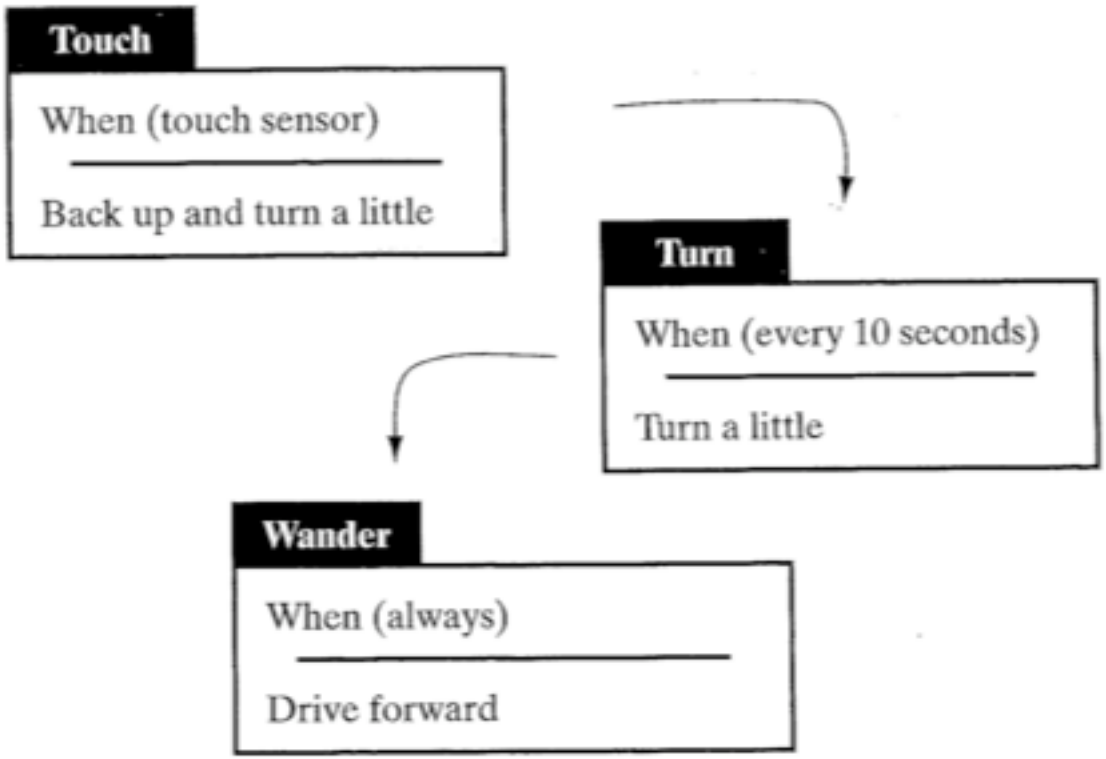
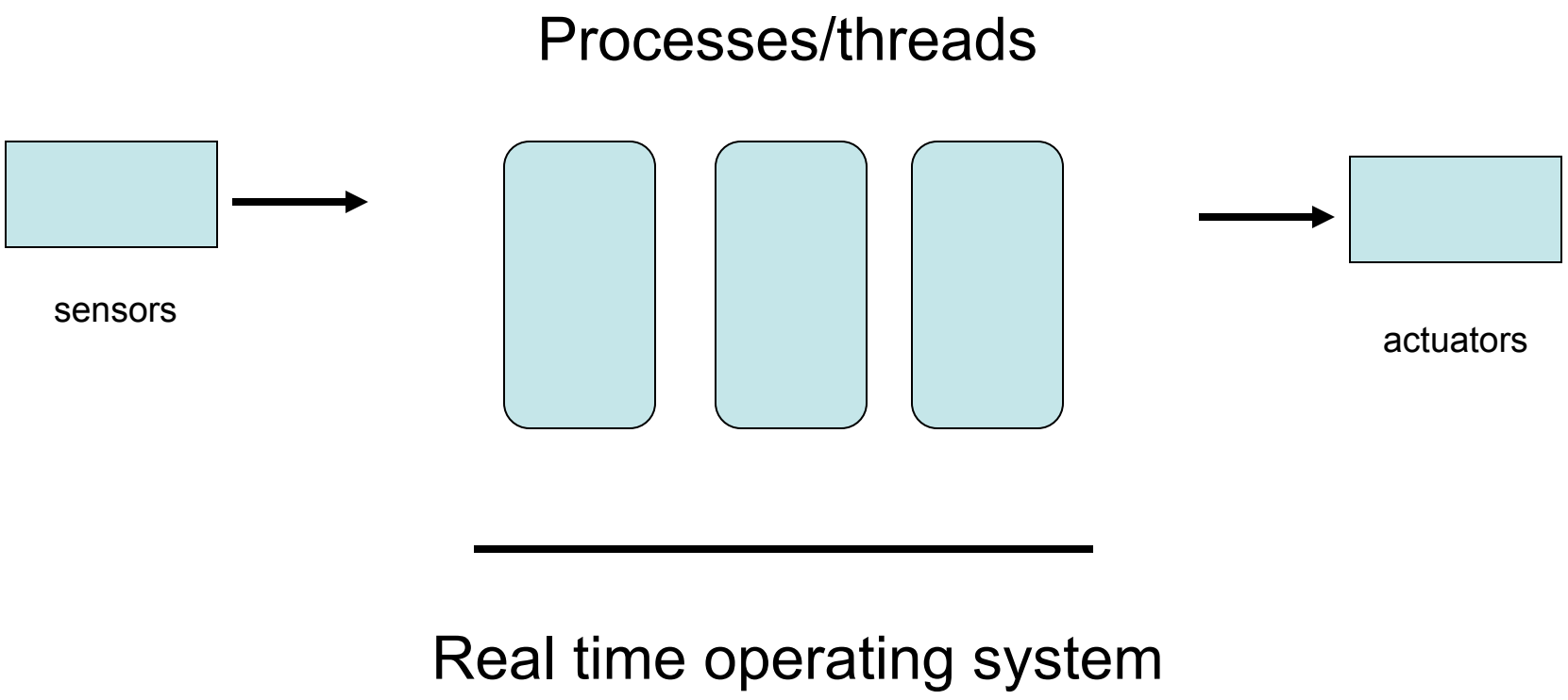
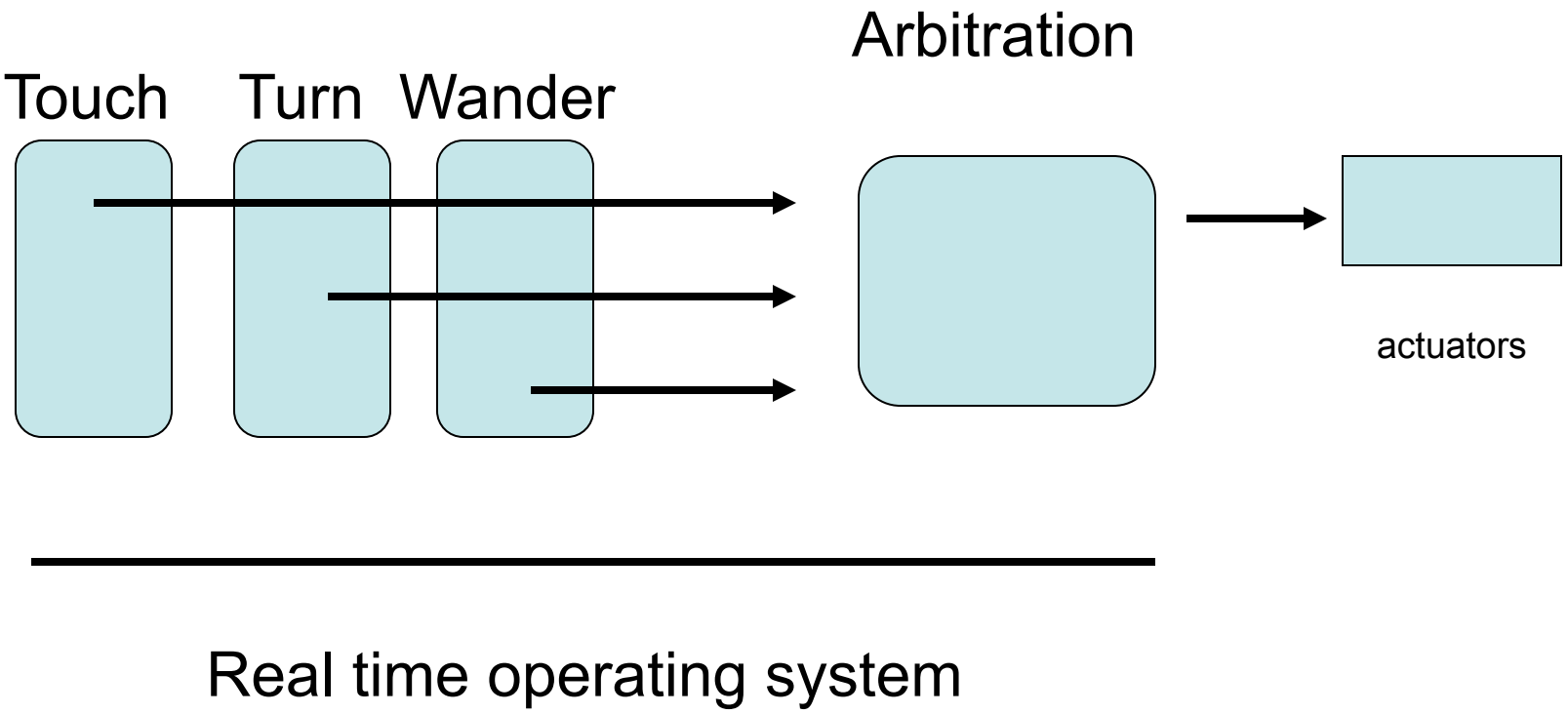
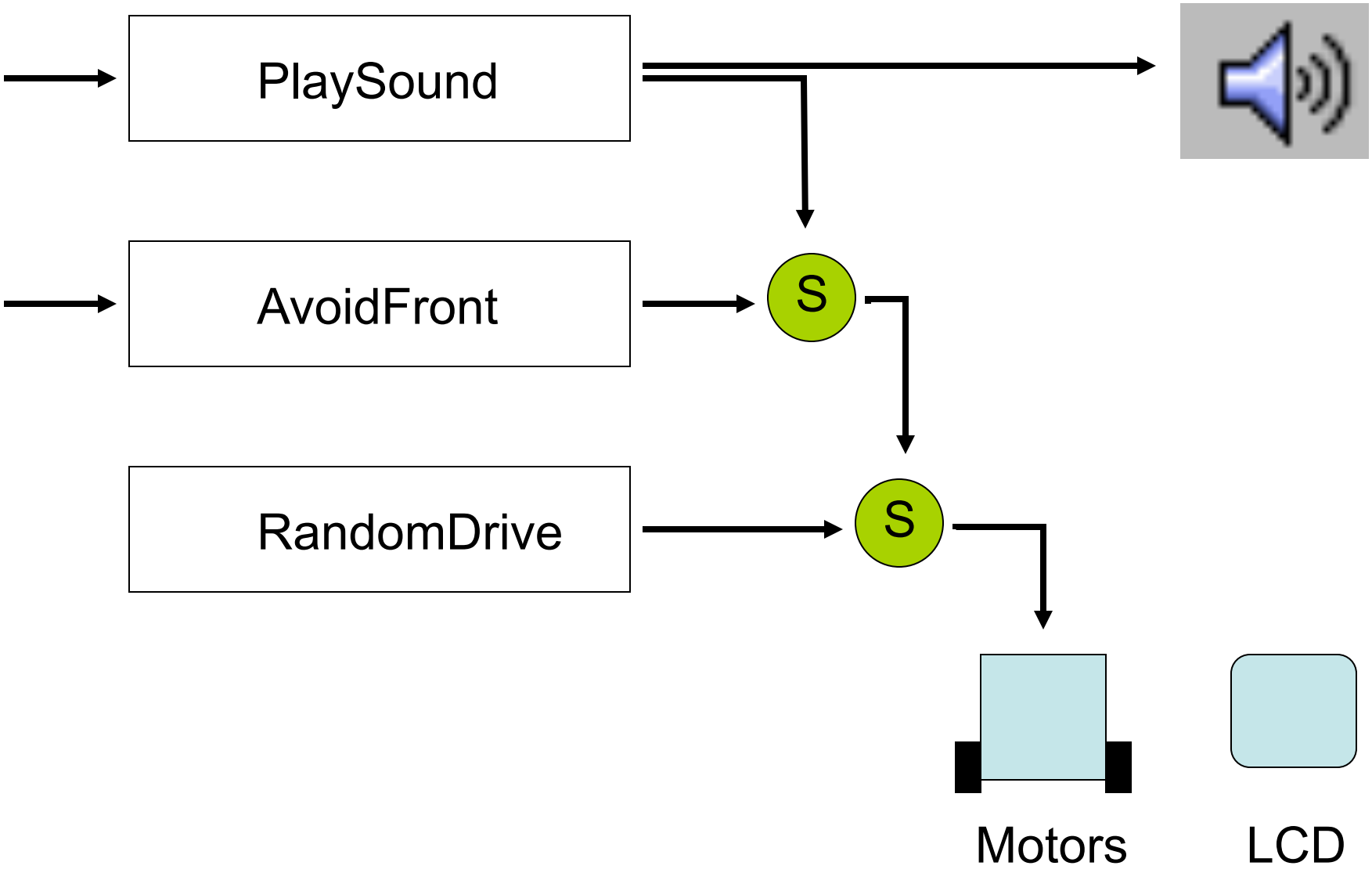
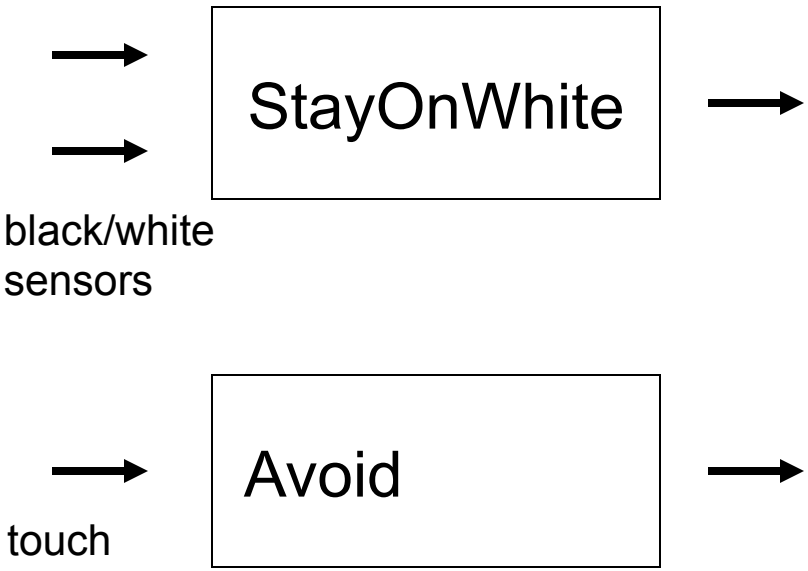

Touch
When (touch sensor)
Back up and turn a little
Turn
When (every 10 seconds)
Turn a little
Wander
When (always)
Drive forward


Processes/threads
sensors
actuators
Real time operating system

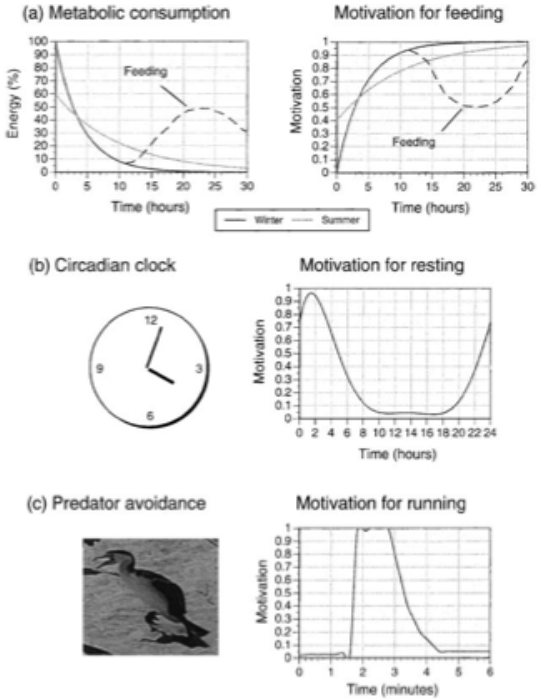

Arbitration
Touch
Turn
Wander
actuators
Real time operating system

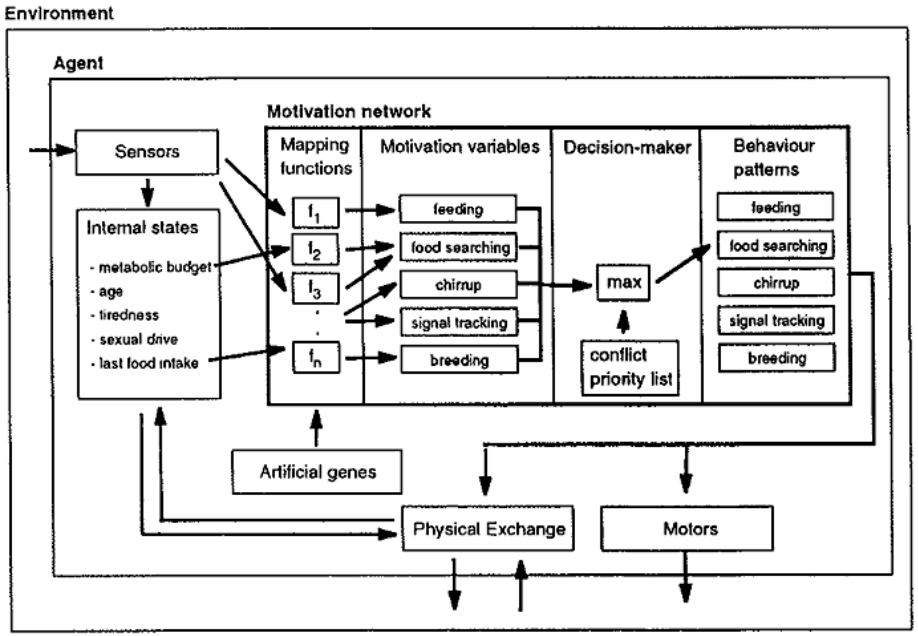

PlaySound
AvoidFront
RandomDrive
S
S
Motors
LCD

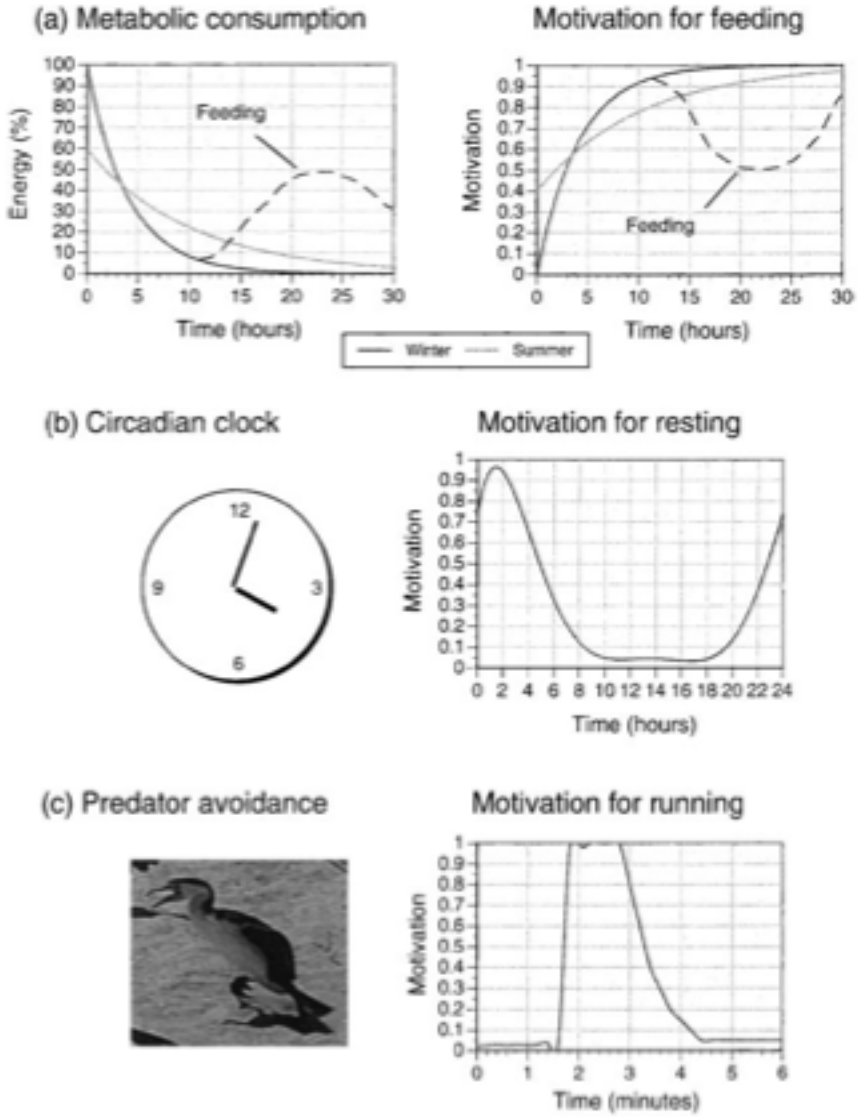
Ghost control program

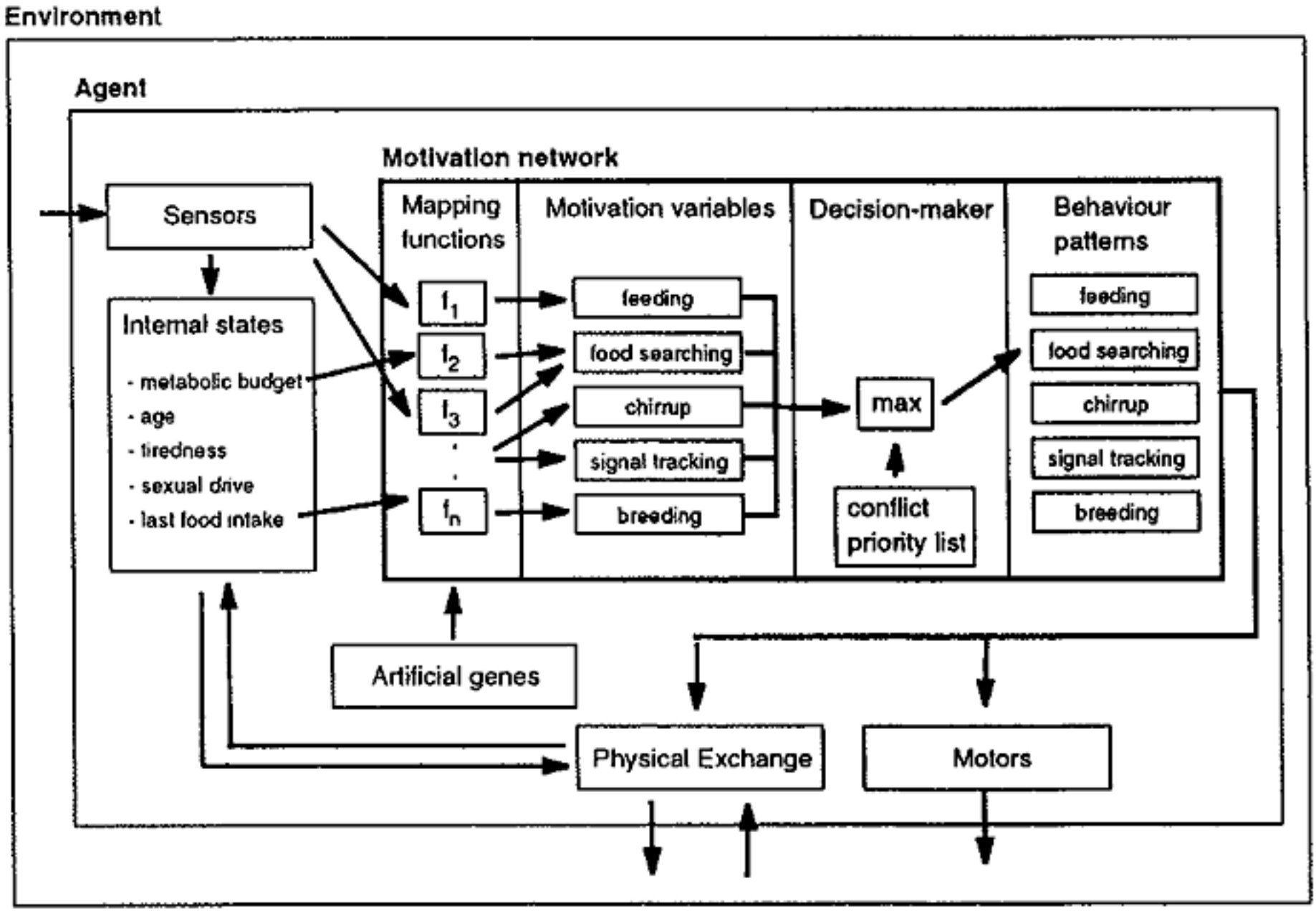


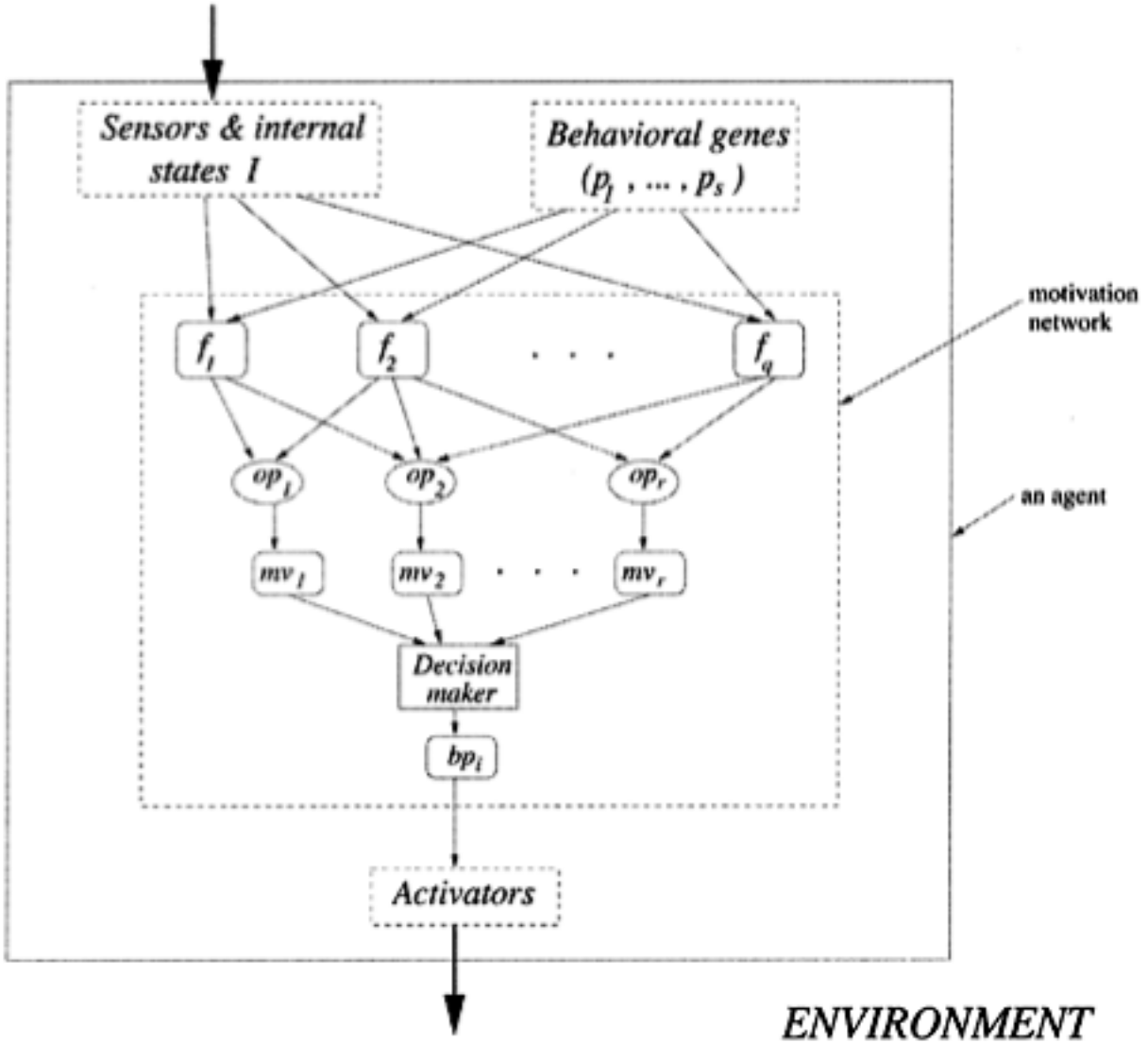
Motivation Networks – A Biological Model for Autonomous Agent Control






(a) Metabolic consumption
Motivation for feeding
Energy (%)
Feeding
Time (hours)
Motivation
Feeding
Time (hours)
Winter
Summer
(b) Circadian clock
Motivation for resting
12
3
6
9
Motivation
Time (hours)
(c) Predator avoidance
Motivation for running
Motivation
Time (minutes)


Environment
Agent
Motivation network
Sensors
Internal states
- metabolic budget
- age
- tiredness
- sexual drive
- last food intake
Mapping functions
f_1
f_2
f_3
f_n
Motivation variables
feeding
food searching
chirrup
signal tracking
breeding
Decision-maker
max
conflict priority list
Behaviour patterns
feeding
food searching
chirrup
signal tracking
breeding
Artificial genes
Physical Exchange
Motors


Sensors & internal states I
Behavioral genes (p_1 , ... , p_s)
f_1
f_2
. . .
f_q
op_1
op_2
op_r
mv_1
mv_2
. . .
mv_r
Decision maker
bp_i
Activators
motivation network
an agent
ENVIRONMENT

The ALIVE System:

Wireless, Full-body Interaction with Autonomous Agents

Pattie Maes, Trevor Darrell, Bruce Blumberg, Alex Pentland

Figure 1: The ALIVE "Magic-Mirror": a user sees himself in a virtual world.

Figure 3: Image of user is composited with computer graphics. Here the Dog responds to pointing gesture by sitting.

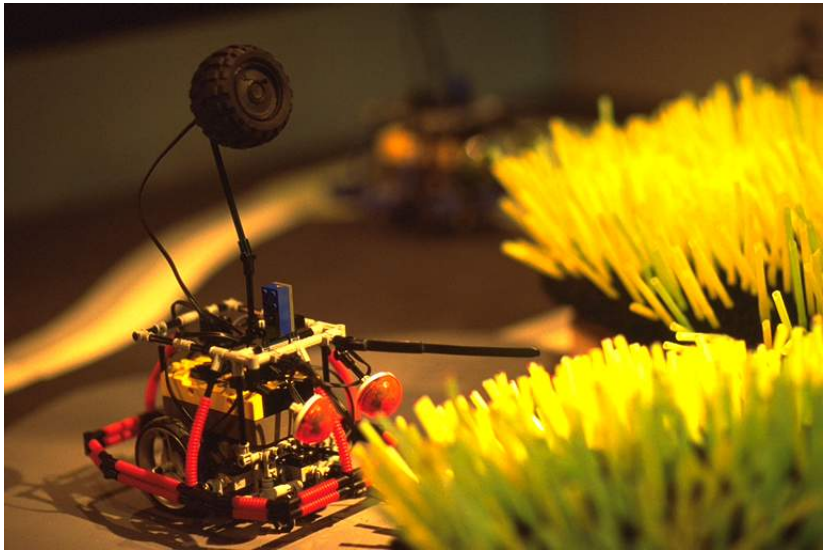
Figure 4: Another example of a recognized gesture. Dog walks in direction indicated by user.

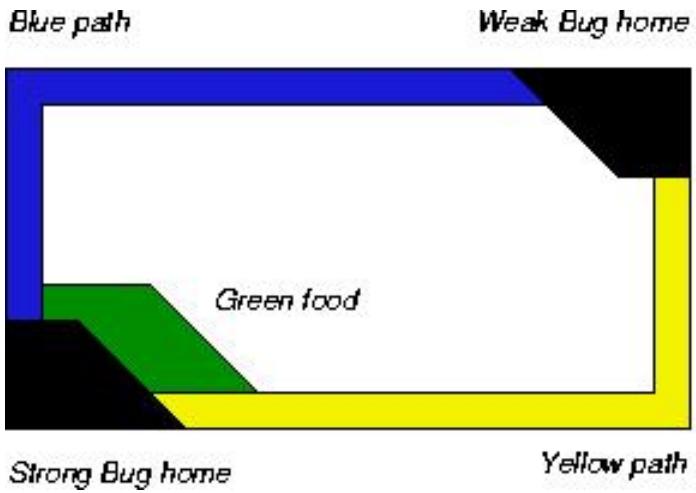
Figure 5: Dog shakes hands with user. Dog respods to hand gestures differently depending on stance of user.

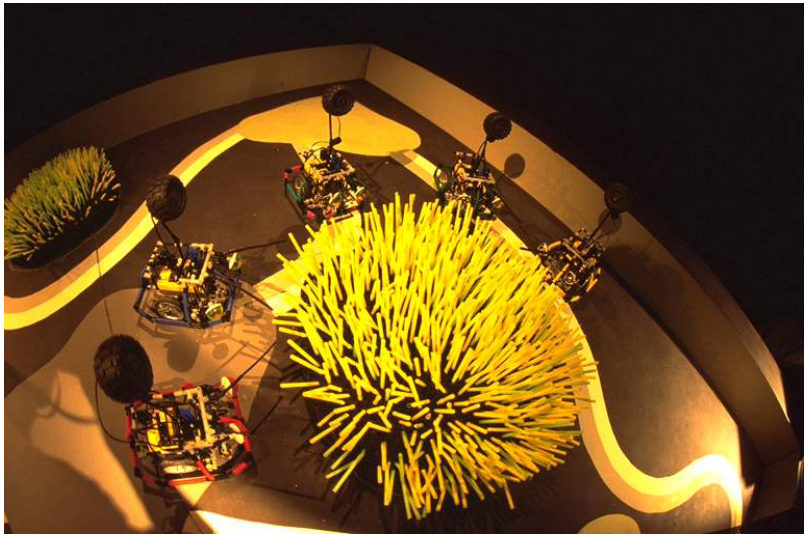
Figure 6: Image of Dog standing on hind legs to mimic user's gesture.

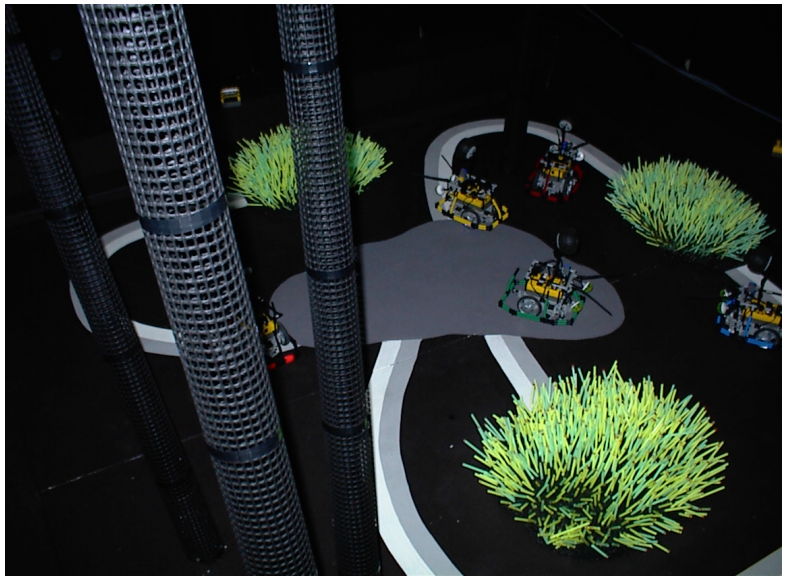
WORLD OF ZOO™

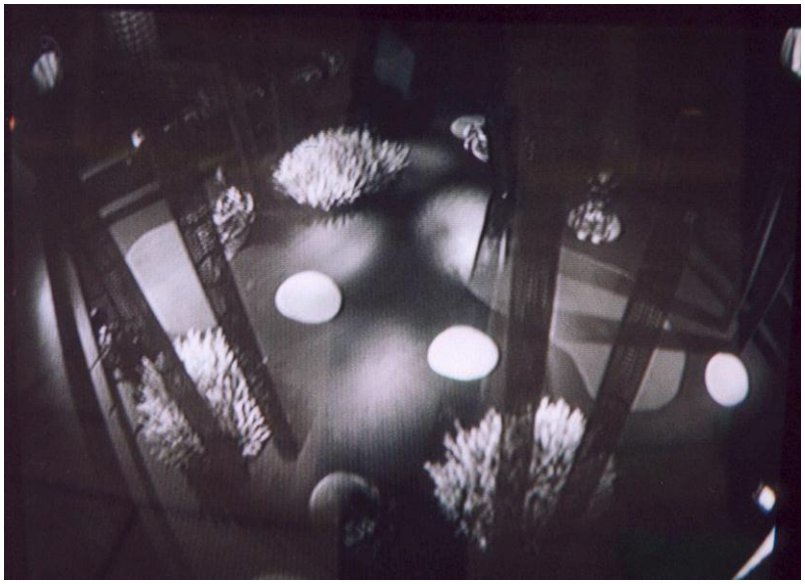
Jungle Kuben

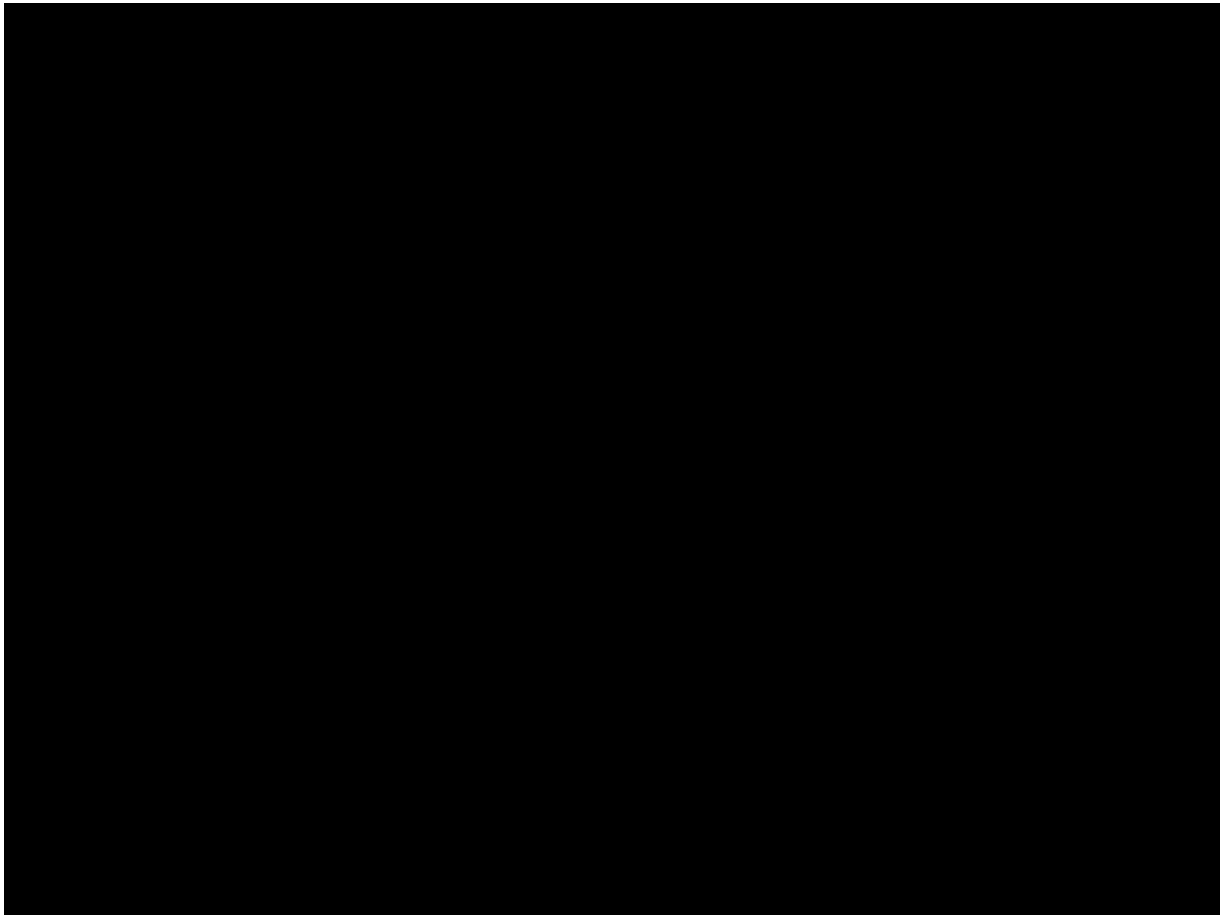


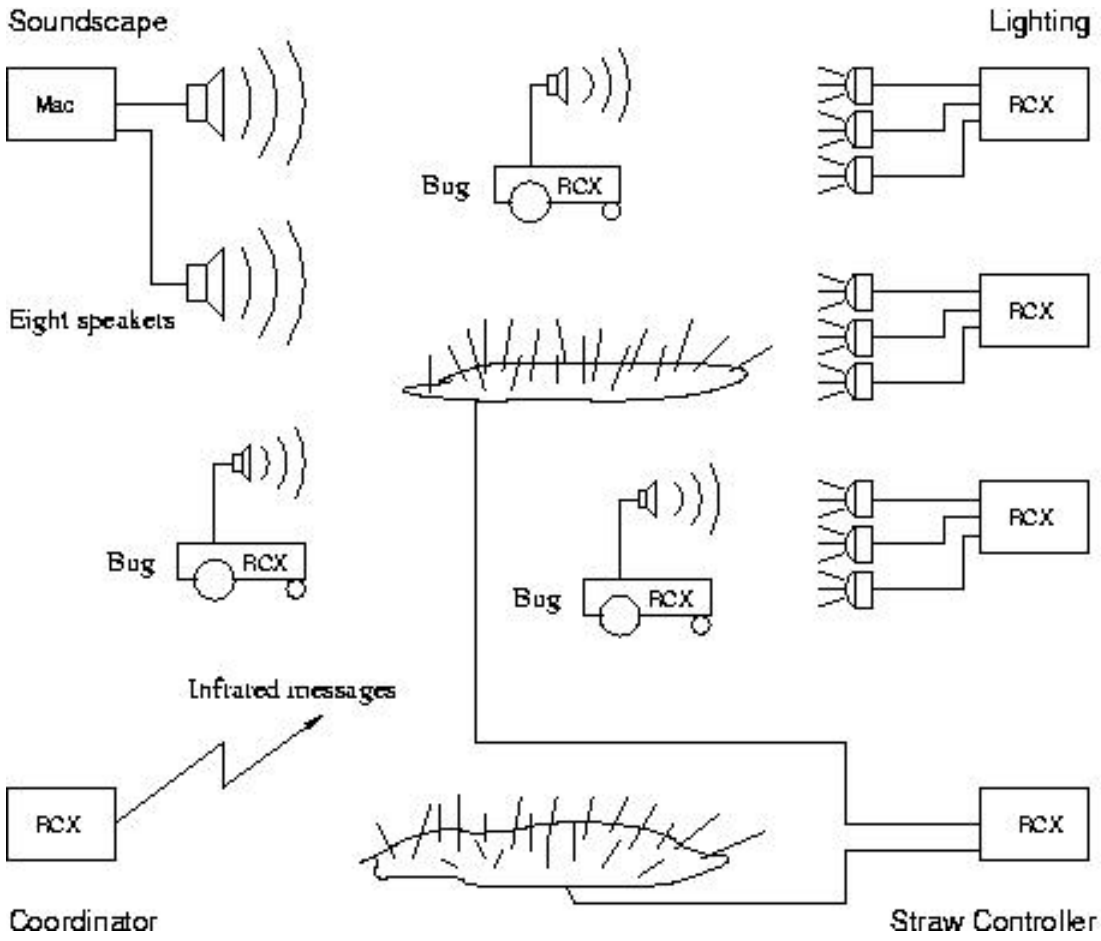

Blue path
Weak Bug home
Green food
Strong Bug home
Yellow path

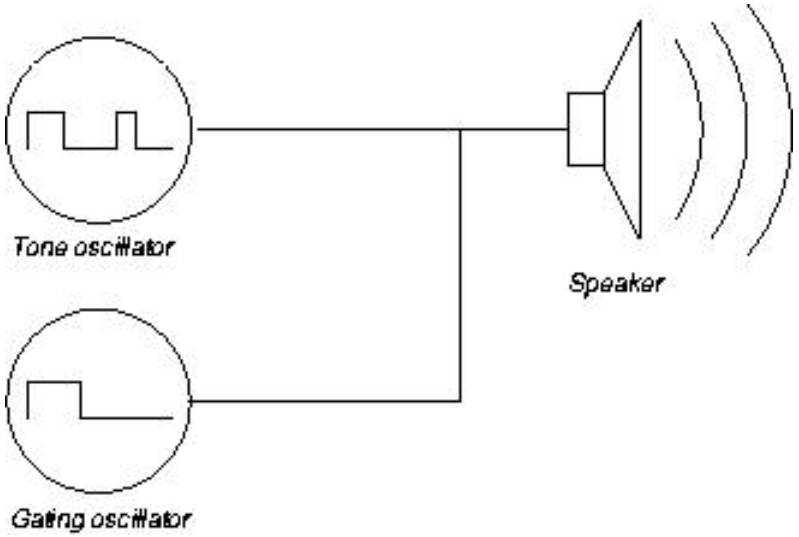


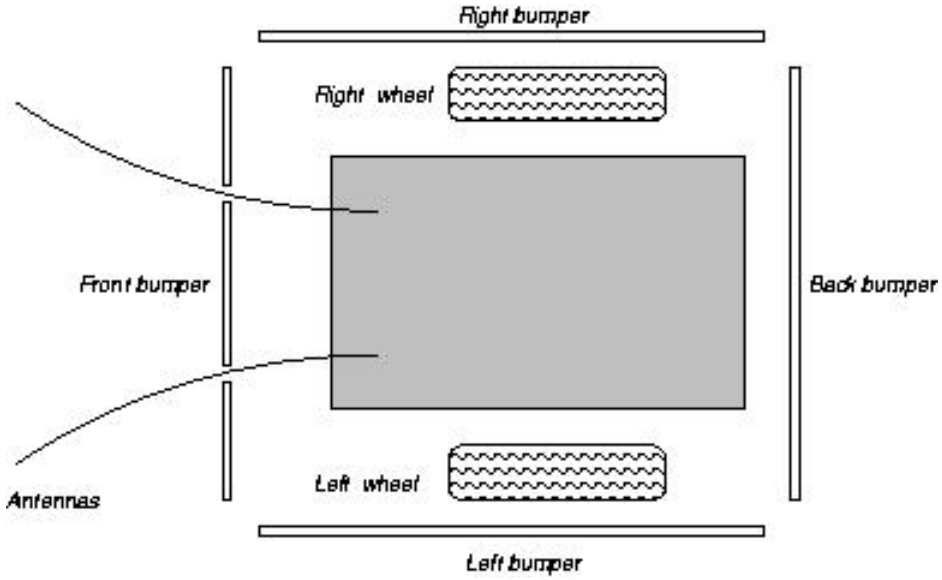


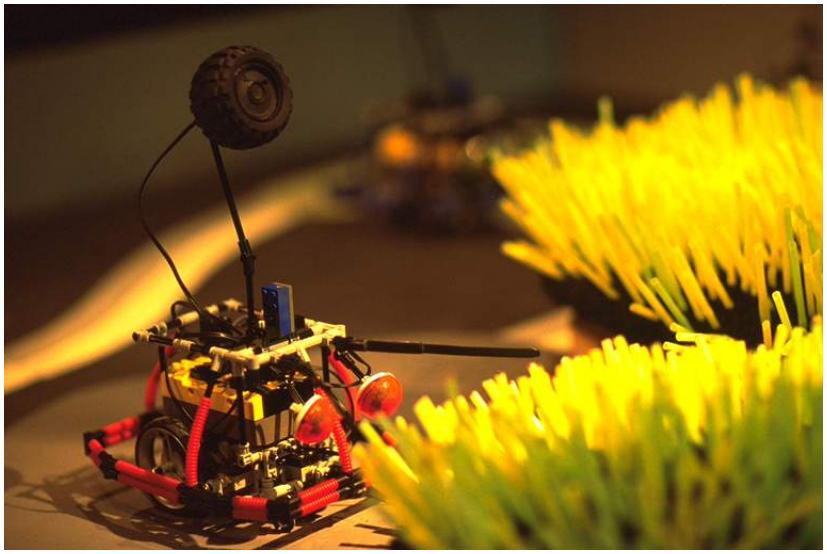





Soundscape
Lighting
Mac
Eight speakers
Bug
RCX
RCX
RCX
RCX
Bug
RCX
Bug
RCX
Infrared messages
RCX
RCX
Coordinator
Straw Controller


Tone oscillator
Speaker
Gating oscillator

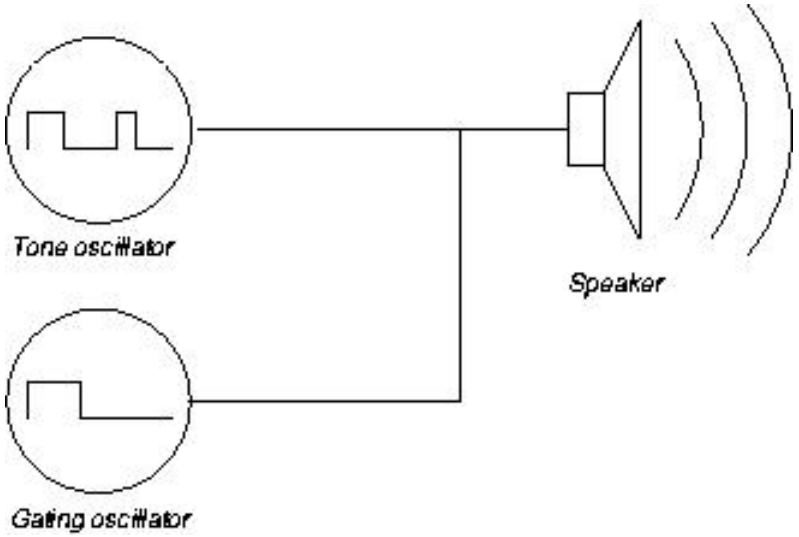

Right bumper
Right wheel
Front bumper
Back bumper
Antennas
Left wheel
Left bumper

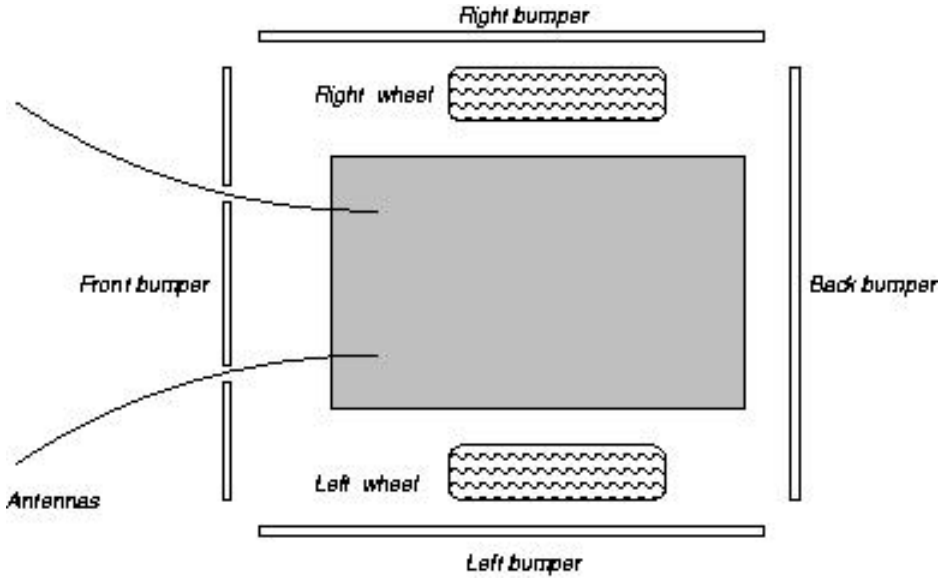


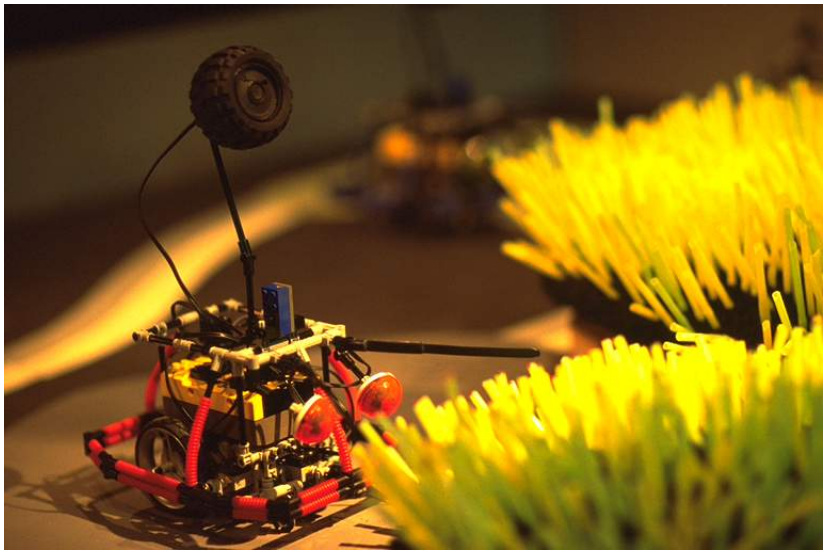
Antennas

Two bend sensors connected to the same port.

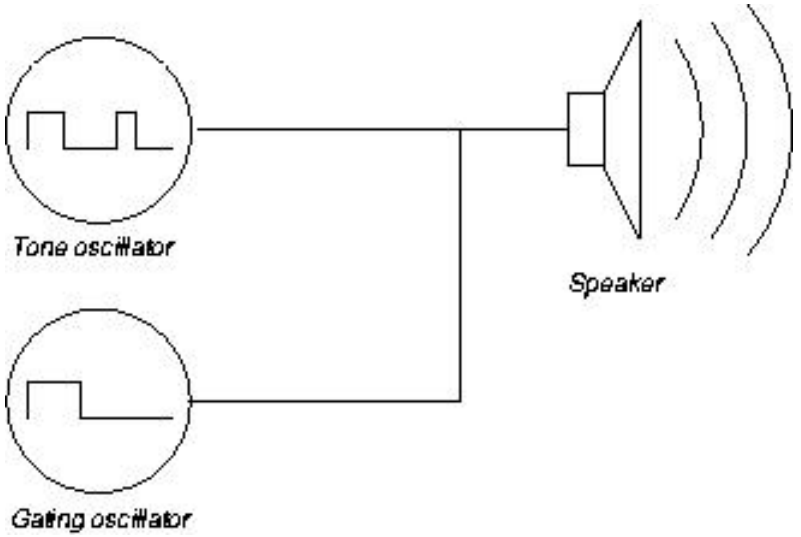
Raw value interpreted as NoBend or Bend

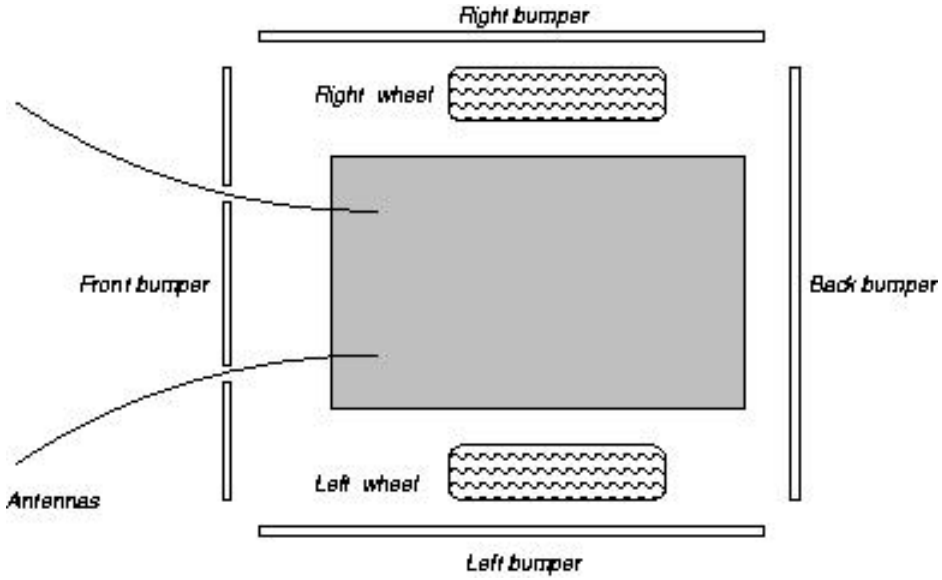


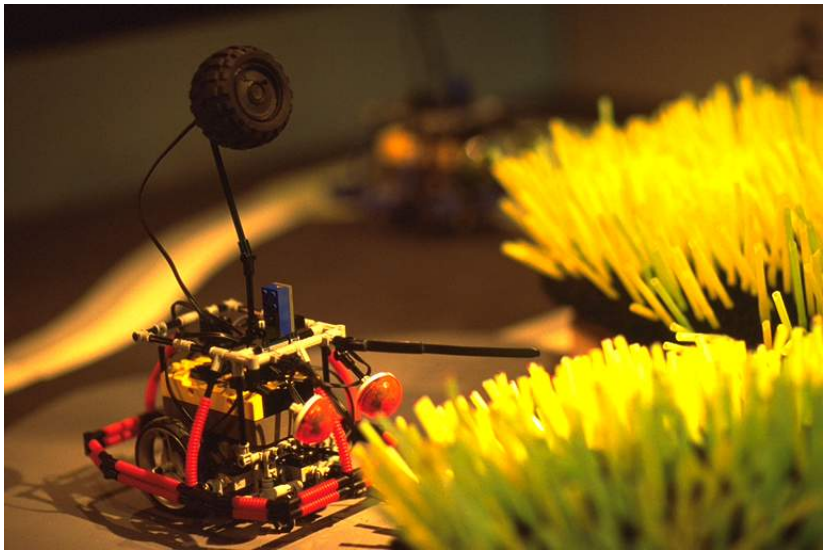




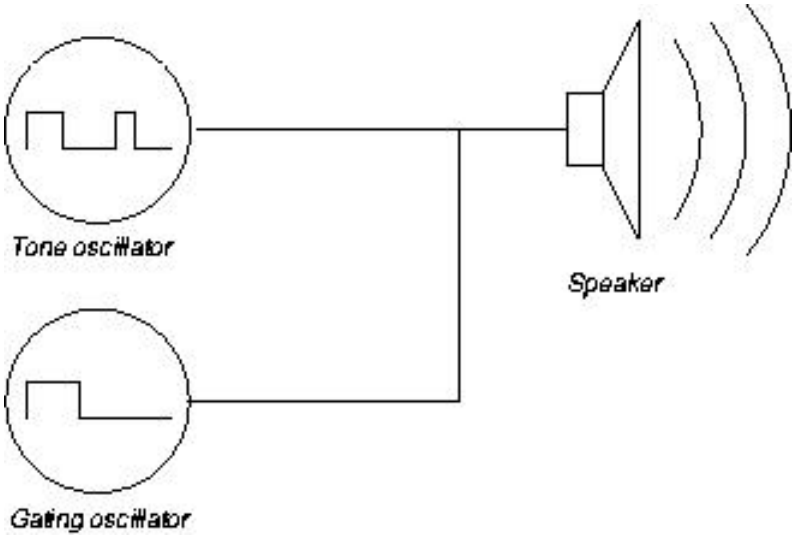
A bumper activates a touch sensor.
There are four touch sensors connected to one port.
The touch sensors are modified to have different resistance when pressed so that it is possible to Identify the bumper that is activated.

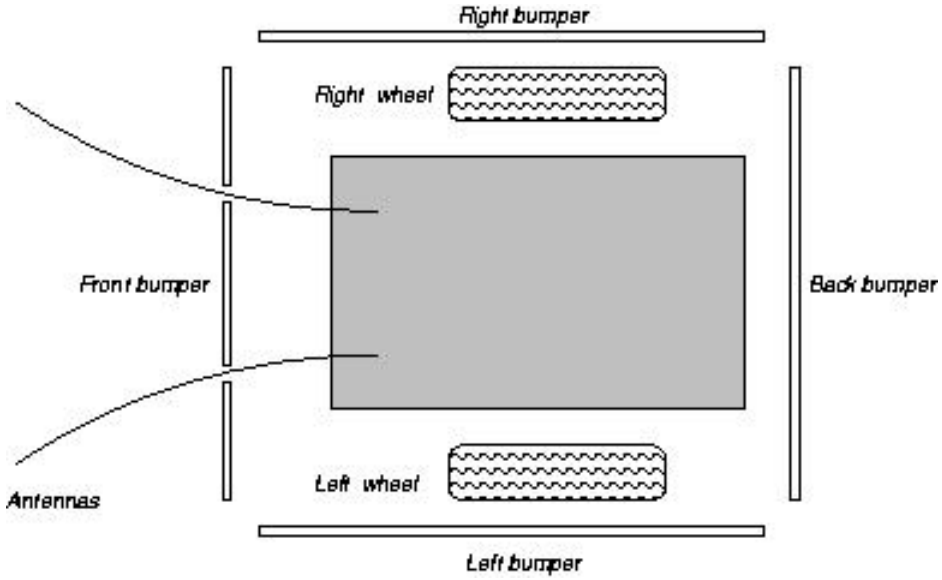


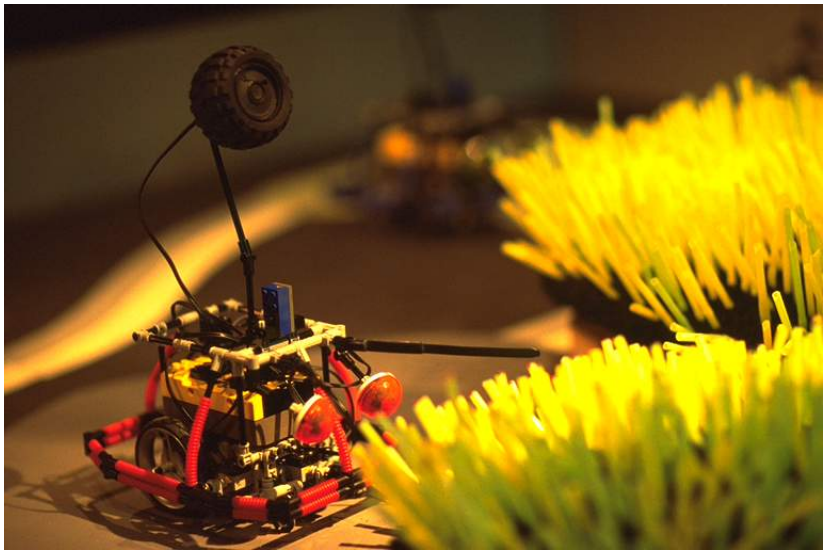


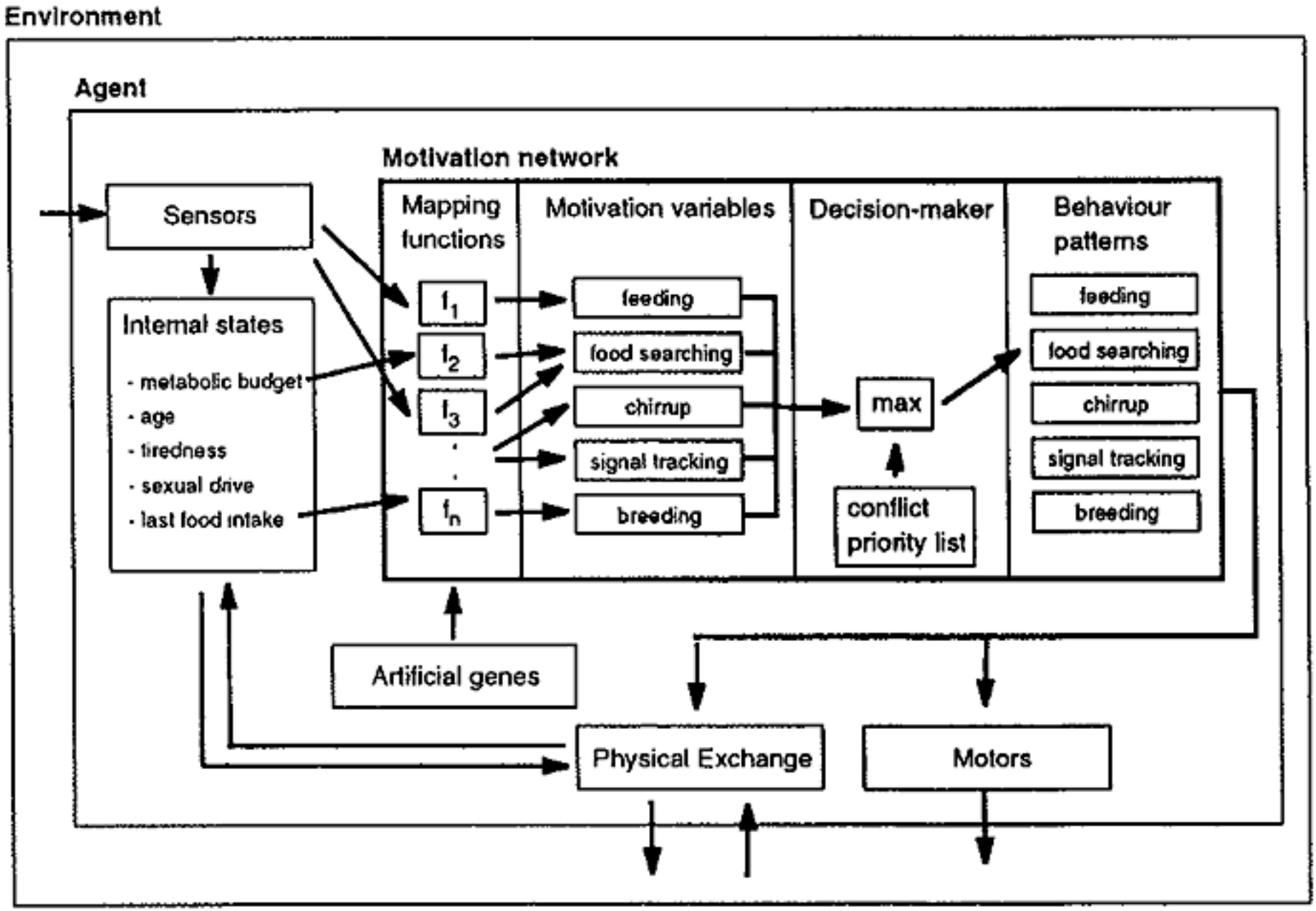


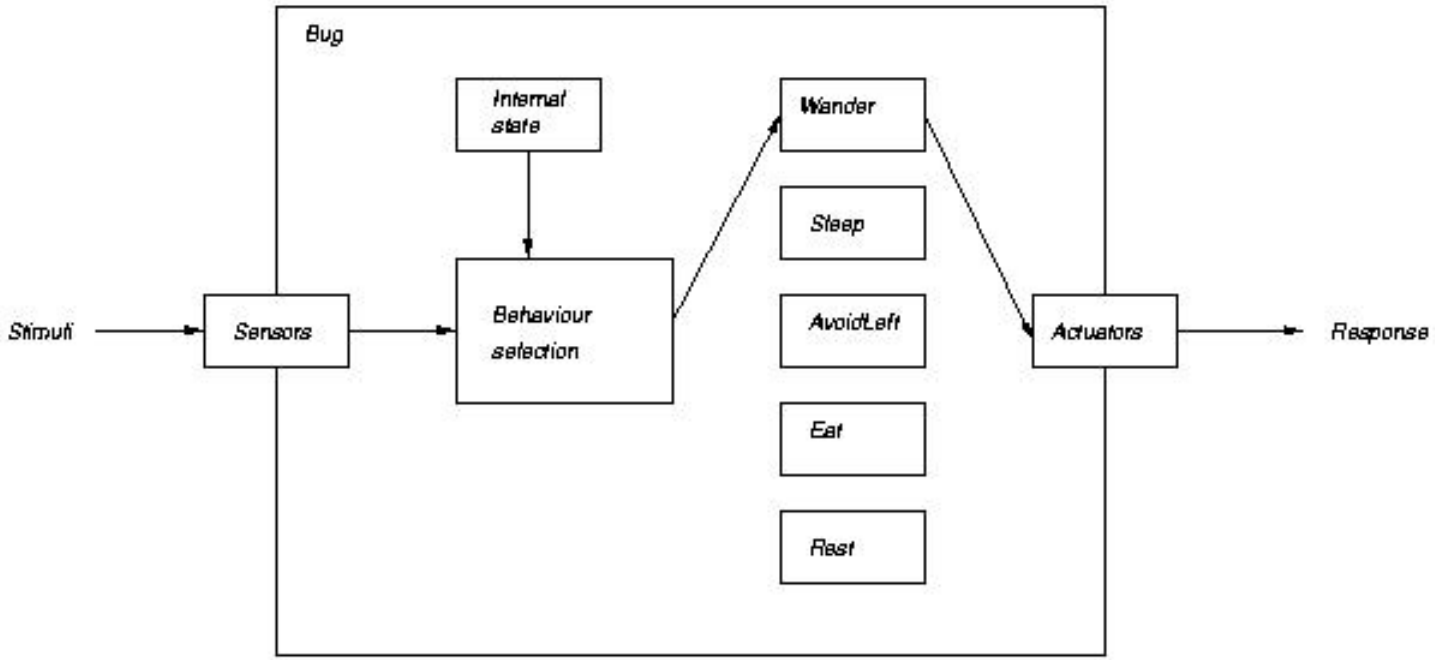
A magnetic speaker is connected to an output port. Carefully timed sequences of on/off commands produce square waves with different periods and duty cycles. This is used to make musical/signal sounds through the speaker.

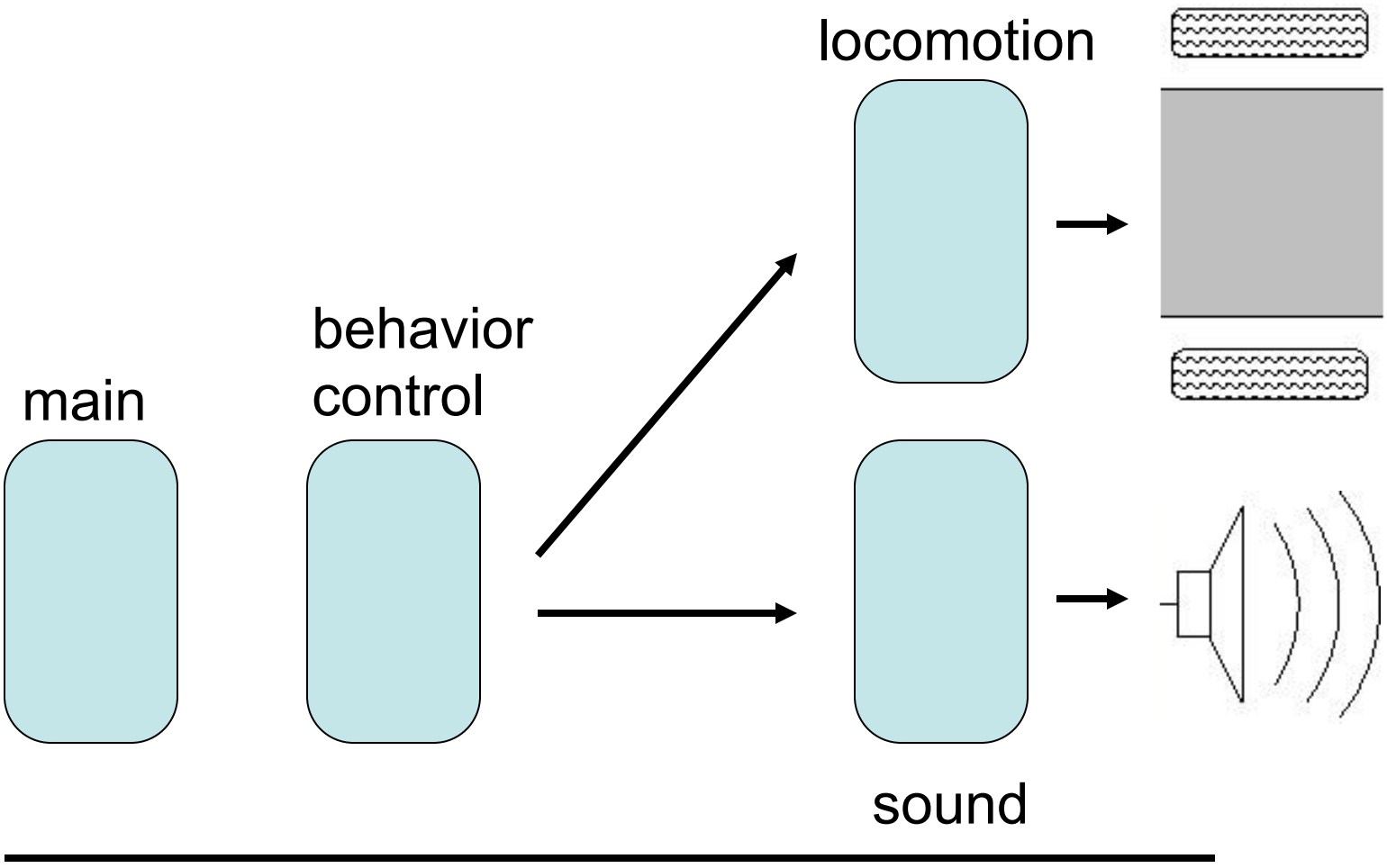




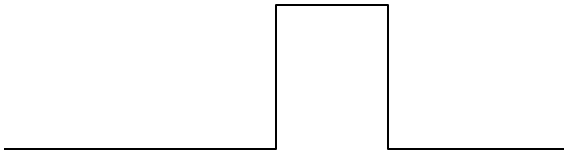



Environment
Agent
Motivation network
Sensors
Internal states
- metabolic budget
- age
- tiredness
- sexual drive
- last food intake
Mapping functions
f_1
f_2
f_3
f_n
Motivation variables
feeding
food searching
chirrup
signal tracking
breeding
Decision-maker
max
conflict priority list
Behaviour patterns
feeding
food searching
chirrup
signal tracking
breeding
Artificial genes
Physical Exchange
Motors


Bug
Internal state
Wander
Sleep
AvoidLeft
Eat
Rest
Stimuli
Sensors
Behaviour selection
Actuators
Response

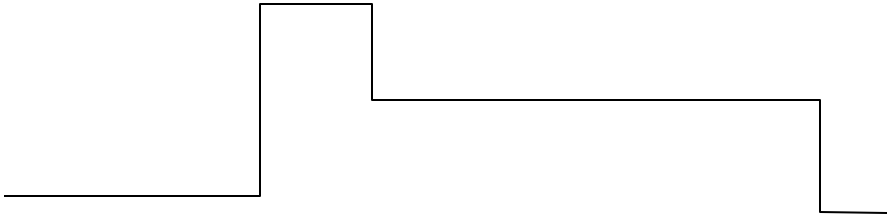

locomotion
behavior
control
main
sound
real time scheduler

Level triggered



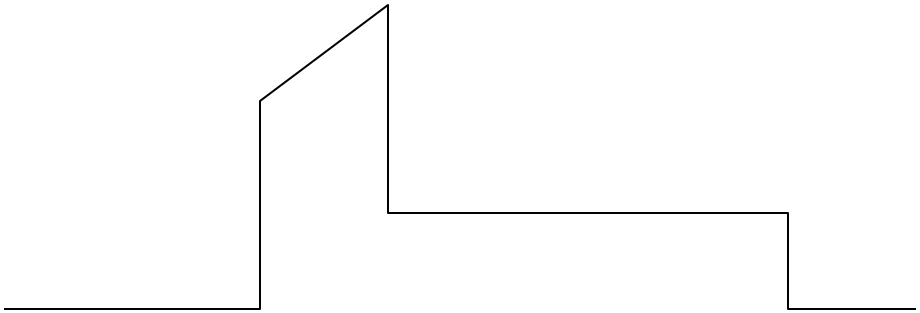
! cond cond ! cond

Edge triggered



! cond & ! active cond ! cond & active

Priority triggered



! cond & ! active cond ! cond & active

Lesson 9 leJOS Behavior API

Behavior Programming

- Programming Behavior with leJOS NXJ
- The Behavior API
  - Behavior class
  - Arbitrator class
- Coding behaviors
- Recommended design
- Summary

Package lejos.robotics.subsumption

Support for subsumption architecture.

See:

Description

Interface Summary	
Behavior	The Behavior interface represents an object embodying a specific behavior belonging to a robot.

Class Summary	
Arbitrator	Arbitrator controls which behavior should become active in a behavior control system.

lejos.robotics.subsumption

Interface Behavior

```
public interface Behavior
```

The Behavior interface represents an object embodying a specific behavior belonging to a robot. Each behavior must define three things:
1) The circumstances to make this behavior seize control of the robot. e.g. When the touch sensor determines the robot has collided with an object.
2) The action to perform when this behavior takes control. e.g. Back up and turn.
3) A way to quickly exit from the action when the Arbitrator selects a higher priority behavior to take control. These are represented by defining the methods takeControl(), action(), and suppress() respectively.
A behavior control system has one or more Behavior objects. When you have defined these objects, create an array of them and use that array to initialize an Arbitrator object.

Version:
0.9 May 2011
See Also:
Arbitrator

Method Summary

void	**action**() The code in action() represents the tasks the robot performs when this behavior becomes active.
void	**suppress**() The code in suppress() should cause the current behavior to exit.
boolean	**takeControl**() The boolean return indicates if this behavior should seize control of the robot.

lejos.robotics.subsumption

Class Arbitrator

java.lang.Object
  └ **lejos.robotics.subsumption.Arbitrator**

```
public class Arbitrator
extends Object
```

Arbitrator controls which Behavior object will become active in a behavior control system. Make sure to call start() after the Arbitrator is instantiated.
This class has three major responsibilities:
1. Determine the highest priority behavior that returns **true** to takeControl()
2. Suppress the active behavior if its priority is less than highest priority.
3. When the action() method exits, call action() on the Behavior of highest priority.
The Arbitrator assumes that a Behavior is no longer active when action() exits,
therefore it will only call suppress() on the Behavior whose action() method is running.
It can make consecutive calls of action() on the same Behavior.
Requirements for a Behavior:
When suppress() is called, terminate action() immediately.
When action() exits, the robot is in a safe state (e.g. motors stopped)

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;

/**
 * Demonstration of the Behavior subsumption classes.
 *
 * Requires a wheeled vehicle with two independently controlled
 * motors connected to motor ports A and C, and
 * a touch sensor connected to sensor  port 1 and
 * an ultrasonic sensor connected to port 3;
 *
 * @author Brian Bagnall and Lawrie Griffiths, modified by Roger Glassey
 *
 */
public class BumperCar
{

  public static void main(String[] args)
  {
    Motor.A.setSpeed(400);
    Motor.C.setSpeed(400);
    Behavior b1 = new DriveForward();
    Behavior b2 = new DetectWall();
    Behavior[] behaviorList =
    {
      b1, b2
    };
    Arbitrator arbitrator = new Arbitrator(behaviorList);
    LCD.drawString("Bumper Car",0,1);
    Button.waitForPress();
    arbitrator.start();
  }
}
```

```
class DriveForward implements Behavior
{

  private boolean _suppressed = false;

  public boolean takeControl()
  {
    return true;  // this behavior always wants control.
  }

  public void suppress()
  {
    _suppressed = true;// standard practice for suppress methods
  }

  public void action()
  {
    _suppressed = false;
    Motor.A.forward();
    Motor.C.forward();
    while (!_suppressed)
    {
      Thread.yield(); //don't exit till suppressed
    }
    Motor.A.stop(); // not strictly necessary, but good programming practice
    Motor.C.stop();
  }
}
```

```
class DetectWall implements Behavior
{

  public DetectWall()
  {
    touch = new TouchSensor(SensorPort.S1);
    sonar = new UltrasonicSensor(SensorPort.S3);
  }

  public boolean takeControl()
  {
    sonar.ping();
    //Sound.pause(20);
    return touch.isPressed() || sonar.getDistance() < 25;
  }

  public void suppress()
  {
    //Since  this is highest priority behavior, suppress will never be called.
  }

  public void action()
  {
    Motor.A.rotate(-180, true);// start Motor.A rotating backward
    Motor.C.rotate(-360);  // rotate C farther to make the turn
  }
  private TouchSensor touch;
  private UltrasonicSensor sonar;
}
```

```
private class Monitor extends Thread
{

  boolean more = true;
  int maxPriority = _behavior.length - 1;

  public void run()
  {
    while (more)
    {
      //FIND HIGHEST PRIORITY BEHAVIOR THAT WANTS CONTROL
      synchronized (this)
      {
         _highestPriority = NONE;
        for (int i = maxPriority; i >= 0; i--)
        {
          if (_behavior[i].takeControl())
          {
            _highestPriority = i;
            break;
          }
        }
        int active = _active;// local copy: avoid out of bounds error in 134
        if (active != NONE && _highestPriority > active)
        {
          _behavior[active].suppress();
        }
      }// end synchronize block - main thread can run now
      Thread.yield();
    }
  }
}
```

```
public void start()
{
  monitor.start();
  while (_highestPriority == NONE)
  {
    Thread.yield();//wait for some behavior to take contro
  }
  while (true)
  {
    synchronized (monitor)
    {
      if (_highestPriority != NONE)
      {
        _active = _highestPriority;

      } else if (_returnWhenInactive)
      {// no behavior wants to run
        monitor.more = false;//9 shut down monitor thread
        return;
      }
    }// monotor released before action is called
    if (_active != NONE)  //_highestPrioirty could be NONE
    {
      _behavior[_active].action();
      _active = NONE;  // no active behavior at the moment
    }
    Thread.yield();
  }
}
```