2 Stimulus–Response Agents

2.1 Perception and Action

In this chapter, I consider machines that have no internal state and that simply react to immediate stimuli in their environments. We'll call these machines *stimulus-response (S-R)* agents. A variety of robots can be built that exhibit surprisingly interesting behavior based on motor responses to rather simple functions of immediate sensory inputs. One of the earliest examples of this kind of robot was Grey Walter's *Machina speculatrix*—a wheeled device with motors, photocells, and two vacuum tubes [Walter 1953] that moved toward light of moderate intensity and avoided bright light. Similar machines are described by Braitenberg [Braitenberg 1984].

I begin my discussion with an illustrative example. Consider the robot in the two-dimensional grid-space world shown in Figure 2.1. This robot's world is completely enclosed by boundaries and may contain other large, unmovable objects, as shown. The world has no "tight spaces" (spaces between objects and boundaries that are only one cell wide), and the design of our robot will take advantage of that fact.

We want this robot to execute the following behavior: go to a cell adjacent to a boundary or object and then follow that boundary along its perimeter forever. To be capable of such boundary-following behavior, the robot must be able to sense whether or not certain cells are free for it to occupy, and it must be able to perform certain primitive actions.

Note: For elaboration of the no-tight-space condition, see *www.mkp.com/nils/clarified.*

The robot is able to sense whether or not the eight cells surrounding itare free. These sensory inputs are denoted by the binary-valued variables s_1, s_2, s_3, s_4, s_5, s_6, s_7, and s_8. They have value 0 whenever the corresponding cell (relative to the robot, as shown in Figure 2.1) can be occupied by the robot; otherwise, they have

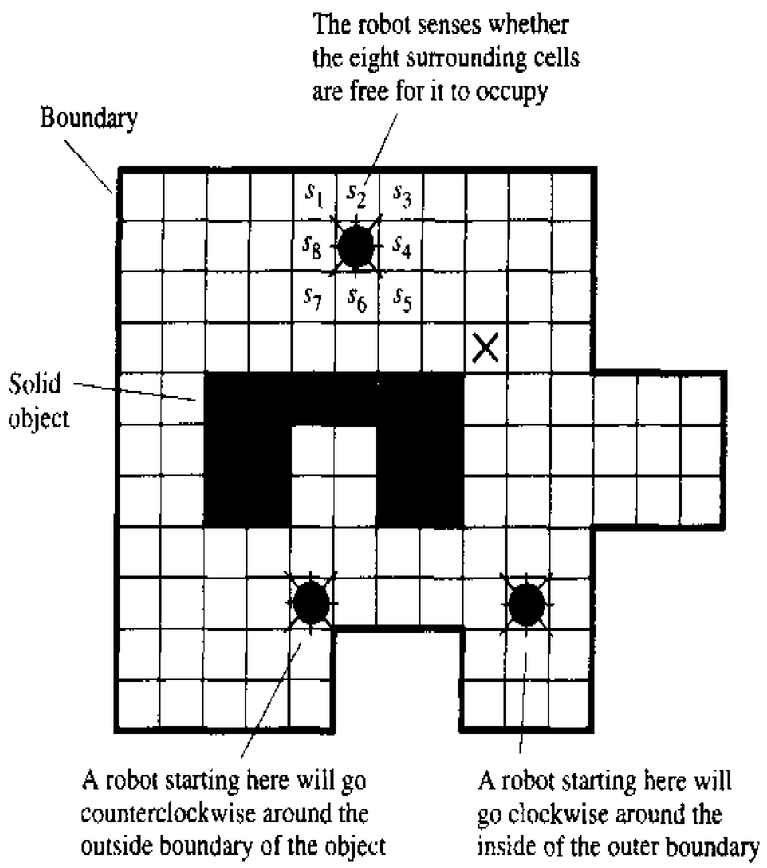

The robot senses whether the eight surrounding cells are free for it to occupy
Boundary
s_1
s_2
s_3
s_8
s_4
s_7
s_6
s_5
X
Solid object
A robot starting here will go counterclockwise around the outside boundary of the object
A robot starting here will go clockwise around the inside of the outer boundary

Figure 2.1

A Robot in a Two-Dimensional Grid World

value 1. If the robot were in the position marked by an X, the values of the sensory inputs (starting with s_1 and proceeding clockwise) would be (0,0,0,0,0,0,1,0).

The robot can move to a (free) adjacent cell in its column or row. There are four such actions:

1. north moves the robot one cell up in the cellular grid
2. east moves the robot one cell to the right
3. south moves the robot one cell down
4. west moves the robot one cell to the left

All of the actions have their indicated effects unless the robot attempts to move into a cell that is not free; in that case, the action has no effect.

Given the properties of the kinds of worlds that a robot might inhabit (Figure 2.1, for example), the task that the robot is to perform (following a boundary), and the robot's sensory and motor abilities, the designer's job is to specify a function of the sensory inputs ($s_1, \ldots, s_8$, in our example) that selects actions appropriate for the task. It is common to divide the processes

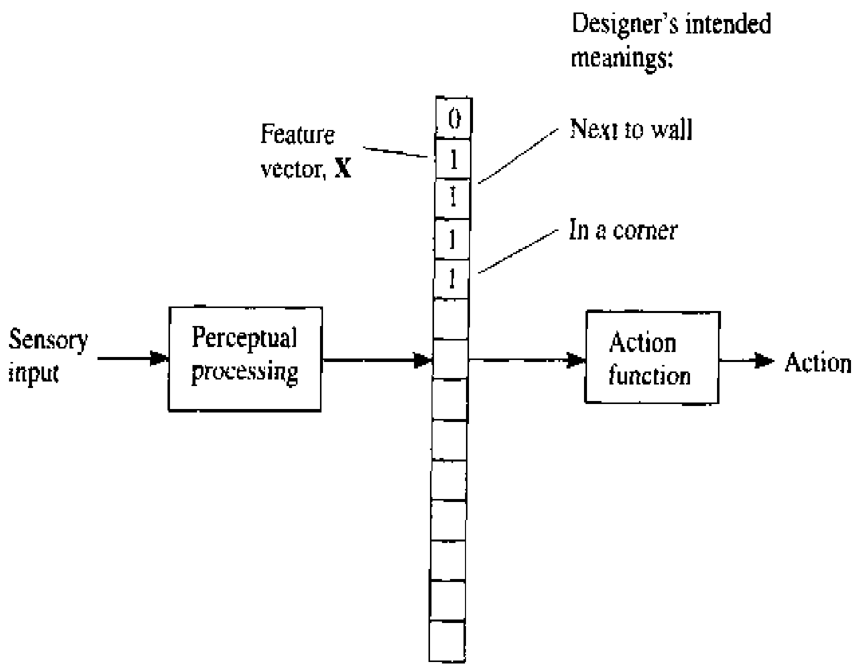
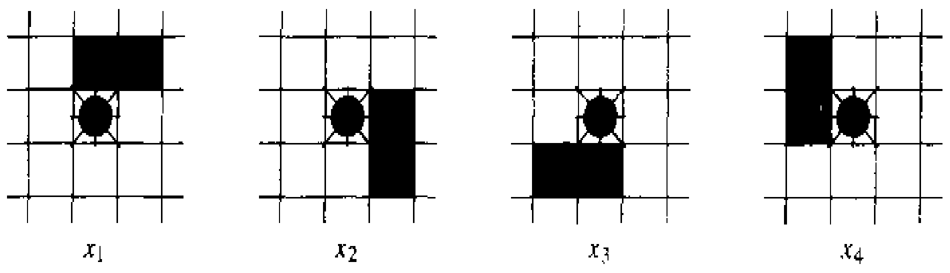


Figure 2.2

Perception and Action Components

of computing an action from the sensory signals into two separate phases, as illustrated in Figure 2.2. A perceptual processing phase produces a vector, **X**, of *features*, $(x_1, \ldots, x_i, \ldots, x_n)$, and an action computation phase selects an action based on the feature vector. The values of the features can be either real numbers (*numeric features*) or categories (*categorical features*). (A categorical feature is one whose value is a name or a property. For example, the value of the feature "color" might be "red," "blue," or "green.") The special case of binary-valued features can be regarded as either numeric (0,1) or categorical (*T*, *True*, and *F*, *False*). The features are selected by the designer to correlate with those properties of the robot's environment that are relevant to which action should be performed in the state described by the features.

Of course, the split between perception and action is completely arbitrary. I could have lumped the whole process as either perception (the world is perceived to be in a state in which action north is appropriate) or as action (action north is computed to be appropriate based on the raw sensory data). Usually, the split is made in such a way that the same features would be used repeatedly in a variety of tasks to be performed. Different tasks would have the same feature vector but different action functions. Viewed as computer programs, the computation of features from sensory signals can be regarded as often used library routines—needed by many different action computations. Deciding how



In each diagram, the indicated feature has value 1 if and only if at least one of the shaded cells is *not* free.

Figure 2.3

Features for Boundary Following

to split into the two processes is part of the art of the design of these machines, and I will not have very much to say about that art here.

After deciding how to split, we are left with two problems: (1) converting raw sensory data into a feature vector, and (2) specifying an action function. I will discuss each of these problems briefly in the context of our example

2.1.1 Perception

The sensory input for our boundary-following robot consists of the values of $s_1, \ldots, s_8$. There are $2^8 = 256$ different combinations of these values. In our environment, some of the combinations are ruled out because of my restriction against tight spaces. For the task at hand, it happens that there are four binary-valued features of the sensory values that are useful for computing an appropriate action. I denote these features by x_1, x_2, x_3, and x_4. Their definitions are given in Figure 2.3. For example, $x_1 = 1$ if and only if $s_2 = 1$ or $s_3 = 1$.

For this example, perceptual processing consists of relatively simple computations. For more complex worlds, and for robots with more complex sensors and tasks, designing appropriate perceptual processing can be challenging. Also, in many real tasks, perceptual processing might occasionally give erroneous, ambiguous, or incomplete information about the robot's environment. Such errors might evoke inappropriate actions, although depending on the task and on the environment, poorly selected actions may not cause too much harm if they are infrequent. I will return to the subject of perception in Chapter 6.

2.1.2 Action

Given the four features, we must now specify a function of them that selects the appropriate boundary-following action. We note first that if *none* of the features has value 1 (that is, if the robot senses that all of its surrounding cells are free), the robot can move in *any* direction until it encounters a boundary. Let's have it move north. Whenever at least one of the other features has value 1, boundary-following behavior is achieved by the following rules for action:

if $x_1 = 1$ and $x_2 = 0$, move east

if $x_2 = 1$ and $x_3 = 0$, move south

if $x_3 = 1$ and $x_4 = 0$, move west

if $x_4 = 1$ and $x_1 = 0$, move north

The conditions under which the robot should take these various actions happen, in this case, to be Boolean combinations of the features. The features themselves are also Boolean combinations of the sensory inputs. Since several important perceptual and action selection methods involve Boolean functions, it will be helpful to digress briefly here to discuss them before continuing with our example.

2.1.3 Boolean Algebra

A Boolean function, $f(x_1, x_2, \ldots, x_n)$ maps an n tuple of (0,1) values to $\{0, 1\}$. ***Boolean algebra*** is a convenient notation for representing Boolean functions. Boolean algebra uses the connectives $\cdot$, $+$, and $\bar{\ }$. For example, the *and* function of two variables is written $x_1 \cdot x_2$. By convention, the connective $\cdot$ is usually suppressed, and the *and* function is written x_1x_2. The function x_1x_2 has value 1 if and only if *both* x_1 and x_2 have value 1; otherwise, it has value 0. The (inclusive) *or* function of two variables is written $x_1 + x_2$. $x_1 + x_2$ has value 1 if and only if either or both of x_1 or x_2 has value 1; otherwise, it has value 0. The *complement* or *negation* of a variable, x, is written $\bar{x}$. $\bar{x}$ has value 1 if and only if x has value 0; otherwise, it has value 0.

These definitions are compactly given by the following rules for Boolean algebra:

$1 + 1 = 1,\ 1 + 0 = 1,\ 0 + 0 = 0$

$1 \cdot 1 = 1,\ 1 \cdot 0 = 0,\ 0 \cdot 0 = 0$

$\bar{1} = 0,\ \bar{0} = 1$

As an example, the condition under which our boundary-following robot should move north is given by the expression $\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 + x_4\bar{x}_1$. The functions that compute features from sensory signals also happen to be Boolean in this case. For example, $x_4 = s_1 + s_8$. The other features and action rules are given by similar functions.

Sometimes the arguments and values of Boolean functions are expressed in terms of the constants T (*True*) and F (*False*) instead of 1 and 0, respectively.

The connectives $\cdot$ and $+$ are commutative. Thus, $x_1x_2 = x_2x_1$ and $x_1 + x_2 = x_2 + x_1$. They are also associative; thus $x_1(x_2x_3) = (x_1x_2)x_3$ and $x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$. Therefore, we can drop the parentheses and write expressions like $x_1x_2x_3$ and $x_1 + x_2 + x_3$ without ambiguity.

A Boolean formula consisting of a single variable, such as x_1, is called an *atom*. One consisting of either a single variable or its complement, such as $\overline{x_1}$, is called a *literal*.

We cannot interchange the order of the connectives $\cdot$ and $+$ in complex expressions. Instead, we have DeMorgan's laws (which can be verified by using the preceding definitions):

$$\overline{f_1 f_2} = \overline{f_1} + \overline{f_2}$$

$$\overline{f_1 + f_2} = \overline{f_1}\,\overline{f_2}$$

DeMorgan's laws can often be used to simplify Boolean functions. For example, $x_1\overline{x_2} = (s_2 + s_3)\overline{(s_4 + s_5)} = (s_2 + s_3)\overline{s_4}\,\overline{s_5}$.

Another important law is the *distributive law*:

$$f_1(f_2 + f_3) = f_1 f_2 + f_1 f_3$$

2.1.4 Classes and Forms of Boolean Functions

Boolean functions come in a variety of forms. An important form is $\lambda_1\lambda_2\cdots\lambda_k$, where the λ_i are literals. A function written in this way is called a *conjunction* of literals or a *monomial*. The conjunction itself is called a *term*. Some example terms are x_1x_7 and $x_1x_2\overline{x_4}$. The *size* of a term is the number of literals it contains. The examples are of sizes 2 and 3, respectively.

It is easy to show that there are exactly 3^n possible monomials of n variables (see Exercise 2.4). The number of monomials of size k or less is bounded from above by

$$\sum_{i=0}^{k} C(2n, i) = O(n^k),$$

where

$$C(i, j) = \frac{i!}{(i-j)!j!}$$

is the binomial coefficient.

A *clause* is any expression of the form $\lambda_1 + \lambda_2 + \cdots + \lambda_k$, where the λ_i are literals. Such a form is called a *disjunction* of literals. Some example clauses are $x_3 + x_5 + x_6$ and $x_1 + \overline{x_4}$. The *size* of a clause is the number of literals it contains. There are 3^n possible clauses and fewer than $\sum_{i=0}^{k} C(2n, i)$ clauses of size k or less. If f is a term, then (by DeMorgan's laws) $\overline{f}$ is a clause, and vice versa. Thus, terms and clauses are duals of each other.

A Boolean function is said to be in *disjunctive normal form (DNF)* if it can be written as a *disjunction* of terms. Some examples in DNF are $f = x_1x_2 + x_2x_3x_4$ and $f = x_1\overline{x_3} + \overline{x_2}\,\overline{x_3} + x_1x_2\overline{x_3}$. Any Boolean function can be written in DNF. A DNF expression is called a k-term DNF expression if it is a disjunction of k terms; it is in the class k-DNF if the size of its largest term is k. The preceding examples are 2-term and 3-term expressions, respectively. Both expressions are in the class 3-DNF.

Disjunctive normal form has a dual: *conjunctive normal form (CNF)*. A Boolean function is said to be in CNF if it can be written as a *conjunction* of clauses. An example in CNF is $f = (x_1 + x_2)(x_2 + x_3 + x_4)$. All Boolean functions also have a CNF form. A CNF expression is called a k-clause CNF expression if it is a conjunction of k clauses; it is in the class k-CNF if the size of its largest clause is k. The example is a 2-clause expression in 3-CNF. If f is written in DNF, an application of DeMorgan's law renders $\bar{f}$ in CNF, and vice versa.

2.2 Representing and Implementing Action Functions

If there are R possible actions, then we must find an appropriate R-valued function of the feature vector to compute an action. Various ways of representing and implementing action functions have been investigated, and I describe some of them next.

2.2.1 Production Systems

One convenient representational form for an action function is a *production system*. A production system comprises an ordered list of rules called *production rules* or, simply, *productions*. Each rule is written in the form $c_i \longrightarrow a_i$, where c_i is the *condition part*, and a_i is the *action part*. A production system consists of a list of such rules:

$$\begin{aligned}
&c_1 \longrightarrow a_1\\
&c_2 \longrightarrow a_2\\
&\quad\vdots\\
&c_i \longrightarrow a_i\\
&\quad\vdots\\
&c_m \longrightarrow a_m
\end{aligned}$$

In general, the condition part of a rule can be any binary-valued (0,1) function of the features resulting from perceptual processing of the sensory inputs. Often, it is a monomial—a conjunction of Boolean literals. To select an action, the rules are processed as follows: starting with the first rule, namely, $c_1 \longrightarrow a_1$, we look for the first rule in the ordering whose condition part evaluates to 1 and select the action part of that rule. The action part can be either a primitive action, a call to another production system, or a set of actions to be executed simultaneously. Usually, the last rule in the ordering has 1 as its condition part; if no rule above it has a condition part equal to 1, then the action associated with the last rule is executed by default. As actions are executed, sensory inputs and the values of features based on them change. We assume that the conditions are continuously being checked so that the action being executed at any time corresponds to the first rule whose condition (at that precise time) has value 1.

Using Boolean algebra and the feature literals defined earlier for the boundary-following robot, here is a production system representation of the boundary-following routine:

$x_4\overline{x_1} \longrightarrow$ north

$x_3\overline{x_4} \longrightarrow$ west

$x_2\overline{x_3} \longrightarrow$ south

$x_1\overline{x_2} \longrightarrow$ east

$1 \longrightarrow$ north

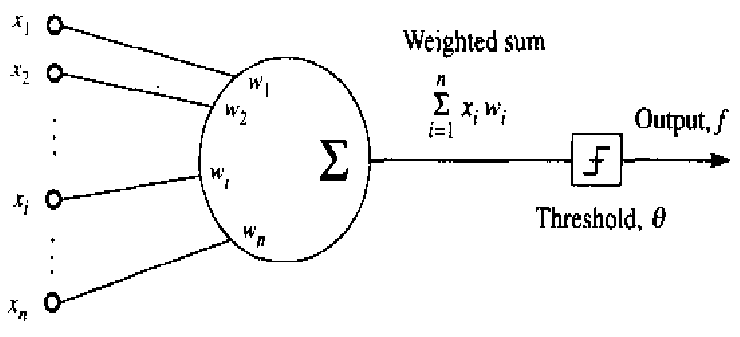
Boundary-following behavior is an example of a *durative* procedure—one that never ends. The robot continues to execute actions forever. In contrast, some tasks require acting only until some specific *goal* condition is achieved and then ceasing activity. The goal is usually expressed as a Boolean condition on the features. As an example, instead of having our robot follow a boundary forever, we might want it to go to a (concave) corner and stay there. Given a corner-detecting feature, say, c, whose value is 1 if and only if the robot is in a corner, the following production system will get the robot to a corner (if there is one to be found):

$c \longrightarrow$ nil

$1 \longrightarrow$ b-f

Here, nil is the null or do-nothing action, and b-f is the boundary-following procedure, which we have just defined.

In goal-achieving production systems, c_1, the condition part of the rule at the top of the list, specifies the overall goal that we want the action program to achieve. Whenever it is satisfied, the agent performs no action. Condition c_2 and action a_2 are usually chosen so that if c_1 is not satisfied, and c_2 is, then the execution of action a_2 will eventually achieve c_1. And so on down the list. This style of production system forms the basis of a formalism called *teleo-reactive (T-R) programs* [Nilsson 1994]. In a T-R program, each properly executed action in the ordering works toward achieving a condition higher in the list. Production systems with this property are usually easy to write, given an overall goal for an agent (stated as a condition on the features). T-R programs are also quite robust; actions proceed inexorably toward the goal. Occasional setbacks, caused perhaps by faulty perception, improperly executed actions, or exogenous processes in the environment, are recouped so long as perception is reasonably accurate and



$$f = 1 \text{ if } \sum_{i=1}^{n} x_i w_i \geq \theta$$
$$= 0 \text{ otherwise}$$

Figure 2.4

A Threshold Logic Unit

the actions usually achieve their designed effects.[1] Besides these features, T-R programs can have parameters that are bound when the programs are called, and they can call other T-R programs and themselves recursively.

2.2.2 Networks

Boolean functions and production systems can easily be implemented by computer programs. Alternatively, they can be implemented directly as electronic circuits.[2] The inputs to the circuitry can be the sensory signals themselves. In logic circuits, Boolean functions are usually implemented by networks of logical gates (AND, NAND, OR, etc.). A popular type of circuit consists of networks of threshold elements or other elements that compute a nonlinear function of a weighted sum of their inputs. An example of such an element is the *threshold logic unit (TLU)* shown in Figure 2.4. It computes a weighted sum of its inputs, compares this sum to a threshold value, and outputs a 1 if the threshold is exceeded. Otherwise, it outputs a 0.

The Boolean functions implementable by a TLU are called the *linearly separable functions.* (A TLU separates the space of input vectors yielding an above-threshold response from those yielding a below-threshold response by a linear

1. T-R programs are ones that are both "pulled" toward their goals and react to their situations. The prefix *teleo* is derived from the Greek word for "end" or "purpose."

2. Even so, the behavior of the circuit is often simulated by some type of circuit-simulation program running on a computer. Nevertheless, thinking of the behavior as being generated by a circuit instead of by a program helps us to understand the dynamic dependence of appropriate action on immediate sensory inputs

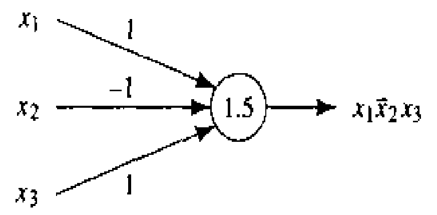


Figure 2.5

Implementation of a Monomial with a TLU

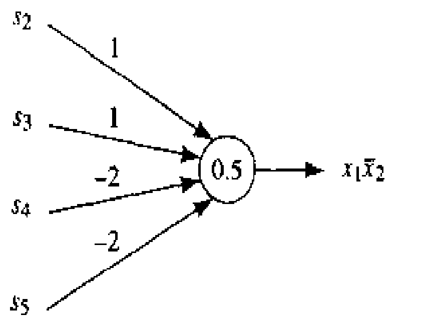


Figure 2.6

Implementing a Function for Boundary Following

surface—called a *hyperplane* in n dimensions.) Many, but far from all, Boolean functions are linearly separable. For example, any monomial (a conjunction of literals) or any clause (a disjunction of literals) is linearly separable. I show a TLU implementation of a monomial in Figure 2.5. The TLU weights are shown drawn next to their corresponding input lines, and the threshold is drawn inside the circle representing the TLU. The exclusive-or function of two variables ($f = x_1\bar{x}_2 + \bar{x}_1x_2$), however, is an example of a function that is not linearly separable.

The functions used by the boundary-following robot can be implemented by TLUs. As an example, an implementation of $x_1\bar{x}_2 = (s_2 + s_3)\overline{(s_4 + s_5)} = (s_2 + s_3)\bar{s}_4\bar{s}_5$ by a single TLU is shown in Figure 2.6.

In applications in which there are only two possible actions, it may be that a single TLU can compute the proper action, given a coded representation of the feature vector as input. For more complex problems, a network of such elements is needed. Such circuits are called *neural networks* because the TLUs are thought to be simple models of biological neurons, which fire or not depending on the summed strength of their inputs across synapses of various strengths. We study neural networks in more detail in the next chapter.

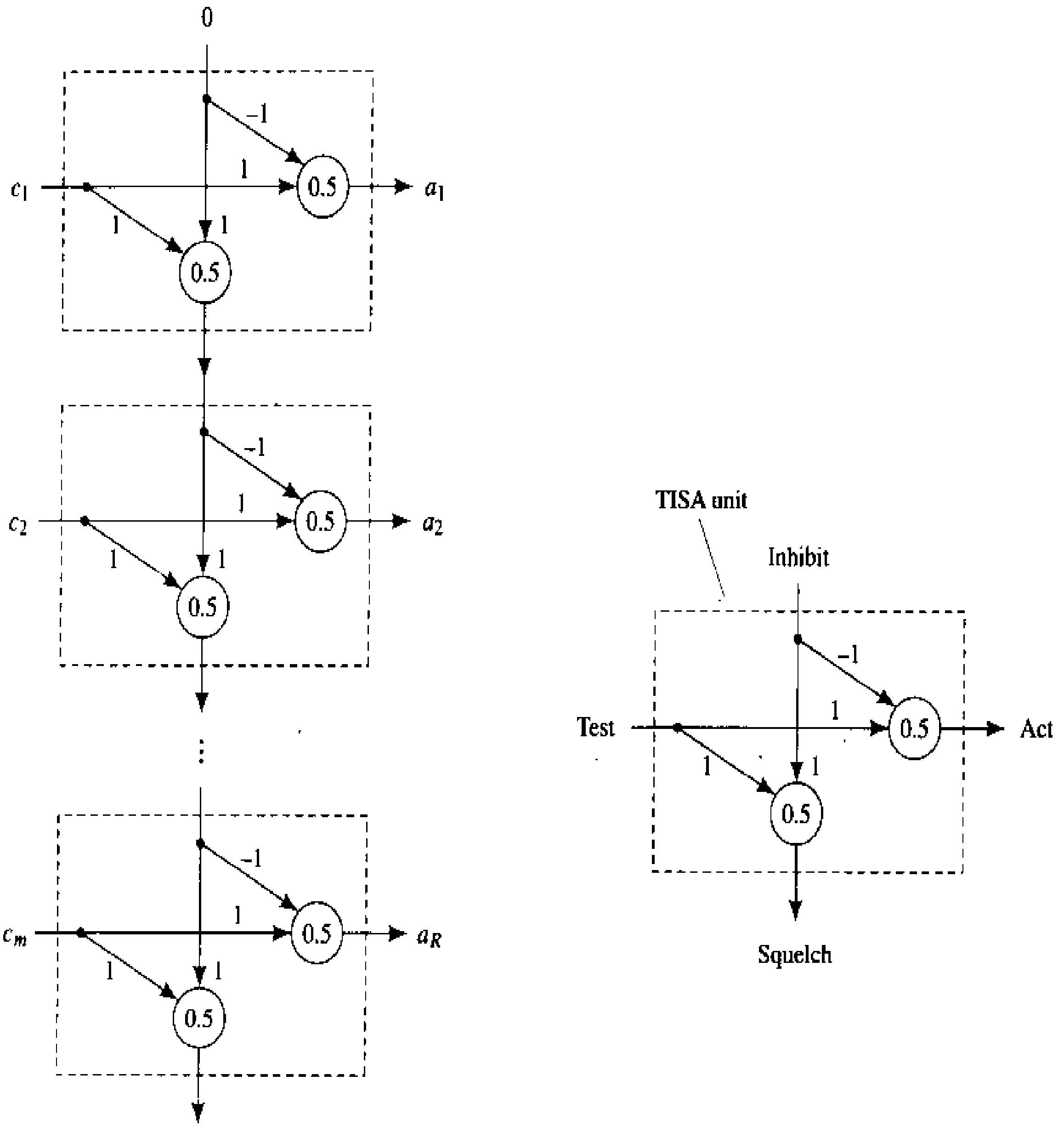


Figure 2.7

A Network of TISA Units

Katz[3] has suggested that a simple network structure with repeated combinations of inverters and AND gates (which can be implemented by TLUs) can be used to implement any T-R program. I show my version of such a network in Figure 2.7. The inputs to the network are the binary (0,1) values of the conditions,

3. Edward Katz, private communication, April 1996.

c_i; the outputs of the network energize the corresponding actions, a_i. (Not shown here are the computations that produce the c_i; since the c_i are often conjunctions of literals, they too might be computed by TLUs.) Each rule in the T-R program is implemented by a subcircuit (called a TISA, Test, Inhibit, Squelch, Act) with two inputs and two outputs. One TLU in the TISA computes the conjunction of one of its inputs with the complement of the other input; the other TLU computes the disjunction of its two inputs. The inhibit input is 0 when *none* of the rules above has a true condition; the test input is 1 only if the condition, c_i, corresponding to this rule is satisfied. If the test input is 1 and the inhibit input is 0, the act output is 1 (energizing the corresponding action, a_i). If either the test input or the inhibit input is 1, the squelch output is 1 (inhibiting all units below). Using programmable gate arrays or some equivalent form of dynamic circuit building, it can be arranged that calling a T-R program dynamically builds the run-time circuit.

2.2.3 The Subsumption Architecture

There have been several other formalisms that have been proposed for converting immediate sensory inputs into actions. Among these is the *subsumption architecture* of Rodney Brooks [Brooks 1986, Brooks 1990, Connell 1990]. Although there seems to be no precise definition of what constitutes such an architecture, the general idea is that an agent's behavior is controlled by a number of "behavior modules." Each module receives sensory information directly from the world. If the sensory inputs satisfy a precondition specific to that module, then a certain behavior program, also specific to that module, is executed. One behavior module can subsume another. In Figure 2.8, each upper module can subsume the one below it. When module i subsumes module j, then if module i's precondition is met, then module i's program replaces that of module j. Brooks terms this a *horizontal* architecture as contrasted with a *vertical* one. Brooks and his students have demonstrated that surprisingly complex behaviors can emerge from the interaction of a relatively simple reactive machine with a complex environment [Mataric 1990, Connell 1990]. As contrasted with much other work in artificial intelligence, Brooks's machines do not depend on complex internal representations of their environments or on reasoning about them [Brooks 1991a, Brooks 1991b].[4] But, of course, one must recognize that all S-R machines, although capable of perhaps very interesting behavior, are nevertheless quite limited. ([Kirsh 1991] has written an interesting commentary on Brooks's approach.)

4 Because many of Brooks's subsumption machines do have small amounts of internal state, it might have been more appropriate to mention them in Chapter 5. We include them here because of their close relationship to T-R programs.

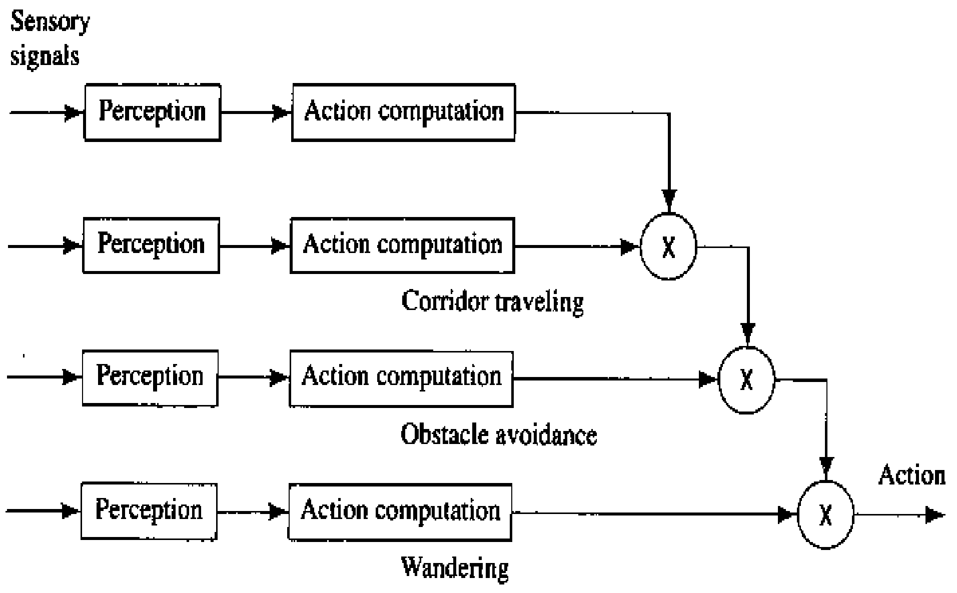


Figure 2.8

Subsumption Modules

2.3 *Additional Readings and Discussion*

S-R agents are ubiquitous in our modern, electronic world. Thermostats for maintaining temperature, cruise control for maintaining automobile speed, interrupt-driven components of a computer operating system, and thousands of different automatic devices in factories are all S-R agents. Ordinarily, these kinds of systems and devices would not be considered part of the subject matter of artificial intelligence, but I include them in my treatment because they inhabit the territory at the beginning of our journey toward more intelligent systems.

In some laboratory courses in robotics and AI, students begin by building S-R robots using Lego components. (See, for example, [Resnick 1993].) AI researchers exploring behavior-based control strategies have experimented with sonar-guided S-R robots capable of wandering, obstacle avoidance, and wall following [Mataric 1990, Connell 1990].

Our proposed split between perception and action is common in agent design. Following [Kaelbling & Rosenschein 1990], we assign to perceptual processing the task of computing a feature vector from raw sensory input, and we assign to the action function the task of selecting an action based on this feature vector. Kaelbling and Rosenschein think of this architectural style as vertical in contrast to Brooks's horizontal structure. Comparing these two styles, they say [Kaelbling & Rosenschein 1990, pp. 36-37]

> Horizontal decompositions that cut across perception and action have been advocated by Brooks as a practical way of approaching agent design. . . .

> The horizontal approach allows the designer to consider simultaneously those limited aspects of perception and action needed to support specific behaviors. . . . The alternative is a vertical strategy based on having separate system modules that recover broadly useful information from multiple sources [i.e., perception] and others that exploit it for multiple purposes [i.e., action]. The inherent combinatorics of information extraction and behavior generation make the vertical approach attractive as a way of making efficient use of a programmer's effort.

Actually, the split between perception and action can accommodate the horizontal approach also. The feature vector can be divided into separate fields—each field computed by specialized perceptual apparatus and evoking separate actions or behaviors, as in the subsumption architecture.

T-R programs and other reactive systems bear a close relationship to various ethological models of animal behavior [McFarland 1987]. In some of these models, a goal-achieving sequence of animal actions is elicited by arranging that the result of one action acts as a stimulus or trigger that "releases" the next one in the sequence. The conditions and actions of a T-R program are selected so that the program works in this manner. My work on T-R programs was inspired partly by a book by [Deutsch 1960].

Animals that seek out and move toward certain stimuli exhibit what are called *tropisms*. Phototropic animals move toward light, for example. [Genesereth & Nilsson 1987] term the kinds of agents discussed in this chapter *tropistic agents*.

Some experimental research uses simulated S-R robots rather than physical ones. Simulations are sometimes criticized as "doomed to succeed" [Brooks & Mataric 1993, p. 209], but those that do not succeed well enough help us refine perceptual and action strategies. And sometimes the simulation is actually the "real thing." For example, in computer-based education, entertainment, and dramatic art, animated characters interact with their simulated environments and with a user [Bates 1994, Blumberg 1996]. (Blumberg's ALIVE system is another example of one that uses principles from ethology to build interactive characters.)

For more on Boolean functions, see [Unger 1989]. Production systems can be thought of as a generalization of Boolean functions called *decision lists* [Rivest 1987].

Exercises

2.1 Write the following Boolean function in DNF:

$$f = (x_1 + x_2)(x_3 + x_4)$$

2.2 Show that $x_1x_2x_3 + \overline{x_1}x_2x_3 = x_2x_3$.

2.3 Indicate which of the following Boolean functions of three input variables can be realized by a single threshold element with weighted connections to the inputs. You do *not* need to calculate the weight and threshold values:

1. x_1
2. $x_1x_2x_3$
3. $x_1 + x_2 + x_3$
4. $(x_1x_2x_3) + (\overline{x_1}\,\overline{x_2}\,\overline{x_3})$
5. 1

2.4 Prove that there are exactly 3^n monomials of n dimensions and 3^n clauses of n dimensions.

2.5 Refer to the definitions of the features, x_1, x_2, x_3, x_4 on page 24 and to the rules for action on page 25. Show that the assumption that there are no "tight spaces" in the two-dimensional grid world implies that no two of the action rules can be satisified simultaneously.

2.6 Design (by hand) a neural network that accepts as inputs the sensory signals $s_1, s_2, \ldots, s_8$ and produces as outputs the conditions needed by a network of TLSA units to implement the action rules on page 25 for the wall-following robot.

2.7 In this chapter, I stated that the actions prescribed by T-R programs "proceed inexorably toward the goal." Is this statement strictly true? Can you think of situations in which a T-R program does not terminate with the top condition becoming true?