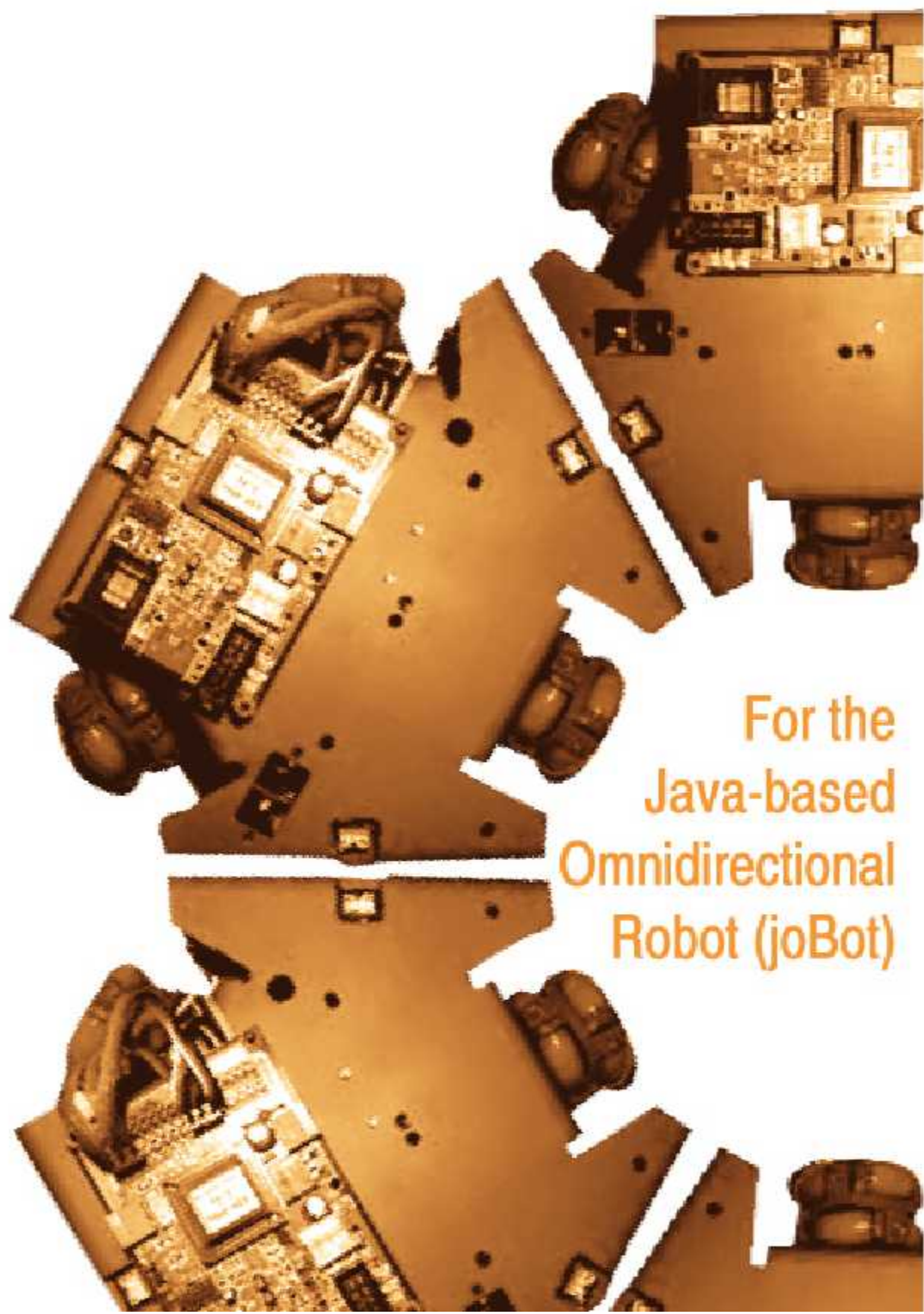




MULTI MOTIONS
interactive animatronics

UVMdemo

DEMO PROGRAM



For the
Java-based
Omnidirectional
Robot (joBot)

Table of Contents

Chapter 1 Introduction.....	3
1.1 UVMDemo Demonstrations.....	3
1.2 Behaviors.....	3
1.3 Changing behaviors.....	4
1.3.1 <i>Finite State Machines</i>	5
1.3.2 <i>The Command Interface</i>	6
1.3.4 <i>The DIP switches interface</i>	7
1.3.5 <i>The UVMdemo behaviors</i>	8
Chapter 2 JoBot UVM Demo.....	10
2.1 joBot UVMdemo program.....	10
2.1.1 <i>The main line</i>	10
2.1.2 <i>Periodic Timers</i>	11
2.1.3 <i>The HeartBeat</i>	12
2.1.4 <i>Web Service interface</i>	12
2.1.5 <i>Drive function</i>	12
2.1.6 <i>Vector Drive function</i>	13
2.1.7 <i>Servo linearization</i>	13
2.1.8 <i>Calculating distance</i>	14
2.1.9 <i>State Machines</i>	15
2.1.10 <i>getState, setState and reportState</i>	16
2.1.11 <i>PeriodicTimer</i>	16
2.1.12 <i>JobotBaseController.java</i>	16
2.1.13 <i>The Behavior class</i>	17
2.1.14 <i>Making changes to the code</i>	18
2.1.15 <i>Programming joBot</i>	19
2.2 Additional Demonstrations.....	19
Chapter 3 Rescue Demo	20
Chapter 4 Soccer Demo	21
4.1 Percepts.....	22
4.2 States	23
4.3 Behaviors.....	23
4.4 Calibration.....	24
Chapter 5 Speech Demo	26
Chapter 6 Mouse Demo	28

Chapter 1 Introduction

To build an autonomous robot, a program is needed that implements a number of behaviors and criteria that select from this repertoire of behaviors.

The UVMdemo program is a simple program that demonstrates how to program certain behaviors and how to make a robot select and change behaviors. There are a number of these demonstration programs of which UVMdemo is the basic version. This simple version explains much of the basics, the other demonstrations deal with more advanced topics like:

- RoboCup Rescue behavior, demonstrating a line follower robot
- RoboCup Soccer, demonstrating a soccer playing robot
- Mouse Sensor demonstrates the use of the JoBot mouse sensor
- Speech Processing demonstrates a simple speech processing program

This manual reflects the situation of the JoBot Simulator version 1.0.25

1.1 UVMdemo Demonstrations

The UVMdemo programs consist of a completely self-contained package that contains everything that is needed to run a JoBot. A sound software engineering principle is to share code between modules and optimize the architecture of software components as part of a whole structure.

Since the demonstrations must run both inside the simulation environment but must also run stand-alone in a JoBot, care has been taken to not integrate the functionality of the demonstrations with other software. In this way, the entire package may be taken out of the simulation environment and may be run on its own.

This of course has the disadvantage that some duplication occurs between the various demonstrations. However if we would not do this, separating out the modules for a certain demonstration would create the situation that parts of the software that runs inside a JoBot could be scattered all over the modules of the simulation environment. In the current setup you can be certain that ALL software for a demo that is to be loaded into a JoBot's memory can be found inside the package you are working with, without any outside references, except for the standard Java and muVium components.

1.2 Behaviors

Behaviors are small routines that perform a certain function. They are normally executed continually until an external condition makes it necessary for the robot to change its behavior. As an example we will look at a soccer playing robot as is used in the RoboCup Junior tournament.



A simple soccer-playing robot will first need to find the ball, then move towards the ball, then move the ball towards the goal and finally kick the ball into the goal. In a simple robot we thus have the following behaviors:

1. *Find the ball.* This could be done with a simple sensor. In the RoboCup Junior games an active infrared ball is used for this purpose. In other RoboCup leagues cameras are used to analyze the image and determine where the ball is. Such a behavior usually returns when the ball is found and provides the coordinates where the ball is found. If not found it returns an indication that no ball was spotted. In many cases finding the ball involves rotating around its own axis to look around for the ball.
2. *Move towards the ball.* After the robot has determined where the ball is, it will move into the direction of the ball. During this behavior the robot continually needs to check if it has reached the ball or if the ball has move to a different location in the meantime.
3. *Move ball towards goal.* When the robot has possession of the ball it will dribble with it and moved into the direction of the goal. During this behavior it needs to check if it still has the ball. This is a complicated behavior since it has to move into a certain direction while at the same time checking for obstacles and making sure that the ball was not lost.
4. *Kick the ball.* The robot needs to make a kicking movement. Pushing is not allowed in RoboCup so it needs to make a quick movement to get the ball into the goal. In many cases the robot will be equipped with a kicker or some rotating mechanism that keeps the ball near the robot and when reversing the rotation, kicking the ball away.

During each of these four basic behaviors external factors could influence the actions and make it necessary for the robot to change behaviors. A real RoboCup program is of course much more complicated. The robot needs to know where it is but also needs to avoid obstacles like opponents and walls. When the robot gets into a corner it needs to be able to escape from it.

So behaviors are executed continually until some external factor determines that a different behavior is needed.

In the UVMdemo program behaviors are coded in the `behavior` class. Each behavior has its own class.

1.3 Changing behaviors

To change behavior, program code outside of the behavior needs to check if certain conditions arise which make it necessary to change the current behavior. This kind of control is normally programmed in the form of a finite state machine (FSM) where the program keeps track of the current state of the system.

A state thus corresponds with a certain temporary kind of behavior. If the robot is in the 'dribbling' state it is executing the behavior of *moving with the ball*.

A higher level control module in the robot software takes care of determining the desired state and when needed, a state change will take place. When designing a robot it is common practice to first draw a state-change-diagram in which is depicted how state changes usually take place.

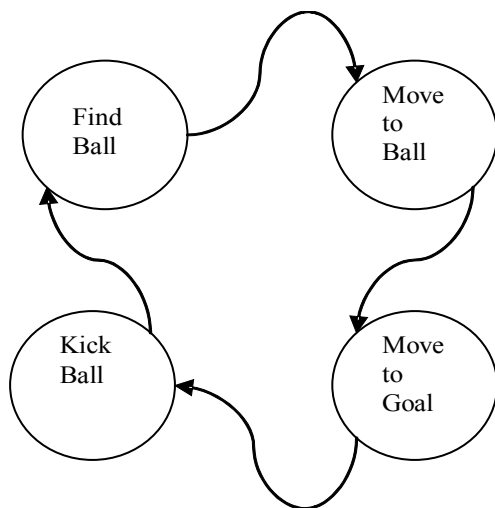
The UVMdemo program can change states in the following manner:



1. By changing the DIP switches on the robot or in the simulator. In the demo program the DIP switches directly correspond with the current state of the robot. This is by far the simplest and least sophisticated way to change behaviors.
2. By external control, using the WebService interface. With this interface commands may be sent to the robot from the simulator. One of the commands is a setState. This allows control over behavior by an online user.
3. Under program control. The basic demo program does not contain any state change under program control, this is left as an exercise. You could easily create a new behavior that checks the status of for instance the sensors and based on the value, select a new behavior. More advanced state control mechanisms can be found in the Rescue and Soccer demonstrations.

1.3.1 Finite State Machines

Finite State Machines are commonly used to implement systems that perform certain actions in a pre-defined sequence.



Usually the designer of a FSM based system will start by designing a state transition diagram in which is indicated which states exist and what state transitions are considered legal. Generally speaking state transitions do not take place at random but have to adhere to certain logical rules, depending on the type of problem. To illustrate the very simple soccer-playing robot could have the following state transition diagram:

1.3.2 The Command Interface

Every UVMagent is required to implement the command interface that is using the muVium Web Service interface. With this interface simple commands may be sent to the agent and it may respond by sending back data.



Agent selection

DIP switches

DIP switch readings

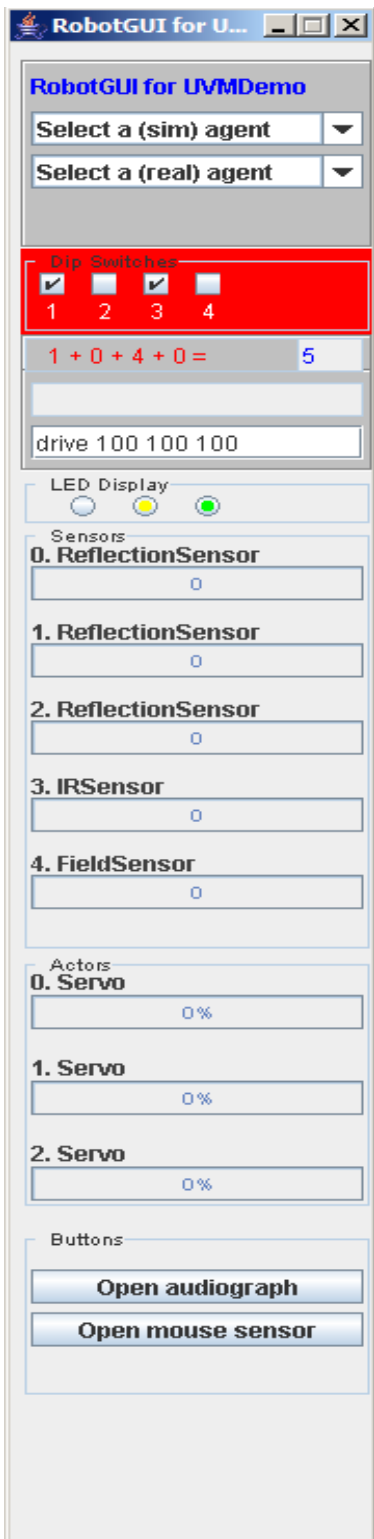
Web Interface StatusLine

Web Interface CommandLine

LEDs

Sensors

servos



The interface is titled 'RobotGUI for UVM Demo'. It features two dropdown menus for 'Select a (sim) agent' and 'Select a (real) agent'. Below these is a red 'Dip Switches' section with four switches (1, 2, 3, 4) and a calculation '1 + 0 + 4 + 0 = 5'. A 'Web Interface CommandLine' field contains 'drive 100 100 100'. The 'LED Display' section shows three indicator lights (red, yellow, green). The 'Sensors' section lists: 0. ReflectionSensor (0), 1. ReflectionSensor (0), 2. ReflectionSensor (0), 3. IRSensor (0), and 4. FieldSensor (0). The 'Actors' section lists: 0. Servo (0%), 1. Servo (0%), and 2. Servo (0%). At the bottom are two buttons: 'Open audiograph' and 'Open mouse sensor'.

1.3.3

This interface is only important if the simulator is used to send commands. When you write your own Web Service client application you may implement other interfaces.

We are assuming here that the default muVium Web Service interface is used throughout the examples.



Commands are typed into the commandLine of the RobotGUI in the form of a command code and a maximum of 3 numeric parameters. This information is sent to the agent and it will respond with appropriate actions.

Currently the following commands are provided:

1. **Drive x y z** – Sends data to the X, Y and Z servo values and activates these. This allows the servos to be set by a command. Make sure the agent is not overwriting this information so a mode must be selected in which the agent is silent. In the joBot Junior version the Z motor is used to drive the (optional) dribbler.
2. **VectorDrive x y r** – Sends the X and Y direction of travel to the agent and the Rotation of the body. Once the command is received it is executed.
3. **Sensor x** – Returns the current value of sensor X. The value is shown in the StatusLine above the commandLine.
4. **Action x** – Sends an action command to the agent. This is a single byte code that may be interpreted by the agent in any way the programmer wants.
5. **Report x** – Sets the reporting level in the robot to X. This allows the robot to selectively send reporting messages to the output device so the user can determine what is going on while maintaining control over the amount of output. This feature only works if you implement a reporting function in your program. Unless this is done, no action will be taken.
6. **Start** – Restarts execution of the program after issuing a stop command.
7. **Stop** – Stops execution of the program.

These commands are only executed by an UVMagent when the proper Web Service interface has been implemented. See the demo programs for examples on how to implement these features.

For more information about the Drive and VectorDrive settings see the “Hardware” section in the joBot manual.

The WebService interface may also be used by other programs that are able in this manner to control the robot externally under program control. A PC based program could in this way determine the actions of a remote robot.

1.3.4 The DIP switches interface

The JPB board has a DIP switch that may be read by the program. The value of the switch consists of a nibble (4 bits) and is used for mode settings or other facilities.

In the simulator, the position of the DIP switches is set through the RobotGUI

The switches are numbered from left to right as 1,2,3,4. Any combination may be made, where the value of the switches is read with 1 and 2 representing the values 1 and 2 and switches 3 and 4 representing the values 4 and 8.

The value represented in the picture above is $1 + 4 = 5$.

1.3.5 The UVMdemo behaviors

The UVMdemo program that is loaded in the JPB's memory when delivered, uses the DIP switches to select a behavior. The following functions are provided:



- 0 = STATE_IDLE
- 1 = STATE_CALIBRATE
- 4 = STATE_CURIOUS
- 5 = STATE_FLEE
- 8 = STATE_WALL_HUG
- 11 = STATE_CHASE_BALL

By selecting the switches the current behavior may be changed. For instance, to select the YOYO functions, select switch 1 and 4 or 2 and 4.

0 = STATE_IDLE

In the IDLE state the servos are put in the 0 position and the lamps are off. Only the heartbeat lamp is blinking. In this state commands may be sent from the user interface to allow online testing. No behavior is running in this state and only the heartbeat is active.

1 = STATE_CALIBRATE

In CALIBRATE mode all servos are set into the 0 position and in addition the sensors are read. When a sensor has a value higher than 128, output is generated that shows the value read, using the serial port.

In addition a lamp is turned on, RED for the 0 sensor and YEL for the left sensor 2 and GRN for the right sensor 1. These colors correspond with the position of the sensor, YEL is left and GRN is right.

A cutoff value is used that reacts to objects at a distance of about 20 cm to make testing easier.

4 = STATE_CURIOUS

This state is not implemented and is included to allow adding a new behavior. It is left as an exercise for the user to implement the CURIOUS behavior which is the opposite of the FLEE behavior.

5 = STATE_FLEE

In the FLEE state, the robot continually reads all three sensors. As soon as one of the sensors detects an obstacle, it will activate the motors, opposite to the sensor to drive away from the obstacle. When two sensors measure an obstacle at the same time, only the one with the lowest number will react. When no more obstacles are detected, all motors will stop.

Like in the TEST mode, the leds will show which sensor detects an obstacle. Green is right, yellow is left and red is the middle sensor. The red LED is shared with the heartbeat and may therefore either blink or light continuously.

The sensitivity of the sensor is here set at about 40 cm and the motors are always set to 100%. It would be interesting to make a version that reacts proportional to the distance of the object.



8 = STATE_WALL_HUG

joBot reads sensor 1 continuously and when it detects an obstacle it tries to maintain a constant distance to the obstacle while moving forward. This is essentially a single sensor wall follower.

More complicated wall following behavior is possible, but this is left to the user as an exercise. An example would be a joBot that first drives in a straight line until it detects an obstacle. That could be any of the sensors. It would then turn in such a way that two of its sensors detect the wall. Using the two sensors it keeps the robot parallel to the wall.

11 = STATE_CHASE_BALL

This state is only partly implemented and is also meant as an exercise. Its main goal is to first find the ball and then chase it around, using the standard distance sensors. More advanced behaviors may be built, using additional sensors, allowing joBot to play a very simple game of soccer.

Chapter 2 JoBot UVM Demo

2.1 joBot UVMdemo program

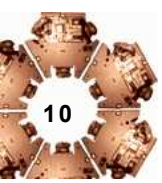
The UVMdemo program can be invoked by the simulator through the menu selection Insert > Robot Demo > UVMdemo and is implemented by the class `javaBot.UVM.UVMdemo.java`. This class extends `com.muvium.UVMRunnable` so it can be used as an agent for a real joBot. In fact all explanations in the manuals assume that this particular class is used to program a real joBot. It also implements `javaBot.UVM.AgentWebService` so it can be accessed through the muVium Web Service interface. As the name indicates, `UVMRunnable` implements the standard Java interface `java.lang.Runnable` and so it is called to start execution through the `run()` method, which is comparable with the `main()` method in standard Java programs.

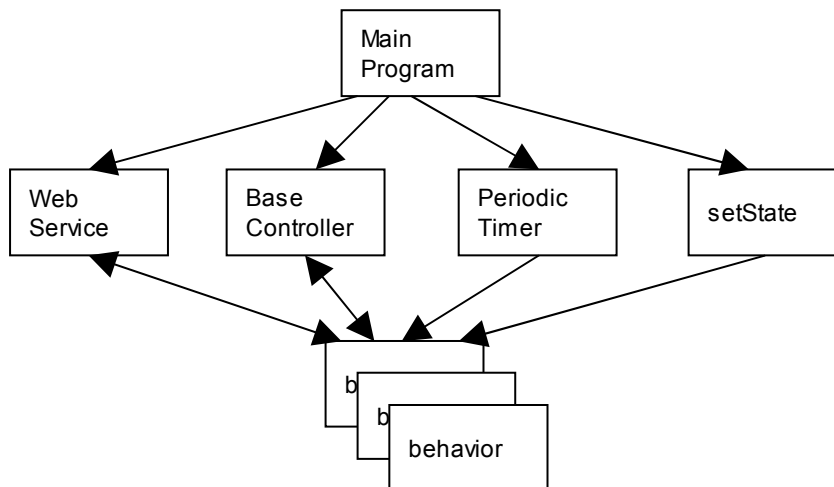
2.1.1 The main line

The UVMdemo class takes care of initialization and defines all system variables and constants. The main method is `run()` like in a normal `java.lang.Runnable`. In this method the port settings for port B or E (depending on the type of processor board) are set to allow the LEDs and DIP switches to operate. Then two `periodicTimers` are created, one for the heartbeat and another one for the behavior ticks. These elements are discussed in more detail later on. It then starts the timers and the clock ticks of each of the timers determine when the actual processing will take place.

The UVMdemo class also contains the WebService interface which allows the simulator to send commands to the program. In this way execution may be influenced under control of an external program.

The BaseController (`javaBot.UVM.JobotBaseController`) is a separate class that contains most of the support functions. An instance of this class is created and may be referred to from all behaviors.





The periodic Timer determines when a clock tick is generated and a certain behavior is executed. The `setState()` method determines what the current state of the robot should be.

The main work is done inside the behavior modules. Each behavior is a separate class and is selected by `setState()`. Once a behavior is running it continues running until it is stopped again by `setState()`.

In order to run software on joBot or in the simulator, the user program must adhere to a number of standards.

Every muVium program must inherit from `UVMRunnable` in order to run on a muVium controller. It also needs to implement the `AgentWebService` in order to use the muVium Web Service interface that is provided by the simulator.

```
public class UVMdemo extends UVMRunnable implements AgentWebService,
TimerListener {}
```

2.1.2 Periodic Timers

A periodic timer is a self running timer that is created during setup and is given a fixed interval. When a timer event occurs, a listener gets control and executes the timer event. In the UVMdemo there are two of such timers, the heartbeat event and the `behaviorServiceTick`.

The `behaviorServiceTick` transfers control to the currently running behavior and executes one cycle of the action during that event. The timer thus determines how frequently a behavior is executed.

In a robot the speed with which the sensors are read and also when the actuators are enabled, determines the reaction speed. In a slow robot like the joBot which is able to move at a speed of about 8 cm/sec a high reaction speed is not very useful, so it is set at 1 Hz. For some other applications however a speed of 10 Hz might be more appropriate.

Remember that both the speed of the processor and of the simulator must be large enough to handle the load on the processor, generated by the periodic timers. Although the simulator generally runs on a high-speed PC, the amount of overhead of a low-level simulator is considerable and could turn out to be lower than that of the actual UVM device.



2.1.3 The HeartBeat

The heartbeat of a robot is generated to show that the program is still alive. In the UVMdemo program it blinks at a rate of once every second. Actually it flashes on every half second and off every half second also.

2.1.4 Web Service interface

The Web Service interface allows the program to accept commands from the simulator and execute them while having an online connection with the simulator. For this the following functions need to be implemented:

- `getSensor(int sensor)` returns the value of the given sensor
- `getState()` returns the current state of the robot.
- `setState (int value)` forces the robot into the given state. Always make sure that there is a default state in case a value is entered for which there is no defined behavior.
- `ReportState (int level)` tells the robot if continuous status feedback must be sent to the Sysout device. This way debugging output from the robot may be sent to the simulator, using the serial communication link.
- `drive(int x, int y, int z)` sets the values of the three servos.
- `vector(int x, int y, int rot)` makes the robot move in the given direction with a rotation specified.

This interface is realized in the robotGUI of the simulator and allows you to enter commands that are sent to the JoBot directly. Using the webservice interface new commands may be implemented as well.

2.1.5 Drive function

This function is provided by the class `javaBot.UVM.JobotBaseController`

```
drive (s0, s1, s2)
```

This function drives each of the three servos according to the setting of the corresponding parameter. The servos are numbered counterclockwise starting with the wheel opposite the JPB

Normal rotation (as indicated by the arrows in the diagram) is indicated by a number between 1 and 100.

Reverse rotation is indicated by a number between -1 and -100.

0 stops rotation of the servo.

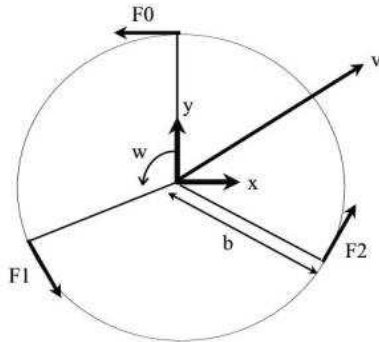
The number is an indication for the speed of the servo. Since the used servo servos are not linear, the software has to compensate for this and calculates the drive factor that has to be given to the servo. There is especially a difference between the forward and backward direction. -80 does not rotate as fast as 80 and can therefore create problems when trying to drive a straight line.

This needs to be compensated for in software and is depending on the used servo type. See the linearization functions for more details.



2.1.6 Vector Drive function

This function also is provided by the class `javaBot.UVM.JobotBaseController`



With the drive function you have to specify the exact motions for all three wheels. Although that is fine for fixed movements, when a trajectory needs to be calculated it is much simpler to allow the definition of a direction.

`vectorDrive` simplifies these tasks and takes three parameters that allow the definition of direction, speed and rotation of the robot:

```
vectorDrive (X, Y, Rot);
```

The X and Y coordinates specify a vector that in essence is a direction in standard Euclidian coordinates, where the Y axes points in the direction of servo 0, the servo opposite the JPB board.

The numbers indicate the relative speed, so the following specifications are:

```
vectorDrive (0,0,0);           // Stand still
vectorDrive (100, 0, 0);       // Move right (positive X) at full speed
vectorDrive (-100, 0, 0);      // Move left (negative X)
vectorDrive (0, 100, 0);       // Move forward (positive Y) at full speed
vectorDrive (0, -100, 0);      // Move backward (negative Y)
vectorDrive (70, 30, 0);       // Move toward v full speed
vectorDrive (35, 15, 0);       // Move toward v approx. half speed
```

The third parameter specifies the rotation speed. At 0 the robot will maintain the current position. At 100, the robot will rotate along its own axis.

```
vectorDrive (0, 0, 100);       // Spin in same position
vectorDrive (0, 100, 50);      // Move forward rotating clockwise
```

2.1.7 Servo linearization

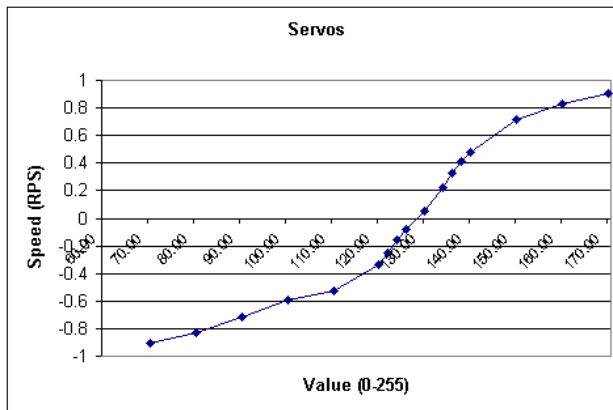
Small RC servos are used to drive joBot. They allow control of speed and direction, are small and affordable. Several types have been tested and in joBot the standard Futaba S3003 servos are used. The JoBot Junior uses the Futaba S148 servo.

A disadvantage of these kind of servos is that their speed is not proportional to the pulse value that is given in a value ranging from -100 to +100.

The graph underneath shows the pattern that is true for several of these servo types.

In many cases, fixed values are used and the non-linearity is no issue. However when using the `VectorDrive` function, the rotational speed of all servos is calculated and should be linear in order to function properly.





The simulator takes this non-linearity of the servos into account. Therefore when you are developing agents you have to deal with it.

The `drive` and `vectorDrive` methods use the table below to convert the values forward and backward. This is still not 100% correct. The muVium controller cannot handle doubles and therefore some precision is lost because of rounding errors.

<i>method parameter</i>	<i>servo command</i>
0	1
13	4
25	5
38	8
50	9
63	10
75	11
88	14
100	34

So if you use these methods, you do not have to make any corrections yourself. Please note that this table is valid for the S3003 servo. Other servo types are used in other robots and each servo needs to implement its own conversion table. Details can be found in the servo definitions for the various robots in the demonstration programs.

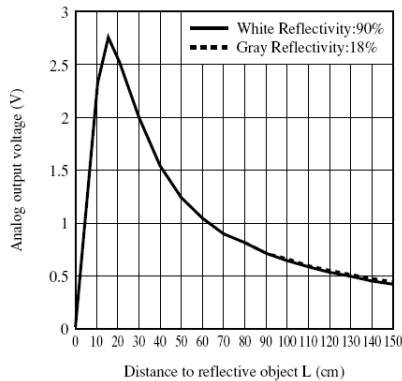
2.1.8 Calculating distance

The Sharp GP2D12 infrared triangulation sensors are used to measure distance to an obstacle. This type of sensor uses linear position sensitive devices (PSD) and can measure with an accuracy of about 1cm. joBot returns a value from the sensor that ranges from 0 to 512. This value can be converted to a distance using the formula:

$$\text{distance} = 2141.7 * (s \wedge -1.08)$$



where s is the sensor value from 0 to 512. Outputs of these sensors are connected to the JBP analog input pins, however their power is connected to unregulated 4.8V and the readings are therefore influenced by the power usage of the other sensors and the servos. This has an effect on the accuracy of the readings.



JoBot uses the GP2D12 sensor, the JoBot Junior uses the GP2D120 sensor which has an active range between 3 and 30 cm.

The Sharp sensors are using triangulation to calculate the distance. This makes it insensitive to colors, but has a dead zone below which the sensor value is not reliable and should not be used. The graph shows the general characteristics of the sensor.

The simulator implements the sensors in two different versions, using the DistanceSensor and DistanceShortSensor classes. In these classes a table is defined that specifies the values for the different distances. These tables allow the simulator to use values that are close to the values generated by the real sensors.

2.1.9 State Machines

Robot control programs are best written in the form of a finite state machine or a number of nested stated machines.

The program runs a timed loop in which every cycle a full set of functions is performed. Such a cycle is called a heartbeat. Usually the red LED is flashed on and off during a heartbeat to show that the system is still alive.

The system keeps track of where we are in the process by keeping the current state in a state variable. When a certain event occurs, for instance when a sensor reaches a certain value, the internal state of the system is changed. That makes sure that during the next cycle another action is taken by the program.

Although you could program all statements sequentially and insert waiting loops at various places, this will not allow the program to be interrupted and change actions under external control. State machines have been designed to cope with the changing real-time aspects that are important to autonomous robots.

For an example study the code of the UVMdemo program where the heartbeat and finite state machine approach is used.

2.1.10 getState, setState and reportState

The robot's state can be controlled by changing the DIP switches and by using the setState command. Using the setState command, behaviors may be altered under program control. The getState function is used to retrieve the current state, while reportState is used to allow the robot to send messages to the Sysout device selectively.

When using Sysout statements information is sent through the COM or USB serial interface. When no PC is connected to the robot, however this may cause the program to block. Although connecting the robot to a PC briefly resolves this problem, allowing output to be sent selectively is a more elegant solution.

Therefore the reportState command sets the level of reporting output and may be used by the robot to either suppress output or select different reporting schemes.

No fixed reporting schemes have been implemented which gives total freedom to the developer to use this facility in any desired way.

2.1.11 PeriodicTimer

A periodic timer is a function that is called at fixed intervals. It is a standard Java construction that allows the robot to generate clock ticks.

In UVM Demo two such timers are used, the heartBeat and the behavior tick. The tick frequency determines how often a behavior is executed. For most JoBot implementations a frequency of 1 cycle per second is sufficient. The heartbeat is executed every half second so that the heartbeat is effectively 1 second also.

When a higher reaction speed is required the behavior clock could be set faster but remember that the processor needs to have sufficient capacity to cope with the desired clock frequency.

2.1.12 JobotBaseController.java

All commonly used standard JoBot functions are contained in javaBot.UVM.JobotBaseController.java. An instance of the class is created when the program is started and is referred to as `JoBot`. for all standard functions. Please note that the various implementations may differ based on the facilities of the robot and the processor. The descriptions underneath are the most common facilities that exist in the various implementations.

The following functions are present:

- `heartbeat()` - Shows the heartbeat
- `setStatusLeds(r, y, g, b)` - Sets the status lights
- `setServo(a, x)` - Gets the current servo value
- `getSensor(x)` - Get the current sensor value
- `drive(x, y, z)` - Drives three servos
- `scaleToServoSpeed(a)` - Scales the given servo value
- `vectorDrive(x, y, z)` - Drives servos using a vector and rotation
- `tone(tone)` - Sounds a tone on the JPB2 speaker
- `reportState(level)` - Sets the reporting level
- `setState(s)` - Changes the robot state



2.1.13 The Behavior class

Behavior is the base class that all behaviors are derived from. It is an abstract class that defines the methods that need to be implemented by every behavior. Currently only `doBehavior()` is required. There are two methods defined in the base class itself:

- `timer()` This gets control when a Timer event occurs for the periodic timer for the behavior and then calls the `doBehavior()` method.
- `Stop()`. Is called by the `setState()` method when the current behavior needs to stop. Once stopped a behavior can't be started again.

Basically what the behavior class does is like running a thread for the given behavior until a stop command is given. Using the `periodicTimer` the behavior is called on every clock tick and executes one cycle of the given behavior. The `JoBotBaseController`, `periodicTimer` and `timerPeriod` are made known to the Behavior through its constructor:

```
/**
 * Creates a new Behavior object.
 *
 * @param initJoBot can be used to access the sensors and actuators of the
 * robot.
 * @param initServiceTick The Timer on which ticks this Behavior should be
 * executed.
 * @param period The period in milliseconds to be set on initServiceTick.
 */
public Behavior(JoBotBaseController initJoBot, PeriodicTimer
                initServiceTick, int period)
{....}
```

Although the UVM Demo contains a number of behaviors, we will describe two typical behaviors as an illustration of the capabilities of the UVM software:

- **FleeBehavior** – flees away from an object that is detected by one of the sensors.
- **MapReaderBehavior** – accepts an input map of drive parameters and executes these, resulting in pre-programmed movements as encoded in the map.
- **ChaseBallBehavior** – locates a nearby object and assumes this is the ball. It then moves toward this object and pushes it forward, thus chasing the ball.

FleeBehavior

This behavior takes the input readings of the three sensors and determines if an object is sensed. In the example a fixed value is taken, but using the actual value may result in more advanced behavior.

When a sensor detects an object, the robot will generate a movement away from that object.



When developing behaviors like this one please observe that more than one sensor could be sensing an object at the same time and that therefore any combination of sensor values must be taken into account.

Therefore we only take the value of the highest sensor. A more complicated solution could be to calculate the movement required to react to multiple sensors at the same time but this is left as an exercise for the user.

MapReaderBehavior

The UVM Demo defines a String called `gyrateMacro` and `testMacro`, consisting of binary values that represent servo settings for each of the three servos. When the `MapReaderBehavior` reads this so-called map, it will take three values on every cycle and set the servos accordingly using the `drive` or the `vectorDrive` function. It will continue the movement for some preset time and then read the next map values.

The values in the map define a gyrating movement, making the robot move in a circle without changing its orientation. When the same values are given to a normal drive function however a completely different movement pattern results which seems unpredictable.

Using this technique you may encapsulate many different movements that can be selected at random from a map, thus allowing very flexible additions of 'canned' behavior patterns. You might even send a map of movements to a robot, using a `WebService` interface.

ChaseBallBehavior

The `JoBot` may be programmed to locate and chase a ball. For RoboCup Junior games, a `IRBall` sensor is available that allows the robot to detect the infrared-emitting ball. Additionally a `IR floor` sensor is available that reads the greyscale map of the RoboCup soccer field. With this field and the sensor, the robot is able to determine where it is in relation to the field.

This simple demonstration uses only the `distancesensors` and chases a ball around. Please note that when the robot encounters another object like a wall, it will not be able to distinguish this from the ball and chasing the ball will no longer work then.

2.1.14 Making changes to the code

If you make a change to your code and the agent does not work anymore, make sure you first have made a backup which can be used to find the differences. You may also rename your own code and then copy the backup code back into the `JavaBot.XXX` package to start with an unmodified version.

Copying and Renaming is done by selecting the package name in the Eclipse explorer and using the right mouse button to select the context menu. In that menu copy command is included. Renaming is done, using the `Refactor` menu, in which the `Rename` is a submenu.



2.1.15 Programming JoBot

See the JoBot manual chapter 8 for detailed instructions on how to program the JoBot after you have made changes and tested your program with the simulator.

When the behavior of the agent has reached the stage where it can be tested on a real JoBot, the agent's program code needs to be compiled to PIC machine code and transmitted to the flash memory of the JPB board.

This chapter deals with the muVium architecture and describes how the UVM IDE, that is provided as a separate program, can be used to perform those tasks.

2.2 Additional Demonstrations

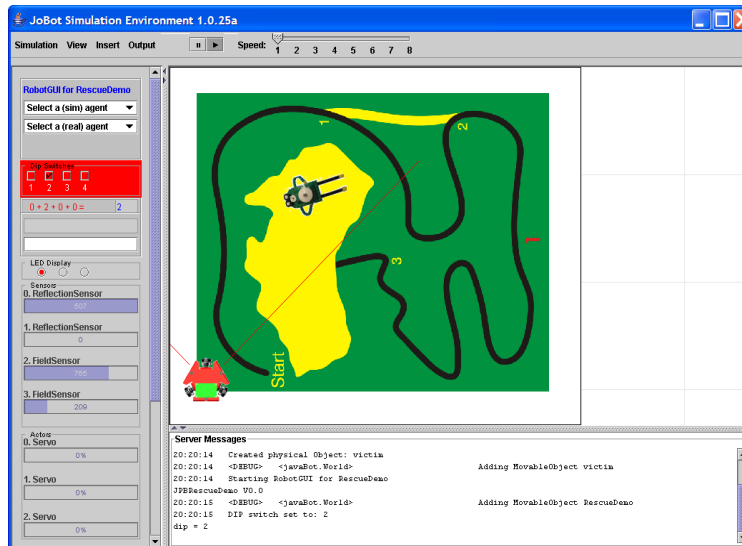
The basic UVMDemo program shows the most important functions. There are a number of more advanced demonstrations, each showing more complicated facilities of JoBot and the simulator. They are the following:

- Rescue Demo - Shows how a simple line following robot may be programmed for use in the Rescue competitions.
- Soccer Demo – Shows a basic soccer playing program for the JoBot.
- Speech Demo – Is a demonstration of simple sound analysis and recognition that runs both on the PC and in a JoBot, equipped with a microphone interface.
- Mouse Demo – Is a demonstration of the Mouse sensor. A special mouse sensor device is required to run this demo on a real robot, the simulated version is fully implemented.

Please note that these demonstrations are a first experimental version that is continually updated. Most of the demos are not complete and contain bugs that will be resolved in future versions. Work is being done on these demos by students on a continuing basis and this work is included here as additional information.



Chapter 3 Rescue Demo



The rescue demo is a simple demonstration of a JoBot equipped with two field sensors that is able to follow the black line.

Please note that a JoBot cannot easily be fitted with two field sensors, since there is only space for one of these sensors. The simulator however allows the definition of such a configuration and therefore we are able to run a simulation of such a JoBot.

A future version will include a version of the JoBot Junior which is capable of having two field sensors and will be controlled more easily. For now there is only the simulation version of the Rescue demo.

This demonstration is based on the lessons, created for RoboCup Junior and shows the first of a series of exercises in which simple line following with one and two sensors is explained. More demonstrations will need to be created, this is only the first one.

In order to run the demonstration, place the robot over the black line and it will start following the line. Please note that once in a while the robot loses track of the line. This is mainly due to small color differences and the lack of proper calibration of the demonstration. More work is required to make this demo follow the educational material.

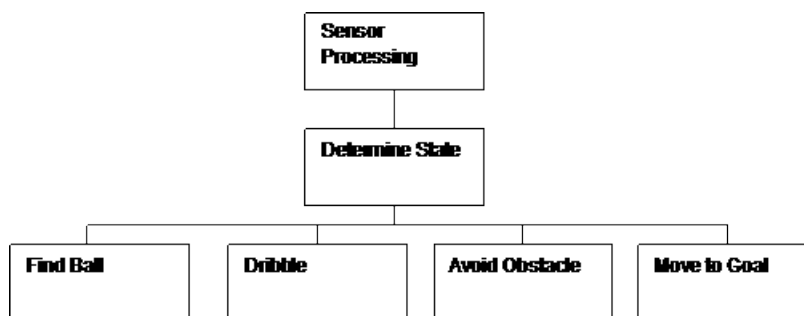
Chapter 4 Soccer Demo

The soccer demo is the most extensive and complicated demonstration. It consists of a number of behaviors that are organized according to the reactive behavior pattern.

Reactive behaviors are programmed functions of the robot that respond to a certain situation, called a state. So in order to react properly, the system must continually determine the state it is in, which is done by constantly monitoring all its sensors.

The picture underneath shows how this is organized. First the sensors are read and its values processed. Based on the sensor readings and the current state of the robot, the new state is determined. Based on the state a certain behavior is then selected. In this example we have four behaviors:

- Find Ball - will look around and tries to find the ball.
- Dribble – Moves forward iwhile keeping the ball at a close distance
- Avoid Obstacle – Will navigate around an obstacle if one is detected
- Move to Goal will move into the direction of the goal



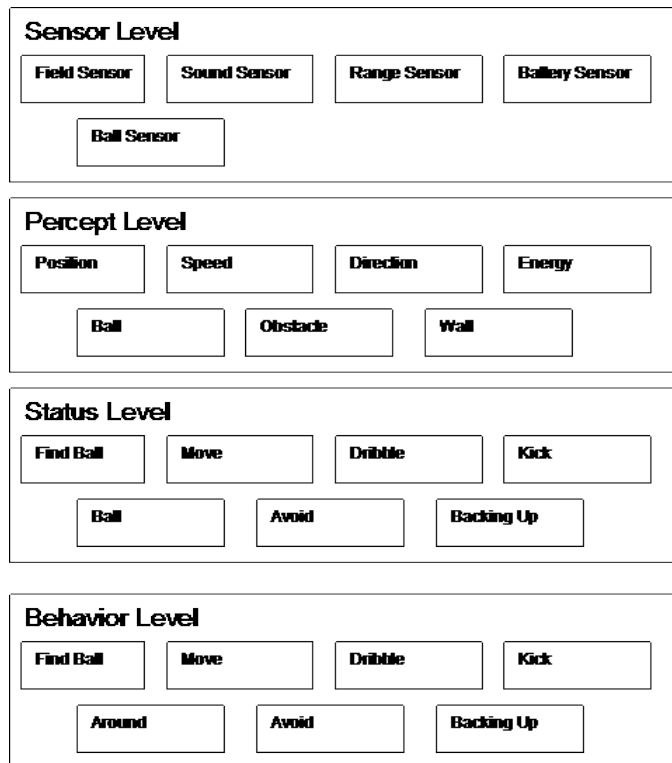
The most important thing is to determine the proper priority and the conditions under which these behaviors are executed. In this demo we only deal with a limited number of states and behaviors. In a real application many more states and behaviors will be present.

A behavior is triggered by the robot's state, which is based on the current state and the sensor readings. So several task levels exists that are executing more or less simultaneously. We distinguish the following levels:

- **The Sensor Level** – At this level all sensors are read continually and their values are used to determine what observations can be made. These observations are collected at the Percept level.
- **The Percept Level** – Percepts correspond to concepts in the real world. By using the Infrared Sensors and the Distance sensors, the robot can determine that it sees the ball and at what distance it sees it. Using its distance sensors it can also determine if it sees an obstacle. These Percepts are then used to determine the current state.
- **The Status Level** – Using the Percepts the robot now registers if it sees the ball and how close it is, if it is moving or standing still and if moving, in what direction and can also determine if there is an obstacle. These states are then used to determine what the appropriate behavior is.



- **The Behavioral Level** – Based on the current state, the robot will determine what the best behavior is and will execute this behavior until the state changes.



4.1 Percepts

Determining the percepts is depending on the sensor readings. Let us take for instance the field sensor. Using the grayscale on the soccer field floor we read the value of the sensor. Assuming that we know what the highest values of Black and White on the field are, we can now determine where we are on the length axis of the field.

By comparing the current position with the last known position and assuming that we update the field position at fixed intervals the difference between the current and previous reading indicates how fast the robot is moving and into what direction.

Suppose now that we have ordered the robot to move with 100% power and if we know what the maximum speed is, we can compare the actual speed to the speed at 100% and use this to determine if we are stuck or if we are moving diagonally.

If the actual speed is zero but the motors are powered, then we apparently are stuck. If this is lasting for a few seconds, this is a state in which we need to move away from the obstacle we apparently have encountered.

The concepts of position at the field, the current speed and direction of movement are called Percepts and correspond to concepts the robot can determine with its sensors.

In this example we are using the following percepts:

- **Position** – The robot's position in the length axis of the field



- **Speed** – Speed in grayscale steps per clock tick
- **Direction** – Movement in direction towards black goal
- **Energy** – Current battery levels
- **Ball** – Is the ball in sight or not
- **Obstacle** – Has an obstacle been detected
- **Wall** – Has a wall been detected

4.2 States

If the robot determines, based on its percepts that it is moving in the wrong direction of that it is stuck, this creates a certain State. For instance, Stuck is a state, Obstacle Detected is another state. The difference is that Stuck signals that there is an obstacle that the sensors have not been able to detect, while Obstacle Detected means that the robot needs to navigate around the detected obstacle.

Actually determining the state means that we take all known percepts and determine what behavior needs to be selected based on the current known situation. This then determines the current state, which is closely associated with the selected behavior.

In this example we are using the following states:

- **Finding Ball** – When the ball is not in sight, the robot needs to look for it
- **Moving** – When the robot has determined that it should move, it is in the moving state.
- **Dribbling** – When the ball is in sight, the robot will start moving in a dribbling mode
- **Kick** – When the robot gets close to the opponent's goal it should attempt to kick the ball
- **Ball** – Robot sees the ball
- **Avoid** – Robot is avoiding an obstacle
- **Backup Up** – Robot is backing up from a collision with a wall

Based on the current state, the appropriate behavior is now selected.

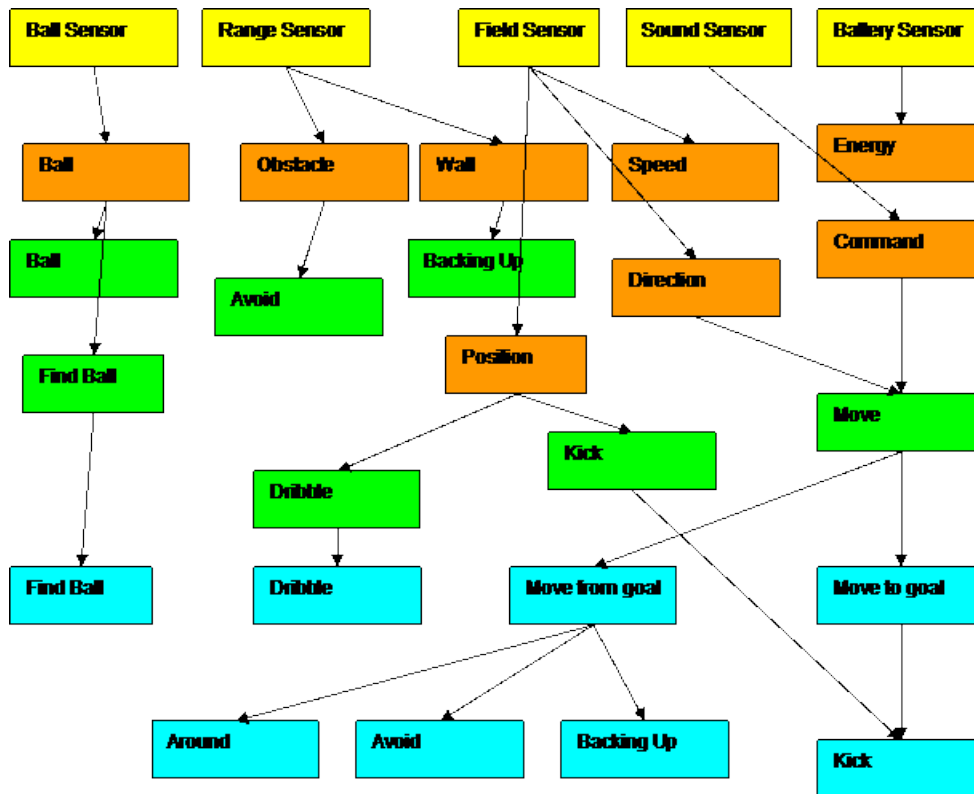
4.3 Behaviors

Like we have seen in the basic UVMDemo, all behaviors are inherited from the Behavior abstract class. All new behaviors need to implement the doBehavior method and define the required actions there.

A behavior is executed continually after it is created by setState after the system detects a change in the robot's state. It first stops the current behavior and then starts a new behavior.

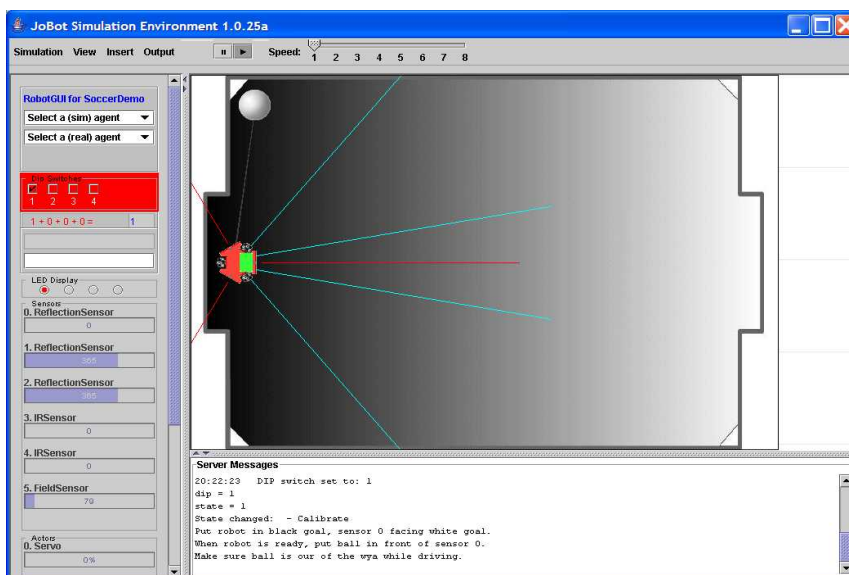
The picture underneath shows the relationships between Sensors, Percepts, States and Behaviors. Please note that this picture does represent the general idea and not the actual implementation, which does contain more detail than is depicted here.





4.4 Calibration

In order for the sensors to determine the Percepts and States based on the sensor values, all sensors need to be calibrated if a change of environment may cause sensor readings to differ.



In this demo we collect information about the greyscale values and the speed of the robot when driving the motors at different power levels. Please note that the current state of the batteries do have an effect on the speed of the motors. Although not

covered in this demo, the actual power level may be used to correct the speed indications for power level fluctuations.

The demonstration contains an aut-calibration routine that starts of with driving the robot from black to white and measuring the maximum and minimum values. It uses the distance sensors to track the sidewalls of the soccer field and keep the robot in the middle of the field. Once it reaches the white goal it turns around and then starts measuring the actual travel speed against the motor power levels and the greyscale values. These values are stored and are used as a reference.

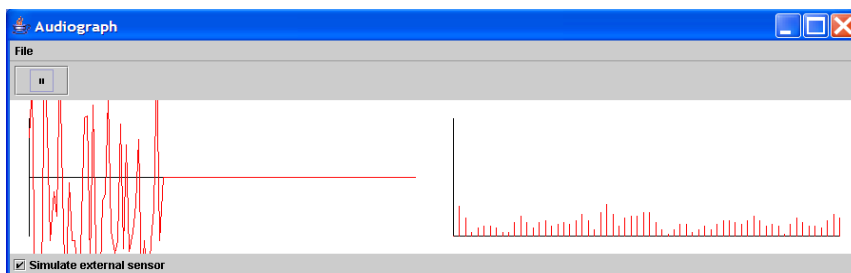
Therefore calibration only needs to take place once as long as none of the other parameters change.

The calibration starts when you move the ball in front of the distance sensor, Make sure you place it back in the corner to avoid interference with the travel pattern during calibration.

Chapter 5 Speech Demo

To demonstrate the possibility of simple sound analysis and phoneme recognition a simple example has been built in which a microphone sensor is used to collect a number of sound samples.

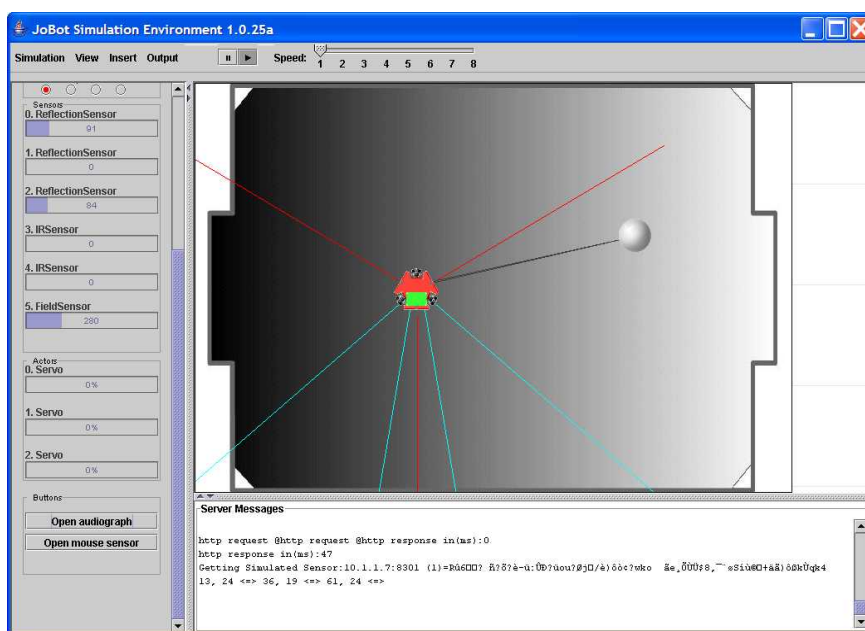
This sample is then analyzed using a Fast Fourier Transform and generates a power spectrum of the given signal.



A power spectrum shows the amplitude of the input signal, divided into a number of frequencies that exist in the input. Using these levels the program is then able to determine the sound that is generated.

Every phoneme consists of a number of so-called formants and the combination of the first three or four formants determine the actual phoneme. In most cases the lower frequencies are resonations that are not relevant to the sound and the second and third formants are usually sufficient to recognize a phoneme.

In order to run the demo in the simulator, select the “Simulate External Sensor” check box in the Audiograph window and you will see signals that are generated at random. In a future version the PC's microphone will be used to generate these signals instead.



If this box is not checked the system assumes that the microphone on the robot is being used and the actual values are then retrieved from the robot, using the webservice interface. The FFT will then be executed inside the robot and not in the simulator and the calculated values are sent back to the simulator to show what the robot is doing.

When all this works OK, the robot is capable to do a full analysis independent of the PC and will be able to recognize simple sounds and react to these.

This part has yet to be developed. The main change required is that an integer-only version of the FFT needs to be implemented so the robot may perform this calculation autonomously.



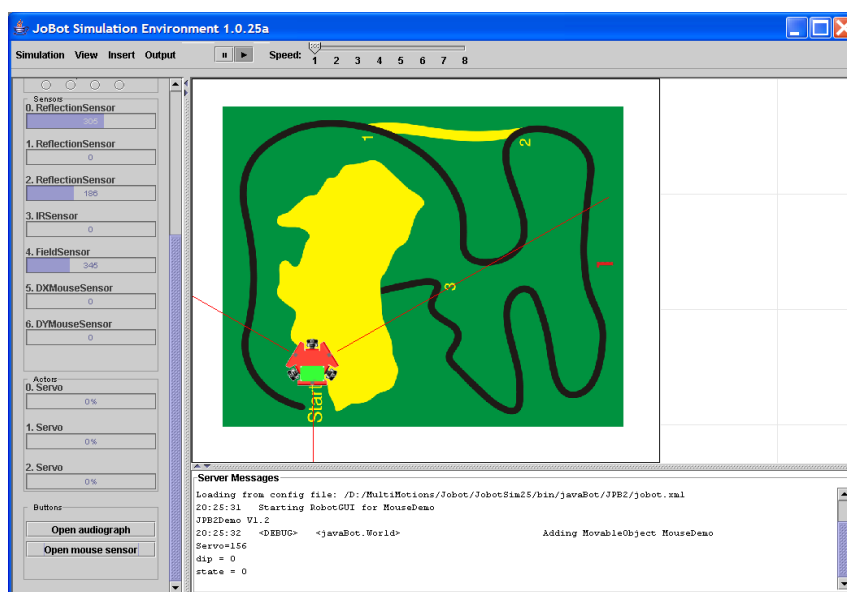
Chapter 6 Mouse Demo

The mouse demo requires a special optimal mouse interface.

A simulation of this interface is included with which information may be retrieved that is similar to what the hardware reports back.

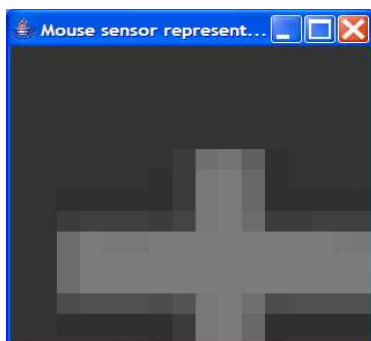
A small part of the underlying bitmap is inspected continually and a black-and-white picture of the 16x16 pixel image that is seen by the sensor is shown.

The hardware version reports back how many pixels the image has shifted in the X en Y direction since the last sample was taken.



When checking this information at fixed intervals this information can be used to calculate the position of the robot as well as its speed.

In the picture above the simulated mouse sensor is situated above the '**Start**' sign and shows the sensor detecting the top of the last T.



More work is required to fully integrate the hardware and software version of this new sensor.