
        while(!seeBallSem && !obSem && !haveBallSem && !enableShoot)
          {
            if(ballReading > HAVE_BALL)
              {
                printf("\n\nI got it!"); beep();
                haveBallSem = 1;
                reverse(50);
                msleep(900L);
                attack();
            }

            forward(80);
            sleep(2.0);

             // make sure I haven't turned in a complete circle yet
            for(i = 0; i < 50; i++)
              {
                if(!seeBallSem && !obSem && !haveBallSem && !enableShoot)
                  {
                    turnLeft(18.0);
                    msleep(250L);
                    if(ballReading > HAVE_BALL)
                      {
                        printf("\n\nI got it!"); beep();
                        haveBallSem = 1;
                        reverse(50);
                        msleep(900L);
                        attack();
                    }
                }
                // if I see the ball, break out
                else
                  {
                    break;
                }
            }

            // randomly turn, move and repeat
            if(random(2) == 0 && !seeBallSem && !obSem && !haveBallSem && !enableShoot)
              {
                turnTo(SHOOTING_DIRECTION);
            }
            else
            {
                 if(!seeBallSem && !obSem && !haveBallSem && !enableShoot)
                {
                       turnTo(DEFENDING_DIRECTION);
                 }
            }

            if(ballReading > HAVE_BALL)
              {
                printf("\n\nI got it!"); beep();
                haveBallSem = 1;
                reverse(50);
                msleep(900L);
                attack();
            }

            forward(100);
            msleep(1400L);
            off(LEFT_MOTOR);
            off(RIGHT_MOTOR);
        }
    }
}
```

AVOID

The AVOID layer has the capability to override the wandering layer. Using the four bump sensors distributed around the robot's chassis, this layer detects walls and other robots and ensures that the striker can maneuver away from them. Using the semaphore **obSem**, this process alerts the wandering process to pause execution while evasive action is taken (see Listing 2).

Listing 2: Obstacle Avoidance

```
void avoidObstacles()
{
    // number of seconds to go away after hitting an object
    float avoidTime = 1.0;

    while(1)
      {
        if(!haveBallSem)
          {
            if(digital(LF_BUMP))
              {
                hog_processor();
                obSem = 1;
                reverse(100);
                sleep(avoidTime);
                turnRight(45.0+((float)random(45)));
                forward(90);
                sleep(avoidTime/2.0);
                forward(0);
                obSem = 0;
                defer();
            }
            if(digital(RF_BUMP))
              {
                hog_processor();
                obSem = 1;
                reverse(100);
                sleep(avoidTime);
                turnLeft(45.0+((float)random(45)));
                forward(90);
                sleep(avoidTime/2.0);
                forward(0);
                obSem = 0;
                defer();
            }
            if(digital(LB_BUMP))
              {
                hog_processor();
                obSem = 1;
                forward(100);
                sleep(avoidTime);
                turnLeft(45.0+((float)random(45)));
                reverse(100);
                sleep(avoidTime/2.0);
                forward(0);
                obSem = 0;
                defer();
            }
            if(digital(RB_BUMP))
              {
                hog_processor();
                obSem = 1;
                forward(100);
                sleep(avoidTime);
                turnRight(45.0+((float)random(45)));
                reverse(100);
                sleep(avoidTime/2.0);
                forward(0);
                obSem = 0;
                defer();
            }

        }
    }
}
```

ACQUIRE_BALL

The ACQUIRE_BALL layer is activated when the camera sees the ball. The camera takes pictures at a constant 17 fps regardless of the robot's state. When the camera sees something that looks reasonably like a soccer ball, it sets the **seeBallSem** semaphore to signal the ACQUIRE_BALL process to activate and get the ball. This process lines up the ball in front of the dribbler and heads toward it, adjusting course on the way to keep the ball centered. This process is deactivated when the **haveBallSem** is activated, signaling that the striker has the ball in its control (see Listing 3).

Listing 3: Acquiring the ball

```
void acquireBall()
{
    int counter = 0;
    int i=0;

    while(1)
      {
        if(!haveBallSem && !obSem)
          {
            if (track_confidence > MIN_CONFIDENCE)
              {
                hog_processor();
                seeBallSem = 1;
                resetBallSightTimer();

               // while we don't have the ball, center the ball in the FOV and move towards it
               // incrementally
                while(!haveBallSem)
                  {
                    // the ball is to our right
                    if(track_x > 10)
                      {
                        while(track_x > 10 && !obSem && !haveBallSem &&
                                  track_confidence > MIN_CONFIDENCE)
                          {
                            turnRight(10.0);
                            lastBallDirection = RIGHT;
                            msleep(100L);

                            if(ballReading > HAVE_BALL)
                              {
                                printf("\n\nI got it!"); beep();
                                haveBallSem = 1;
                                attack();
                                break;
                              }
                          }
                      }

                    // the ball is to our left
                    else if(track_x < -10)
                        {
                          while(track_x < -10 && !obSem && !haveBallSem &&
                                  track_confidence > MIN_CONFIDENCE)
                            {
                              turnLeft(10.0);
                              lastBallDirection = LEFT;
                              msleep(100L);
                              if(ballReading > HAVE_BALL)
                                {
                                  printf("\n\nI got it!"); beep();
                                  haveBallSem = 1;
                                  attack();
                                  break;
                                }
                            }
                        }
```

```
                    // the ball is in front of us
                    else
                      {
                        while(track_x >= -10 && track_x <= 10 && !obSem && !haveBallSem &&
                                track_confidence > MIN_CONFIDENCE)
                          {
                            forward(60);
                            for(i=0; i < 10; i++)
                              {
                                if(ballReading > HAVE_BALL)
                                  {
                                    printf("\n\nI got it!"); beep();
                                    haveBallSem = 1;
                                    attack();
                                    break;
                                  }
                              }
                          }
                      }
                }

                hog_processor();
                resetPickupTimer();
                while(pickupTimer() < 3.5)
                  {
                    forward(65);
                    if(ballReading > HAVE_BALL)
                      {
                        printf("\n\nI got it!"); beep();
                        haveBallSem = 1;
                        attack();
                        break;
                      }
                  }
            }
            else
              {
                seeBallSem = 0;
              }
        }
        else
          {
            seeBallSem = 0;
            defer();
          }
    }
}
```

ATTACK

The ATTACK layer is activated from within the ACQUIRE_BALL layer due to threading constraints. When the **haveBallSem** is set, ATTACK is called. ATTACK turns the robot towards the opposing goal using the compass heading, and charges towards the goal. Based on a probabilistic function (due to optosensor unreliability), the ATTACK process invokes the SHOOT process when it determines that the robot is in a shooting position (see Listing 4).

Listing 4: Attack

```
void attack()
{
    int drive_time = 0;
    int ball_sense = 0;

    seeBallSem = 0;

    if(attackInUse != 0)
      {
        return;
    }

    attackInUse = 1;

    if(!obSem)
      {
        if(haveBallSem)
          {
            hog_processor();
            hog_processor();
            hog_processor();

            turnTo(SHOOTING_DIRECTION);
            drive_time = random(1500) + 3000; //BRENT: tune time and speed
            forward(70);
            msleep((long)drive_time);

            enableShoot = 1;
            shoot();
            attackInUse = 0;
            haveBallSem = 0;
            seeBallSem = 0;
        }
         // if we don't have the ball, abort the attack and recommence wandering
        else
          {
            haveBallSem = 0;
            seeBallSem = 0;
            attackInUse = 0;
          }
       }
      return;
}
```

SHOOT

This layer is invoked from the ATTACK layer, and simply runs a shooting routine that flicks the ball diagonally forward. The SHOOT layer then returns control to the ATTACK layer, which cleans up the semaphores so that control can be assumed by the WANDER process (see Listing 5).

Listing 5: Shooting

```
void shoot()
{
    if(!obSem && haveBallSem && enableShoot)
      {
        // make sure we don't get interrupted
        hog_processor();
        hog_processor();
        hog_processor();
        hog_processor();

        // bend it like beckham ;)
        printf("\nfire one!!!");
        servo0 = SERVO_RETRACTED;
        msleep(200L);
        off(DRIBBLE_MOTOR);
        servo0 = SERVO_ANGLE;
        sleep(0.25);
        servo0 = SERVO_RETRACTED;
        forward(0);

        motor(DRIBBLE_MOTOR, 100);
        enableShoot = 0;
    }
}
```

Miscellaneous Functions & Notes

Localization

Localization was achieved using a digital compass which used PWM to send the current heading to the HandyBoard. The compass was polled after the camera took each picture to avoid the interference that originally plagued both subsystems (see Listing 6).

Listing 6: Localization

```
/*
By Andrew Chanler - Fall 2003

Handyboard driver for using Devantech Magnetic Compass - CMPS03
7refer to http://www.cs.uml.edu/~achanler/robotics/
*/
// returns the heading of the robot to the nearest degree

int compass()
{
    int x;
    float z;
    int y;
    float degree;

    x = compass_driver(0);
    z = ((float) ( 0x0F & x) )+( 256.0*((float) ( x >> 8) )) ;

    if((z < 7440.0)&&(compass_wrap))
      z+=65536.0;

    degree = ((z - 1792.0)*360.0)/71183.0;
    y = (int) (degree+.5);
    y = y % 360;
    return y;
}
// takes pictures and localizes constantly
// the robot can't take pictures and use the compass at the same time
// so they must be coordinated in the same function
void takePicture()
{
    while(1)
      {
        track_orange();

        if(track_confidence > MIN_CONFIDENCE)
          {
            resetBallSightTimer();
            seeBallSem = 1;
        }
        else
          {
            seeBallSem = 0;
        }

        currentHeading = compass();

        ballReading = analog(BALL_SENSOR);
    }
}
```

Driving & Steering

Utility functions for forward, reverse, and left and right turns were implemented for code readability. A function that turned to a specific heading was also written to abstract away the need for the ATTACK process to know how the motors work. This function was optimized to turn the shortest distance in order to attain the desired heading (see Listing 7).

Listing 7: Turn to heading

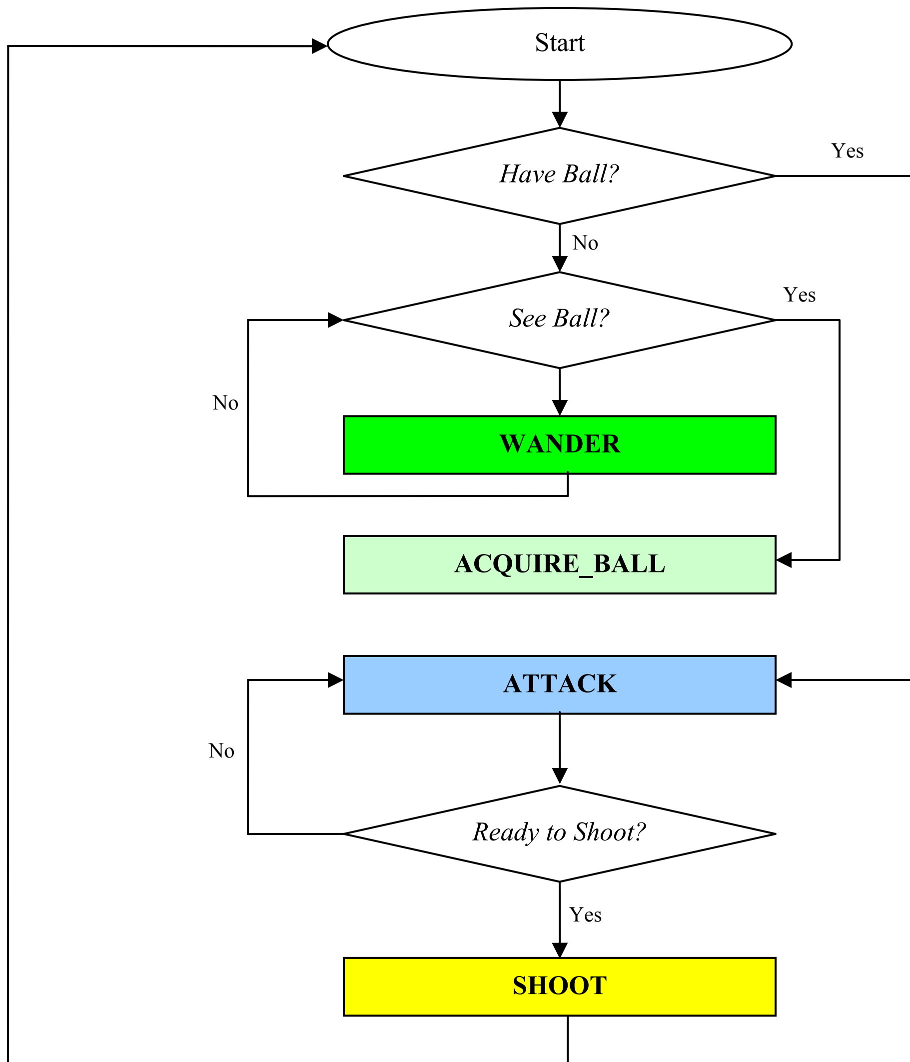
```
// turns to a specific heading +/- 12 degrees
void turnTo(int desiredHeading)
{
    // am I already facing in the desired direction?
    if((currentHeading + 12) % 360 > desiredHeading && (currentHeading - 12) % 360 < desiredHeading)
      {
        printf("\nfinished turnTo\n");
        return;
    }

    // is it easier to turn left?
    if((currentHeading + 180) % 360 < desiredHeading)
      {
        while((currentHeading < (desiredHeading - 12) % 360 ||
                 currentHeading > (desiredHeading + 12) % 360))
          {
            if(!seeBallSem)
              {
                turnLeft(15.0);
                msleep(200L);
            }
            else
              {
                return;
            }
        }
        printf("\nfinished turnTo\n");
        return;
    }

    // is it easier to turn right?
    if((currentHeading + 180) % 360 > desiredHeading)
      {
        while(currentHeading < (desiredHeading - 12) % 360 ||
                 currentHeading > (desiredHeading + 12) % 360)
          {
            if(!seeBallSem)
              {
                turnRight(15.0);
                msleep(200L);
            }
            else
              {
                return;
            }
        }
        printf("\nfinished turnTo\n");
        return;
    }
}
```

Layer/Sequence Combination

Due to the thread limitations of the HandyBoard, it was necessary to combine some layers in order for the code to perform as required. Figure 3 shows the final flowchart for the layered sequenced controller.



NOTE: This flowchart does not take into account the obstacle avoidance behavior. The robot will avoid obstacles whenever possible, and adding this feature to the flowchart would decrease readability. See Figure 1 for an explanation of obstacle avoidance.

Figure 2: Layer/Sequence Flowchart

Conclusion

Using a rule-based architecture as the controller for a soccer-playing robot increases modularity and robustness while simplifying the development process and dramatically shortening the software development time. The robustness and modularity were shown when this robot lost an optosensor used for localization on the floor gradient. The team was able to compensate by simply disabling the floor-based localization layer for the remainder of the tournament. If the HandyBoard's threading system was not so limited, this architecture would have morphed into a subsumption-style architecture, with each layer running in parallel in a different thread.