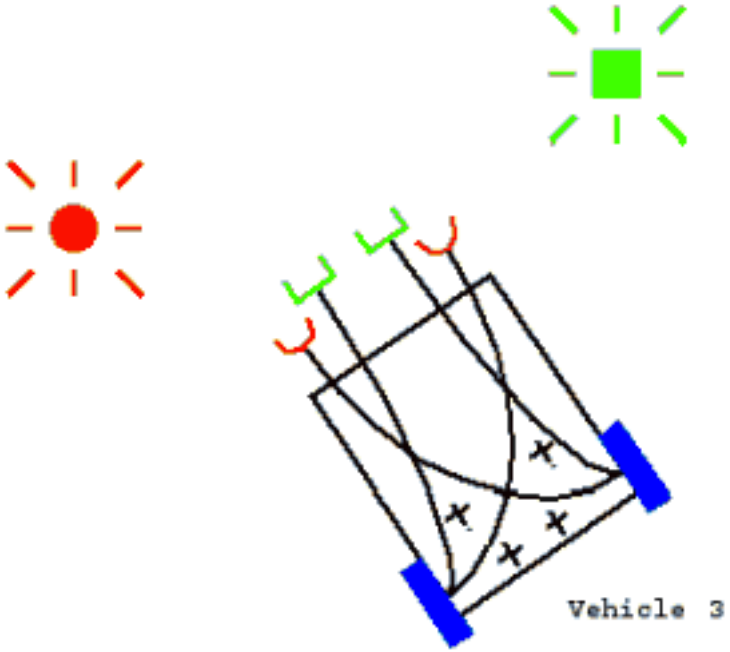
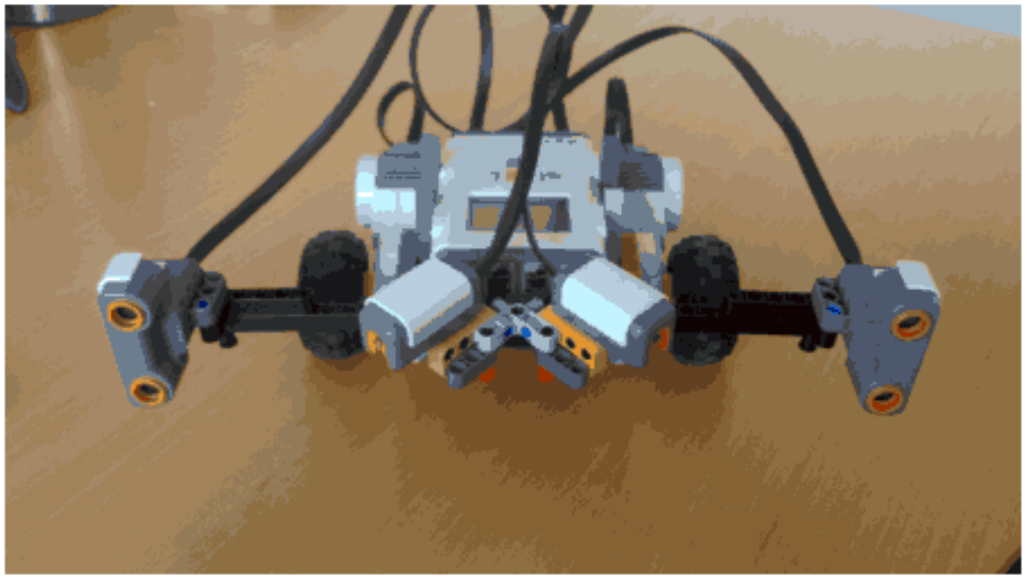
Braitenberg Vehicle 3

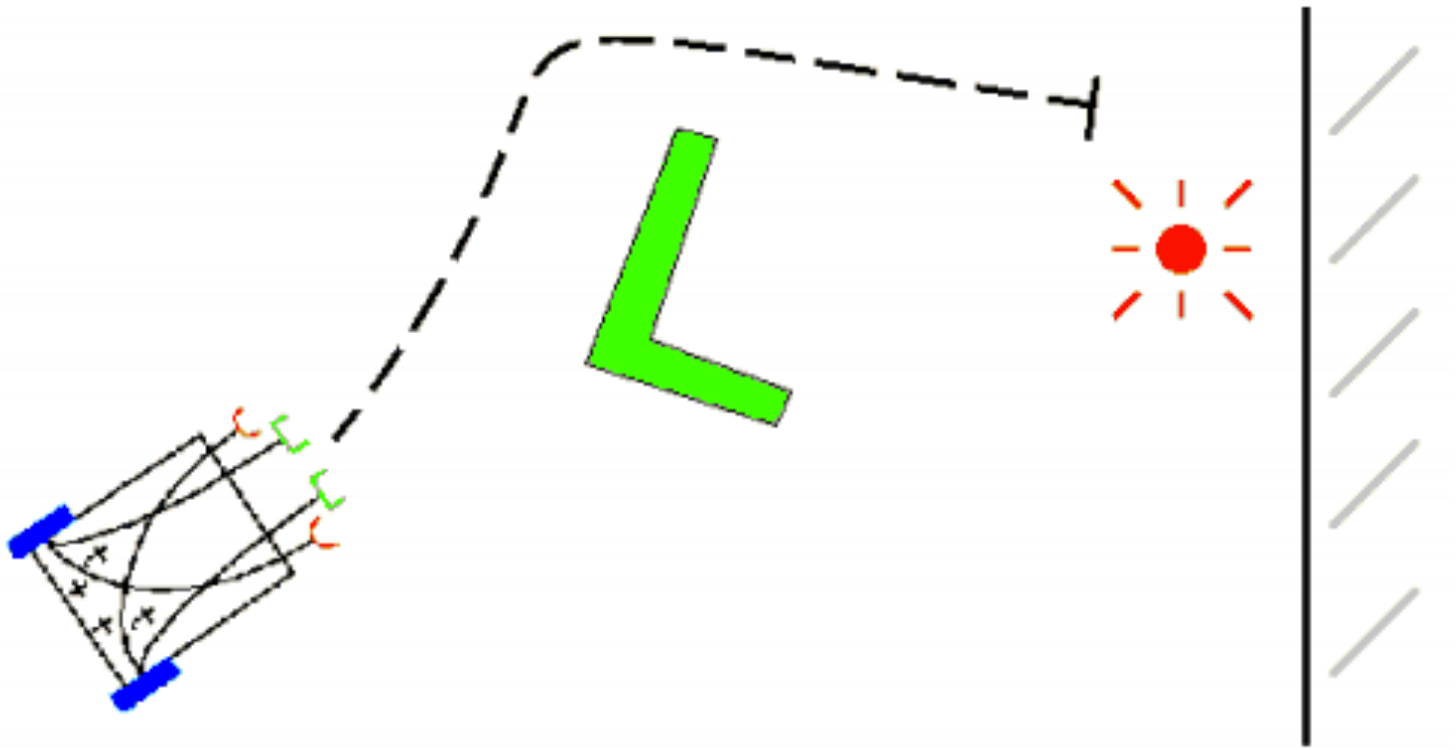


Figur 4: Braitenberg vehicles 3.



Figur 3: Viser Vehicle 3, hvor der er fire sensorer monteret.

Braitenberg Vehicle 3



Braitenberg Vehicle 3

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo 864 September, 1985

A ROBUST LAYERED CONTROL SYSTEM FOR A MOBILE ROBOT

Rodney A. Brooks

Abstract. We describe a new architecture for controlling mobile robots. Layers of control system are built to let the robot operate at increasing levels of competence. Layers are made up of asynchronous modules which communicate over low bandwidth channels. Each module is an instance of a fairly simple computational machine. Higher level layers can subsume the roles of lower levels by suppressing their outputs. However, lower levels continue to function as higher levels are added. The result is a robust and flexible robot control system. The system is intended to control a robot that wanders the office areas of our laboratory, building maps of its surroundings. In this paper we demonstrate the system controlling a detailed simulation of the robot.

Rodney Brooks

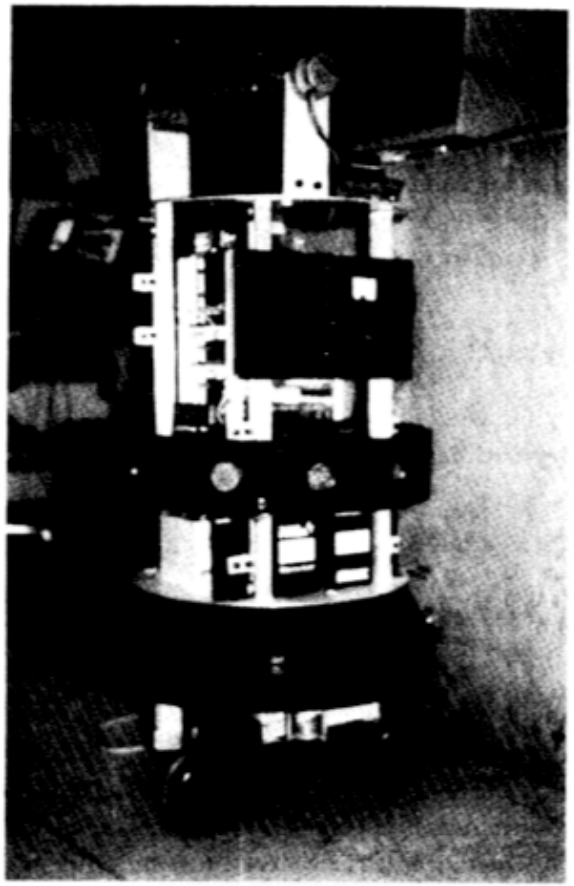
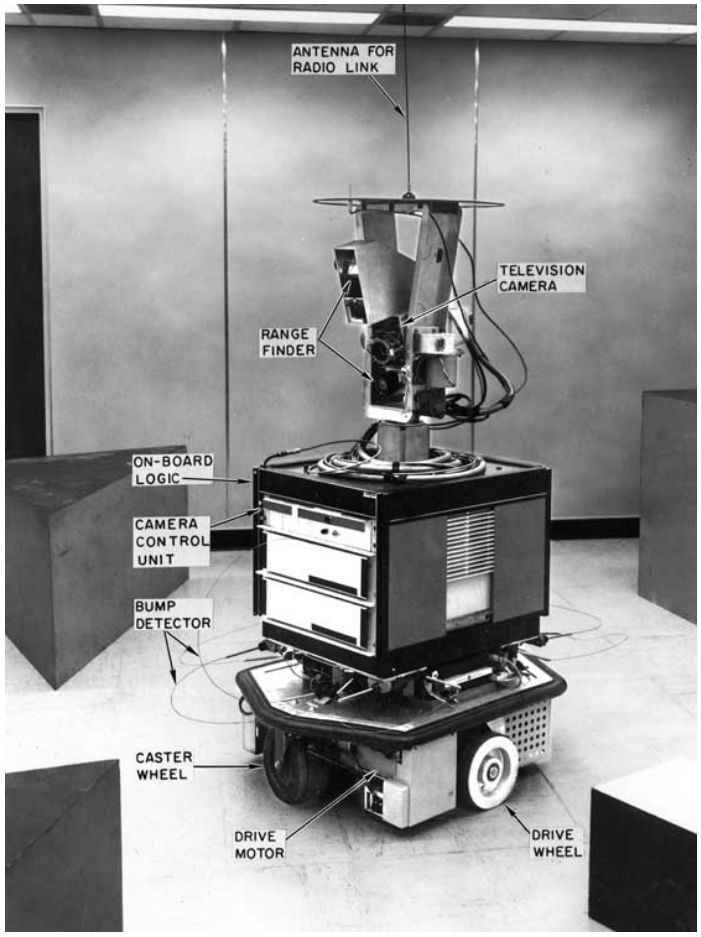


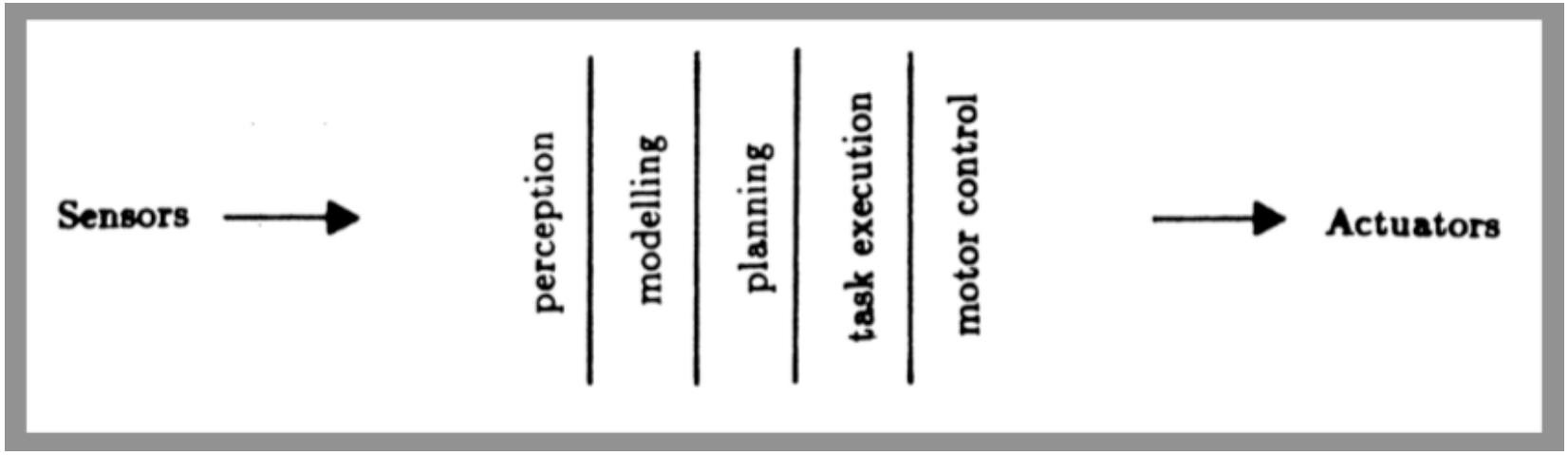
Photo 1. The MIT AI Lab mobile robot.

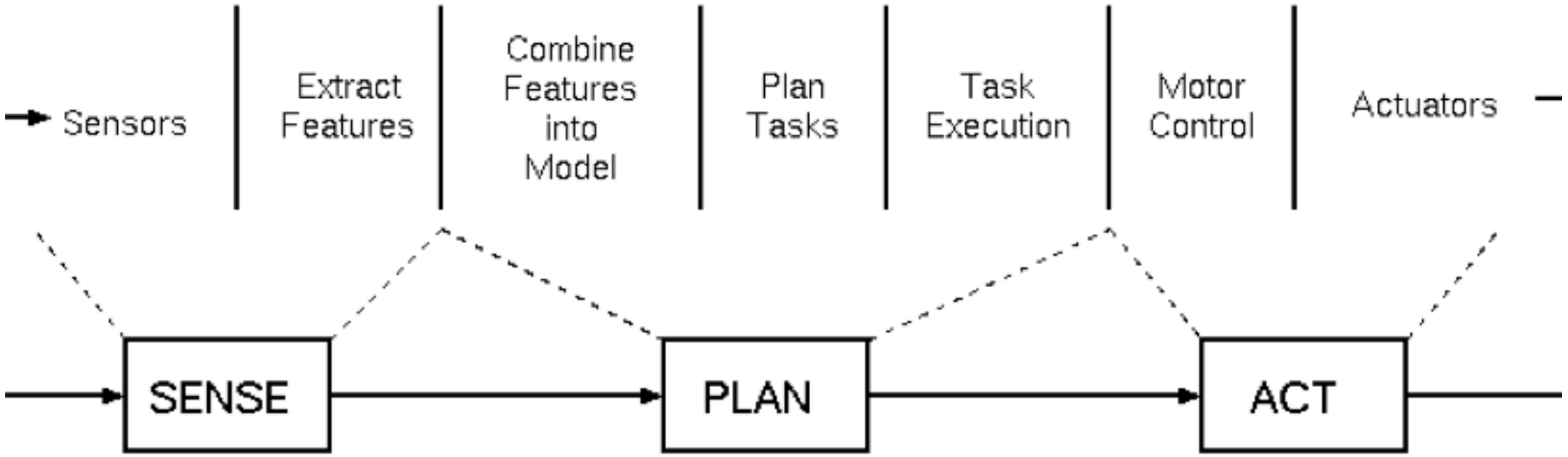


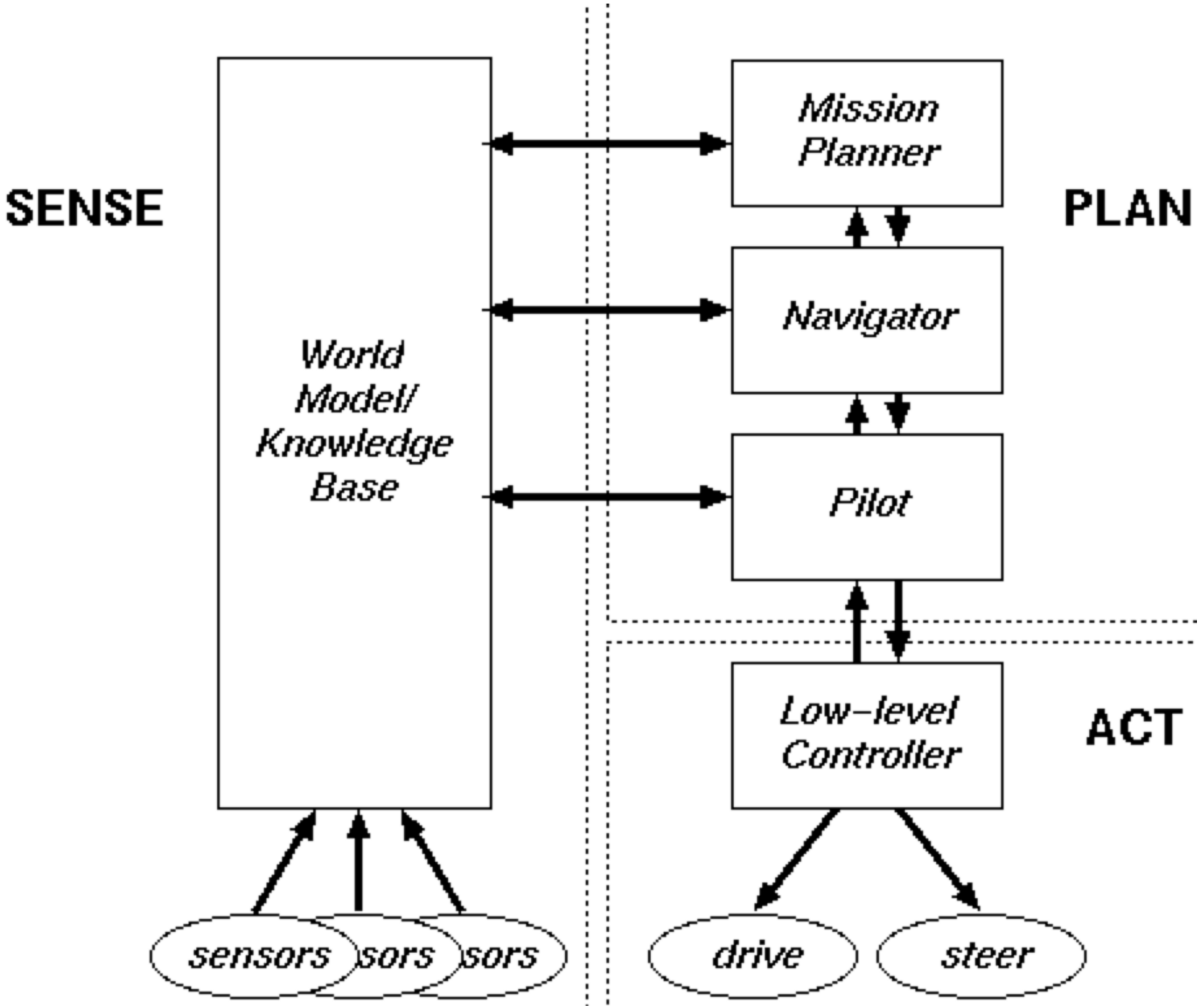
Shakey, 1966

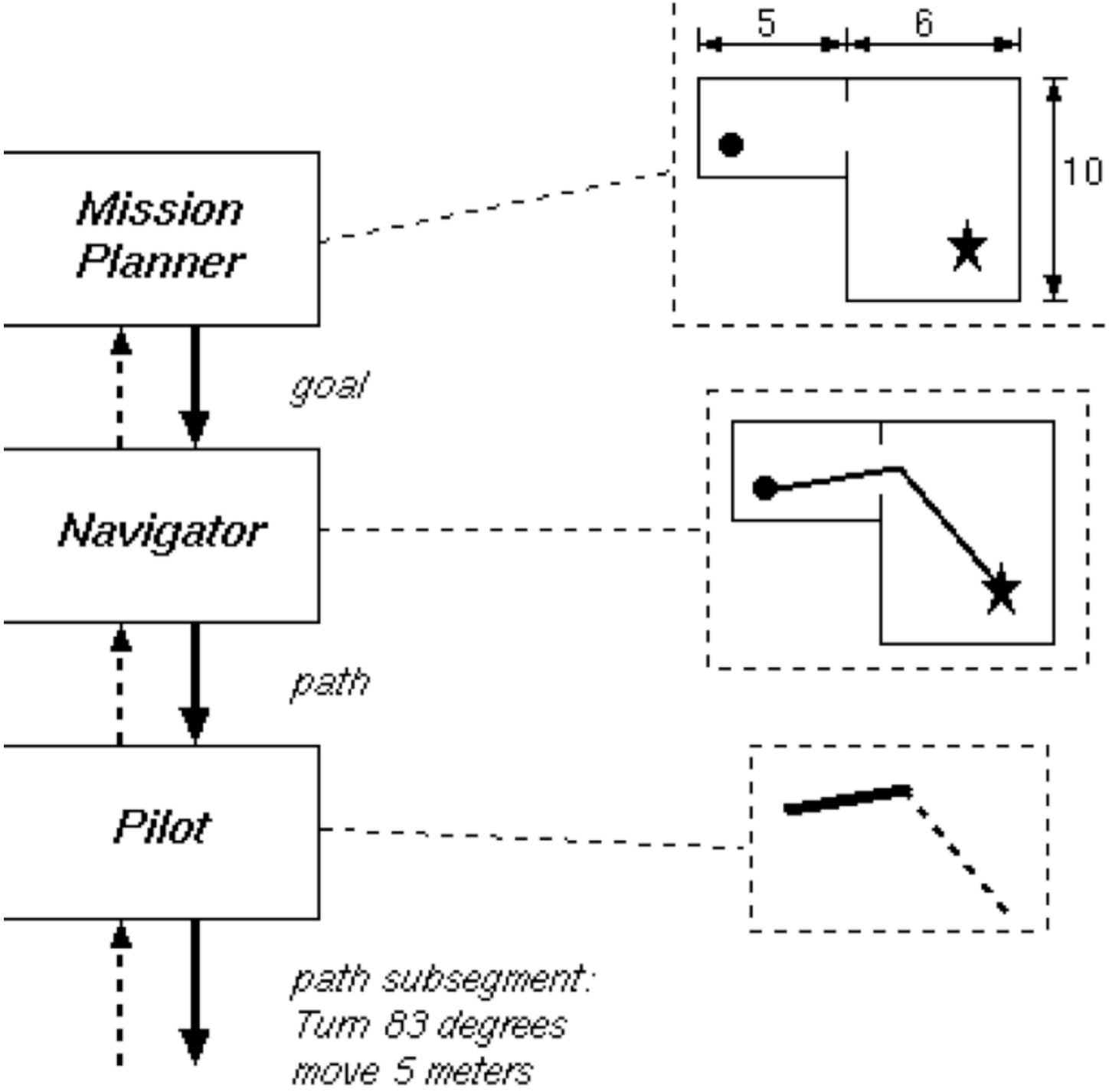
Bump detectors
Camera
Range finders

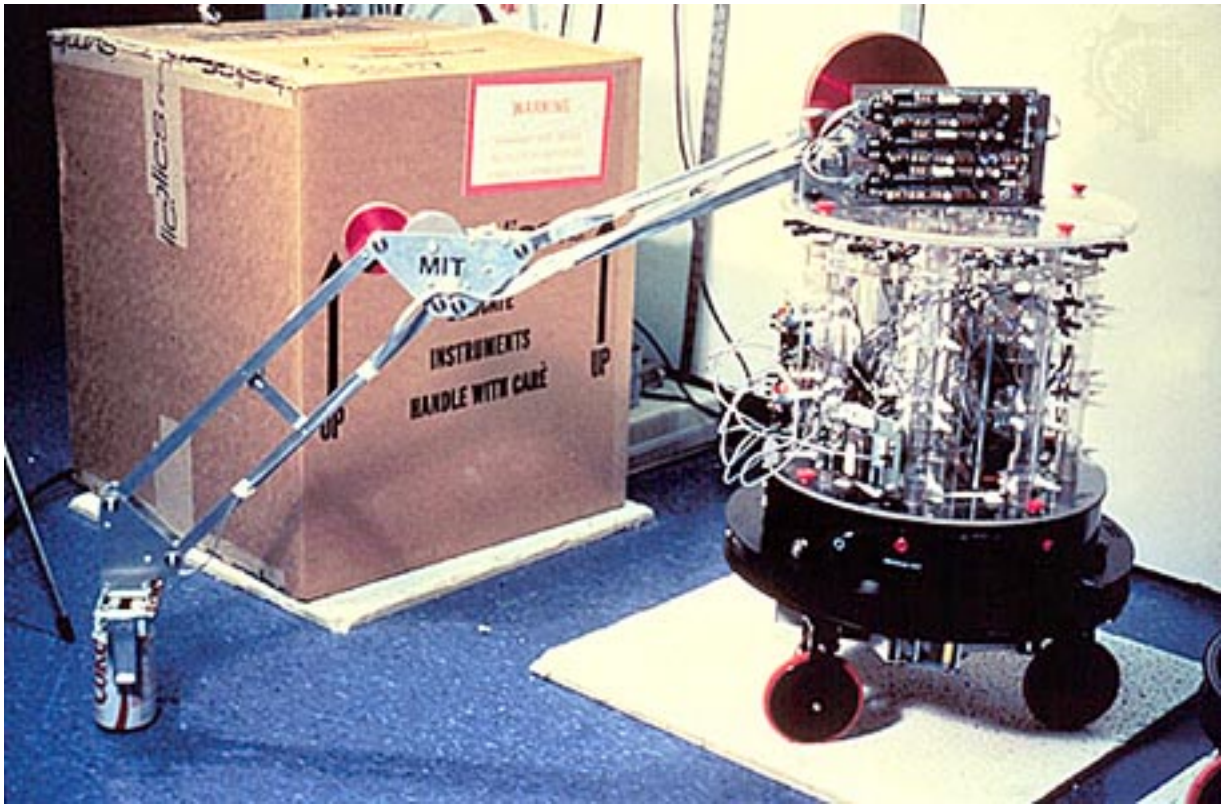
Sense-Plan-Act




Sensors
Extract Features
Combine Features into Model
Plan Tasks
Task Execution
Motor Control
Actuators
SENSE
PLAN
ACT


SENSE
PLAN
ACT
World Model/ Knowledge Base
Mission Planner
Navigator
Pilot
Low-level Controller
sensors
sors
sors
drive
steer


5
6
10
Mission Planner
goal
Navigator
path
Pilot
path subsegment:
Turn 83 degrees
move 5 meters



Rodney Brooks, Herbert, collect coke cans

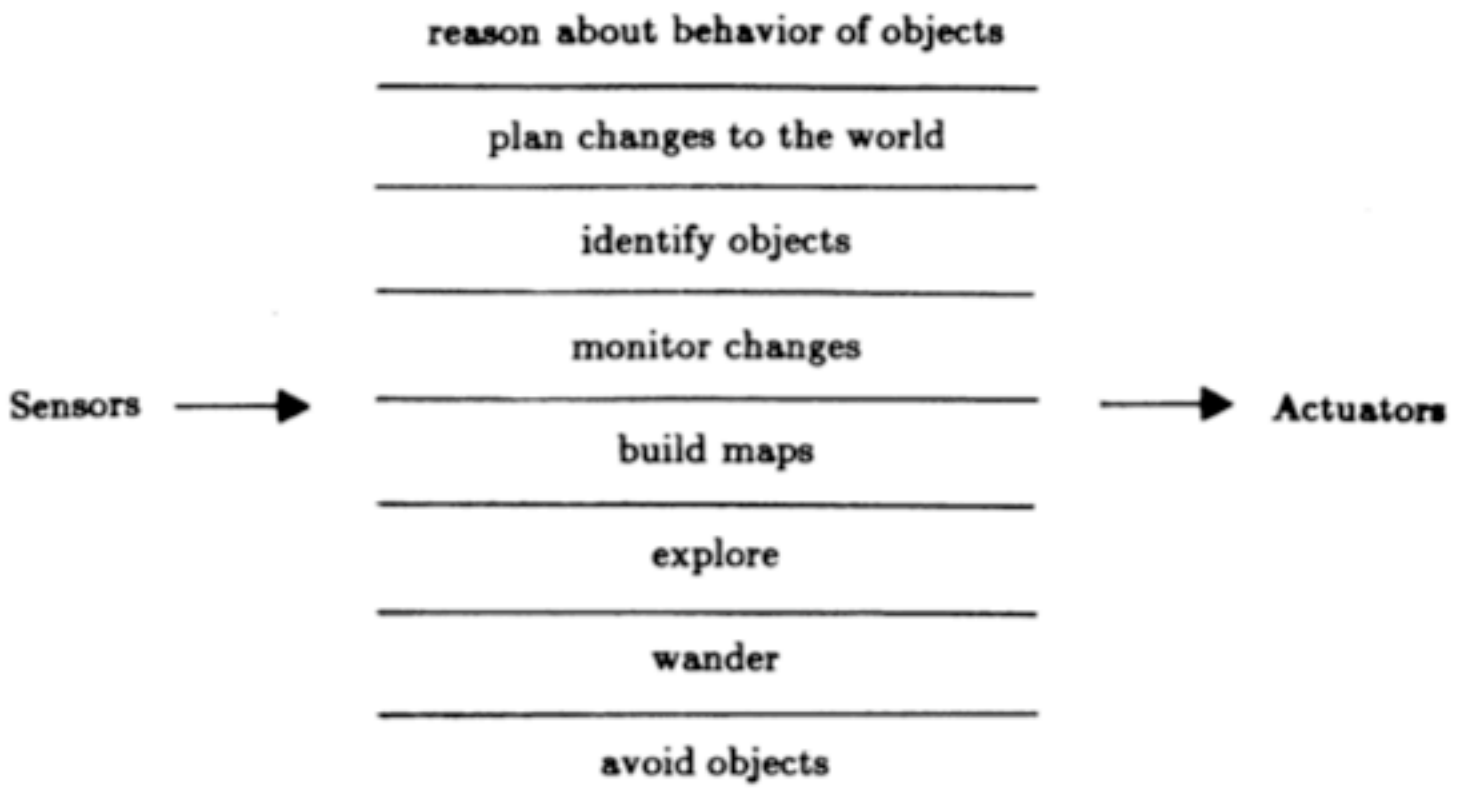
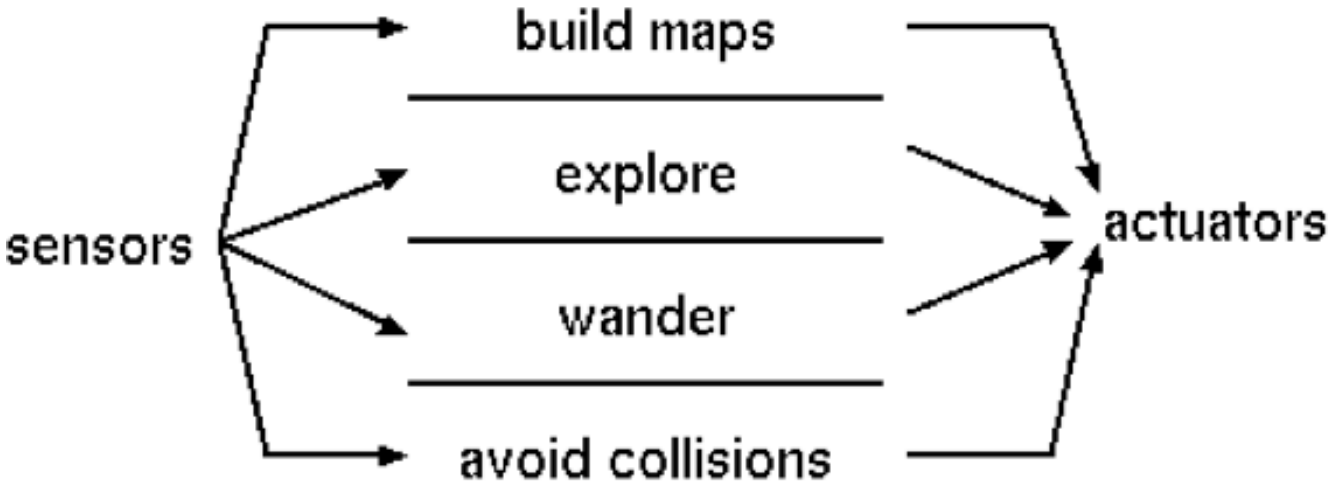
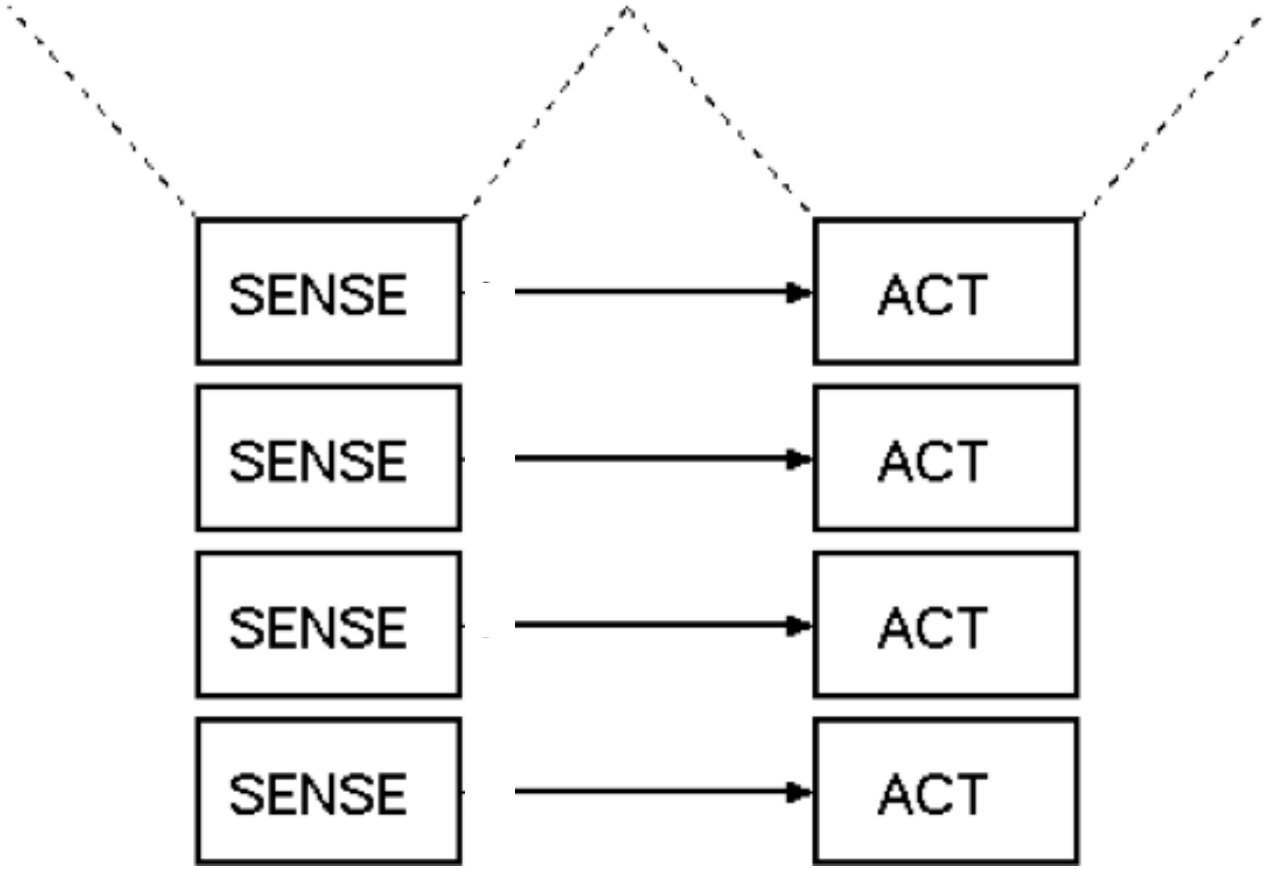
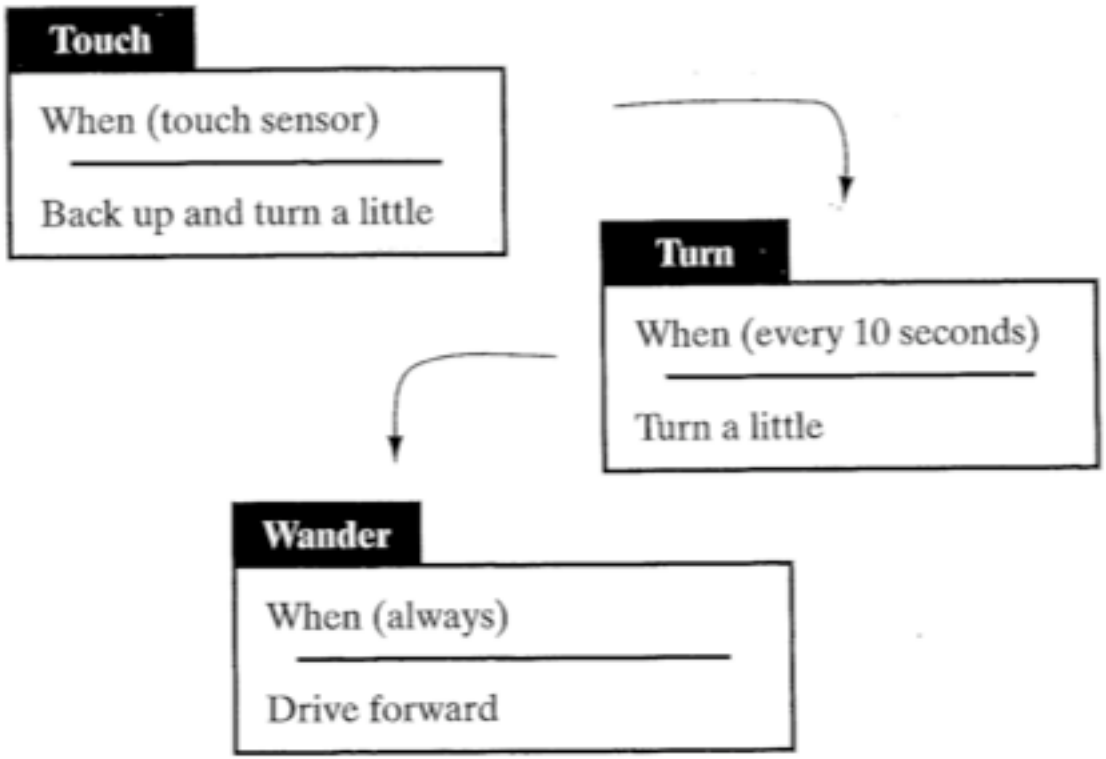


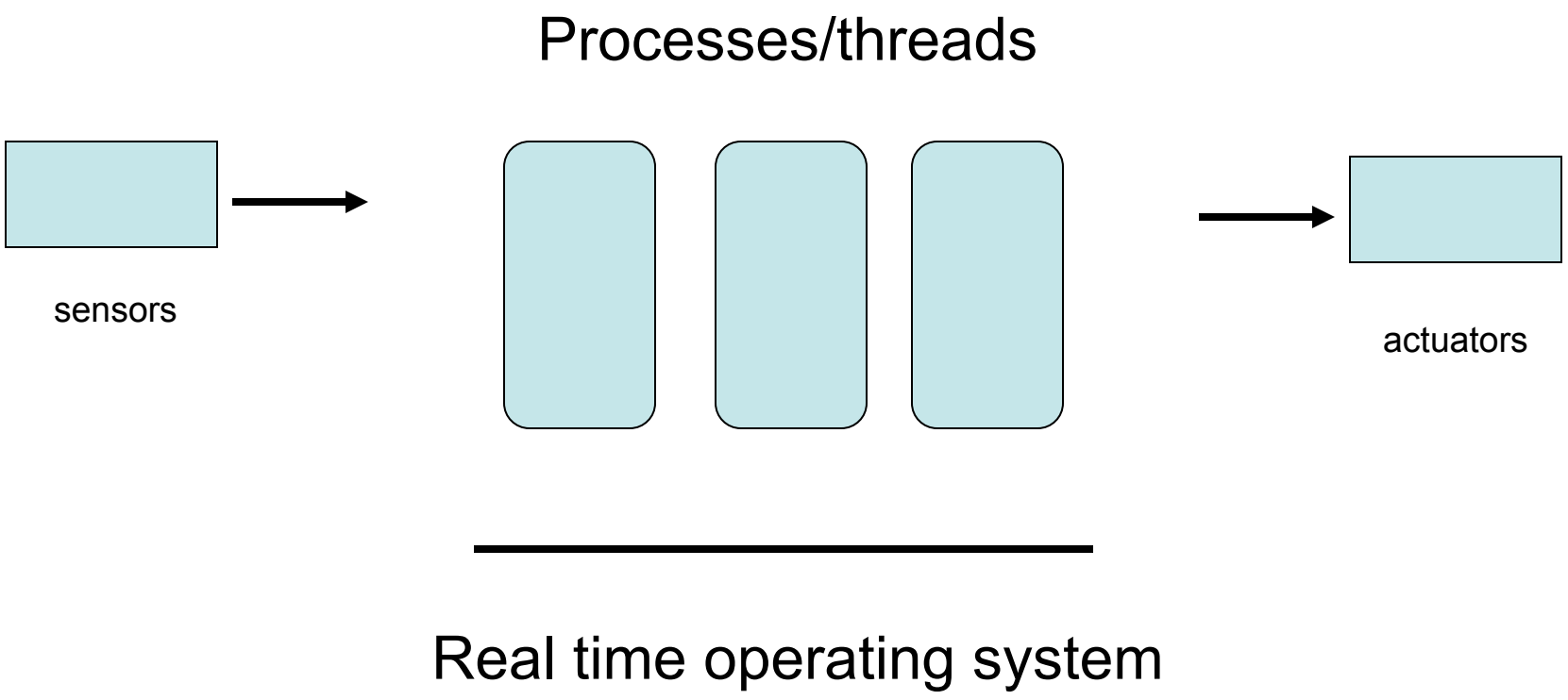
Figure 2. A decomposition of a mobile robot control system based on task achieving behaviors.


build maps
explore
wander
avoid collisions
sensors
actuators

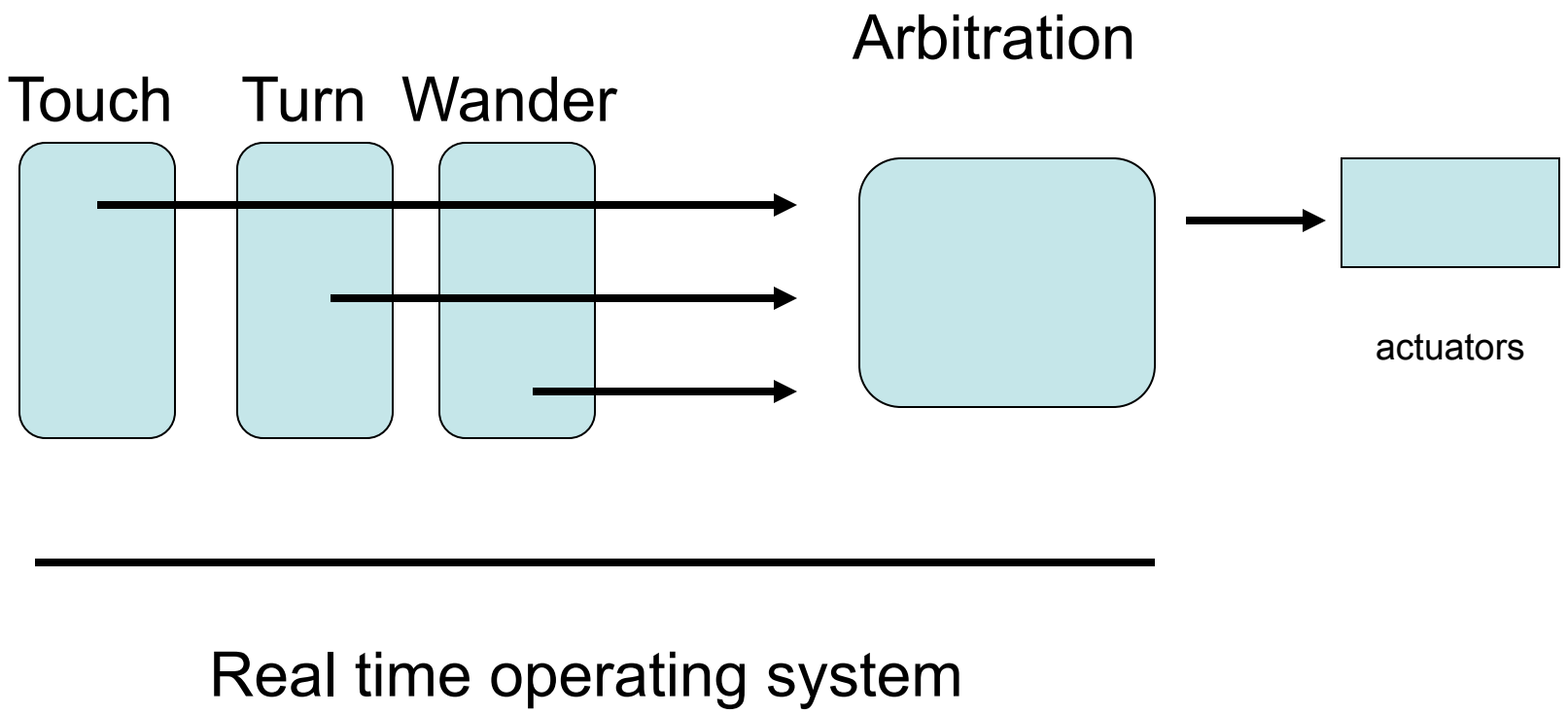

SENSE
ACT
SENSE
ACT
SENSE
ACT
SENSE
ACT

Fred Martin, Reactive Strategy

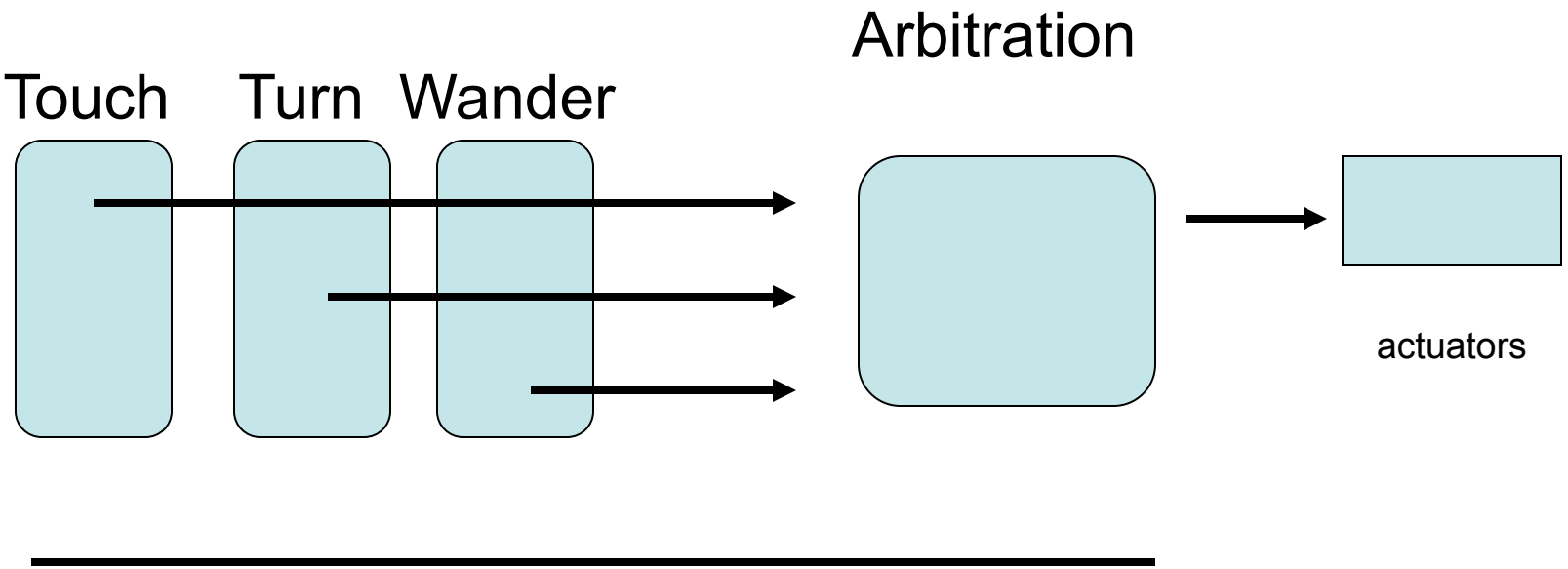



Processes/threads
sensors
actuators
Real time operating system

Fred Martin, Central Arbiter



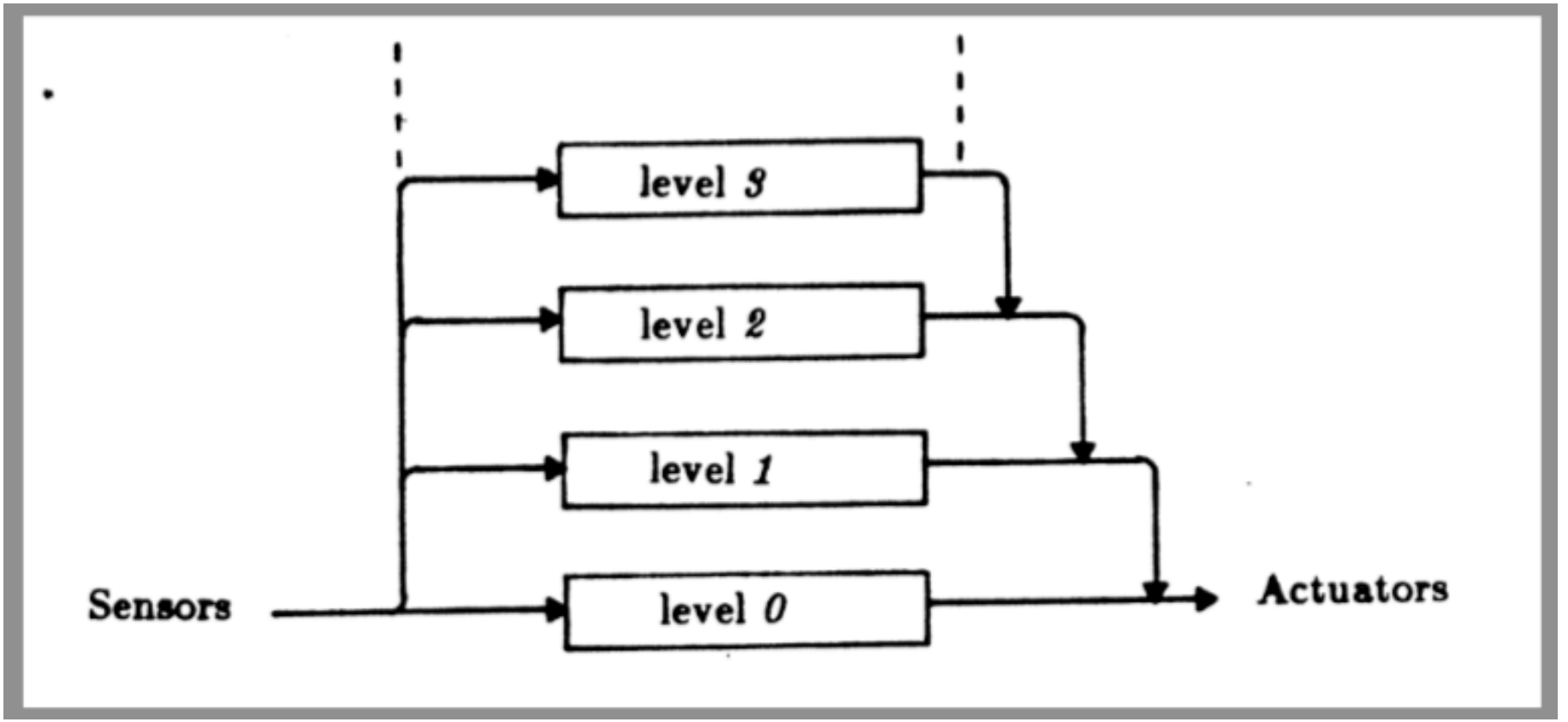
Fred Martin, Central Arbiter

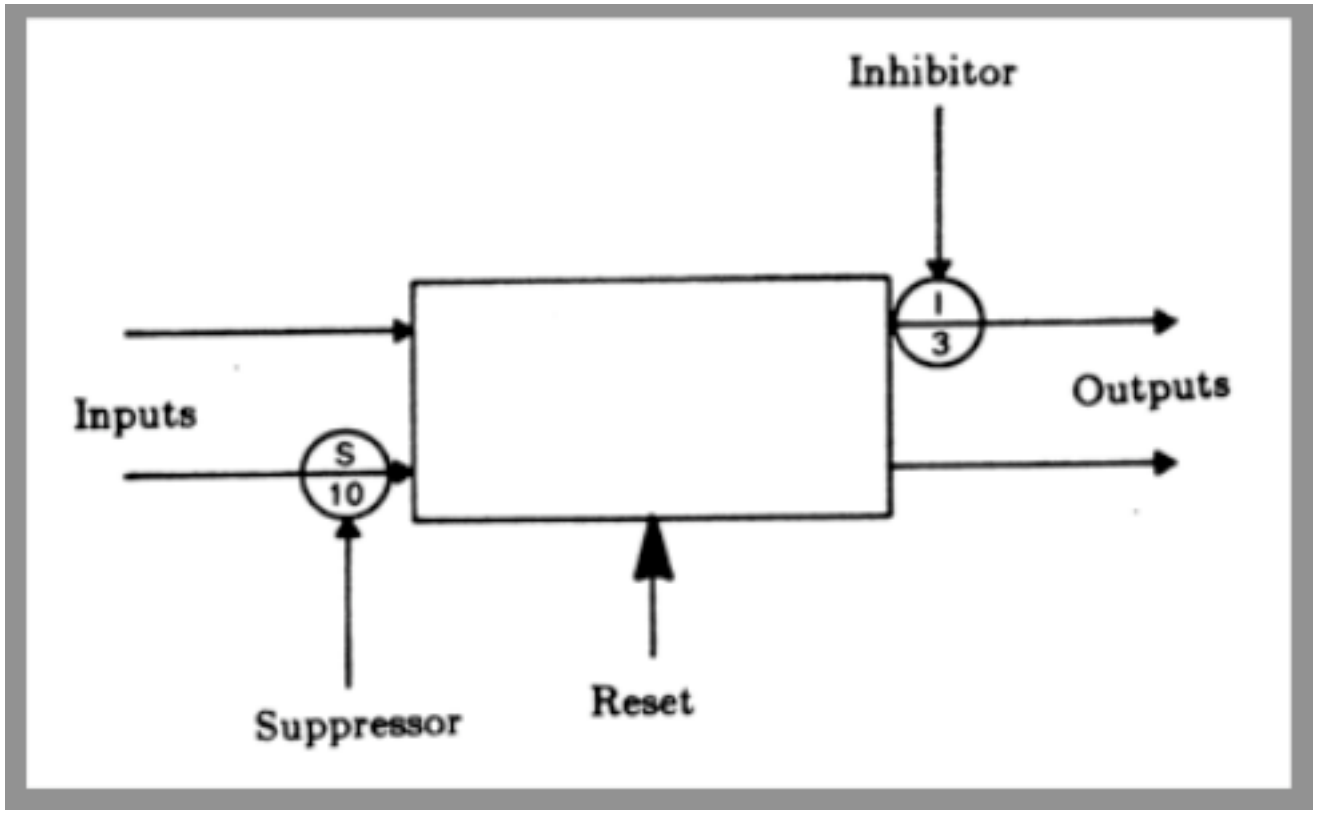


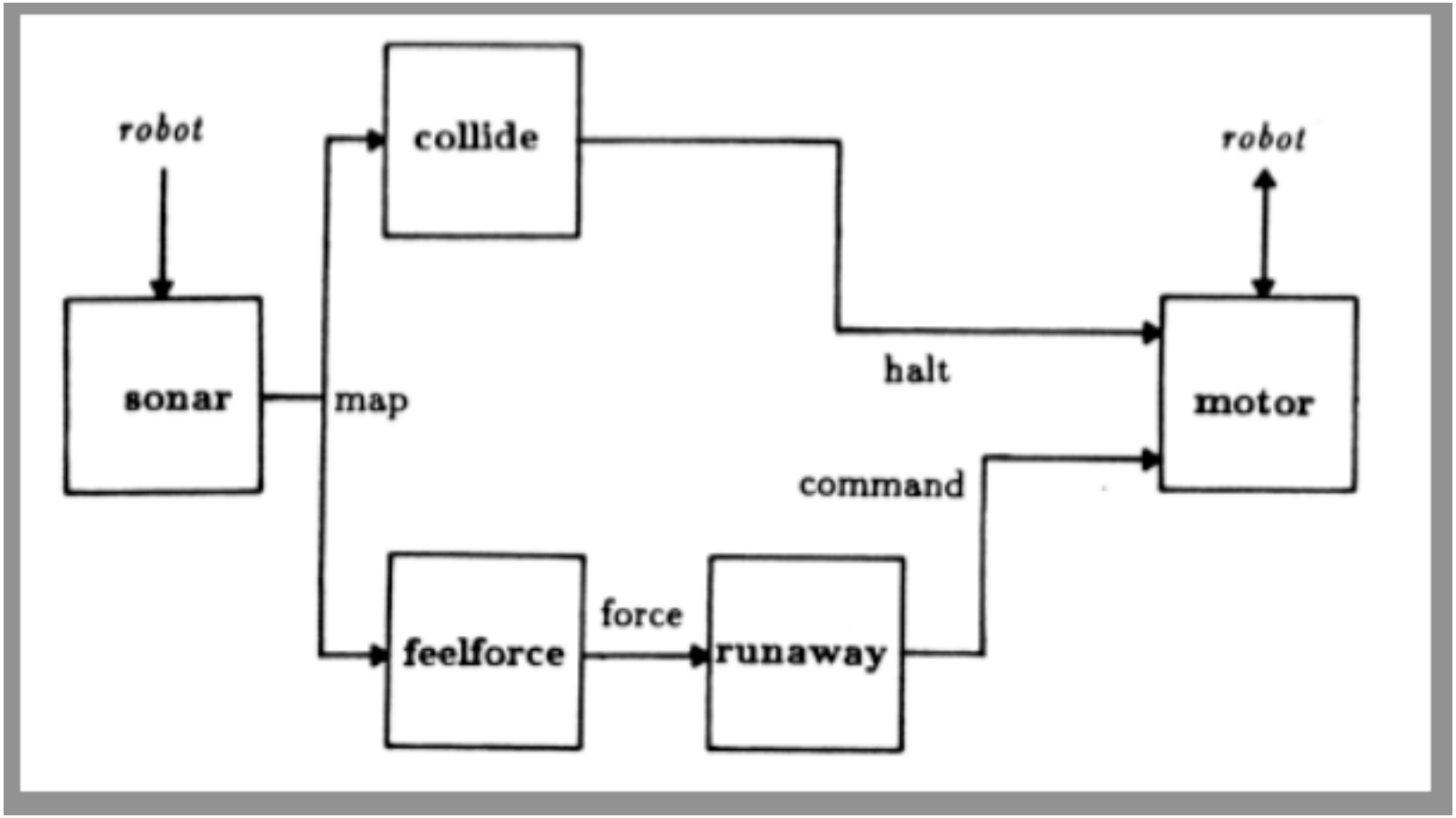
Real time operating system

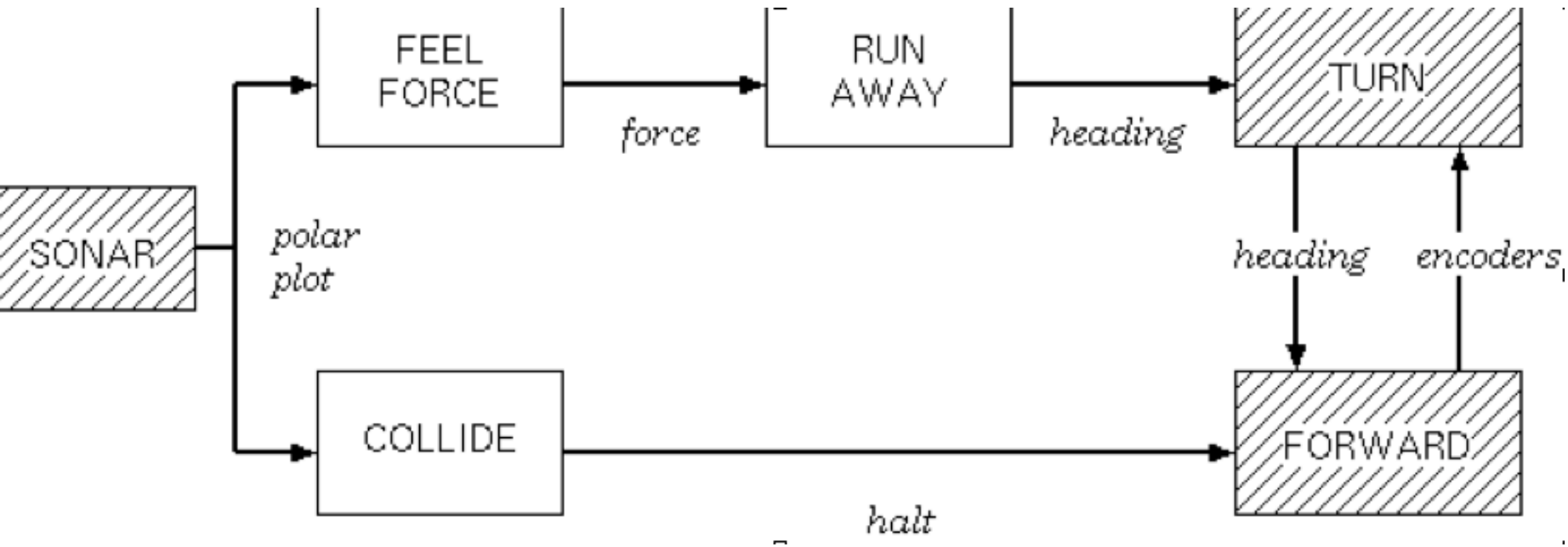
Rodney Brooks, ”No need for a central arbiter”

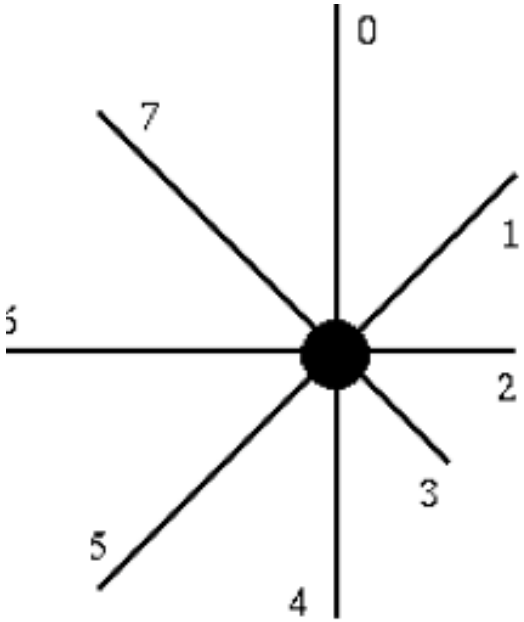
Subsumption architecture

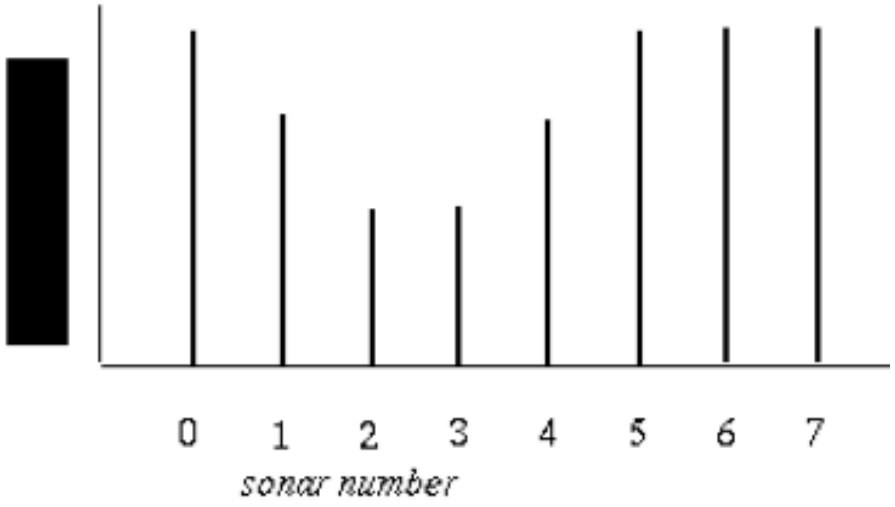



Inhibitor
I
3
Outputs
Inputs
S
10
Suppressor
Reset


robot
collide
robot
sonar
map
halt
motor
command
feelforce
force
runaway


FEEL FORCE
RUN AWAY
TURN
force
heading
SONAR
polar plot
heading
encoders
COLLIDE
FORWARD
halt


0
1
2
3
4
5
6
7


0 1 2 3 4 5 6 7
sonar number

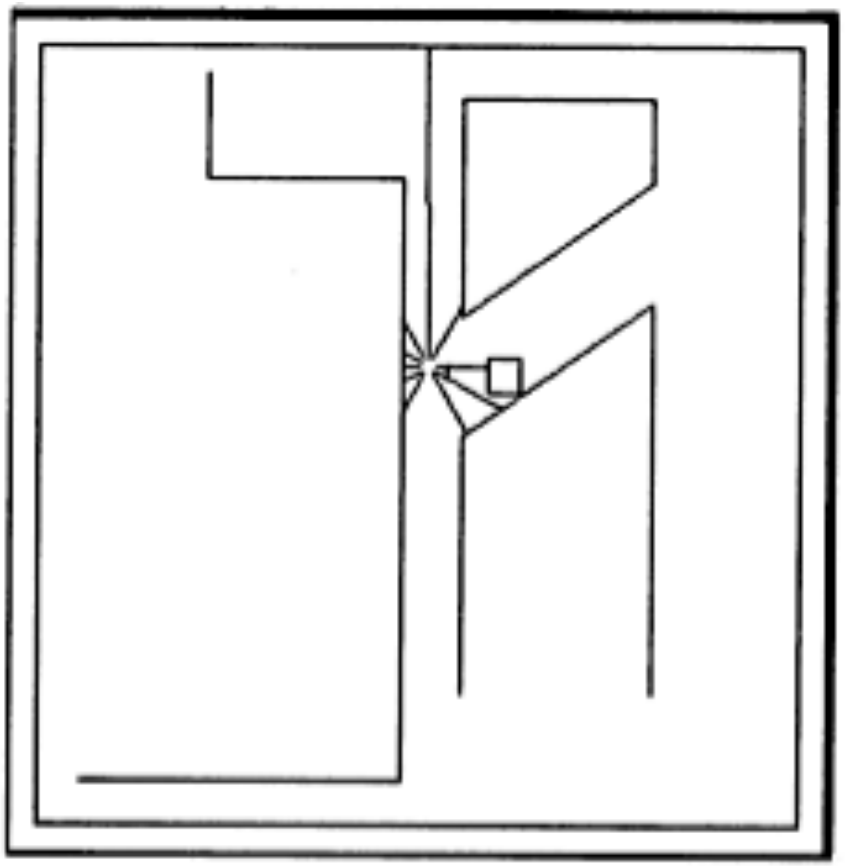
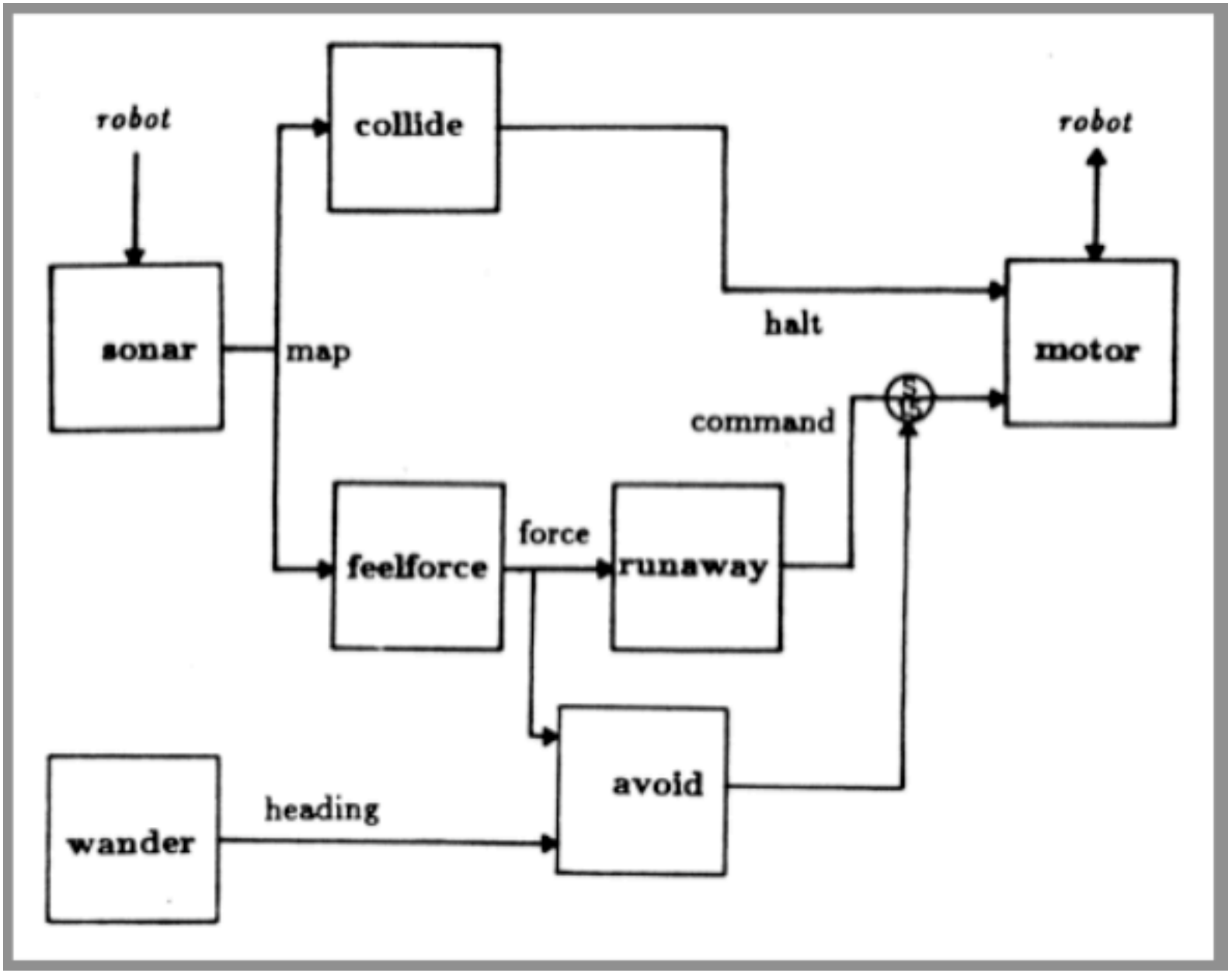
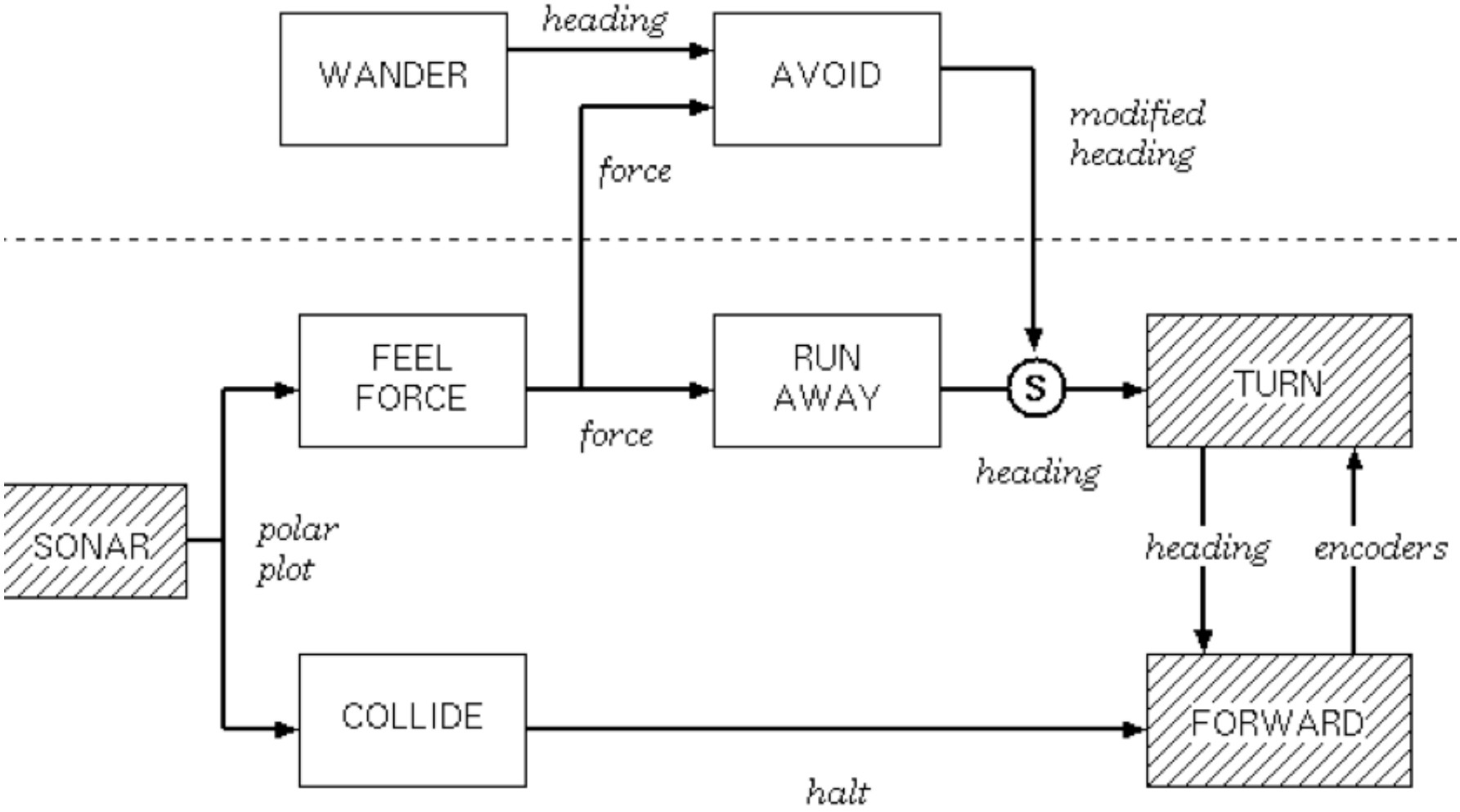
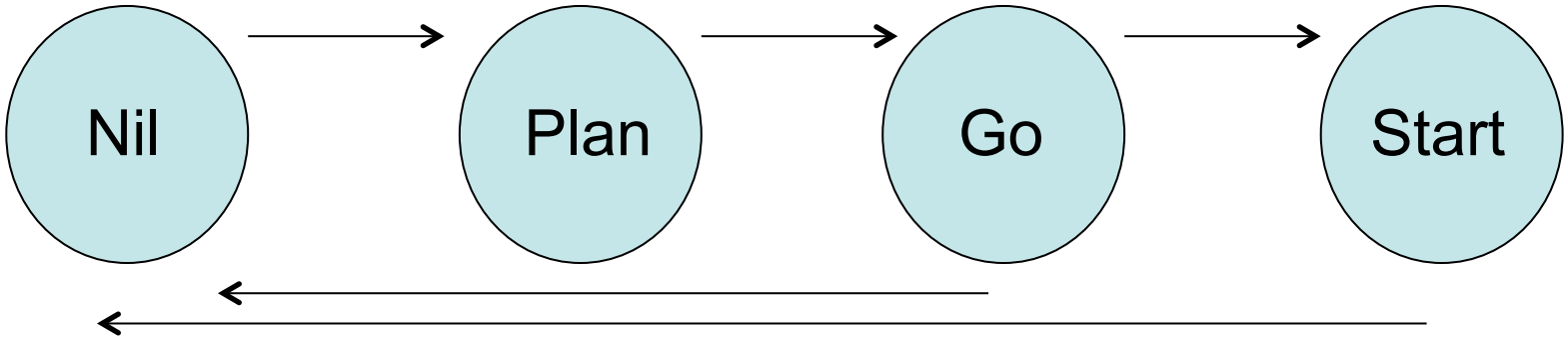


Figure 8. The simulated robot receives 12 sonar readings. Some sonar beams glance off walls and do not return within a certain time.


robot
collide
robot
sonar
map
halt
motor
command
feelforce
force
runaway
avoid
wander
heading


WANDER
heading
AVOID
modified
heading
force
FEEL
FORCE
RUN
AWAY
S
TURN
force
heading
SONAR
polar
plot
heading
encoders
COLLIDE
FORWARD
halt



Lisp

```
(defmodule avoid
  :inputs (force heading)
  :outputs (command)
  :instance-vars (resultforce)
  :states
    ((nil (event-dispatch (and force heading) plan))
     (plan (setf resultforce (select-direction force heading))
           go)
     (go (conditional-dispatch (significant-force-p resultforce 1.0)
                               start
                               nil))
     (start (output command (follow-force resultforce))
            nil)))
```

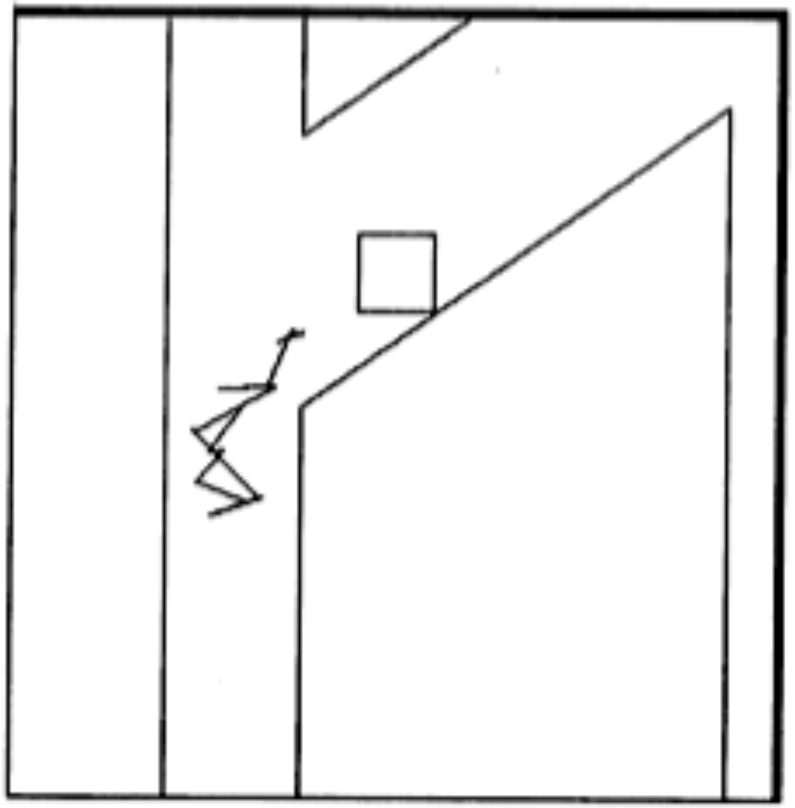
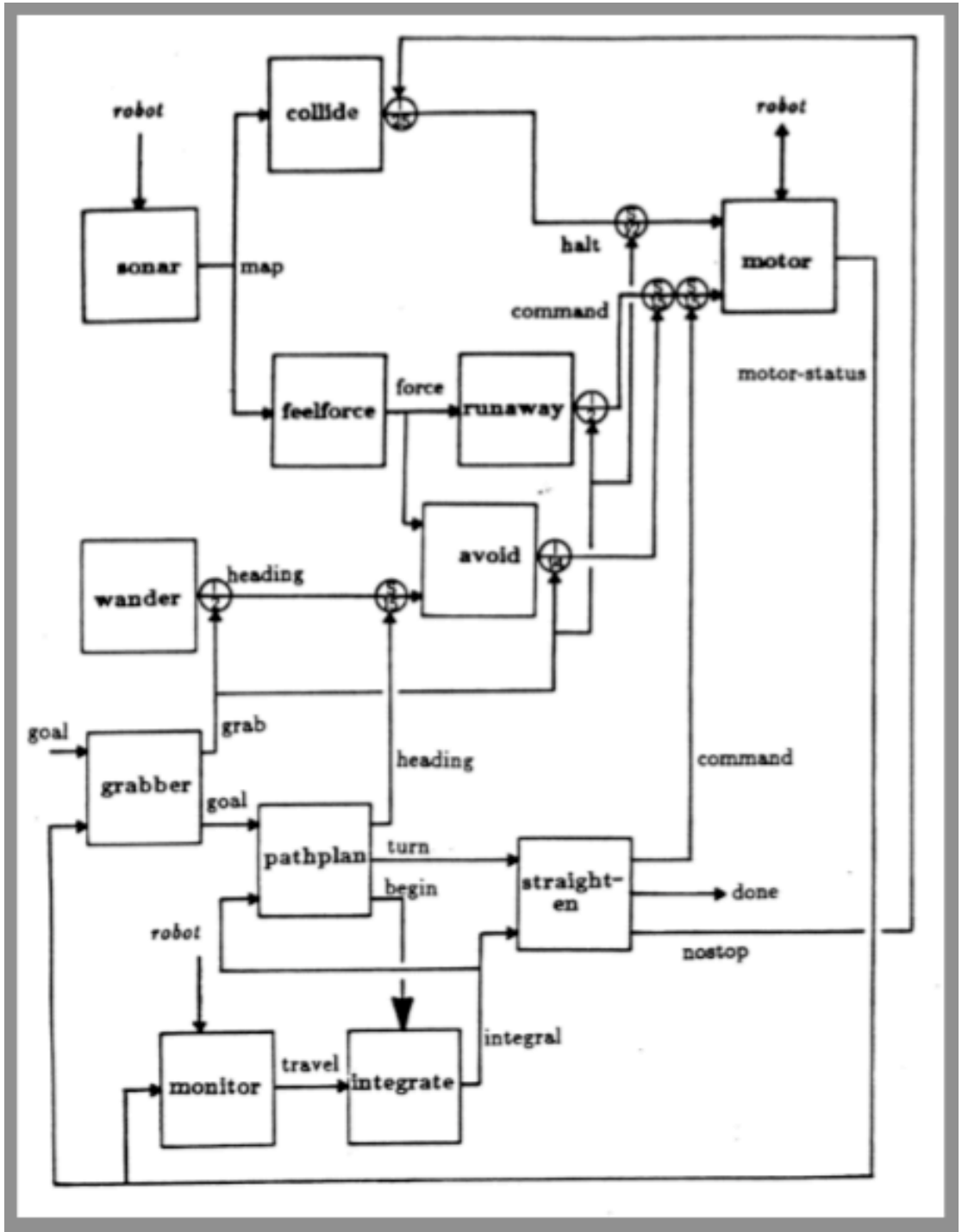
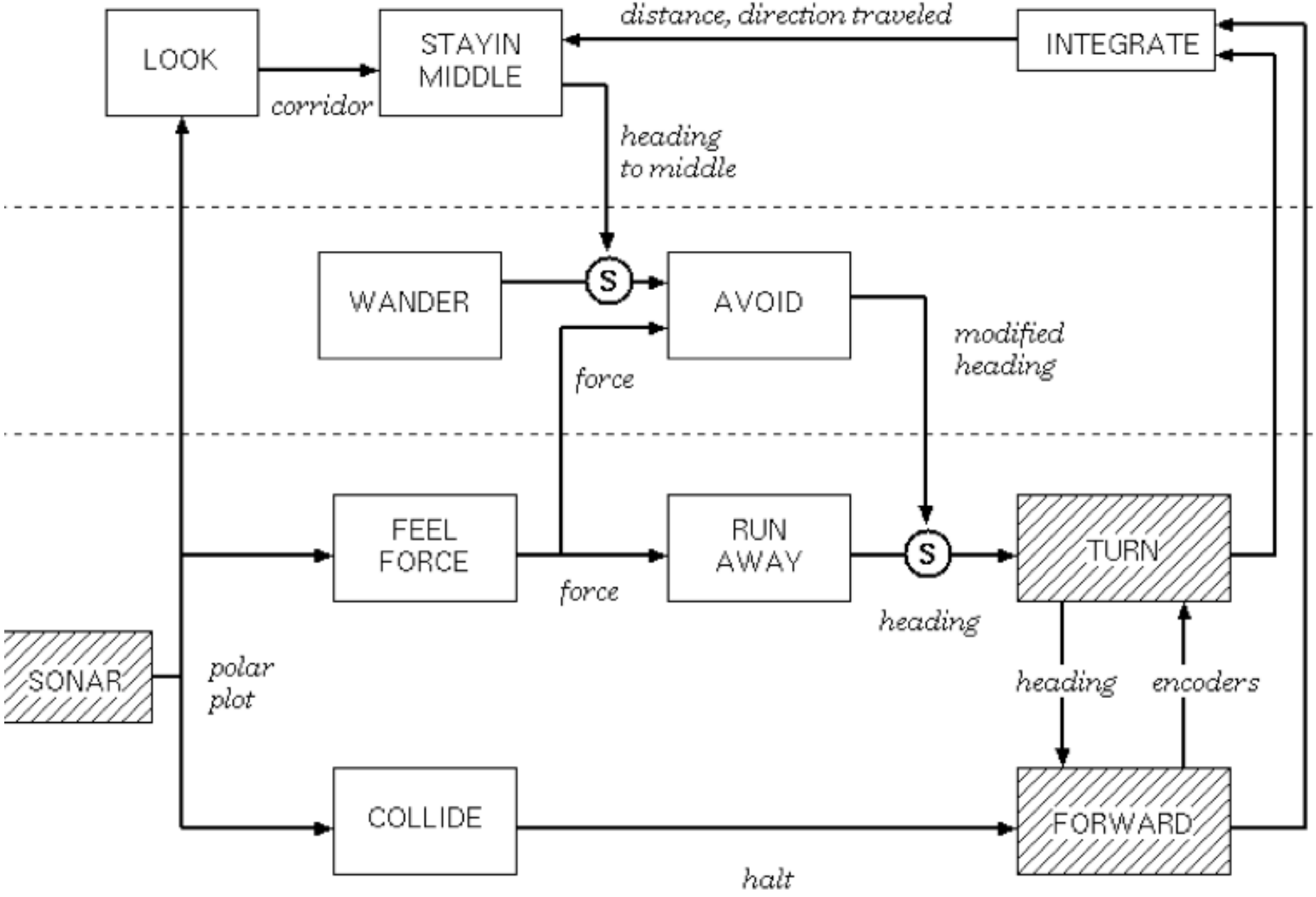


Figure 9. Under levels *0* and *1* control the robot wanders around aimlessly. It does not hit obstacles.


robot
collide
robot
sonar
map
halt
motor
command
motor-status
feelforce
force
runaway
avoid
wander
heading
goal
grab
grabber
heading
command
goal
pathplan
turn
straight-
en
done
begin
robot
nostop
integral
monitor
travel
integrate


LOOK
STAYIN MIDDLE
distance, direction traveled
INTEGRATE
corridor
heading to middle
WANDER
S
AVOID
modified heading
force
FEEL FORCE
RUN AWAY
S
TURN
force
heading
SONAR
polar plot
heading
encoders
COLLIDE
FORWARD
halt

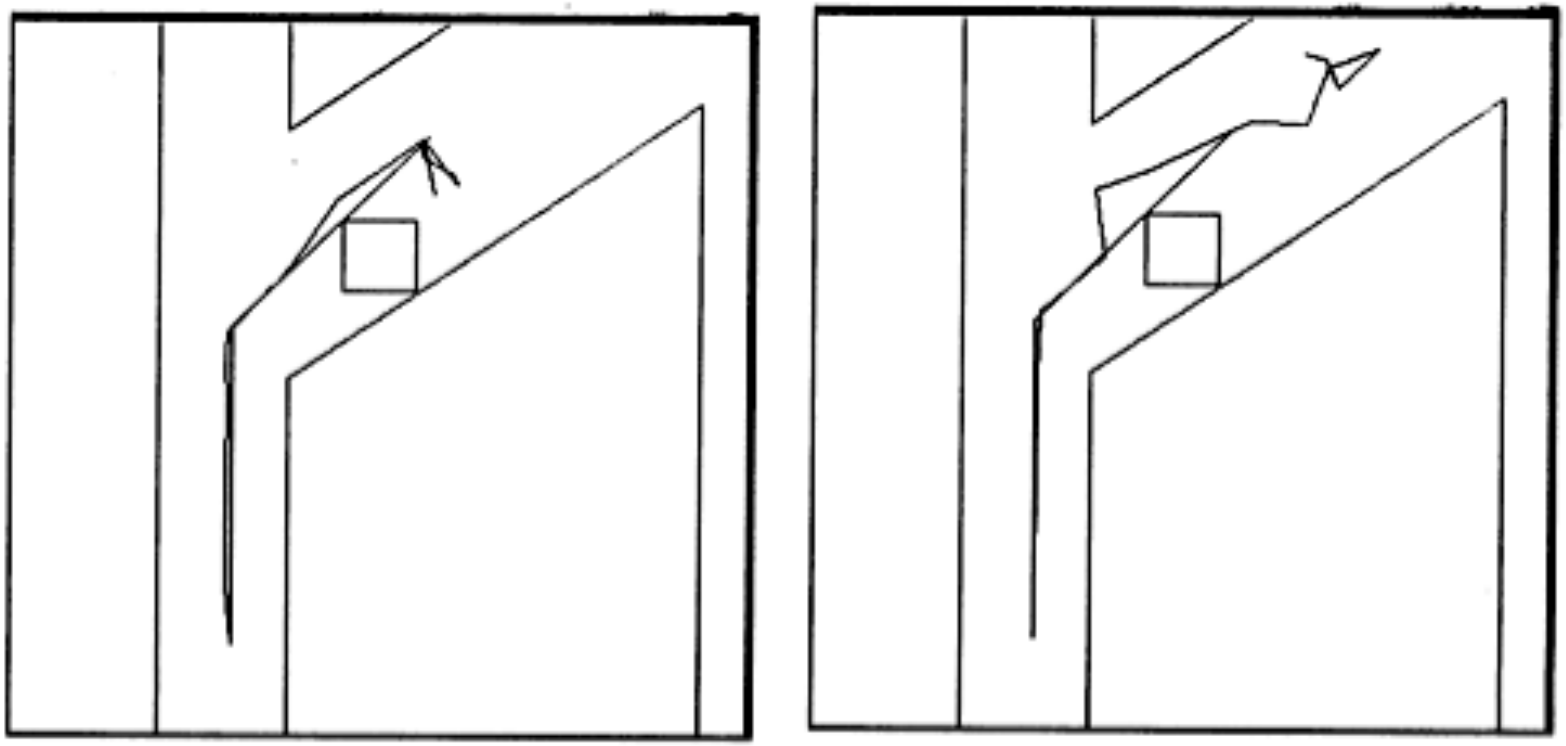
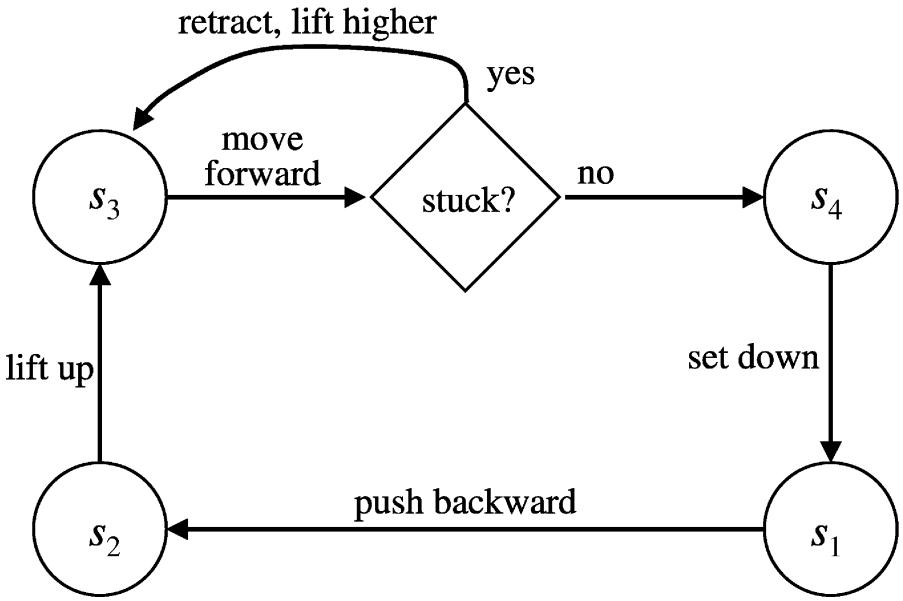
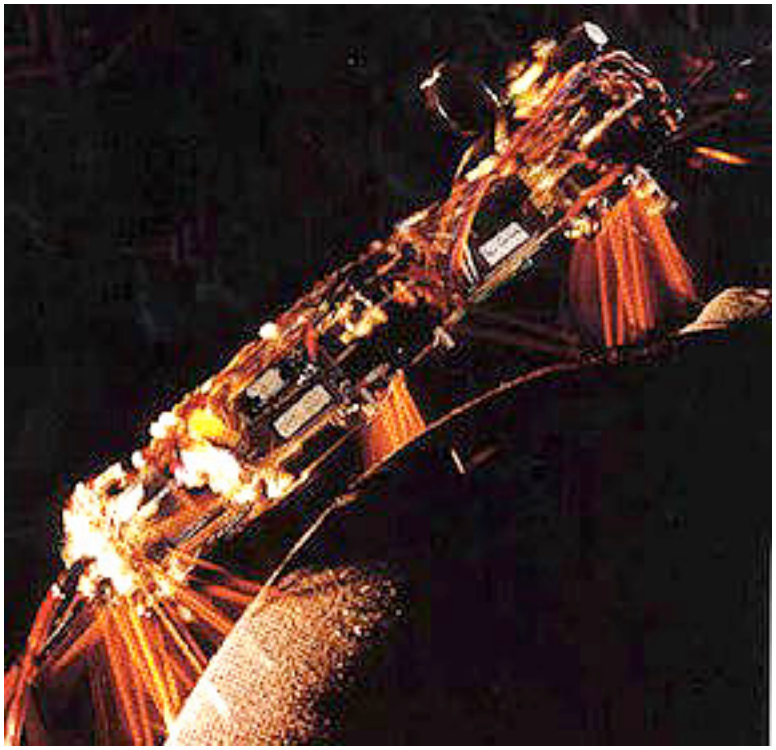


Figure 10. With level 2 control the robot tries to achieve commanded goals. The nominal goals are the two straight lines. After reaching the second goal, since there are no new goals forthcoming, the robot reverts to aimless level 1 behavior.

Genghis


retract, lift higher
yes
move
forward
stuck?
no
s_3
s_4
lift up
set down
push backward
s_2
s_1



Teaching powerful ideas with autonomous mobile robots

Rolf Pfeifer

AI Lab, Department of Computer Science

University of Zurich, Switzerland

pfeifer@ifi.unizh.ch

to appear in: Journal of Computer Science Eduation, Vol. 7, No. 2

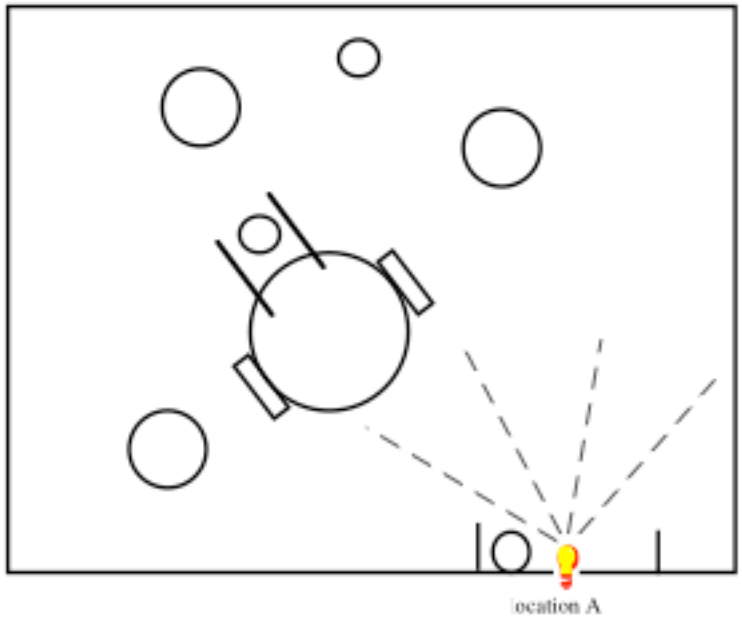


Figure 5: Simplified environment for the garbage-collecting robot. There are two different types of objects, small ones which it should pick up and bring to location A, the large ones are too heavy: they should be left alone. The robot has IR sensors distributed around its front half. It uses them for obstacle avoidance, for moving along objects, and for detecting ambient light (in order to bring the objects to location A). The robot also has sensors for measuring how fast its wheels turn, the so-called wheel-encoders.

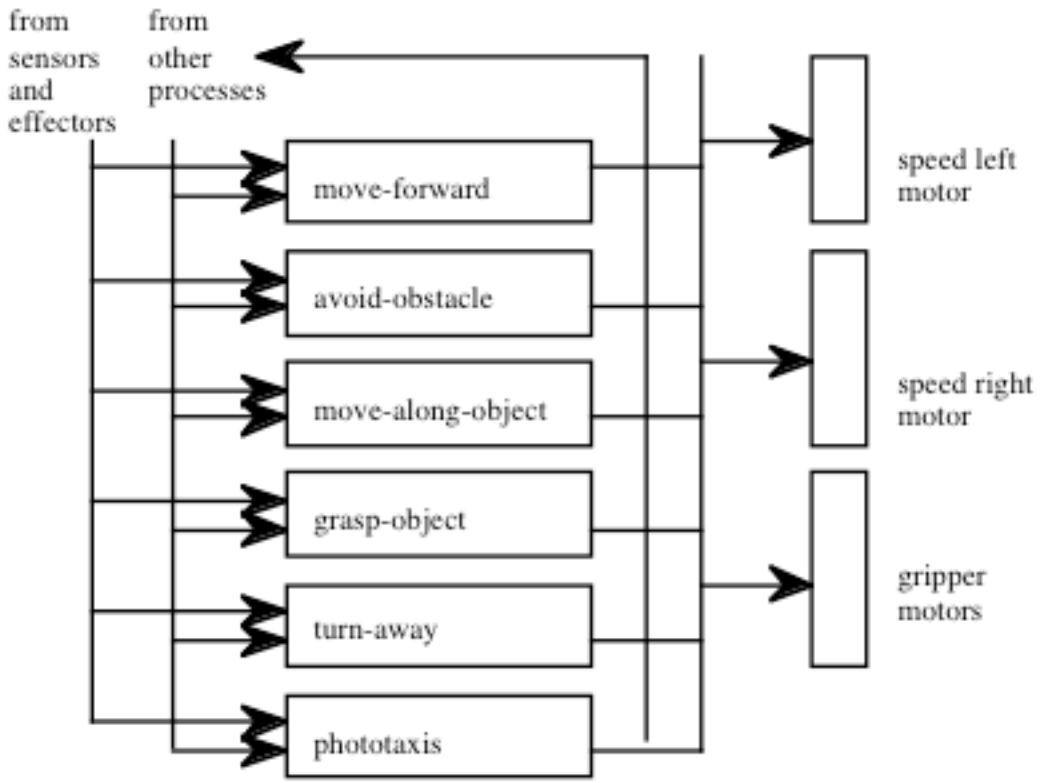
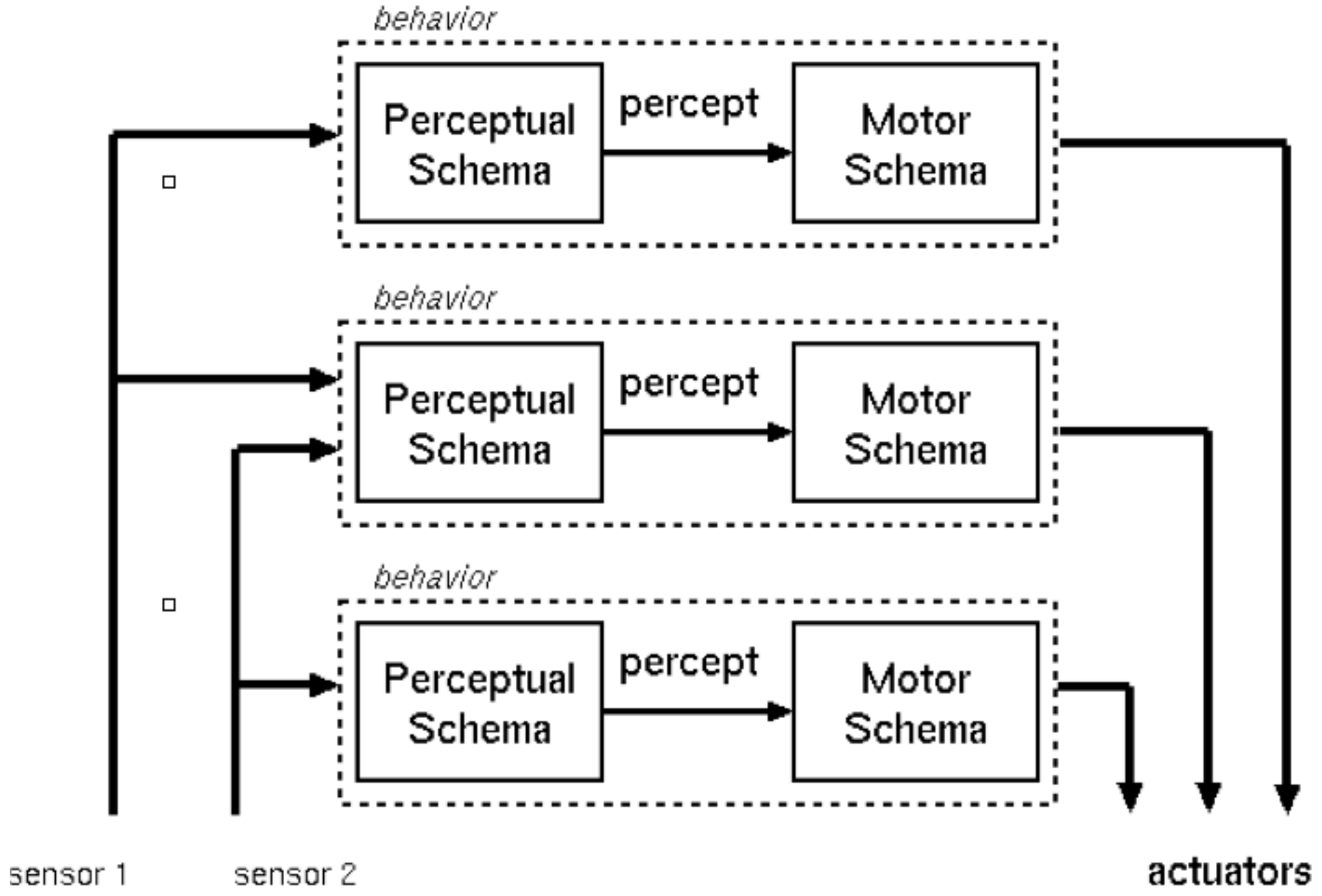
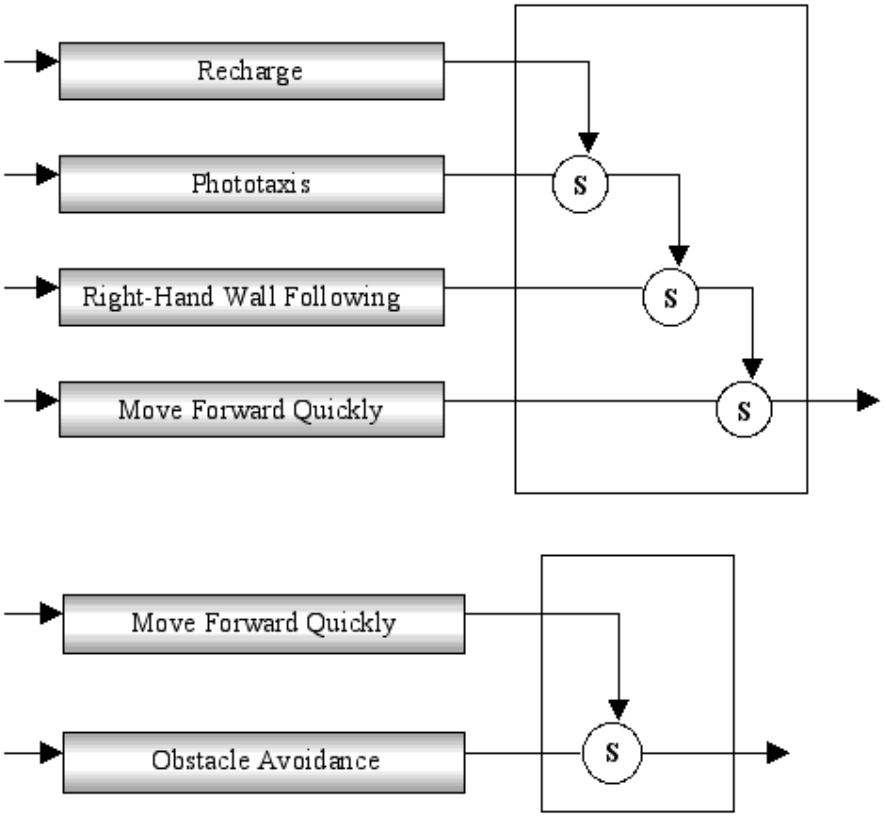


Figure 6: Architecture of the robot. There are many parallel processes that connect sensors to effectors (wheel motors, gripper motors) without undergoing high-level central processing in between.


behavior
Perceptual Schema
percept
Motor Schema
behavior
Perceptual Schema
percept
Motor Schema
behavior
Perceptual Schema
percept
Motor Schema
sensor 1
sensor 2
actuators


Recharge
Phototaxis
Right-Hand Wall Following
Move Forward Quickly
S
S
S
Move Forward Quickly
Obstacle Avoidance
S

Lesson 7

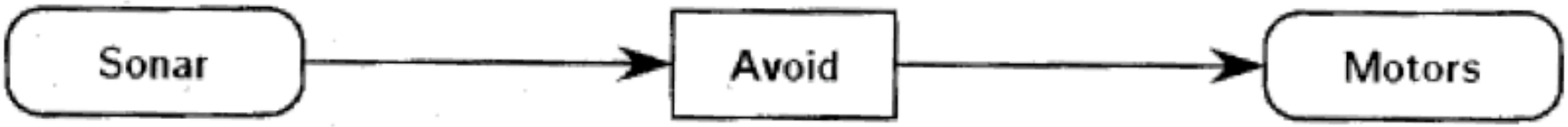


Figure 9.3. A block diagram of a simple behavior control program. Rounded boxes represent both the physical sensors or actuators and the software drivers that directly control them. The square-cornered box contains code that performs computations that transform sensor readings into actuator commands. Arrows can be thought of as wires that allow the components to communicate.

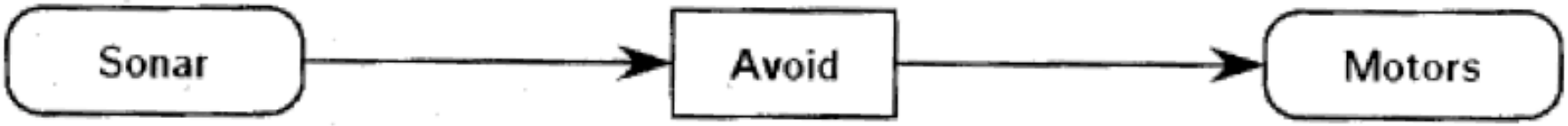


Figure 9.3. A block diagram of a simple behavior control program. Rounded boxes represent both the physical sensors or actuators and the software drivers that directly control them. The square-cornered box contains code that performs computations that transform sensor readings into actuator commands. Arrows can be thought of as wires that allow the components to communicate.

```
PrivateCar car = new PrivateCar();
int power = 70, ms = 500;
UltrasonicSensor sonar = new UltrasonicSensor(SensorPort.S1);
int frontDistance, leftDistance, rightDistance;
int stopThreshold = 30;
```

```
while ( true )
{
    // Go forward
    car.forward(power, power);

    // Monitor the distance in front of the car and stop
    // when an object gets to close
    frontDistance = sonar.getDistance();
    while ( frontDistance > stopThreshold )
    {
        frontDistance = sonar.getDistance();
    }
    car.stop();

    // Get the distance to the left
    car.forward(0, power);
    Delay.msDelay(ms);
    leftDistance = sonar.getDistance();

    // Get the distance to the right
    car.backward(0, power);
    Delay.msDelay(ms);
    car.forward(power, 0);
    Delay.msDelay(ms);
    rightDistance = sonar.getDistance();

    // Turn in the direction with most space in front of the car
    if ( leftDistance > rightDistance ){
        car.backward(power, 0);
        Delay.msDelay(ms);
        car.forward(0, power);
        Delay.msDelay(ms);
    }
}
```

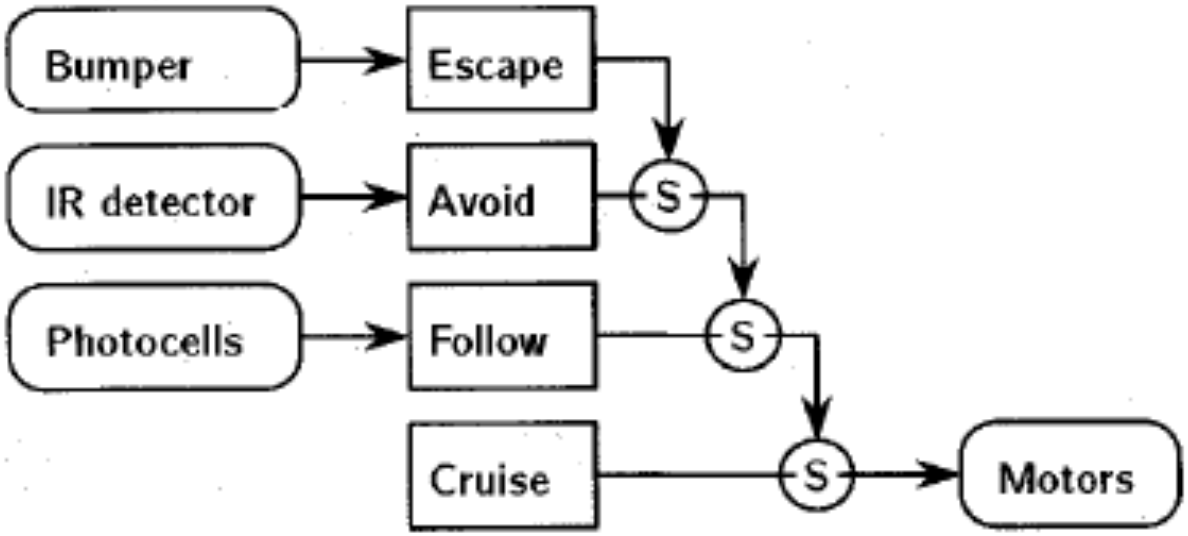
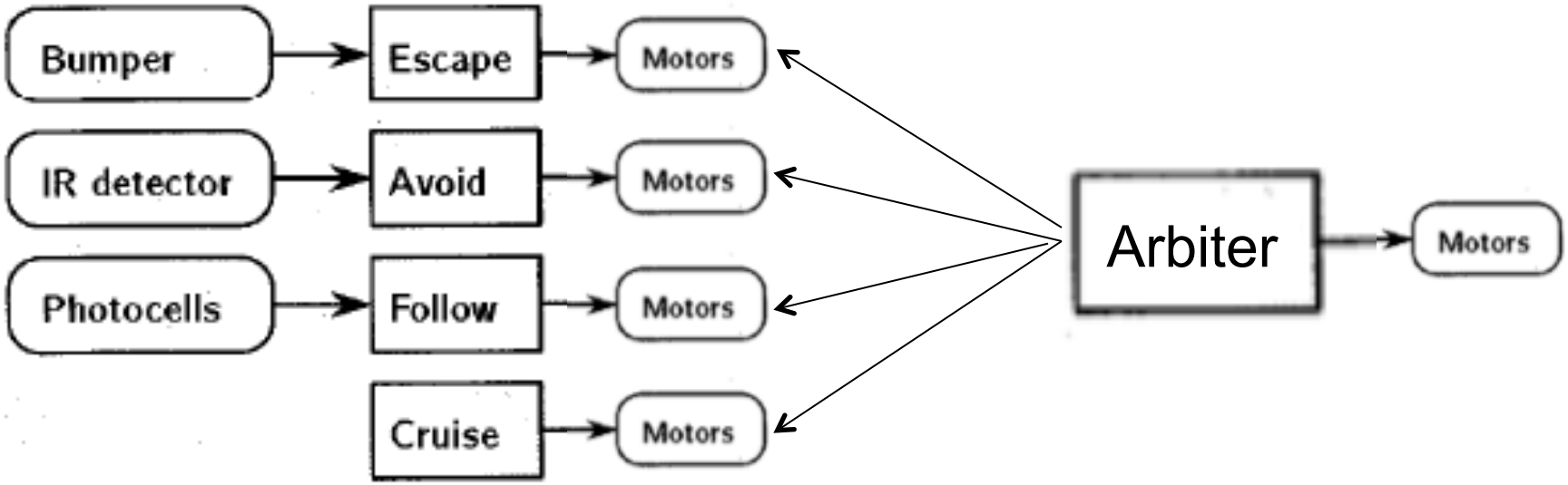
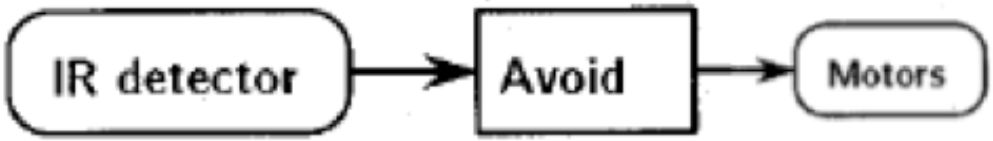


Figure 9.9. We implement this portion of Rug Warrior's behavior control network using **IC** and its process constructs.

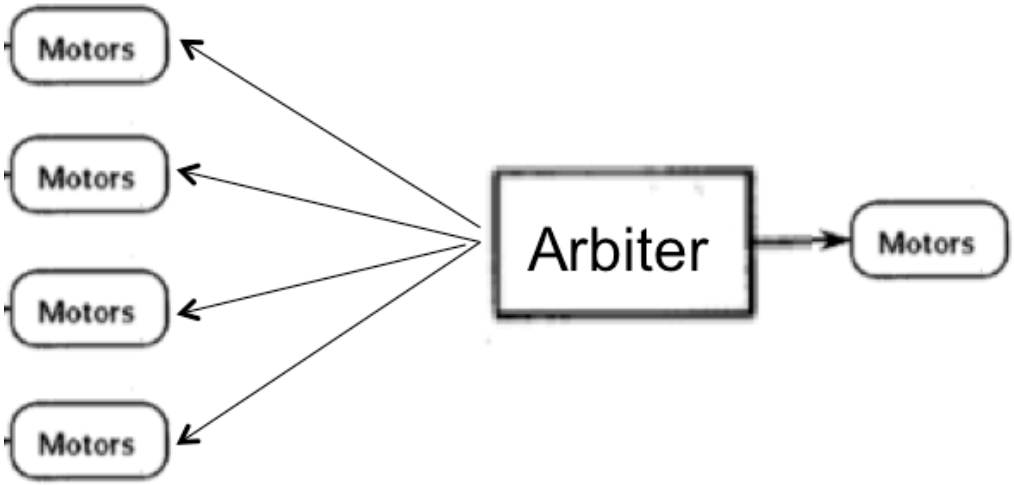

Bumper
Escape
Motors
IR detector
Avoid
Motors
Photocells
Follow
Motors
Cruise
Motors
Arbiter
Motors

```
public class RobotFigure9_9 {


    public static void main(String [] args)  throws Exception {

        SharedCar [] car = { new SharedCar(), new SharedCar(),
                             new SharedCar(), new SharedCar()};

        CarDriver cd = new CarDriver();

        //Escape escape = new Escape(car[0]);
        Avoid avoid   = new Avoid(car[1]);
        Follow follow = new Follow(car[2]);
        Cruise cruise = new Cruise(car[3]);

        Arbiter arbiter = new Arbiter(car, cd);

        LCD.drawString("Robot 9.9", 0, 0);
        Button.waitForAnyPress();

        arbiter.setDaemon(true);
        arbiter.start();
        avoid.setDaemon(true);
        avoid.start();
        follow.setDaemon(true);
        follow.start();
        cruise.setDaemon(true);
        cruise.start();

        while ( ! Button.ESCAPE.isDown())
        {
            LCD.drawString("Winner " + arbiter.winner(), 0, 3);
        }

    }
}
```

```
while (true)
{
    // Monitor the distance in front of the car and stop
    // when an object gets to close
    frontDistance = sonar.getDistance();
    while ( frontDistance >= stopThreshold )
    {
        car.noCommand();
        frontDistance = sonar.getDistance();
    }
    car.stop();
    Delay.msDelay(ms);

    // Get the distance to the left
    car.forward(0, power);
    Delay.msDelay(ms);
    leftDistance = sonar.getDistance();

    // Get the distance to the right
    car.backward(0, power);
    Delay.msDelay(ms);
    car.forward(power, 0);
    Delay.msDelay(ms);
    rightDistance = sonar.getDistance();

    // Turn in the direction with most space in front of the car
    if ( leftDistance > rightDistance ){
        car.backward(power, 0);
        Delay.msDelay(ms);
        car.forward(0, power);
        Delay.msDelay(ms);
    }
}
```



```
public class Arbiter extends Thread
{
    private SharedCar [] car;
    private CarDriver cd;
    private int winner;

    public Arbiter(SharedCar [] car, CarDriver cd)
    {
        this.car = car;
        this.cd = cd;
    }

    public void run()
    {
        while ( true )
        {
            for (int i=0; i < car.length; i++)
            {
                CarCommand carCommand = car[i].getCommand();
                if ( carCommand != null)
                {
                    cd.perform(carCommand);
                    winner = i;
                    break;
                }
            }

        }

    }
```



```
while (true)
{
    // Monitor the light in front of the car and start to follow
    // the light if light level is above the threshold
    frontLight = light.getLightValue();
    while ( frontLight <= lightThreshold )
    {
        car.noCommand();
        frontLight = light.getLightValue();
    }

    // Follow light as long as the light level is above the threshold
    while ( frontLight > lightThreshold )
    {
        // Get the light to the left
        car.forward(0, power);
        Delay.msDelay(ms);
        leftLight = light.getLightValue();

        // Get the light to the right
        car.backward(0, power);
        Delay.msDelay(ms);
        car.forward(power, 0);
        Delay.msDelay(ms);
        rightLight = light.getLightValue();

        // Turn back to start position
        car.backward(power, 0);
        Delay.msDelay(ms);

        // Follow light for a while
        delta = leftLight-rightLight;
        car.forward(power-delta, power+delta);
        Delay.msDelay(ms);

        frontLight = light.getLightValue();
    }

    car.stop();
    Delay.msDelay(ms);
```

Alishan train track