3 *Biological Foundations of the Reactive Paradigm*

Chapter Objectives:

- Describe the three levels in a *computational theory*.
- Explain in one or two sentences each of the following terms: *reflexes, taxes, fixed-action patterns, schema, affordance.*
- Be able to write pseudo-code of an animal's behaviors in terms of *innate releasing mechanisms*, identifying the releasers for the behavior.
- Given a description of an animal's sensing abilities, its task, and environment, identify an *affordance* for each behavior.
- Given a description of an animal's sensing abilities, its task, and environment, define a set of behaviors using schema theory to accomplish the task.

3.1 Overview

Progress in robotics in the 1970's was slow. The most influential robot was the Stanford Cart developed by Hans Moravec, which used stereo vision to see and avoid obstacles protruding from the ground. In the late 1970's and early 80's, Michael Arbib began to investigate models of animal intelligence from the biological and cognitive sciences in the hopes of gaining insight into what was missing in robotics. While many roboticists had been fascinated by animals, and many, including Moravec, had used some artifact of animal behavior for motivation, no one approached the field with the seriousness and dedication of Arbib.

At nearly the same time, a slender volume by Valentino Braitenberg, called *Vehicles: Experiments in Synthetic Psychology*,[25] appeared. It was a series of gedanken or entirely hypothetical thought experiments, speculating as to how machine intelligence could evolve. Braitenberg started with simple thermal sensor-motor actuator pair (Vehicle 1) that could move faster in warm areas and slower in cold areas. The next, more complex vehicle had two thermal sensor-motor pairs, one on each side of the vehicle. As a result of the differential drive effect, Vehicle 2 could turn around to go back to cold areas. Throughout the book, each vehicle added more complexity. This layering was intuitive and seemed to mimic the principles of evolution in primates. *Vehicles* became a cult tract among roboticists, especially in Europe.

Soon a new generation of AI researchers answered the siren's call of biological intelligence. They began exploring the biological sciences in search of new organizing principles and methods of obtaining intelligence. As will be seen in the next chapter, this would lead to the Reactive Paradigm. This chapter attempts to set the stage for the Reactive Paradigm by recapping influential studies and discoveries, and attempting to cast them in light of how they can contribute to robotic intelligence.

The chapter first covers animal behaviors as the fundamental primitive for sensing and acting. Next, it covers the work of Lorenz and Tinbergen in defining how concurrent simple animal behaviors interact to produce complex behaviors through Innate Releasing Mechanisms (IRMs). A key aspect of an animal behavior is that perception is needed to support the behavior. The previous chapter on the Hierarchical Paradigm showed how early roboticists attempted to fuse all sensing into a global world map, supplemented with a knowledge base. This chapter covers how the work of cognitive psychologists Ulrich Neisser[109] and J.J. Gibson[59] provides a foundation for thinking about robotic perception. Gibson refuted the necessity of global world models, a direct contradiction to the way perception was handled in the hierarchical paradigm. Gibson's use of *affordances*, also called *direct perception*, is an important key to the success of the Reactive Paradigm. Later work by Neisser attempts to define when global models are appropriate and when an affordance is more elegant.

ETHOLOGY
COGNITIVE PSYCHOLOGY

Many readers find the coverage on *ethology* (study of animal behavior) and *cognitive psychology* (study of how humans think and represent knowledge) to be interesting, but too remote from robotics. In order to address this concern, the chapter discusses specific principles and how they can be applied to robotics. It also raises issues in transferring animal models of behavior to robots. Finally, the chapter covers *schema theory*, an attempt in cognitive

psychology to formalize aspects of behavior. Schema theory has been used successfully by Arbib to represent both animal and robot behavior. It is implicitly object-oriented and so will serve as the foundation of discussions through out the remainder of this book.

3.1.1 Why explore the biological sciences?

Why should roboticists explore biology, ethology, cognitive psychology and other biological sciences? There is a tendency for people to argue against considering biological intelligence with the analogy that airplanes don't flap their wings. The counter-argument to that statement is that almost everything else about a plane's aerodynamics duplicates a bird's wing: almost all the movable surfaces on the wing of a plane perform the same functions as parts of a bird's wing. The advances in aeronautics came as the Wright Brothers and others extracted aerodynamic principles. Once the principles of flight were established, mechanical systems could be designed which adhered to these principles and performed the same functions but not necessarily in the same way as biological systems. The "planes don't flap their wings" argument turns out to be even less convincing for computer systems: animals make use of innate capabilities, robots rely on compiled programs.

Many AI roboticists often turn to the biological sciences for a variety of reasons. Animals and man provide existence proofs of different aspects of intelligence. It often helps a researcher to know that an animal can do a particular task, even if it isn't known how the animal does it, because the researcher at least knows it is possible. For example, the issue of how to combine information from multiple sensors (sensor fusion) has been an open question for years. At one point, papers were being published that robots shouldn't even try to do sensor fusion, on the grounds that sensor fusion was a phenomenon that sounded reasonable but had no basis in fact. Additional research showed that animals (including man) do perform sensor fusion, although with surprisingly different mechanisms than most researchers had considered.

OPEN WORLD ASSUMPTION
CLOSED WORLD ASSUMPTION
FRAME PROBLEM

The principles of animal intelligence are extremely important to roboticists. Animals live in an *open world*, and roboticists would like to overcome the *closed world assumption* that presented so many problems with Shakey. Many "simple" animals such as insects, fish, and frogs exhibit intelligent behavior yet have virtually no brain. Therefore, they must be doing something that avoids the frame problem.

3.1.2 Agency and computational theory

Even though it seems reasonable to explore biological and cognitive sciences for insights in intelligence, how can we compare such different systems: carbon and silicon "life" forms? One powerful means of conceptualizing the different systems is to think of an abstract intelligent system. Consider something we'll call an *agent*. The agent is self-contained and independent. It has its own "brains" and can interact with the world to make changes or to sense what is happening. It has self-awareness. Under this definition, a person is an agent. Likewise, a dog or a cat or a frog is an agent. More importantly, an intelligent robot would be an agent, even certain kinds of web search engines which continue to look for new items of interest to appear, even after the user has logged off. Agency is a concept in artificial intelligence that allows researchers to discuss the properties of intelligence without discussing the details of how the intelligence got in the particular agent. In OOP terms, "agent" is the superclass and the classes of "person" and "robot" are derived from it.

AGENT

Of course, just referring to animals, robots, and intelligent software packages as "agents" doesn't make the correspondences between intelligence any clearer. One helpful way of seeing correspondences is to decide the level at which these entities have something in common. The set of levels of commonality lead to what is often called a *computational theory*[88] after David Marr. Marr was a neurophysiologist who tried to recast biological vision processes into new techniques for computer vision. The levels in a computational theory can be greatly simplified as:

COMPUTATIONAL THEORY

Level 1: Existence proof of what can/should be done. Suppose a roboticist is interested in building a robot to search for survivors trapped in a building after an earthquake. The roboticist might consider animals which seek out humans. As anyone who has been camping knows, mosquitoes are very good at finding people. Mosquitoes provide an existence proof that it is possible for a computationally simple agent to find a human being using heat. At Level 1, agents can share a commonality of purpose or functionality.

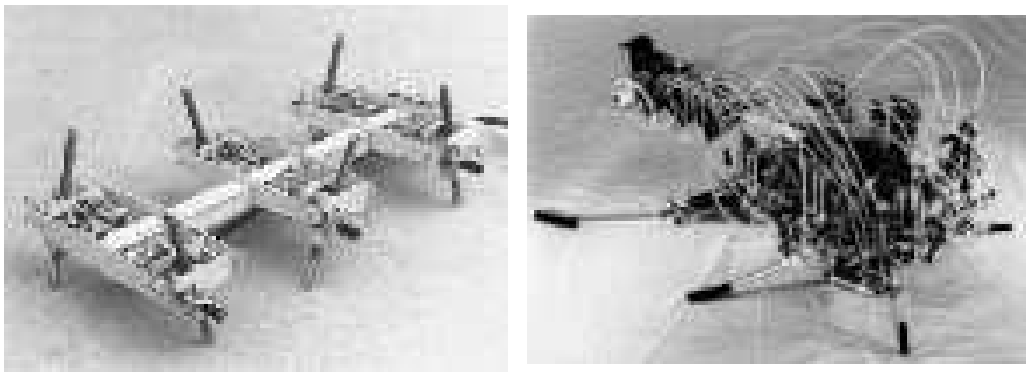
Level 2: Decomposition of "what" into inputs, outputs, and transformations. This level can be thought of as creating a flow chart of "black boxes." Each box represents a transformation of an input into an output. Returning to the example of a mosquito, the roboticist might realize from biology that the mosquito finds humans by homing on the heat of a hu-

man (or any warm blooded animal). If the mosquito senses a hot area, it flies toward it. The roboticist can model this process as: `input=thermal image`, `output=steering command`. The "black box" is how the mosquito transforms the input into the output. One good guess might be to take the centroid of the thermal image (the centroid weighted by the heat in each area of the image) and steer to that. If the hot patch moves, the thermal image will change with the next sensory update, and a new steering command will be generated. This might not be exactly how the mosquito actually steers, but it presents an idea of how a robot could duplicate the functionality. Also notice that by focusing on the process rather than the implementation, a roboticist doesn't have to worry about mosquitoes flying, while a search and rescue robot might have wheels. At Level 2, agents can exhibit common processes.

Level 3: How to implement the process. This level of the computational theory focuses on describing how each transformation, or black box, is implemented. For example, in a mosquito, the steering commands might be implemented with a special type of neural network, while in a robot, it might be implemented with an algorithm which computes the angle between the centroid of heat and where the robot is currently pointing. Likewise, a researcher interested in thermal sensing might examine the mosquito to see how it is able to detect temperature differences in such a small package; electro-mechanical thermal sensors weigh close to a pound! At Level 3, agents may have little or no commonality in their implementation.

It should be clear that Levels 1 and 2 are abstract enough to apply to any agent. It is only at Level 3 that the differences between a robotic agent and a biological agent really emerge. Some roboticists actively attempt to emulate biology, reproducing the physiology and neural mechanisms. (Most roboticists are familiar with biology and ethology, but don't try to make exact duplicates of nature.) Fig. 3.1 shows work at Case Western Reserve's Bio-Bot Laboratory under the direction of Roger Quinn, reproducing a cockroach on a neural level.

In general, it may not be possible, or even desirable, to duplicate how a biological agent does something. Most roboticists do not strive to precisely replicate animal intelligence, even though many build creatures which resemble animals, as seen by the insect-like Genghis in Fig. 3.2. But by focusing on Level 2 of a computational theory of intelligence, roboticists can gain insights into how to organize intelligence.



a. b.

Figure 3.1 Robots built at the Bio-Bot Laboratory at Case Western Reserve University which imitate cockroaches at Level 3: a.) Robot I, an earlier version, and b.) Robot III. (Photographs courtesy of Roger Quinn.)

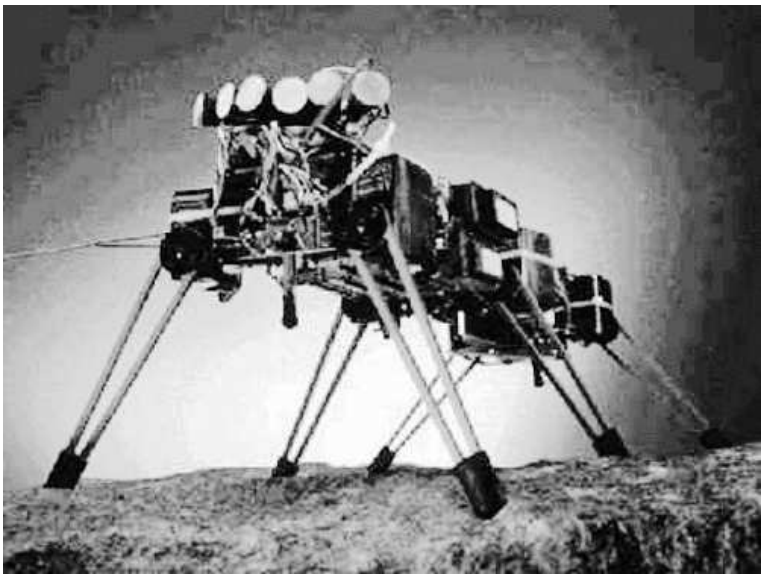


Figure 3.2 Genghis, a legged robot built by Colin Angle, IS Robotics, which imitates an insect at Levels 1 and 2. (Photograph courtesy of the National Aeronautics and Space Administration.)

Figure 3.3 Graphical definition of a behavior.

3.2 What Are Animal Behaviors?

BEHAVIOR

Scientists believe the fundamental building block of natural intelligence is a behavior. A *behavior* is a mapping of sensory inputs to a pattern of motor actions which then are used to achieve a task. For example, if a horse sees a predator, it flattens its ears, lowers its head, and paws the ground. In this case, the sensory input of a predator triggers a recognizable pattern of a defense behavior. The defensive motions make up a pattern because the actions and sequence is always the same, regardless of details which vary each episode (e.g., how many times the horse paws the ground). See Fig. 3.3.

Scientists who study animal behaviors are called *ethologists*. They often spend years in the field studying a species to identify its behaviors. Often the pattern of motor actions is easy to ascertain; the challenging part is to discover the sensory inputs for the behavior and why the behavior furthers the species survival.

REFLEXIVE BEHAVIOR
STIMULUS-RESPONSE
REACTIVE BEHAVIOR
CONSCIOUS BEHAVIOR

Behaviors can be divided into three broad categories.[10] *Reflexive behaviors* are *stimulus-response (S-R)*, such as when your knee is tapped, it jerks upward. Essentially, reflexive behaviors are "hardwired"; neural circuits ensure that the stimulus is directly connected to the response in order to produce the fastest response time. *Reactive behaviors* are learned, and then consolidated to where they can be executed without conscious thought. Any behavior that involves what is referred to in sports as "muscle memory" is usually a reactive behavior (e.g., riding a bike, skiing). Reactive behaviors can also be changed by conscious thought; a bicyclist riding over a very narrow bridge might "pay attention" to all the movements. *Conscious behaviors* are deliberative (assembling a robot kit, stringing together previously developed behaviors, etc.).

The categorization is worthy of note for several reasons. First, the Reactive Paradigm will make extensive use of reflexive behaviors, to the point that some architectures only call a robot behavior a behavior if it is S-R. Second, the categorization can help a designer determine what type of behavior to use, leading to insights about the appropriate implementation. Third, the

use of the word "reactive" in ethology is at odds with the way the word is used in robotics. In ethology, reactive behavior means learned behaviors or a skill; in robotics, it connotes a reflexive behavior. If the reader is unaware of these differences, it may be hard to read either the ethological or AI literature without being confused.

3.2.1 Reflexive behaviors

Reflexive types of behaviors are particularly interesting, since they imply no need for any type of cognition: if you sense it, you do it. For a robot, this would be a hardwired response, eliminating computation and guaranteed to be fast. Indeed, many kit or hobby robots work off of reflexes, represented by circuits.

Reflexive behaviors can be further divided into three categories:[10]

REFLEXES
1. *reflexes*: where the response lasts only as long as the stimulus, and the response is proportional to the intensity of the stimulus.

TAXES
2. *taxes*: where the response is to move to a particular orientation. Baby turtles exhibit *tropotaxis*; they are hatched at night and move to the brightest light. Until recently the brightest light would be the ocean reflecting the moon, but the intrusion of man has changed that. Owners of beach front property in Florida now have to turn off their outdoor lights during hatching season to avoid the lights being a source for tropotaxis. Baby turtles hatch at night, hidden from shore birds who normally eat them. It had been a mystery as to how baby turtles knew which way was the ocean when they hatched. The story goes that a volunteer left a flashlight on the sand while setting up an experiment intended to show that the baby turtles used magnetic fields to orient themselves. The magnetic field theory was abandoned after the volunteers noticed the baby turtles heading for the flashlight! Ants exhibit a particular taxis known as *chemotaxis*; they follow trails of pheromones.

FIXED-ACTION PATTERNS
3. *fixed-action patterns*: where the response continues for a longer duration than the stimulus. This is helpful for fleeing predators. It is important to keep in mind that a taxis can be any orientation relative to a stimulus, not just moving towards.

The above categories are not mutually exclusive. For example, an animal going over rocks or through a forest with trees to block its view might persist

(fixed-action patterns) in orienting itself to the last sensed location of a food source (taxis) when it loses sight of it.

The tight coupling of action and perception can often be quantified by mathematical expressions. An example of this is orienting in angelfish. In order to swim upright, an angelfish uses an internal (*idiothetic*) sense of gravity combined with its vision sense (*allothetic*) to see the external percept of the horizon line of the water to swim upright. If the fish is put in a tank with prisms that make the horizon line appear at an angle, the angelfish will swim cockeyed. On closer inspection, the angle that the angelfish swims at is the vector sum of the vector parallel to gravity with the vector perpendicular to the perceived horizon line! The ability to quantify animal behavior suggests that computer programs can be written which do likewise.

IDIOTHETIC
ALLOTHETIC

3.3 Coordination and Control of Behaviors

KONRAD LORENZ
NIKO TINBERGEN

Konrad Lorenz and Niko Tinbergen were the founding fathers of ethology. Each man independently became fascinated not only with individual behaviors of animals, but how animals acquired behaviors and selected or coordinated sets of behaviors. Their work provides some insight into four different ways an animal might acquire and organize behaviors. Lorenz and Tinbergen's work also helps with a computational theory Level 2 understanding of how to make a process out of behaviors.

The four ways to acquire a behavior are:

INNATE

1. to be born with a behavior (*innate*). An example is the feeding behavior in baby arctic terns. Arctic terns, as the name implies, live in the Arctic where the terrain is largely shades of black and white. However, the Arctic tern has a bright reddish beak. When babies are hatched and are hungry, they peck at the beak of their parents. The pecking triggers a regurgitation reflex in the parent, who literally coughs up food for the babies to eat. It turns out that the babies do not recognize their parents, per se. Instead, they are born with a behavior that says: if hungry, peck at the largest red blob you see. Notice that the only red blobs in the field of vision should be the beaks of adult Arctic terns. The largest blob should be the nearest parent (the closer objects are, the bigger they appear). This is a simple, effective, and computationally inexpensive strategy.

SEQUENCE OF INNATE BEHAVIORS

2. to be born with a *sequence of innate behaviors*. The animal is born with a sequence of behaviors. An example is the mating cycle in digger wasps.

A female digger wasp mates with a male, then builds a nest. Once it sees the nest, the female lays eggs. The sequence is logical, but the important point is the role of stimulus in triggering the next step. The nest isn't built until the female mates; that is a change in internal state. The eggs aren't laid until the nest is built; the nest is a visual stimulus releasing the next step. Notice that the wasp doesn't have to "know" or understand the sequence. Each step is triggered by the combination of internal state and the environment. This is very similar to Finite State Machines in computer science programming, and will be discussed later in Ch. 5.

INNATE WITH MEMORY

3. to be born with behaviors that need some initialization (*innate with memory*). An animal can be born with innate behaviors that need customizing based on the situation the animal is born in. An example of this is bees. Bees are born in hives. The location of a hive is something that isn't innate; a baby bee has to learn what its hive looks like and how to navigate to and from it. It is believed that the curious behavior exhibited by baby bees (which is innate) allows them to learn this critical information. A new bee will fly out of the hive for a short distance, then turn around and come back. This will get repeated, with the bee going a bit farther along the straight line each time. After a time, the bee will repeat the behavior but at an angle from the opening to the hive. Eventually, the bee will have circumnavigated the hive. Why? Well, the conjecture is that the bee is learning what the hive looks like from all possible approach angles. Furthermore, the bee can associate a view of the hive with a motor command ("fly left and down") to get the bee to the opening. The behavior of zooming around the hive is innate; what is learned about the appearance of the hive and where the opening is requires memory.

LEARN

4. to *learn* a set of behaviors. Behaviors are not necessarily innate. In mammals and especially primates, babies must spend a great deal of time learning. An example of learned behaviors is hunting in lions. Lion cubs are not born with any hunting behaviors. If they are not taught by their mothers over a period of years, they show no ability to fend for themselves. At first it might seem strange that something as fundamental as hunting for food would be learned, not innate. However, consider the complexity of hunting for food. Hunting is composed of many sub-behaviors, such as searching for food, stalking, chasing, and so on. Hunting may also require teamwork with other members of the pride. It requires great sensitivity to the type of the animal being hunted and the terrain. Imagine trying to write a program to cover all the possibilities!

While the learned behaviors are very complex, they can still be represented by innate releasing mechanisms. It is just that the releasers and actions are learned; the animal creates the program itself.

Note that the number of categories suggests that a roboticist will have a spectrum of choices as to how a robot can acquire one or more behaviors: from being pre-programmed with behaviors (innate) to somehow learning them (learned). It also suggests that behaviors can be innate but require memory. The lesson here is that while S-R types of behaviors are simple to preprogram or hardwire, robot designers certainly shouldn't exclude the use of memory. But as will be seen in Chapter 4, this is a common constraint placed on many robot systems. This is especially true in a popular style of hobby robot building called BEAM robotics (biology, electronics, aesthetics, and mechanics), espoused by Mark Tilden. Numerous BEAM robot web sites guide adherents through construction of circuits which duplicate memoryless innate reflexes and taxes.

INTERNAL STATE
MOTIVATION

An important lesson that can be extracted from Lorenz and Tinbergen's work is that the internal state and/or motivation of an agent may play a role in releasing a behavior. Being hungry is a stimulus, equivalent to the pain introduced by a sharp object in the robot's environment. Another way of looking at it is that motivation serves as a stimulus for behavior. Motivations can stem from survival conditions (like being hungry) or more abstract goals (e.g., need to check the mail). One of the most exciting insights is that behaviors can be sequenced to create complex behaviors. Something as complicated as mating and building a nest can be decomposed into primitives or certainly more simple behaviors. This has an appeal to the software engineering side of robotics.

3.3.1 Innate releasing mechanisms

INNATE RELEASING
MECHANISMS

RELEASER

Lorenz and Tinbergen attempted to clarify their work in how behaviors are coordinated and controlled by giving it a special name *innate releasing mechanisms* (IRM). An IRM presupposes that there is a specific stimulus (either internal or external) which releases, or triggers, the stereotypical pattern of action. The IRM activates the behavior. A *releaser* is a latch or a Boolean variable that has to be set. One way to think of IRMs is as a process of behaviors. In a computational theory of intelligence using IRMs, the basic black boxes of the process would be behaviors. Recall that behaviors take sensory input and produce motor actions. But IRMs go further and specify when a behav-

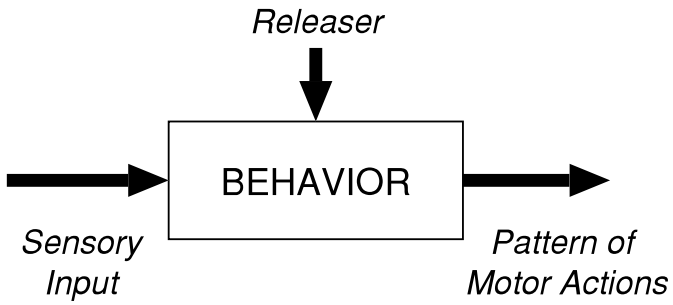


Figure 3.4 Innate Releasing Mechanism as a process with behaviors.

ior gets turned on and off. The releaser acts as a control signal to activate a behavior. If a behavior is not released, it does not respond to sensory inputs and does not produce motor outputs. For example, if a baby arctic tern isn't hungry, it doesn't peck at red, even if there is a red beak nearby.

Another way to think of IRMs is as a simple computer program. Imagine the agent running a C program with a continuous `while` loop. Each execution through the loop would cause the agent to move for one second, then the loop would repeat.

```
enum            Releaser={PRESENT, NOT_PRESENT};
Releaser        predator;
while (TRUE)
{
   predator = sensePredators();
   if (predator == PRESENT)
     flee();
}
```

In this example, the agent does only two things: sense the world and then flees if it senses a predator. Only one behavior is possible: `flee`. `flee` is released by the presence of a predator. A predator is of type `Releaser` and has only two possible values: it is either present or it is not. If the agent does not sense the releaser for the behavior, the agent does nothing. There is no "default" behavior.

This example also shows filtering of perception. In the above example, the agent only looks for predators with a dedicated detection function, `sense-Predators()`. The dedicated predator detection function could be a specialized sense (e.g., retina is sensitive to the frequency of motions associated

with predator movement) or a group of neurons which do the equivalent of a computer algorithm.

COMPOUND RELEASERS

Another important point about IRMs is that the releaser can be a *compound of releasers*. Furthermore, the releaser can be a combination of either external (from the environment) or internal (motivation). If the releaser in the compound isn't satisfied, the behavior isn't triggered. The pseudo-code below shows a compound releaser.

```
enum            Releaser={PRESENT, NOT_PRESENT};
Releaser        food;
while (TRUE)
{
   food = senseFood();
   hungry = checkState();
      if (food == PRESENT && hungry==PRESENT)
      feed();
}
```

IMPLICIT CHAINING

The next example below shows what happens in a sequence of behaviors, where the agent eats, then nurses its young, then sleeps, and repeats the sequence. The behaviors are implicitly chained together by their releasers. Once the initial releaser is encountered, the first behavior occurs. It executes for one second (one "movement" interval), then control passes to the next statement. If the behavior isn't finished, the releasers remain unchanged and no other behavior is triggered. The program then loops to the top and the original behavior executes again. When the original behavior has completed, the internal state of the animal may have changed or the state of the environment may have been changed as a result of the action. When the motivation and environment match the stimulus for the releaser, the second behavior is triggered, and so on.

```
enum            Releaser={PRESENT, NOT_PRESENT};
Releaser        food, hungry, nursed;
while (TRUE) {
   food = sense();
   hungry = checkStateHunger();
   child = checkStateChild();
   if (hungry==PRESENT)
      searchForFood(); //sets food = PRESENT when done
   if (hungry==PRESENT && food==PRESENT)
      feed(); // sets hungry = NOT_PRESENT when done
```

```
   if (hungry== NOT_PRESENT && parent==PRESENT)
      nurse(); // set nursed = PRESENT when done
   if (nursed ==PRESENT)
      sleep();
}
```

The example also reinforces the nature of behaviors. If the agent sleeps and wakes up, but isn't hungry, what will it do? According to the releasers created above, the agent will just sit there until it gets hungry.

In the previous example, the agent's behaviors allowed it to feed and enable the survival of its young, but the set of behaviors did not include fleeing or fighting predators. Fleeing from predators could be added to the program as follows:

```
enum            Releaser={PRESENT, NOT_PRESENT};
Releaser        food, hungry, nursed, predator;
while (TRUE) {
   predator = sensePredator();
   if (predator==PRESENT)
      flee();
   food = senseFood();
   hungry = checkStateHunger();
   parent = checkStateParent();
   if (hungry==PRESENT)
      searchForFood();
   if (hungry==PRESENT && food==PRESENT)
      feed();
   if (hungry== NOT_PRESENT && parent==PRESENT)
      nurse();
   if (nursed ==PRESENT)
      sleep();
}
```

Notice that this arrangement allowed the agent to flee the predator regardless of where it was in the sequence of feeding, nursing, and sleeping because predator is checked for first. But fleeing is temporary, because it did not change the agent's internal state (except possibly to make it more hungry which will show up on the next iteration). The code may cause the agent to flee for one second, then feed for one second.

One way around this is to *inhibit*, or turn off, any other behavior until fleeing is completed. This could be done with an `if-else` statement:

```
while (TRUE) {
   predator = sensePredator();
   if (predator==PRESENT)
      flee();
   else {
      food = senseFood();
      hungry = checkStateHunger();
      ...
   }
}
```

The addition of the `if-else` prevents other, less important behaviors from executing. It doesn't solve the problem with the predator releaser disappearing as the agents runs away. If the agent turns and the predator is out of view (say, behind the agent), the value of `predator` will go to `NOT_PRESENT` in the next update. The agent will go back to foraging, feeding, nursing, or sleeping. Fleeing should be a fixed-pattern action behavior which persists for some period of time, `T`. The fixed-pattern action effect can be accomplished with:

```
#define T LONG_TIME
while (TRUE) {
   predator = sensePredator();
   if (predator==PRESENT)
      for(time = T; time > 0; time--)
         flee();
   else {
      food = senseFood();
      ...
   }
}
```

The C code examples were implemented as an implicit sequence, where the order of execution depended on the value of the releasers. An implementation of the same behaviors with an explicit sequence would be:

```
Releaser         food, hungry, nursed, predator;
while (TRUE) {

   predator = sensePredator();
   if (predator==PRESENT)
```

```
      flee();
   food = senseFood();
   hungry = checkStateHunger();
   parent = checkStateParent();
   if (hungry==PRESENT)
      searchForFood();
   feed();
   nurse();
   sleep();
}
```

The explicit sequence at first may be more appealing. It is less cluttered and the compound releasers are hidden. But this implementation is not equivalent. It assumes that instead of the loop executing every second and the behaviors acting incrementally, each behavior takes control and runs to completion. Note that the agent cannot react to a predator until it has finished the sequence of behaviors. Calls to the fleeing behavior could be inserted between each behavior or fleeing could be processed on an interrupt basis. But every "fix" makes the program less general purpose and harder to add and maintain.

The main point here is: *simple behaviors operating independently can lead to what an outside observer would view as a complex sequence of actions.*

3.3.2 Concurrent behaviors

An important point from the examples with the IRMs is that behaviors can, and often do, execute concurrently and independently. What appears to be a fixed sequence may be the result of a normal series of events. However, some behaviors may violate or ignore the implicit sequence when the environment presents conflicting stimuli. In the case of the parent agent, fleeing a predator was mutually exclusive of the feeding, nursing, and sleeping behaviors.

Interesting things can happen if two (or more) behaviors are released that usually are not executed at the same time. It appears that the strange interactions fall into the following categories:

EQUILIBRIUM

- *Equilibrium (the behaviors seem to balance each other out):* Consider feeding versus fleeing in a squirrel when the food is just close enough to a person on a park bench. A squirrel will often appear to be visibly undecided as to whether to go for the food or to stay away.

DOMINANCE
- *Dominance of one (winner take all):* you're hungry and sleepy. You do one or the other, not both simultaneously.

CANCELLATION
- *Cancellation (the behaviors cancel each other out):* Male sticklebacks (fish) when their territories overlap get caught between the need to defend their territory and to attack the other fish. So the males make another nest! Apparently the stimuli cancels out, leaving only the stimulus normally associated with nest building.

Unfortunately, it doesn't appear to be well understood when these different mechanisms for conflicting behaviors are employed. Clearly, there's no one method. But it does emphasize that a roboticist who works with behaviors should pay close attention to how the behaviors will interact. This will give rise to the differences in architectures in the Reactive and Hybrid Paradigms, discussed in later chapters.

3.4 Perception in Behaviors

While Lorenz and Tinbergen's work provides some insights into behaviors, it's clear that behaviors depend on perception. Ulrich Neisser, who literally created the term "cognitive psychology" in his book, *Cognition and Reality*, argued that perception cannot be separated from action.[109] As will be seen in this section, J.J. Gibson, a very controversial cognitive psychologist, spent his career advocating an ecological approach to perception. The ecological approach is the opposite of the top-down, model-based reasoning about the environment approach favored by psychologists, including Neisser. Interestingly enough, Neisser took a position at Cornell where J.J. Gibson was, and they became close colleagues. Since then, Neisser has spent significant time and thought trying to reconcile the two views based on studies; this has led to his identification of two perceptual systems.

3.4.1 Action-perception cycle

ACTION-PERCEPTION CYCLE
The *action-perception cycle* illustrates that perception is fundamental to any intelligent agent. A simple interpretation of the cycle is: When an agent acts, it interacts with its environment because it is situated in that environment; it is an integral part of the environment. So as it acts, it changes things or how it perceives it (e.g., move to a new viewpoint, trigger a rock slide, etc.). Therefore the agent's perception of the world is modified. This new perception is

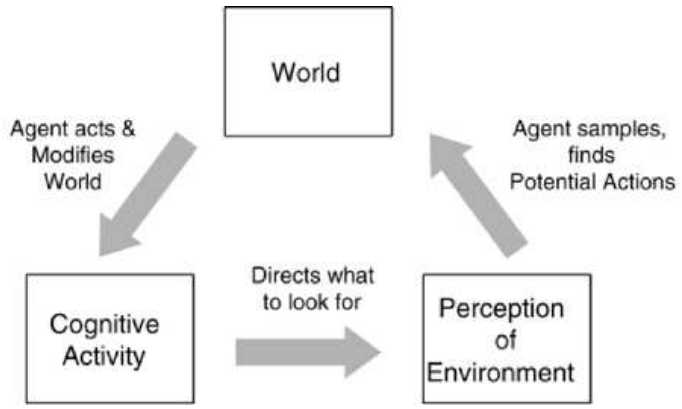


Figure 3.5 Action-Perception Cycle.[7]

then used for a variety of functions, including both cognitive activities like planning for what to do next as well as reacting. The term cognitive activity includes the concepts of feedback and feedforward control, where the agent senses an error in what it attempted to do and what actually happened. An equally basic cognitive activity is determining what to sense next. That activity can be something as straightforward as activating processes to look for releasers, or as complex as looking for a particular face in a crowd.

Regardless of whether there is an explicit conscious processing of the senses or the extraction of a stimulus or releaser, the agent is now directed in terms of what it is going to perceive on the next update(s). This is a type of selective attention or focus-of-attention. As it perceives, the agent perceptually samples the world. If the agent actually acts in a way to gather more perception before continuing with its primary action, then that is sometimes referred to as active perception. Part of the sampling process is to determine the potential for action. Lorenz and Tinbergen might think of this as the agent having a set of releasers for a task, and now is observing whether they are present in the world. If the perception supports an action, the agent acts. The action modifies the environment, but it also modifies the agent's assessment of the situation. In the simplest case, this could be an error signal to be used for control or a more abstract difference such as at the level of those used in STRIPS/MEA.

In some regards, the action-perception cycle appears to bear a superficial resemblance to the Hierarchical Paradigm of SENSE, PLAN, ACT. However, note that 1) there is no box which contains ACT, and 2) the cycle does not require the equivalent of planning to occur at each update. Action is implicit

in an agent; the interesting aspect of the cycle is where perception and cognition come in. The agent may have to act to acquire more perception or to accomplish a task. Also, the agent may or may not need to "plan" an action on each update.

3.4.2 Two functions of perception

Perception in behavior serves two functions. First, as we saw with IRMs, it serves to *release* a behavior. However, releasing a behavior isn't necessarily the same as the second function: perceiving the information needed to accomplish the behavior. For example, consider an animal in a forest fire. The fire activates the fleeing. But the fleeing behavior needs to extract information about open spaces to run through obstacles in order to *guide* the behavior. A frightened deer might bolt right past a hunter without apparently noticing.

RELEASE

GUIDE

In both roles as a releaser and as a guide for behavior, perception filters the incoming stimulus for the task at hand. This is often referred to as action-oriented perception by roboticists, when they wish to distinguish their perceptual approach from the more hierarchical global models style of perception. Many animals have evolved specialized detectors which simplify perception for their behaviors. Some frogs which sit in water all day with just half their eyes poking up have a split retina: the lower half is good for seeing in water, the upper half in air.

3.4.3 Gibson: Ecological approach

The central tenet of Gibson's approach is that "... the world is its own best representation." Gibson's work is especially interesting because it complements the role of perception in IRM and is consistent with the action-perception cycle. Gibson postulated (and proved) the existence of affordances. *Affordances are perceivable potentialities of the environment for an action.* For example, to a baby arctic tern, the color red is perceivable and represents the potential for feeding. So an affordance can be a more formal way of defining the external stimulus in IRM. But like IRMs, an affordance is only a potential—it doesn't count until all the other conditions are satisfied (the baby tern is hungry). An affordance can also be the percept that guides the behavior. The presence of red to a hungry baby arctic tern releases the feeding behavior. But the feeding behavior consists of pecking at the red object. So in this case, red is also the percept being used to guide the action, as well as release it.

AFFORDANCES

Gibson referred to his work as an "ecological approach" because he believed that perception evolved to support actions, and that it is silly to try to discuss perception independently of an agent's environment, and its survival behaviors. For example, a certain species of bees prefers one special type of poppy. But for a long time, the scientists couldn't figure out how the bees recognized that type of poppy because as color goes, it was indistinguishable from another type of poppy that grows in the same area. Smell? Magnetism? Neither. They looked at the poppy under UV and IR light. In the non-visible bands that type of poppy stood out from other poppy species. And indeed, the scientists were able to locate retinal components sensitive to that bandwidth. The bee and poppy had co-evolved, where the poppy's color evolved to a unique bandwidth while at the same time the bee's retina was becoming specialized at detecting that color. With a retina "tuned" for the poppy, the bee didn't have to do any reasoning about whether there was a poppy in view, and, if so, was it the right species of poppy. If that color was present, the poppy was there.

Fishermen have exploited affordances since the beginning of time. A fishing lure attempts to emphasize those aspects of a fish's desired food, presenting the strongest stimulus possible: if the fish is hungry, the stimulus of the lure will trigger feeding. As seen in Fig. 3.6, fishing lures often look to a human almost nothing like the bait they imitate.

DIRECT PERCEPTION

What makes Gibson so interesting to roboticists is that an affordance is directly perceivable. Direct perception means that the sensing process doesn't require memory, inference, or interpretation. This means minimal computation, which usually translates to very rapid execution times (near instantaneous) on a computer or robot.

But can an agent actually perceive anything meaningful without some memory, inference, or interpretation? Well, certainly baby arctic terns don't need memory or inference to get food from a parent. And they're definitely not interpreting red in the sense of: "oh, there's a red blob. It's a small oval, which is the right shape for Mom, but that other one is a square, so it must be a graduate ethology student trying to trick me." For baby arctic terns, it's simply: `red = food, bigger red = better`.

Does this work for humans? Consider walking down the hall and somebody throws something at you. You will most likely duck. You also probably ducked without recognizing the object, although later you may determine it was only a foam ball. The response happens too fast for any reasoning: "Oh look, something is moving towards me. It must be a ball. Balls are usually hard. I should duck." Instead, you probably used a phenomena so basic that

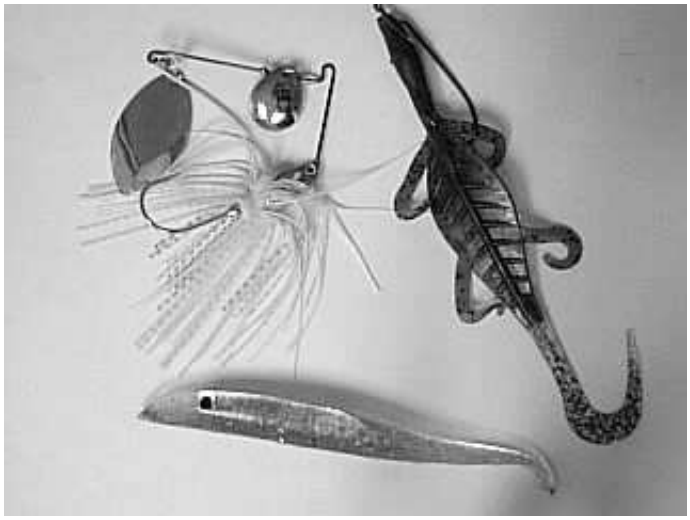


Figure 3.6 A collection of artificial bait, possibly the first example of humans exploiting affordances. Notice that the lures exaggerate one or more attributes of what a fish might eat.

OPTIC FLOW

you haven't noticed it, called *optic flow*. Optic flow is a neural mechanism for determining motion. Animals can determine time to contact quite easily with it. You probably are somewhat familiar with optic flow from driving in a car. When driving or riding in a car, objects in front seem to be in clear focus but the side of the road is a little blurry from the speed. The point in space that the car is moving to is the focus of expansion. From that point outward, there is a blurring effect. The more blurring on the sides, the faster the car is going. (They use this all the time in science fiction movies to simulate faster-than-light travel.) That pattern of blurring is known as a flow field (because it can be represented by vectors, like a gravitational or magnetic field). It is straightforward, neurally, to extract the time to contact, represented in the cognitive literature by τ.

TIME TO CONTACT

Gannets and pole vaulters both use optic flow to make last-minute, precise movements as reflexes. Gannets are large birds which dive from high altitudes after fish. Because the birds dive from hundreds of feet up in the air, they have to use their wings as control surfaces to direct their dive at the targeted fish. But they are plummeting so fast that if they hit the water with their wings open, the hollow bones will shatter. Gannets fold their wings just before hitting the water. Optic flow allows the time to contact, τ, to be a stimulus: when the time to contact dwindles below a threshold, fold those wings!

Pole vaulters also make minute adjustments in where they plant their pole as they approach the hurdle. This is quite challenging given that the vaulter is running at top speed. It appears that pole vaulters use optic flow rather than reason (slowly) about where the best place is for the pole. (Pole vaulting isn't the only instance where humans use optic flow, just one that has been well-documented.)

In most applications, a fast computer program can extract an affordance. However, this is not the case (so far) with optic flow. Neural mechanisms in the retina have evolved to make the computation very rapid. It turns out that computer vision researchers have been struggling for years to duplicate the generation of an optical flow field from a camera image. Only recently have we seen any algorithms which ran in real-time on regular computers.[48] The point is that affordances and specialized detectors can be quite challenging to duplicate in computers.

Affordances are not limited to vision. A common affordance is knowing when a container is almost filled to the top. Think about filling a jug with water or the fuel tank of a car. Without being able to see the cavity, a person knows when the tank is almost filled by the change in sound. That change in sound is directly perceivable; the person doesn't need to know anything about the size or shape of the volume being filled or even what the liquid is.

One particularly fascinating application of affordances to robotics, which also serves to illustrate what an affordance is, is the research of Louise Stark and Kevin Bowyer.[135] A seemingly unsurmountable problem in computer vision has been to have a computer recognize an object from a picture. Literally, the computer should say, "that's a chair" if the picture is of a chair.

STRUCTURAL MODELS

The traditional way of approaching the problem has been to use *structural models*. A structural model attempts to describe an object in terms of physical components: "A chair has four legs, a seat, and a back." But not all chairs fit the same structural model. A typing chair has only one leg, with supports at the bottom. Hanging baskets don't have legs at all. A bench seat doesn't have a back. So clearly the structural approach has problems: instead of one structural representation, the computer has to have access to many different models. Structural models also lack flexibility. If the robot is presented with a new kind of chair (say someone has designed a chair to look like your toilet or an upside down trash can), the robot would not be able to recognize it without someone explicitly constructing another structural model.

Stark and Bowyer explored an alternative to the structual approach called GRUFF. GRUFF identifies chairs by *function* rather than form. Under Gibsonian perception, a chair should be a chair because it affords sitting, or serves

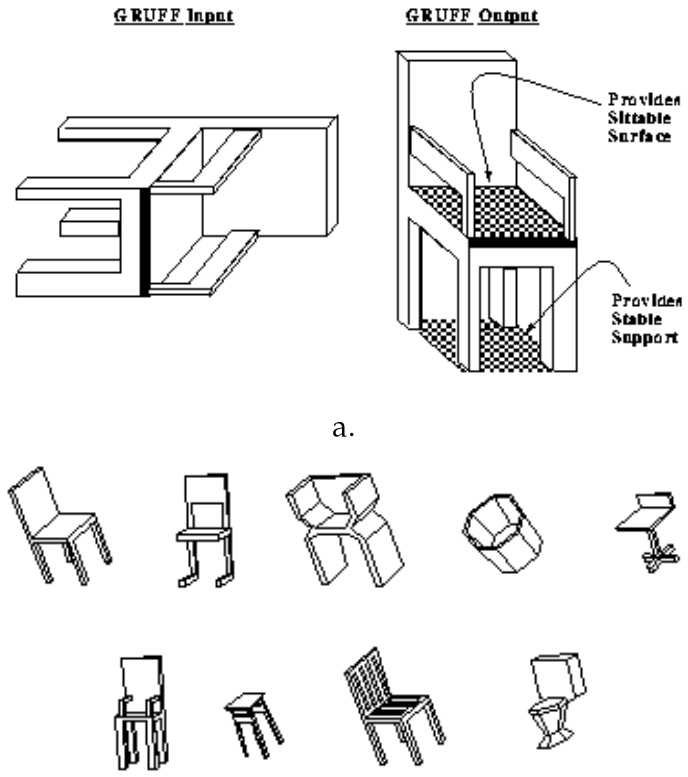


Figure 3.7 The GRUFF system: a.) input, and b.) different types of chairs recognized by GRUFF. (Figures courtesy of Louise Stark.)

the function of sittability. And that affordance of sittability should be something that can be extracted from an image:

- Without memory (the agent doesn't need to memorize all the chairs in the world).
- Without inference (the robot doesn't need to reason: "if it has 4 legs, and a seat and a back, then it's a chair; we're in an area which should have lots of chairs, so this makes it more likely it's a chair").
- Without an interpretation of the image (the robot doesn't need to reason: "there's an arm rest, and a cushion, ..."). A computer should just be able to look at a picture and say if something in that picture is sittable or not.

Stark and Bowyer represented sittability as a reasonably level and continuous surface which is at least the size of a person's butt and at about the height of their knees. (Everything else like seat backs just serve to specify the kind of chair.) Stark and Bowyer wrote a computer program which accepted CAD/CAM drawings from students who tried to come up with non-intuitive things that could serve as chairs (like toilets, hanging basket chairs, trash cans). The computer program was able to correctly identify sittable surfaces that even the students missed.

It should be noted that Stark and Bowyer are hesitant to make claims about what this says about Gibsonian perception. The computer vision algorithm can be accused of some inference and interpretation ("that's the seat, that's the right height"). But on the other hand, that level of inference and interpretation is significantly different than that involved in trying to determine the structure of the legs, etc. And the relationship between seat size and height could be represented in a special neural net that could be *released* whenever the robot or animal got tired and wanted to sit down. The robot would start noticing that it could sit on a ledge or a big rock if a chair or bench wasn't around.

3.4.4 Neisser: Two perceptual systems

At this point, the idea of affordances should seem reasonable. A chair is a chair because it affords sittability. But what happens when someone sits in *your* chair? It would appear that humans have some mechanism for recognizing specific instances of objects. Recognition definitely involves memory ("my car is a blue Ford Explorer and I parked it in slot 56 this morning"). Other tasks, like the kind of sleuthing Sherlock Holmes does, may require inference and interpretation. (Imagine trying to duplicate Sherlock Holmes in a computer. It's quite different than mimicking a hungry baby arctic tern.)

So while affordances certainly are a powerful way of describing perception in animals, it is clearly not the only way animals perceive. Neisser postulated that there are two perceptual systems in the brain (and cites neurophysiological data):[110]

DIRECT PERCEPTION
1. *direct perception*. This is the "Gibsonian," or ecological, track of the brain, and consists of structures low in the brain which evolved earlier on and accounts for affordances.

RECOGNITION
2. *recognition*. This is more recent perceptual track in the brain, which ties in with the problem solving and other cognitive activities. This part accounts

for the use of internal models to distinguish "your coffee cup" from "my coffee cup." This is where top-down, model-based perception occurs.

On a more practical note, Neisser's dichotomy suggests that the first decision in designing a behavior is to determine whether a behavior can be accomplished with an affordance or requires recognition. If it can be accomplished with an affordance, then there may be a simple and straightforward way to program it in a robot; otherwise, we will most certainly have to employ a more sophisticated (and slower) perceptual algorithm.

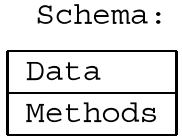
3.5 Schema Theory

Schema theory provides a helpful way of casting some of the insights from above into an object-oriented programming format.[6] Psychologists have used schema theory since the early 1900's. It was first brought to the serious attention of AI roboticists by Michael Arbib while at the University of Massachusetts, and later used extensively by Arkin and Murphy for mobile robotics, Lyons and Iberall for manipulation,[75] and Draper et al. for vision.[46]

SCHEMA

Schemas were conceived of by psychologists as a way of expressing the basic unit of activity. A *schema* consists both of the knowledge of how to act and/or perceive (knowledge, data structures, models) as well as the computational process by which it is uses to accomplish the activity (the algorithm). The idea of a schema maps nicely onto a class in object-oriented programming (OOP). A *schema class* in C++ or Java would contain both data (knowledge, models, releasers) and methods (algorithms for perceiving and acting), as shown below.

SCHEMA CLASS



A schema is a generic template for doing some activity, like riding a bicycle. It is a template because a person can ride different bicycles without starting the learning process all over. Since a schema is parameterized like a class, parameters (type of bicycle, height of the bicycle seat, position of the handlebars) can be supplied to the object at the time of instantiation (when an object is created from the class). As with object-oriented programming, the creation of a specific schema is called a *schema instantiation (SI)*.

SCHEMA INSTANTIATION (SI)

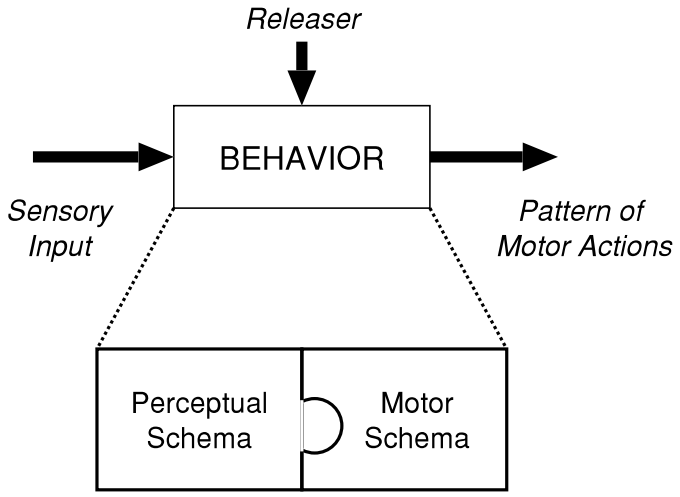


Figure 3.8 Behaviors decomposed into perceptual and motor schemas.

The schema instantiation is the object which is constructed with whatever parameters are needed to tailor it to the situation. For example, there could be a `move_to_food` schema where the agent always heads in a straight line to the food. Notice that the "always heads in a straight line" is a template of activity, and a reusable algorithm for motion control. However, it is just a method; until the `move_to_food` schema is instantiated, there is no specific goal to head for, e.g., the candy bar on the table. The same schema could be instantiated for moving to a sandwich.

3.5.1 Behaviors and schema theory

COMPOSITION OF A BEHAVIOR
MOTOR SCHEMA
PERCEPTUAL SCHEMA

In the Arbibian application of schema theory towards a computational theory of intelligence, a *behavior* is a schema which is composed of a motor schema and a perceptual schema. The *motor schema* represents the template for the physical activity, the *perceptual schema* embodies the sensing. The motor schema and perceptual schema are like pieces of a puzzle; both pieces must be together in place before there is a behavior. This idea is shown below in Fig. 3.8.

Essentially, the perceptual and motor schema concept fits in with ethology and cognitive psychology as follows:

- A behavior takes sensory inputs and produces motor actions as an output.

- A behavior can be represented as a schema, which is essentially an object-oriented programming construct.
- A behavior is activated by releasers.
- The transformation of sensory inputs into motor action outputs can be divided into two sub-processes: a perceptual schema and a motor schema.

In OOP terms, the motor schema and perceptual schema classes are derived from the schema class. A primitive behavior just has one motor and one perceptual schema.

`Behavior::Schema`

Data	
Methods	`perceptual_schema()` `motor_schema()`

Recall from IRMs, more sophisticated behaviors may be constructed by sequencing behaviors. In the case of a sequence of behaviors, the overall behavior could be represented in one of two ways. One way is to consider the behavior to be composed of several primitive behaviors, with the releasing logic to serve as the knowledge as to when to activate each primitive behaviors. This is probably the easiest way to express a "meta" behavior.

A meta-behavior composed of three behaviors can be thought of as:

`Behavior::Schema`

Data	`releaser1`
	`releaser2`
	`releaser3`
	`IRM_logic()`

Methods	`behavior1()`
	`behavior2()`
	`behavior3()`

However, in more advanced applications, the agent may have a choice of either perceptual or motor schemas to tailor its behavior. For example, a person usually uses vision (the default perceptual schema) to navigate out of a room (motor schema). But if the power is off, the person can use touch (an alternate perceptual schema) to feel her way out of a dark room. In this case, the schema-specific knowledge is knowing which perceptual schema

to use for different environmental conditions. Schema theory is expressive enough to represent basic concepts like IRMs, plus it supports building new behaviors out of primitive components. This will be discussed in more detail in later chapters.

This alternative way of creating a behavior by choosing between alternative perceptual and motor schemas can be thought of as:

Behavior::Schema

Data	environmental_state
Methods	choose_PS(environmental_state) perceptual_schema_1() perceptual_schema_2() motor_schema()

RANA COMPUTATRIX

Arbib and colleagues did work constructing computer models of visually guided behaviors in frogs and toads. They used schema theory to represent the toad's behavior in computational terms, and called their model *rana computatrix* (rana is the classification for toads and frogs). The model explained Ingle's observations as to what occasionally happens when a toad sees two flies at once.[33] Toads and frogs can be characterized as responding visually to either small, moving objects and large, moving objects. Small, moving objects release the feeding behavior, where the toad orients itself towards the object (taxis) and then snaps at it. (If the object turns out not to be a fly, the toad can spit it out.) Large moving objects release the fleeing behavior, causing the toad to hop away. The feeding behavior can be modeled as a behavioral schema, or template, shown in Fig. 3.9.

When the toad sees a fly, an instance of the behavior is instantiated; the toad turns toward that object and snaps at it. Arbib's group went one level further on the computational theory.[7] They implemented the taxis behavior as a vector field: rana computatrix would literally feel an attractive force along the direction of the fly. This direction and intensity (magnitude) was represented as a vector. The direction indicated where rana had to turn and the magnitude indicated the strength of snapping. This is shown in Fig. 3.10.

What is particularly interesting is that the rana computatrix program predicts what Ingle saw in real toads and frogs when they are presented with two flies simultaneously. In this case, each fly releases a separate instance of the feeding behavior. Each behavior produces the vector that the toad needs to turn to in order to snap at that fly, without knowing that the other be-

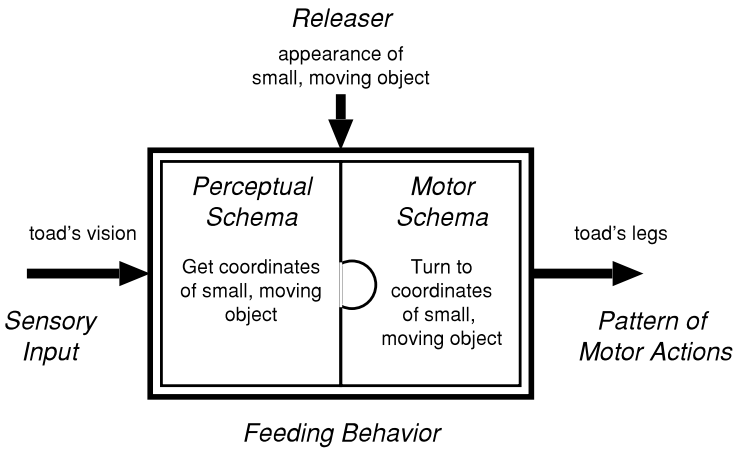


Figure 3.9 Toad's feeding behavior represented as a behavior with schema theory.

havior exists. According to the vector field implementation of the schema model, the toad now receives two vectors, instead of one. What to do? Well, rana computatrix summed the two vectors, resulting in a third vector in between the original two! The toad snaps at neither fly, but in the middle. The unexpected interaction of the two independent instances probably isn't that much of a disadvantage for a toad, because if there are two flies in such close proximity, eventually one of them will come back into range.

This example illustrates many important lessons for robotics. First, it validates the idea of a computational theory, where functionality in an animal and a computer can be equivalent. The concept of behaviors is Level 1 of the computational theory, schema theory (especially the perceptual and motor schemas) expresses Level 2, and Level 3 is the vector field implementation of the motor action. It shows the property of emergent behavior, where the agent appears to do something fairly complex, but is really just the result of interaction between simple modules. The example also shows how behaviors correspond to object-orienting programming principles.

Another desirable aspect of schema theory is that it supports reflex behaviors. Recall that in reflex behaviors, the strength of the response is proportional to the strength of the stimulus. In schema theory, the perceptual schema is permitted to pass both the percept and a gain to the motor schema. The motor schema can use the gain to compute a magnitude on the output action. This is an example of how a particular schema can be tailored for a behavior.

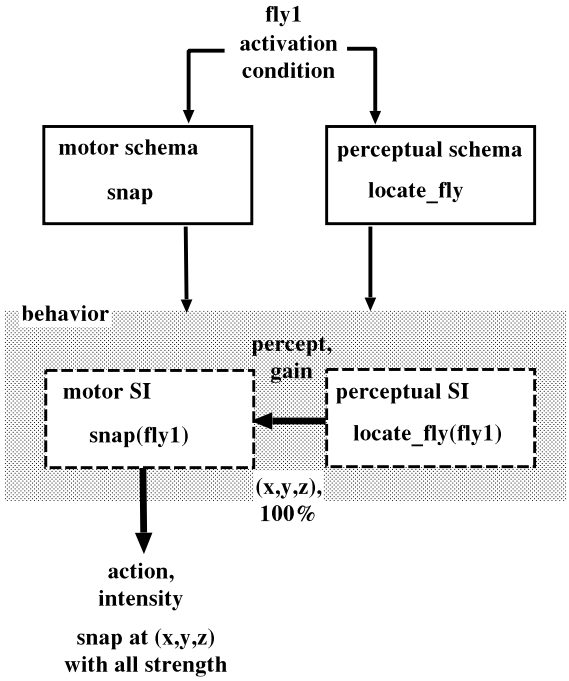


Figure 3.10 Schema theory of a frog snapping at a fly.

Schema theory does not specify how the output from concurrent behaviors is combined; that is a Level 3, or implementation, issue. Previous examples in this chapter have shown that in some circumstances the output is combined or summed, in others the behaviors would normally occur in a sequence and not overlap, and sometimes there would be a winner-take-all effect. The winner-take-all effect is a type of inhibition, where one behavior inhibits the instantiation of another behavior.

INHIBITION

Arbib and colleagues also modeled an instance of *inhibition* in frogs and toads.[7] Returning to the example of feeding and fleeing, one possible way to model this behavior is with two behaviors. The `feeding` behavior would consist of a motor schema for moving toward an object, with a perceptual schema for finding small, moving objects. The `fleeing` behavior would be similar only with a motor schema for moving away from the perception of large moving objects. Lesion studies with frogs showed something different. The `feeding` behavior actually consists of moving toward *any* moving object. So the perceptual schema is more general than anticipated. The frog would try to eat anything, including predators. The perceptual schema in the `fleeing` behavior detects large moving objects. It flees from them, but

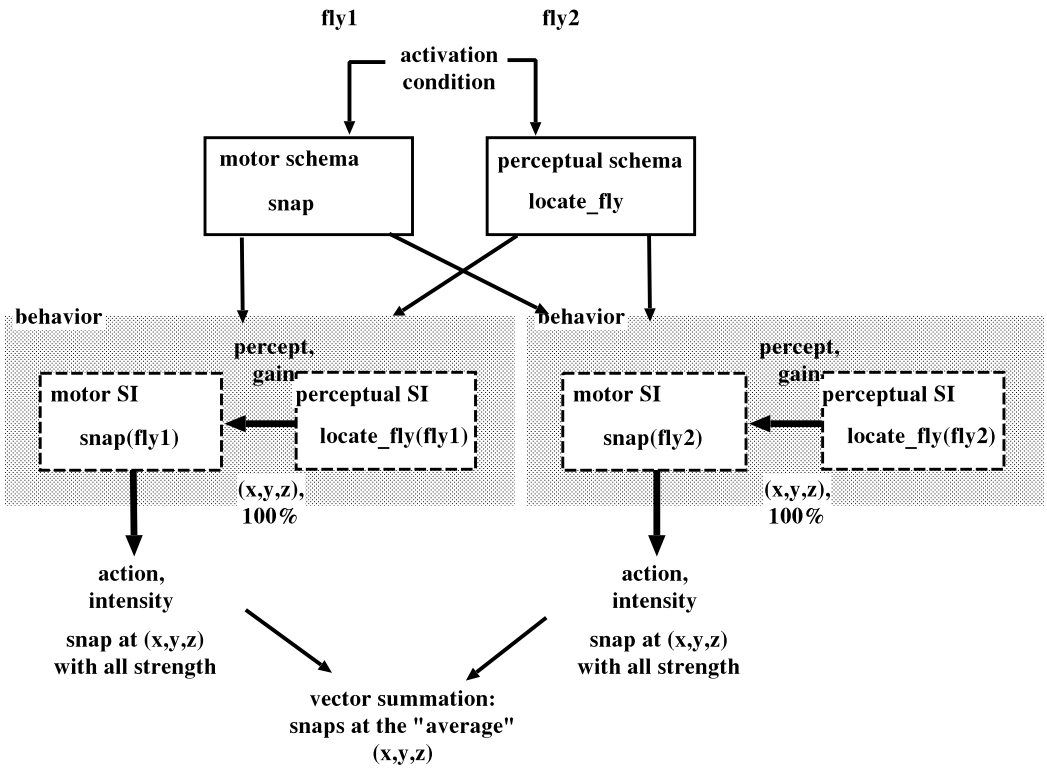


Figure 3.11 Schema theory of a frog snapping at a fly when presented with two flies equidistant.

it also inhibits the perceptual schema for feeding. As a result, the inhibition keeps the frog from trying to both flee from predators and eat them.

3.6 Principles and Issues in Transferring Insights to Robots

PRINCIPLES FOR PROGRAMMING

To summarize, some general principles of natural intelligence which may be useful in programming robots:

- Programs should decompose complex actions into independent behaviors, which tightly couple sensing and acting. Behaviors are inherently parallel and distributed.

- In order to simplify control and coordination of behaviors, an agent should rely on straightforward, boolean activation mechanisms (e.g. IRM).

- In order to simplify sensing, perception should filter sensing and consider only what is relevant to the behavior (action-oriented perception).
- Direct perception (affordances) reduces the computational complexity of sensing, and permits actions to occur without memory, inference, or interpretation.
- Behaviors are independent, but the output from one 1) may be combined with another to produce a resultant output, or 2) may serve to inhibit another (competing-cooperating).

Unfortunately, studying natural intelligence does not give a complete picture of how intelligence works. In particular there are several unresolved issues:

UNRESOLVED ISSUES

- *How to resolve conflicts between concurrent behaviors?* Robots will be required to perform concurrent tasks; for example, a rescue robot sent in to evacuate a building will have to navigate hallways while looking for rooms to examine for people, as well as look for signs of a spreading fire. Should the designer specify dominant behaviors? Combine? Let conflicting behaviors cancel and have alternative behavior triggered? Indeed, one of the biggest divisions in robot architectures is how they handle concurrent behaviors.
- *When are explicit knowledge representations and memory necessary?* Direct perception is wonderful in theory, but can a designer be sure that an affordance has not been missed?
- *How to set up and/or learn new sequences of behaviors?* Learning appears to be a fundamental component of generating complex behaviors in advanced animals. However, the ethological and cognitive literature is unsure of the mechanisms for learning.

It is also important to remember that natural intelligence does not map perfectly onto the needs and realities of programming robots. One major advantage that animal intelligence has over robotic intelligence is evolution. Animals evolved in a way that leads to survival of the species. But robots are expensive and only a small number are built at any given time. Therefore, individual robots must "survive," not species. This puts tremendous pressure on robot designers to get a design right the first time. The lack of evolutionary pressures over long periods of time makes robots extremely vulnerable to design errors introduced by a poor understanding of the robot's ecology.

Ch. 5 will provide a case study of a robot which was programmed to follow white lines in a path-following competition by using the affordance of white. It was distracted off course by the white shoes of a judge. Fortunately that design flaw was compensated for when the robot got back on course by reacting to a row of white dandelions in seed.

Robots introduce other challenges not so critical in animals. One of the most problematic attributes of the Reactive Paradigm, Ch. 4, is that roboticists have no real mechanism for completely predicting emergent behaviors. Since a psychologist can't predict with perfect certainty what a human will do under a stressful situation, it seems reasonable that a roboticist using principles of human intelligence wouldn't be able to predict what a robot would do either. However, robotics end-users (military, NASA, nuclear industry) have been reluctant to accept robots without a guarantee of what it will do in critical situations.

3.7 Summary

A behavior is the fundamental element of biological intelligence, and will serve as the fundamental component of intelligence in most robot systems.

BEHAVIOR

A *behavior* is defined as a mapping of sensory inputs to a pattern of motor actions which then are used to achieve a task. Innate Releasing Mechanisms are one model of how intelligence is organized. IRMs model intelligence at Level 2 of a computational theory, describing the process but not the implementation. In IRM, releasers activate a behavior. A releaser may be either an internal state (motivation) and/or an environmental stimulus. Unfortunately, IRMs do not make the interactions between concurrent, or potentially concurrent, behaviors easy to identify or diagram.

Perception in behaviors serves two roles, either as a releaser for a behavior or as the percept which guides the behavior. The same percept can be used both as a releaser and a guide; for example, a fish can respond to a lure and follow it. In addition to the way in which perception is used, there appear to be two pathways for processing perception. The direct perception pathway uses affordances: perceivable potentialities for action inherent in the environment. Affordances are particularly attractive to roboticists because they can be extracted without inference, memory, or intermediate representations. The recognition pathway makes use of memory and global representations to identify and label specific things in the world.

Important principles which can be extracted from natural intelligence are:

- Agents programs should decompose complex actions into independent behaviors (or objects), which tightly couple sensing and acting. Behaviors are inherently parallel and distributed.
- In order to simplify control and coordination of behaviors, an agent should rely on straightforward, boolean activation mechanisms (e.g. IRM).
- In order to simplify sensing, perception should filter sensing and consider only what is relevant to the behavior (action-oriented perception).
- Direct perception (affordances) reduces the computational complexity of sensing, and permits actions to occur without memory, inference, or interpretation.
- Behaviors are independent, but the output from one 1) may be combined with another to produce a resultant output, or 2) may serve to inhibit another (competing-cooperating).

Schema theory is an object-oriented way of representing and thinking about behaviors. The important attributes of schema theory for behaviors are:

- Schema theory is used to represent behaviors in both animals and computers, and is sufficient to describe intelligence at the first two levels of a computational theory.
- A behavioral schema is composed of at least one motor schema and at least one perceptual schema, plus local, behavior-specific knowledge about how to coordinate multiple component schemas.
- More than one behavior schema can be instantiated at a time, but the schemas act independently.
- A behavior schema can have multiple instantiations which act independently, and are combined.
- Behaviors or schemas can be combined, sequenced, or inhibit one another.

3.8 Exercises

Exercise 3.1

Describe the three levels in a Computational Theory.

Exercise 3.2

Explain in one or two sentences each of the following terms: reflexes, taxes, fixed-action patterns, schema, affordance.

Exercise 3.3

Represent a schema, behavior, perceptual schema, and motor schema with an Object-Oriented Design class diagram.

Exercise 3.4

Many mammals exhibit a camouflage meta-behavior. The animal freezes when it sees motion (an affordance for a predator) in an attempt to become invisible. It persists until the predator is very close, then the animal flees. (This explains why squirrels freeze in front of cars, then suddenly dash away, apparently flinging themselves under the wheels of a car.) Write pseudo-code of the behaviors involved in the camouflage behavior in terms of innate releasing mechanisms, identifying the releasers for each behavior.

Exercise 3.5

Consider a mosquito hunting for a warm-blooded mammal and a good place to bite them. Identify the affordance for a warm-blooded mammal and the associated behavior. Represent this with schema theory (perceptual and motor schemas).

Exercise 3.6

One method for representing the IRM logic is to use finite state automata (FSA), which are commonly used in computer science. If you have seen FSAs, consider a FSA where the behaviors are states and releasers serve as the transitions between state. Express the sequence of behaviors in a female digger wasp as a FSA.

Exercise 3.7

Lego Mindstorms and Rug Warrior kits contain sensors and actuators which are coupled together in reflexive behaviors. Build robots which:

a. Reflexive avoid: turn left when they touch something (use touch sensor and two motors)

b. Phototaxis: follow a black line (use the IR sensor to detect the difference between light and dark)

c. Fixed-action pattern avoid: back up and turn right when robot encounters a "negative obstacle" (a cliff)

Exercise 3.8

What is the difference between direct perception and recognition?

Exercise 3.9

Consider a cockroach, which typically hides when the lights are turned on. Do you think the cockroach is using direct perception or recognition of a hiding place? Explain why. What are the percepts for the cockroach?

Exercise 3.10

Describe how cancellation could happen as a result of concurrency and incomplete FSA.

Exercise 3.11 [*Advanced Reading*]

Read the first four chapters in Braitenberg's *Vehicles*.[25] Write a 2-5 page paper:

a. List and describe the principles of behaviors for robotics in Ch. 3.

b. Discuss how *Vehicles* is consistent with the biological foundations of reactivity. Be specific, citing which vehicle illustrates what principle or attribute discussed in the book.

c. Discuss any flaws in the reasoning or inconsistency between *Vehicles* with the biological foundations of reactivity or computer science.

Exercise 3.12 [*Advanced Reading*]

Read "Sensorimotor transformations in the worlds of frogs and robots," by Arbib and Liaw.[7]

a. List and describe how the principles of schema theory and potential fields for robotics given in Ch. 3.

b. Summarize the main contributions of the paper.

3.9 End Notes

For the roboticist's bookshelf.
Valentino Braitenberg's *Vehicles: Experiments in Synthetic Psychology*[25] is the cult classic of AI roboticists everywhere. It doesn't require any hardware or programming experience, just a couple hours of time and an average imagination to experience this radical departure from the mainstream robotics of the 1970's.

About David Marr.
David Marr's idea of a computational theory was an offshoot of his work bridging the gap between vision from a neurophysiological perspective (his) and computer vision. As is discussed in his book, *Vision*,[88] Marr had come from England to work in the MIT AI lab on computer vision. The book represented his three years there, and he finished it while literally on his deathbed with leukemia. His preface to the book is heartbreaking.

A Brief History of Cognitive Science.
Howard Gardner's *The Mind's New Science*[56] gives a nice readable overview of cognitive psychology. He conveys a bit of the controversy Gibson's work caused.

J.J. and E.J. Gibson.
While J.J. Gibson is very well-known, his wife Jackie (E.J. Gibson) is also a prominent cognitive psychologist. They met when he began teaching at Smith College, where she was a student. She raised a family, finished a PhD, and publishes well-respected studies on learning. At least two of the Gibson's students followed their husband-wife teaming: Herb Pick was a student of J.J. Gibson, while his wife, Anne Pick, was a student of E.J. Gibson. The Picks are at the University of Minnesota, and Herb Pick has been active in the mobile robotics community.

Susan Calvin, robopsychologist.
Isaac Asimov's robot stories often feature Dr. Susan Calvin, the first robopsychologist. In the stories, Calvin is often the only person who can figure out the complex interactions of concurrent behaviors leading to a robot's emergent misbehavior. In some regards, Calvin is the embarrassing Cold War stereotype of a woman scientist: severe, unmarried, too focused on work to be able to make small talk.

4 *The Reactive Paradigm*

Chapter Objectives:

- Define what the reactive paradigm is in terms of i) the three primitives SENSE, PLAN, and ACT, and ii) sensing organization.
- List the characteristics of a reactive robotic system, and discuss the connotations surrounding the reactive paradigm.
- Describe the two dominant methods for combining behaviors in a reactive architecture: *subsumption* and *potential field summation*.
- Evaluate subsumption and potential fields architectures in terms of: *support for modularity, niche targetability, ease of portability to other domains, robustness*.
- Be able to program a behavior using a potential field methodology.
- Be able to construct a new potential field from primitive potential fields, and sum potential fields to generate an emergent behavior.

4.1 Overview

This chapter will concentrate on an overview of the reactive paradigm and two representative architectures. The Reactive Paradigm emerged in the late 1980's. The Reactive Paradigm is important to study for at least two reasons. First, robotic systems in limited task domains are still being constructed using reactive architectures. Second, the Reactive Paradigm will form the basis for the Hybrid Reactive-Deliberative Paradigm; everything covered here will be used (and expanded on) by the systems in Ch. 7.

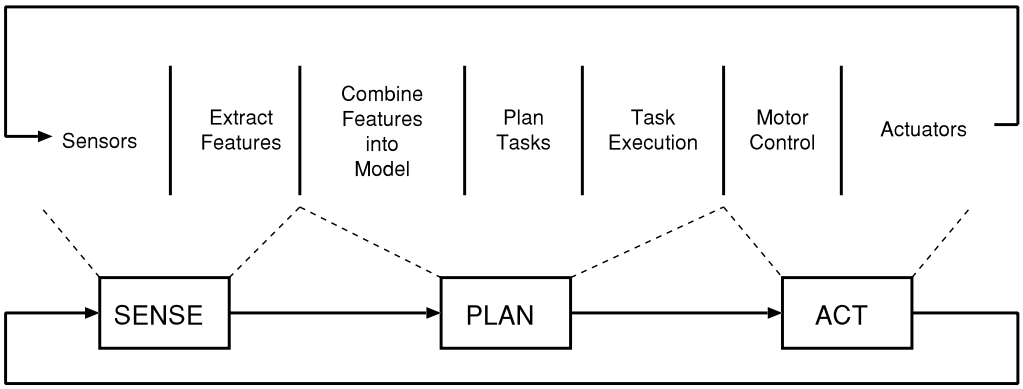


Figure 4.1 Horizontal decomposition of tasks into the S,P,A organization of the Hierarchical Paradigm.

The Reactive Paradigm grew out of dissatisfaction with the hierarchical paradigm and with an influx of ideas from ethology. Although various reactive systems may or may not strictly adhere to principles of biological intelligence, they generally mimic some aspect of biology. The dissatisfaction with the Hierarchical Paradigm was best summarized by Rodney Brooks,[27] who characterized those systems as having a *horizontal decomposition* as shown in Fig. 4.1.

HORIZONTAL DECOMPOSITION

Instead, an examination of the ethological literature suggests that intelligence is layered in a *vertical decomposition*, shown in Fig. 4.2. Under a vertical decomposition, an agent starts with primitive survival behaviors and evolves new layers of behaviors which either reuse the lower, older behaviors, inhibit the older behaviors, or create parallel tracks of more advanced behaviors. The parallel tracks can be thought of layers, stacked vertically. Each layer has access to sensors and actuators independently of any other layers. If anything happens to an advanced behavior, the lower level behaviors would still operate. This return to a lower level mimics degradation of autonomous functions in the brain. Functions in the brain stem (such as breathing) continue independently of higher order functions (such as counting, face recognition, task planning), allowing a person who has brain damage from a car wreck to still breathe, etc.

VERTICAL DECOMPOSITION

Work by Arkin, Brooks, and Payton focused on defining behaviors and on mechanisms for correctly handling situations when multiple behaviors are active simultaneously. Brooks took an approach now known as subsumption and built insect-like robots with behaviors captured in hardware circuitry.

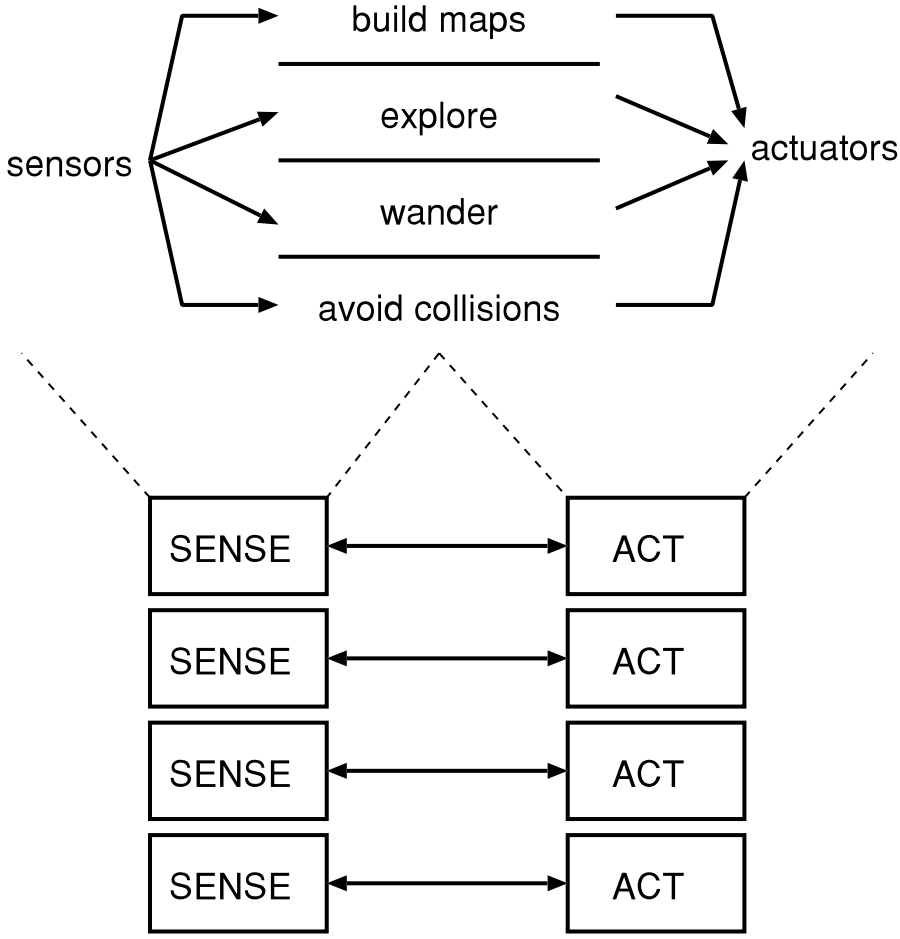


Figure 4.2 Vertical decomposition of tasks into an S-A organization, associated with the Reactive Paradigm.

Arkin and Payton used a potential fields methodology, favoring software implementations. Both approaches are equivalent. The Reactive Paradigm was initially met with stiff resistance from traditional customers of robotics, particularly the military and nuclear regulatory agencies. These users of robotic technologies were uncomfortable with the imprecise way in which discrete behaviors combine to form a rich emergent behavior. In particular, reactive behaviors are not amenable to mathematical proofs showing they are sufficient and correct for a task. In the end, the rapid execution times associated with the reflexive behaviors led to its acceptance among users, just as researchers shifted to the Hybrid paradigm in order to fully explore layering of intelligence.

The major theme of this chapter is that all reactive systems are composed of behaviors, though the meaning of a behavior may be slightly different in each reactive architecture. Behaviors can execute concurrently and/or sequentially. The two representative architectures, subsumption and potential fields, are compared and contrasted using the same task as an example. This chapter will concentrate on how architecture handles concurrent behaviors to produce an emergent behavior, deferring sequencing to the next chapter.

4.2 Attributes of Reactive Paradigm

BEHAVIORS

The fundamental attribute of the reactive paradigm is that all actions are accomplished through behaviors. As in ethological systems, *behaviors are a direct mapping of sensory inputs to a pattern of motor actions that are then used to achieve a task.* From a mathematical perspective, behaviors are simply a transfer function, transforming sensory inputs into actuator commands. For the purposes of this book, a behavior will be treated as a schema, and will consist of at least one motor schema and one perceptual schema. The motor schema contains the algorithm for generating the pattern of action in a physical actuator and the perceptual schema contains the algorithm for extracting the percept and its strength. Keep in mind that few reactive robot architectures describe their behaviors in terms of schemas. But in practice, most behavioral implementations have recognizable motor and perceptual routines, even though they are rarely referred to as schemas.

The Reactive Paradigm literally threw away the PLAN component of the SENSE, PLAN, ACT triad, as shown in Fig. 4.3. The SENSE and ACT components are tightly coupled into behaviors, and all robotic activities emerge as the result of these behaviors operating either in sequence or concurrently. The S-A organization does not specify how the behaviors are coordinated and controlled; this is an important topic addressed by architectures.

SENSE-ACT ORGANIZATION

BEHAVIOR-SPECIFIC (LOCAL) SENSING

Sensing in the Reactive Paradigm is local to each behavior, or behavior-specific. Each behavior has its own dedicated sensing. In many cases, this is implemented as one sensor and perceptual schema per behavior. But in other cases, more than one behavior can take the same output from a sensor and process it differently (via the behavior's perceptual schema). One behavior literally does not know what another behavior is doing or perceiving. Fig. 4.4 graphically shows the sensing style of the Reactive Paradigm.

Note that this is fundamentally opposite of the global world model used in the hierarchical paradigm. Sensing is immediately available to the be-

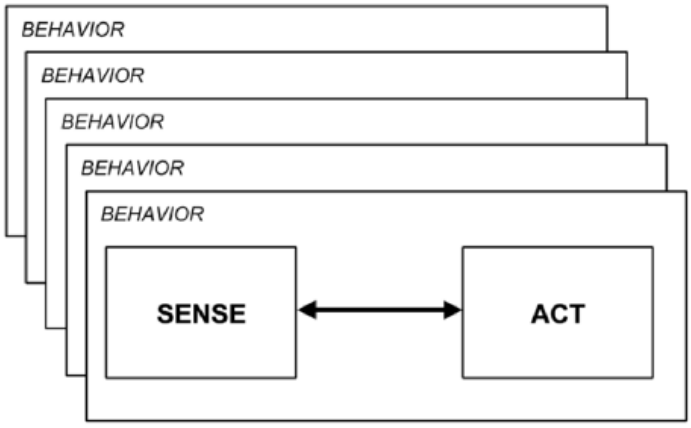


Figure 4.3 S-A organization of the Reactive Paradigm into multiple, concurrent behaviors.

havior's perceptual schema, which can do as little or as much processing as needed to extract the relevant percept. If a computationally inexpensive affordance is used, then the sensing portion of the behavior is nearly instantaneous and action is very rapid.

As can be seen from the previous chapter on the biological foundations of the reactive paradigm, behaviors favor the use of affordances. In fact, Brooks was fond of saying (loudly) at conferences, "we don't need no stinking representations." It should be noted that often the perceptual schema portion of the behavior has to use a behavior-specific representation or data structure to substitute for specialized detectors capable of extracting affordances. For example, extracting a red region in an image is non-trivial with a computer compared with an animal seeing red. The point is that while a computer program may have to have data structures in order to duplicate a simple neural function, the behavior does not rely on any central representation built up from all sensors.

In early implementations of the reactive paradigm, the idea of "one sensor, one behavior" worked well. For more advanced behaviors, it became useful to fuse the output of multiple sensors within one perceptual schema to have increased precision or a better measure of the strength of the stimulus. This type of sensor fusion is permitted within the reactive paradigm as long as the fusion is local to the behavior. Sensor fusion will be detailed in Ch. 6.

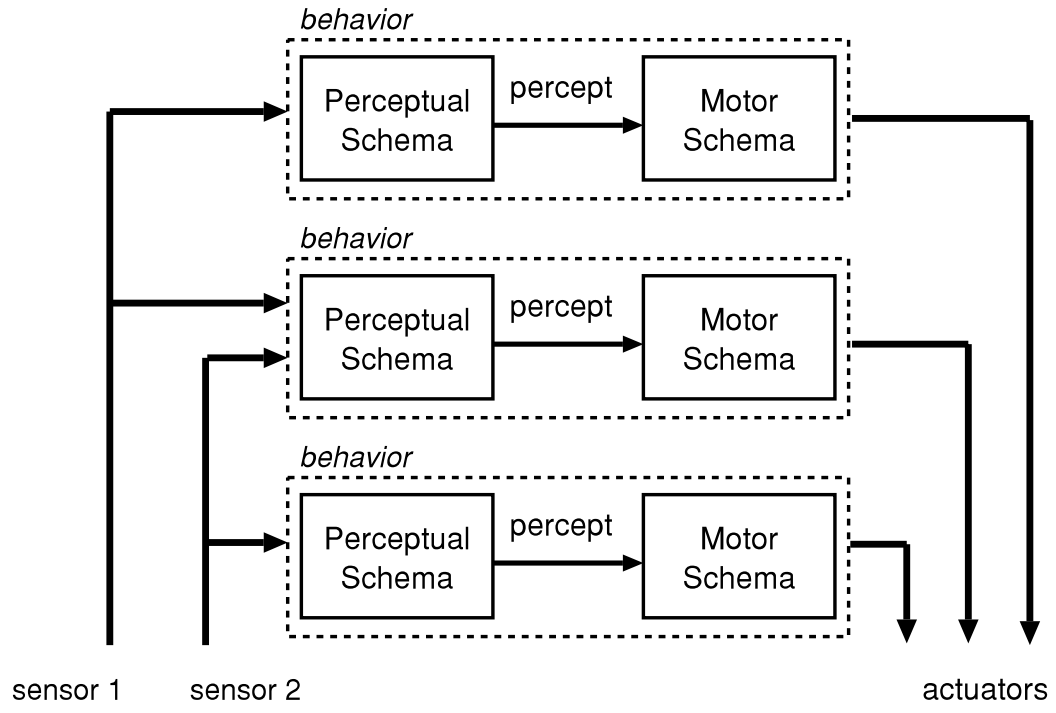


Figure 4.4 Behavior-specific sensing organization in the Reactive Paradigm: sensing is local, sensors can be shared, and sensors can be fused locally by a behavior.

4.2.1 Characteristics and connotations of reactive behaviors

As seen earlier, a reactive robotic system decomposes functionality into behaviors, which tightly couple perception to action without the use of intervening abstract (global) representations. This is a broad, vague definition. Over the years, the reactive paradigm has acquired several connotations and characteristics from the way practitioners have used the paradigm.

The primary connotation of a reactive robotic system is that it executes rapidly. The tight coupling of sensing and acting permits robots to operate in real-time, moving at speeds of 1-2 cm per second. Behaviors can be implemented directly in hardware as circuits, or with low computational complexity algorithms (O(n)). This means that behaviors execute quickly regardless of the processor. Behaviors execute not only fast in their own right, they are particularly fast when compared to the execution times of Shakey and the Stanford Cart. A secondary connotation is that reactive robotic systems have no memory, limiting reactive behaviors to what biologists would call pure stimulus-response reflexes. In practice, many behaviors exhibit a

fixed-action pattern type of response, where the behavior persists for a short period of time without the direct presence of the stimulus. The main point is that behaviors are controlled by what is happening in the world, duplicating the spirit of innate releasing mechanisms, rather than by the program storing and remembering what the robot did last. The examples in the chapter emphasize this point.

The five characteristics of almost all architectures that follow the Reactive Paradigm are:

SITUATED AGENT

1. *Robots are situated agents operating in an ecological niche.* As seen earlier in Part I, *situated agent* means that the robot is an integral part of the world. A robot has its own goals and intentions. When a robot acts, it changes the world, and receives immediate feedback about the world through sensing. What the robot senses affects its goals and how it attempts to meet them, generating a new cycle of actions. Notice that situatedness is defined by Neisser's Action-Perception Cycle. Likewise, the goals of a robot, the world it operates in, and how it can perceive the world form the ecological niche of the robot. To emphasize this, many robotic researchers say they are working on *ecological robotics*.

ECOLOGICAL ROBOTICS

2. *Behaviors serve as the basic building blocks for robotic actions, and the overall behavior of the robot is emergent.* Behaviors are independent, computational entities and operate concurrently. The overall behavior is emergent: there is no explicit "controller" module which determines what will be done, or functions which call other functions. There may be a coordinated control program in the schema of a behavior, but there is no external controller of all behaviors for a task. As with animals, the "intelligence" of the robot is in the eye of the beholder, rather than in a specific section of code. Since the overall behavior of a reactive robot emerges from the way its individual behaviors interact, the major differences between reactive architectures is usually the specific mechanism for interaction. Recall from Chapter 3 that these mechanisms include combination, suppression, and cancellation.

EGO-CENTRIC

3. *Only local, behavior-specific sensing is permitted.* The use of explicit abstract representational knowledge in perceptual processing, even though it is behavior-specific, is avoided. Any sensing which does require representation is expressed in *ego-centric* (robot-centric) coordinates. For example, consider obstacle avoidance. An ego-centric representation means that it does not matter that an obstacle is in the world at coordinates (x,y,z), only

where it is relative to the robot. Sensor data, with the exception of GPS, is inherently ego-centric (e.g., a range finder returns a distance to the nearest object from the transducer), so this eliminates unnecessary processing to create a world model, then extract the position of obstacles relative to the robot.

4. *These systems inherently follow good software design principles.* The modularity of these behaviors supports the decomposition of a task into component behaviors. The behaviors are tested independently, and behaviors may be assembled from primitive behaviors.

5. *Animal models of behavior are often cited as a basis for these systems or a particular behavior.* Unlike in the early days of AI robotics, where there was a conscious effort to not mimic biological intelligence, it is very acceptable under the reactive paradigm to use animals as a motivation for a collection of behaviors.

4.2.2 Advantages of programming by behavior

Constructing a robotic system under the Reactive Paradigm is often referred to as programming by behavior, since the fundamental component of any implementation is a behavior. Programming by behavior has a number of advantages, most of them consistent with good software engineering principles. Behaviors are inherently modular and easy to test in isolation from the system (i.e., they support unit testing). Behaviors also support incremental expansion of the capabilities of a robot. A robot becomes more intelligent by having more behaviors. The behavioral decomposition results in an implementation that works in real-time and is usually computationally inexpensive. Although we'll see that sometimes duplicating specialized detectors (like optic flow) is slow. If the behaviors are implemented poorly, then a reactive implementation can be slow. But generally, the reaction speeds of a reactive robot are equivalent to stimulus-response times in animals.

LOW COUPLING
HIGH COHESION

Behaviors support good software engineering principles through decomposition, modularity and incremental testing. If programmed with as high a degree of independence (also called *low coupling*) as possible, and *high cohesion*, the designer can build up libraries of easy to understand, maintain, and reuse modules that minimize side effects. Low coupling means that the modules can function independently of each other with minimal connections or interfaces, promoting easy reuse. Cohesion means that the data and operations contained by a module relate only to the purpose of that module.

Higher cohesion is associated with modules that do one thing well, like the SQRT function in C. The examples in Sec. 4.3 and 4.4 attempt to illustrate the choices a designer has in engineering the behavioral software of a robot.

4.2.3 Representative architectures

In order to implement a reactive system, the designer must identify the set of behaviors necessary for the task. The behaviors can either be new or use existing behaviors. The overall action of the robot emerges from multiple, concurrent behaviors. Therefore a reactive architecture must provide mechanisms for 1) triggering behaviors and 2) for determining what happens when multiple behaviors are active at the same time. Another distinguishing feature between reactive architectures is how they define a behavior and any special use of terminology. Keep in mind that the definitions presented in Sec. 4.2 are a generalization of the trends in reactive systems, and do not necessarily have counterparts in all architectures.

There are many architectures which fit in the Reactive Paradigm. The two best known and most formalized are the subsumption and potential field methodologies. Subsumption refers to how behaviors are combined. Potential Field Methodologies require behaviors to be implemented as potential fields, and the behaviors are combined by summation of the fields. A third style of reactive architecture which is popular in Europe and Japan is *rule encoding*, where the motor schema component of behaviors and the combination mechanism are implemented as logical rules. The rules for combining behaviors are often ad hoc, and so will not be covered in this book. Other methods for combining behaviors exist, including fuzzy methods and winner-take-all voting, but these tend to be implementation details rather than an over-arching architecture.

RULE ENCODING

4.3 Subsumption Architecture

Rodney Brooks' subsumption architecture is the most influential of the purely Reactive Paradigm systems. Part of the influence stems from the publicity surrounding the very naturalistic robots built with subsumption. As seen in Fig. 4.5, these robots actually looked like shoe-box sized insects, with six legs and antennae. In many implementations, the behaviors are embedded directly in the hardware or on small micro-processors, allowing the robots to have all on-board computing (this was unheard of in the processor-impoverished mid-1980's). Furthermore, the robots were the first to be able

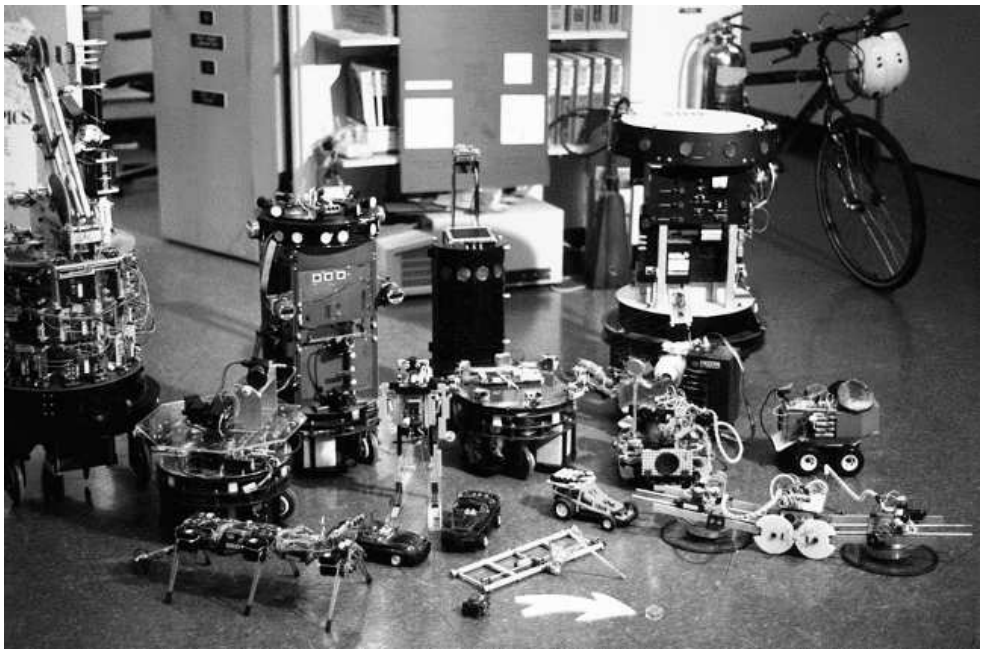


Figure 4.5 "Veteran" robots of the MIT AI Laboratory using the subsumption architecture. (Photograph courtesy of the MIT Artificial Intelligence Laboratory.)

to walk, avoid collisions, and climb over obstacles without the "move-think-move-think" pauses of Shakey.

The term "behavior" in the subsumption architecture has a less precise meaning than in other architectures. A behavior is a network of sensing and acting modules which accomplish a task. The modules are augmented finite state machines AFSM, or finite state machines which have registers, timers, and other enhancements to permit them to be interfaced with other modules. An AFSM is equivalent to the interface between the schemas and the coordinated control strategy in a behavioral schema. In terms of schema theory, a subsumption behavior is actually a collection of one or more schemas into an abstract behavior.

Behaviors are released in a stimulus-response way, without an external program explicitly coordinating and controlling them. Four interesting aspects of subsumption in terms of releasing and control are:

LAYERS OF COMPETENCE

1. Modules are grouped into *layers of competence*. The layers reflect a hierarchy of intelligence, or competence. Lower layers encapsulate basic survival functions such as avoiding collisions, while higher levels create

more goal-directed actions such as mapping. Each of the layers can be viewed as an abstract behavior for a particular task.

LAYERS CAN SUBSUME LOWER LAYERS

2. Modules in a higher layer can override, or subsume, the output from behaviors in the next lower layer. The behavioral layers operate concurrently and independently, so there needs to be a mechanism to handle potential conflicts. The solution in subsumption is a type of winner-take-all, where the winner is always the higher layer.

NO INTERNAL STATE

3. The use of internal state is avoided. Internal state in this case means any type of local, persistent representation which represents the state of the world, or a model. Because the robot is a situated agent, most of its information should come directly from the world. If the robot depends on an internal representation, what it believes may begin to dangerously diverge from reality. Some internal state is needed for releasing behaviors like being scared or hungry, but good behavioral designs minimize this.

TASKABLE

4. A task is accomplished by activating the appropriate layer, which then activates the lower layers below it, and so on. However, in practice, subsumption style systems are not easily *taskable*, that is, they can't be ordered to do another task without being reprogrammed.

4.3.1 Example

These aspects are best illustrated by an example, extensively modified from Brooks' original paper[27] in order to be consistent with schema theory terminology and to facilitate comparison with a potential fields methodology. A robot capable of moving forward while not colliding with anything could be represented with a single layer, Level 0. In this example, the robot has multiple sonars (or other range sensors), each pointing in a different direction, and two actuators, one for driving forward and one for turning.

LEVEL 0: AVOID

Following Fig. 4.6, the `SONAR` module reads the sonar ranges, does any filtering of noise, and produces a *polar plot*. A polar plot represents the range readings in polar coordinates, (r, θ), surrounding the robot. As shown in Fig. 4.7, the polar plot can be "unwound."

POLAR PLOT

If the range reading for the sonar facing dead ahead is below a certain threshold, the `COLLIDE` module declares a collision and sends the halt signal to the `FORWARD` drive actuator. If the robot was moving forward, it now stops. Meanwhile, the `FEELFORCE` module is receiving the same polar plot. It treats each sonar reading as a repulsive force, which can be represented

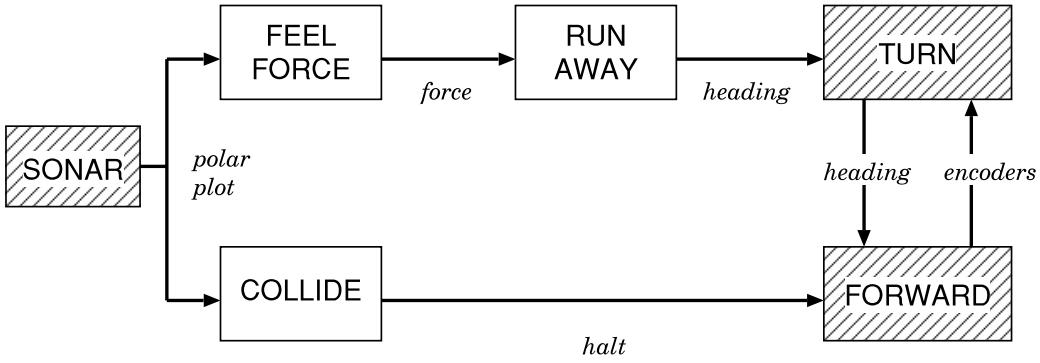


Figure 4.6 Level 0 in the subsumption architecture.

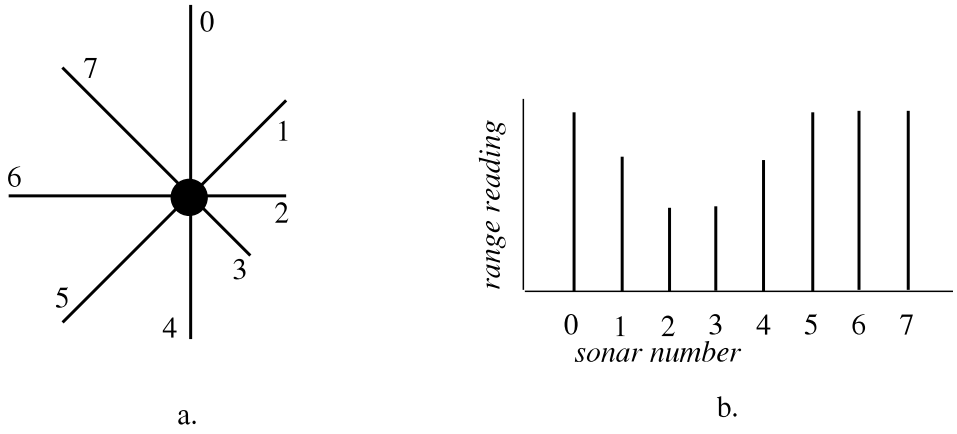


Figure 4.7 Polar plot of eight sonar range readings: a.) "robo-centric" view of range readings along acoustic axes, and b.) unrolled into a plot.

as a vector. Recall that a vector is a mathematical construct that consists of a magnitude and a direction. `FEELFORCE` can be thought of as summing the vectors from each of the sonar readings. This results in a new vector. The repulsive vector is then passed to the `TURN` module. The `TURN` module splits off the direction to turn and passes that to the steering actuators. `TURN` also passes the vector to the `FORWARD` module, which uses the magnitude of the vector to determine the magnitude of the next forward motion (how far or how fast). So the robot turns and moves a short distance away from the obstacle.

The observable behavior is that the robot will sit still if it is in an unoccupied space, until an obstacle comes near it. If the obstacle is on one side of

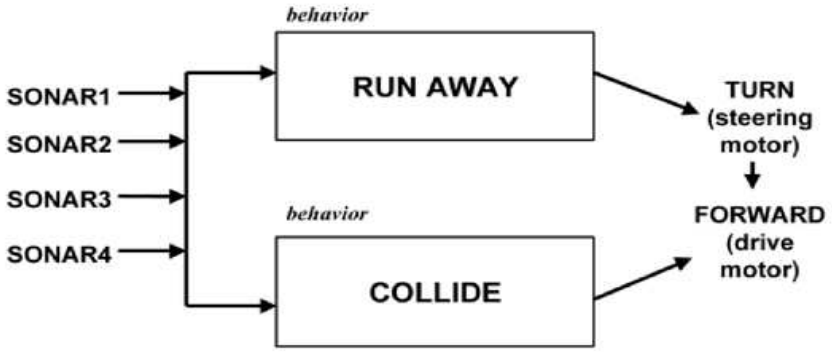


Figure 4.8 Level 0 recast as primitive behaviors.

the robot, the robot will turn 180° the other way and move forward; essentially, it runs away. This allows a person to herd the robot around. The robot can react to an obstacle if the obstacle (or robot) is motionless or moving; the response is computed at each sensor update.

However, if part of the obstacle, or another obstacle, is dead ahead (someone tries to herd the robot into a wall), the robot will stop, then apply the results of `RUNAWAY`. So it will stop, turn and begin to move forward again. Stopping prevents the robot from side-swiping the obstacle while it is turning and moving forward. Level 0 shows how a fairly complex set of actions can emerge from very simple modules.

It is helpful to recast the subsumption architecture in the terms used in this book, as shown in Fig. 4.8. Note how this looks like the vertical decomposition in Fig. 4.2: the sensor data flows through the concurrent behaviors to the actuators, and the independent behaviors cause the robot to do the right thing. The `SONAR` module would be considered a global interface to the sensors, while the `TURN` and `FORWARD` modules would be considered part of the actuators (an interface). For the purposes of this book, a behavior must consist of a perceptual schema and a motor schema. Perceptual schemas are connected to a sensor, while motor schemas are connected to actuators. For Level 0, the perceptual schemas would be contained in the `FEELFORCE` and `COLLIDE` modules. The motor schemas are `RUNAWAY` and `COLLIDE` modules. COLLIDE combines both perceptual processing (extracts the vector for the sonar facing directly ahead) and the pattern of action (halt if there is a

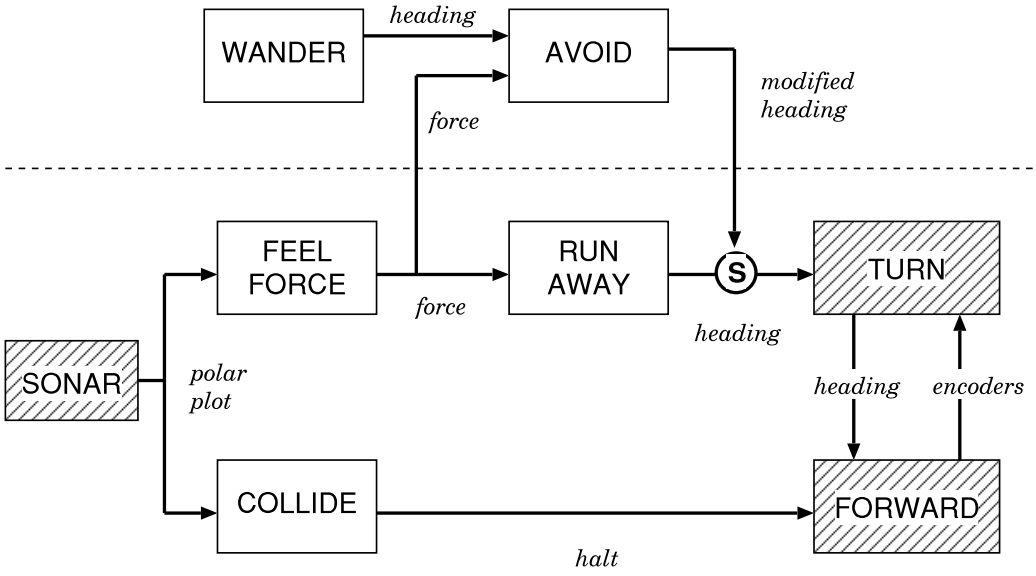


Figure 4.9 Level 1: wander.

reading). The primitive behaviors reflect the two paths through the layer. One might be called the runaway behavior and the other the collide behavior. Together, the two behaviors create a rich obstacle avoidance behavior, or a layer of competence.

It should also be noticed that the behaviors used direct perception, or affordances. The presence of a range reading indicated there was an obstacle; the robot did not have to know what the obstacle was.

LEVEL 1: WANDER

Consider building a robot which actually wandered around instead of sitting motionless, but was still able to avoid obstacles. Under subsumption, a second layer of competence (Level 1) would be added, shown in Fig. 4.9. In this case, Level 1 consists of a `WANDER` module which computes a random heading every n seconds. The random heading can be thought of as a vector. It needs to pass this heading to the `TURN` and `FORWARD` modules. But it can't be passed to the `TURN` module directly. That would sacrifice obstacle avoidance, because `TURN` only accepts one input. One solution is to add another module in Level 1, `AVOID`, which combines the `FEELFORCE` vector with the `WANDER` vector. Adding a new avoid module offers an opportunity to create a more sophisticated response to obstacles. `AVOID` combines the direction of the force of avoidance with the desired heading. This results in the actual heading being mostly in the right direction rather than having the robot turn

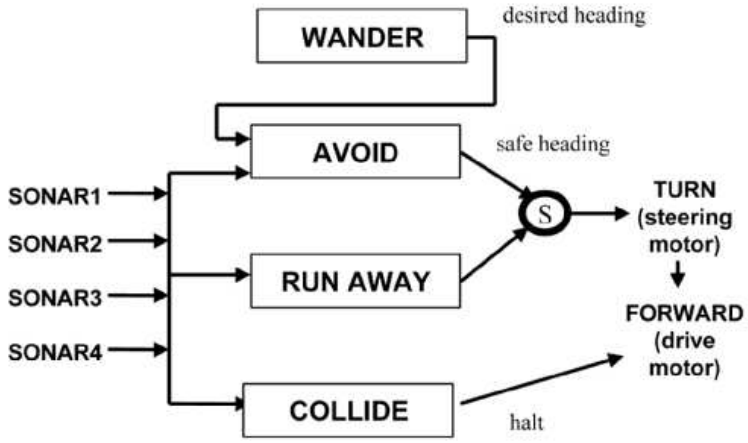


Figure 4.10 Level 1 recast as primitive behaviors.

around and lose forward progress. (Notice also that the AVOID module was able to "eavesdrop" on components of the next lower layer.) The heading output from AVOID has the same representation as the output of RUNAWAY, so TURN can accept from either source.

The issue now appears to be when to accept the heading vector from which layer. Subsumption makes it simple: the output from the higher level subsumes the output from the lower level. Subsumption is done in one of two ways:

INHIBITION

1. *inhibition*. In inhibition, the output of the subsuming module is connected to the output of another module. If the output of the subsuming module is "on" or has any value, the output of the subsumed module is blocked or turned "off." Inhibition acts like a faucet, turning an output stream on and off.

SUPPRESSION

2. *suppression*. In suppression, the output of of the subsuming module is connected to the input of another module. If the output of the subsuming module is on, it *replaces* the normal input to the subsumed module. Suppression acts like a switch, swapping one input stream for another.

In this case, the AVOID module suppresses (marked in the diagram with a S) the output from RUNAWAY. RUNAWAY is still executing, but its output doesn't go anywhere. Instead, the output from AVOID goes to TURN.

The use of layers and subsumption allows new layers to be built on top of less competent layers, without modifying the lower layers. This is good software engineering, facilitating modularity and simplifying testing. It also adds some robustness in that if something should disable the Level 1 behaviors, Level 0 might remain intact. The robot would at least be able to preserve its self-defense mechanism of fleeing from approaching obstacles.

Fig. 4.10 shows Level 1 recast as behaviors. Note that FEELFORCE was used by both RUNAWAY and AVOID. FEELFORCE is the perceptual component (or schema) of both behaviors, with the AVOID and RUNAWAY modules being the motor component (or schema). As is often the case, behaviors are usually named after the observable action. This means that the behavior (which consists of perception and action) and the action component have the same name. The figure does not show that the AVOID and RUNAWAY behaviors share the same FEELFORCE perceptual schema. As will be seen in the next chapter, the object-oriented properties of schema theory facilitate the reuse and sharing of perceptual and motor components.

LEVEL 2: FOLLOW CORRIDORS

Now consider adding a third layer to permit the robot to move down corridors, as shown in Fig. 4.11. (The third layer in Brooks' original paper is "explore," because he was considering a mapping task.) The LOOK module examines the sonar polar plot and identifies a corridor. (Note that this is another example of behaviors sharing the same sensor data but using it locally for different purposes.) Because identifying a corridor is more computationally expensive than just extracting range data, LOOK may take longer to run than behaviors at lower levels. LOOK passes the vector representing the direction to the middle of the corridor to the STAYINMIDDLE module. STAYINMIDDLE subsumes the WANDER module and delivers its output to the AVOID module which can then swerve around obstacles.

But how does the robot get back on course if the LOOK module has not computed a new direction? In this case, the INTEGRATE module has been observing the robots actual motions from shaft encoders in the actuators. This gives an estimate of how far off course the robot has traveled since the last update by LOOK. STAYINMIDDLE can use the dead reckoning data with the intended course to compute the new course vector. It serves to fill in the gaps in mismatches between updates rates of the different modules. Notice that LOOK and STAYINMIDDLE are quite sophisticated from a software perspective.

INTEGRATE is an example of a module which is supplying a dangerous internal state: it is actually substituting for feedback from the real world. If for some reason, the LOOK module never updates, then the robot could op-

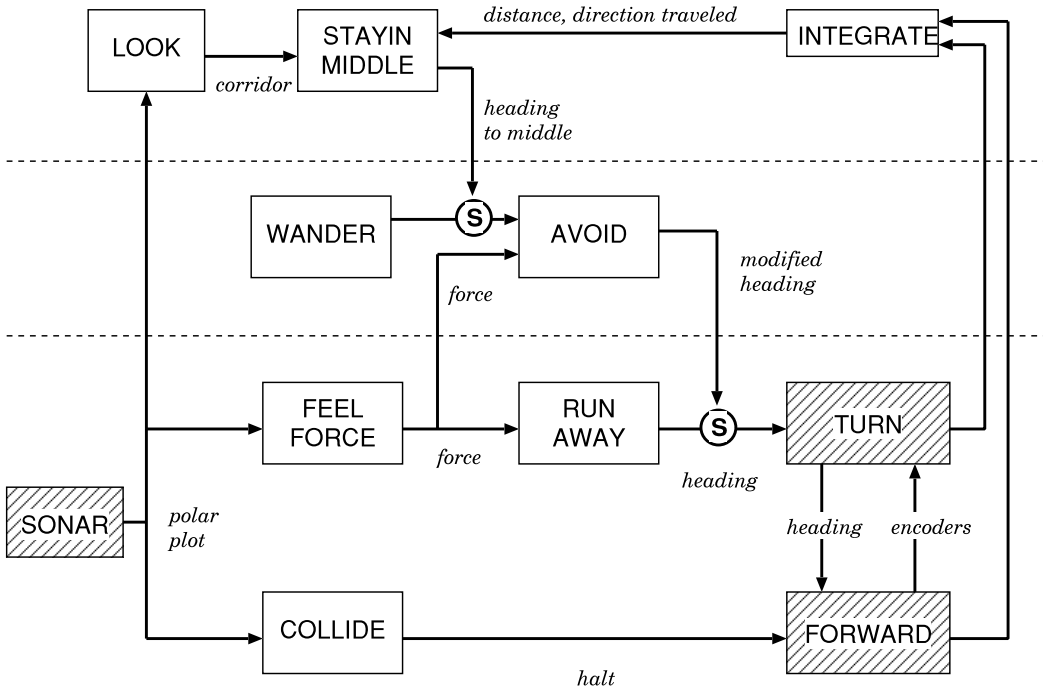


Figure 4.11 Level 2: follow corridors.

erate without any sensor data forever. Or at least until it crashed! Therefore, subsumption style systems include time constants on suppression and inhibition. If the suppression from `STAYINMIDDLE` ran for longer than n seconds with out a new update, the suppression would cease. The robot would then begin to wander, and hopefully whatever problem (like the corridor being totally blocked) that had led to the loss of signal would fix itself.

Of course, a new problem is how does the robot know that it hasn't started going down the hallway it just came up? Answer: it doesn't. The design assumes that that a corridor will always be present in the robot's ecological niche. If it's not, the robot does not behave as intended. This is an example of the connotation that reactive systems are "memory-less."

4.3.2 Subsumption summary

To summarize subsumption:

- Subsumption has a loose definition of behavior as a tight coupling of sensing and acting. Although it is not a schema-theoretic architecture, it can

be described in those terms. It groups schema-like modules into layers of competence, or abstract behaviors.

- Higher layers may subsume and inhibit behaviors in lower layers, but behaviors in lower layers are never rewritten or replaced. From a programming standpoint, this may seem strange. However, it mimics biological evolution. Recall that the fleeing behavior in frogs (Ch. 3) was actually the result of two behaviors, one which always moved toward moving objects and the other which actually suppressed that behavior when the object was large.

- The design of layers and component behaviors for a subsumption implementation, as with all behavioral design, is hard; it is more of an art than a science. This is also true for all reactive architectures.

- There is nothing resembling a STRIPS-like plan in subsumption. Instead, behaviors are released by the presence of stimulus in the environment.

- Subsumption solves the frame problem by eliminating the need to model the world. It also doesn't have to worry about the open world being non-monotonic and having some sort of truth maintenance mechanism, because the behaviors do not remember the past. There may be some perceptual persistence leading to a fixed-action pattern type of behavior (e.g., corridor following), but there is no mechanism which monitors for changes in the environment. The behaviors simply respond to whatever stimulus is in the environment.

- Perception is largely direct, using affordances. The releaser for a behavior is almost always the percept for guiding the motor schema.

- Perception is ego-centric and distributed. In the wander (layer 2) example, the sonar polar plot was relative to the robot. A new polar plot was created with each update of the sensors. The polar plot was also available to any process which needed it (shared global memory), allowing user modules to be distributed. Output from perceptual schemas can be shared with other layers.

4.4 Potential Fields Methodologies

Another style of reactive architecture is based on potential fields. The specific architectures that use some type of potential fields are too numerous to

VECTORS
VECTOR SUMMATION

describe here, so instead a generalization will be presented. Potential field styles of behaviors always use *vectors* to represent behaviors and *vector summation* to combine vectors from different behaviors to produce an emergent behavior.

4.4.1 Visualizing potential fields

The first tenet of a potential fields architecture is that the motor action of a behavior must be represented as a potential field. A potential field is an array, or field, of vectors. As described earlier, a vector is a mathematical construct which consists of a magnitude and a direction. Vectors are often used to represent a force of some sort. They are typically drawn as an arrow, where the length of the arrow is the magnitude of the force and the angle of the arrow is the direction. Vectors are usually represented with a boldface capital letter, for example, $\mathbf{V}$. A vector can also be written as a tuple (m, d), where m stands for magnitude and d for direction. By convention the magnitude is a real number between 0.0 and 1, but the magnitude can be any real number.

ARRAY REPRESENTING A FIELD

The array represents a region of space. In most robotic applications, the space is in two dimensions, representing a bird's eye view of the world just like a map. The map can be divided into squares, creating a (x,y) grid. Each element of the array represents a square of space. Perceivable objects in the world exert a force field on the surrounding space. The force field is analogous to a magnetic or gravitation field. The robot can be thought of as a particle that has entered the field exuded by an object or environment. The vector in each element represents the force, both the direction to turn and the magnitude or velocity to head in that direction, a robot would feel if it were at that particular spot. Potential fields are continuous because it doesn't matter how small the element is; at each point in space, there is an associated vector.

Fig. 4.12 shows how an obstacle would exert a field on the robot and make it run away. If the robot is close to the obstacle, say within 5 meters, it is inside the potential field and will fell a force that makes it want to face directly away from the obstacle (if it isn't already) and move away. If the robot is not within range of the obstacle, it just sits there because there is no force on it. Notice that the field represents what the robot should do (the motor schema) based on if the robot perceives an obstacle (the perceptual schema). The field isn't concerned with how the robot came to be so close to the obstacle; the robot feels the same force if it were happening to move within range or if it was just sitting there and someone put their hand next to the robot.

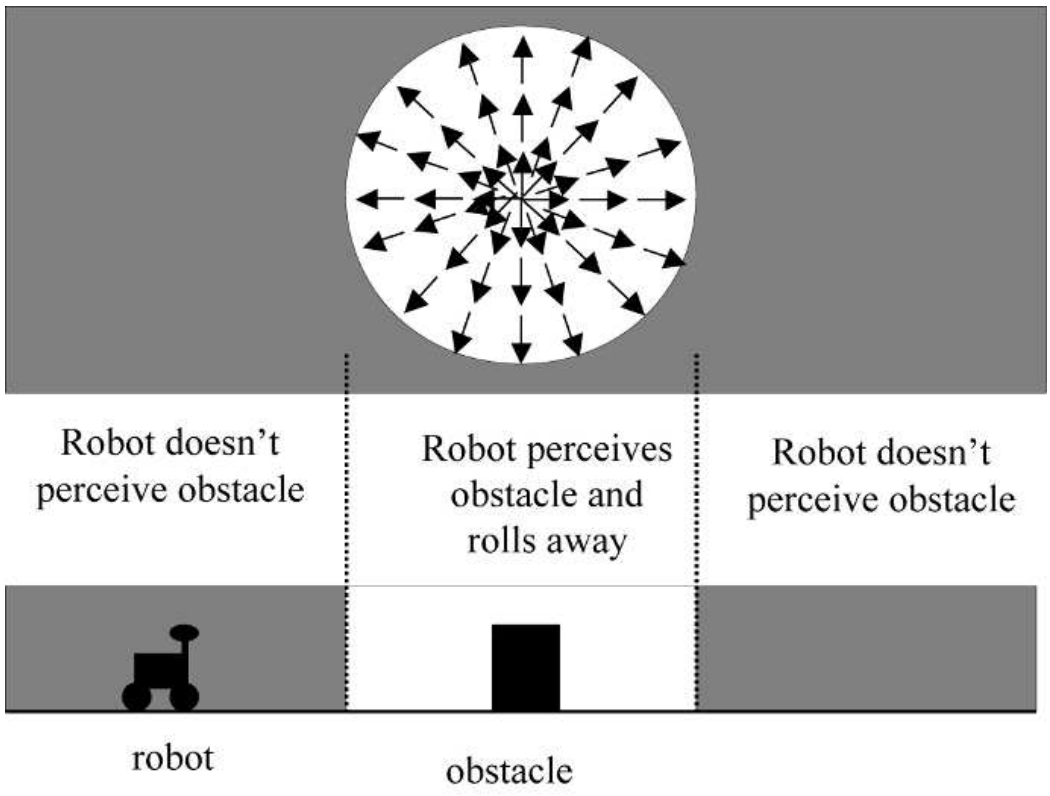


Figure 4.12 Example of an obstacle exerting a repulsive potential field over the radius of 1 meter.

One way of thinking about potential fields is to imagine a force field acting on the robot. Another way is to think of them as a potential energy surface in three dimensions (gravity is often represented this way) and the robot as a marble. In that case, the vector indicates the direction the robot would "roll" on the surface. Hills in the surface cause the robot to roll away or around (vectors would be pointing away from the "peak" of the hill), and valleys would cause the robot to roll downward (vectors pointing toward the bottom).

FIVE PRIMITIVE FIELDS

There are five basic potential fields, or primitives, which can be combined to build more complex fields: *uniform, perpendicular, attractive, repulsive,* and *tangential*. Fig. 4.13 shows a uniform field. In a uniform field, the robot would feel the same force no matter where it was. No matter where it got set down and at what orientation, it would feel a need to turn to align itself to the direction the arrow points and to move in that direction at a velocity propor-

UNIFORM FIELD

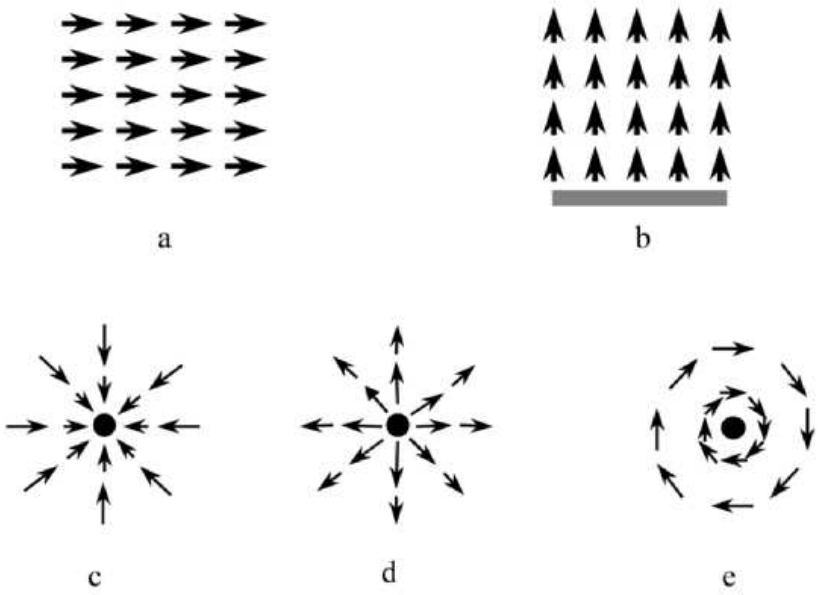


Figure 4.13 Five primitive potential fields: a.) uniform, b.) perpendicular, c.) attraction, d.) repulsion, and e.) tangential.

tional to the length of the arrow. A uniform field is often used to capture the behavior of "go in direction $n°$."

PERPENDICULAR FIELD

Fig. 4.13b shows a perpendicular field, where the robot is oriented perpendicular to some object or wall or border The field shown is directed away from the gray wall, but a perpendicular field can be pointed towards an object as well.

ATTRACTIVE FIELD

Fig. 4.13c illustrates an attractive field. The circle at the center of the field represents an object that is exerting an attraction on the robot. Wherever the robot is, the robot will "feel" a force relative to the object. Attractive fields are useful for representing a taxis or tropism, where the agent is literally attracted to light or food or a goal. The opposite of an attractive field is a repulsive field, shown in Fig. 4.13d. Repulsive fields are commonly associated with obstacles, or things the agent should avoid. The closer the robot is to the object, the stronger the repulsive force 180° away from it.

TANGENTIAL FIELD

The final primitive field is the tangential field in Fig. 4.13e. The field is a tangent around the object (think of a tangent vector as being perpendicular to

radial lines extending outward from the object). Tangential fields can "spin" either clockwise or counterclockwise; Fig. 4.13 shows a clockwise spin. They are useful for directing a robot to go around an obstacle, or having a robot investigate something.

4.4.2 Magnitude profiles

MAGNITUDE PROFILE

Notice that in Fig. 4.13, the length of the arrows gets smaller closer to the object. The way the magnitude of vectors in the field change is called the *magnitude profile*. (The term "magnitude profile" is used here because the term "velocity profile" is used by control engineers to describe how a robot's motors actually accelerate and decelerate to produce a particular movement without jerking.)

Consider the repulsive field in Fig. 4.12. Mathematically, the field can be represented with polar coordinates and the center of the field being the origin (0,0):

$$\begin{aligned} V_{direction} &= -\phi \\ V_{magnitude} &= c \end{aligned} \tag{4.1}$$

In that case, the magnitude was a constant value, c: the length of the arrows was the same. This can be visualized with a plot of the magnitude shown in Fig. 4.14a.

This profile says that the robot will run away (the direction it will run is $-\phi$) at the same velocity, no matter how close it is to the object, as long as it is in the range of the obstacle. As soon as the robot gets out of range of the obstacle, the velocity drops to 0.0, stopping the robot. The field is essentially binary: the robot is either running away at a constant speed or stopped. In practice there is a problem with a constant magnitude. It leads to jerky motion on the perimeter of the range of the field. This is illustrated when a robot is heading in a particular direction, then encounters an obstacle. It runs away, leaving the field almost immediately, and turns back to its original path, encounters the field again, and so on.

REFLEXIVITY

Magnitude profiles solve the problem of a constant magnitude. They also make it possible for a robot designer to represent reflexivity (that a response should be proportional to the strength of a stimulus) and to create interesting responses. Now consider the profile in Fig. 4.13c. It can be described as how

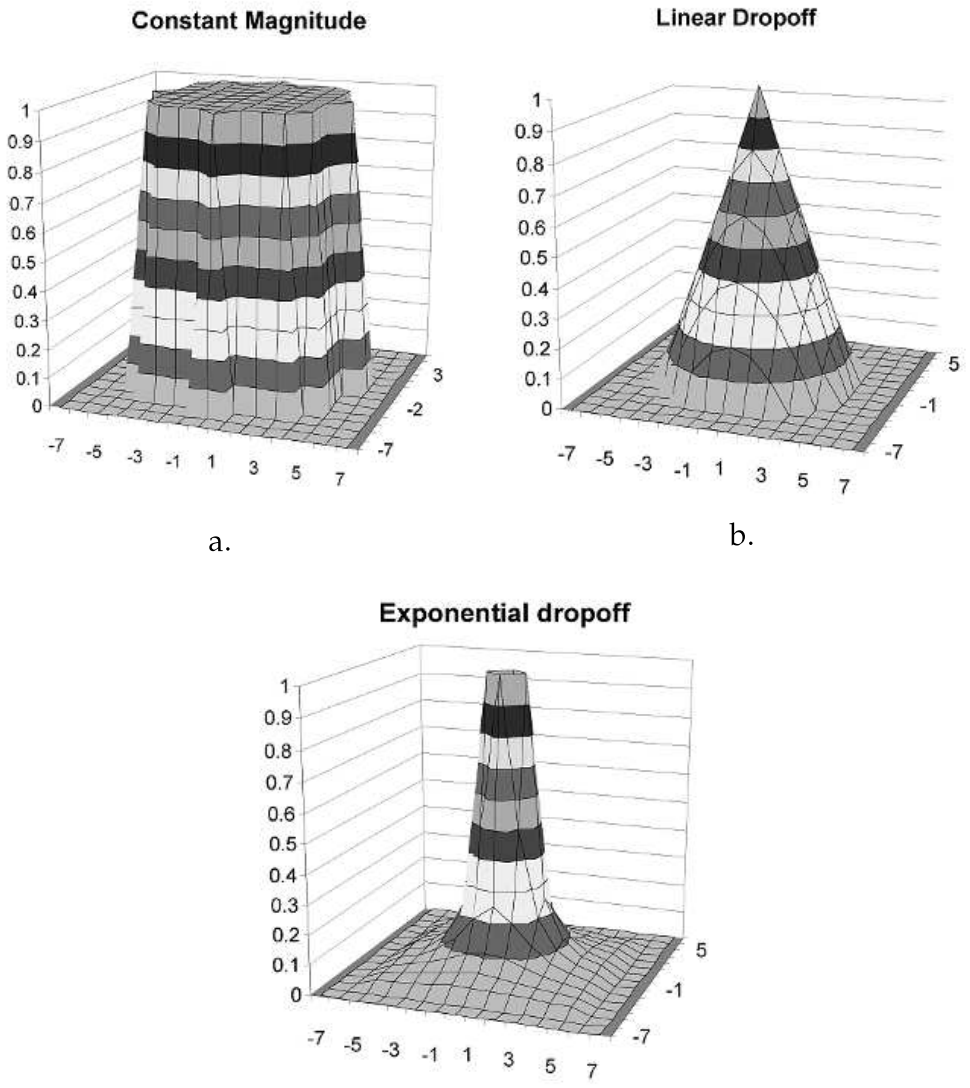


Figure 4.14 Plots of magnitude profiles for a field of radius 5 units: a.) constant magnitude, b.) linear drop off with a slope of -1, and c.) exponential drop off.

an observer would see a robot behave in that field: if the robot is far away from the object, it will turn and move quickly towards it, then slow up to keep from overshooting and hitting the object. Mathematically, this is called a *linear drop off*, since the rate at which the magnitude of the vectors drops off can be plotted as a straight line. The formula for a straight line is $y = mx + b$, where x is the distance and y is magnitude. b biases where the line starts, and m is the slope ($m = \frac{\Delta_y}{\Delta_x}$). Any value of m and b is acceptable. If it is not specified, $m = 1$ or -1 (a 45° slope up or down) and $b = 0$ in linear functions.

LINEAR DROP OFF

The linear profile in Fig. 4.14b matches the desired behavior of the designer: to have the robot react more, the closer it is. But it shares the problem of the constant magnitude profile in the sharp transition to 0.0 velocity. Therefore, another profile might be used to capture the need for a strong reaction but with more of a taper. One such profile is a *exponential drop off* function, where the drop off is proportional to the square of the distance: for every unit of distance away from the object, the force on the robot drops in half. The exponential profile is shown in Fig. 4.14c.

EXPONENTIAL DROP OFF

As can be seen from the previous examples, almost any magnitude profile is acceptable. The motivation for using magnitude profiles is to fine-tune the behavior. It is important to note that the robot only computes the vectors acting on it at its current location. The figures display the entire field for all possible locations of the robot. The question then arises as to why do the figures show an entire field over space? First, it aids visualizing what the robot will do overall, not just at one particular time step. Second, since fields are continuous representations, it simplifies confirming that the field is correct and makes any abrupt transitions readily apparent.

4.4.3 Potential fields and perception

In the previous examples, the force of the potential field at any given point was a function of both the relative distance between the robot and an object and the magnitude profile. The strength of a potential field can be a function of the stimulus, regardless of distance. As an example recall from Ch. 3 the feeding behavior of baby arctic terns where the feeding behavior is guided by the stimulus "red." This can be modeled by an attractive field. The bigger and redder an object in the baby's field of view, the stronger the attraction, suggesting that a magnitude profile using an increasing exponential function. Another important point that has already been mentioned is that potential fields are ego-centric because robot perception is ego-centric.

4.4.4 Programming a single potential field

Potential fields are actually easy to program, especially since the fields are ego-centric to the robot. The visualization of the entire field may appear to indicate that the robot and the objects are in a fixed, absolute coordinate system, but they are not. The robot computes the effect of the potential field, usually as a straight line, at every update, with no memory of where it was previously or where the robot has moved. This should become clear through the following examples.

A primitive potential field is usually represented by a single function. The vector impacting the robot is computed each update. Consider the case of a robot with a single range sensor facing forward. The designer has decided that a repulsive field with a linear drop off is appropriate. The formula is:

$$
\begin{aligned}
V_{direction} &= -180^{\circ} \\
V_{magnitude} &= \begin{cases} \frac{(D-d)}{D} & \text{for } d <= D \\ 0 & \text{for } d > D \end{cases}
\end{aligned} \tag{4.2}
$$

where D is the maximum range of the field's effect, or the maximum distance at which the robot can detect the obstacle. (D isn't always the detection range. It can be the range at which the robot should respond to a stimulus. For example, many sonars can detect obstacles 20 feet away, producing an almost infinitesimal response in emergent behavior but requiring the runtime overhead of a function call. In practice, a roboticist might set a D of 2 meters.) Notice that the formula produces a result where $0.0 \leq V_{magnitude} \leq 1.0$.

Below is a C code fragment that captures the repulsive field.

```
typedef struct {
   double   magnitude;
   double direction;
} vector;

vector repulsive(double d, double D)
{
   if (d <= D) {
     outputVector.direction = -180; //turn around!
     outputVector.magnitude = (D-d)/D; //linear dropoff
   }
   else {
```

```
      outputVector.direction=0.0
      outputVector.magnitude=0.0
   }
   return outputVector;
}
```

At this point, it is easy to illustrate how a potential field can be used by a behavior, `runaway`, for the robot with a single sensor. The `runaway` behavior will use the `repulsive()` function as the motor schema, and a function `readSonar()` as the perceptual schema. The output of the behavior is a vector. `runaway` is called by the robot on every update cycle.

```
vector runaway( ){
   double reading;
   reading=readSonar();//perceptual schema
   vector=repulsive (reading, MAX_DISTANCE); //motor schema
   return Voutput;
}
while (robot==ON)
{
   Vrunaway=runaway(reading); // motor schema
   turn(Vrunaway.direction);
   forward(Vrunaway.magnitude*MAX_VELOCITY);
}
```

4.4.5 Combination of fields and behaviors

As stated earlier in the chapter, the first attribute of a true potential fields methodology is that it requires all behaviors to be implemented as potential fields. The second attribute is that it combines behaviors not by one subsuming others, but by vector summation. A robot will generally have forces acting on it from multiple behaviors, all acting concurrently. This section provides two examples of how multiple behaviors arise and how they are implemented and combined.

The first example is simple navigation, where a robot is heading for a goal (specified as "10.3m in direction Θ") and encounters an obstacle. The motor

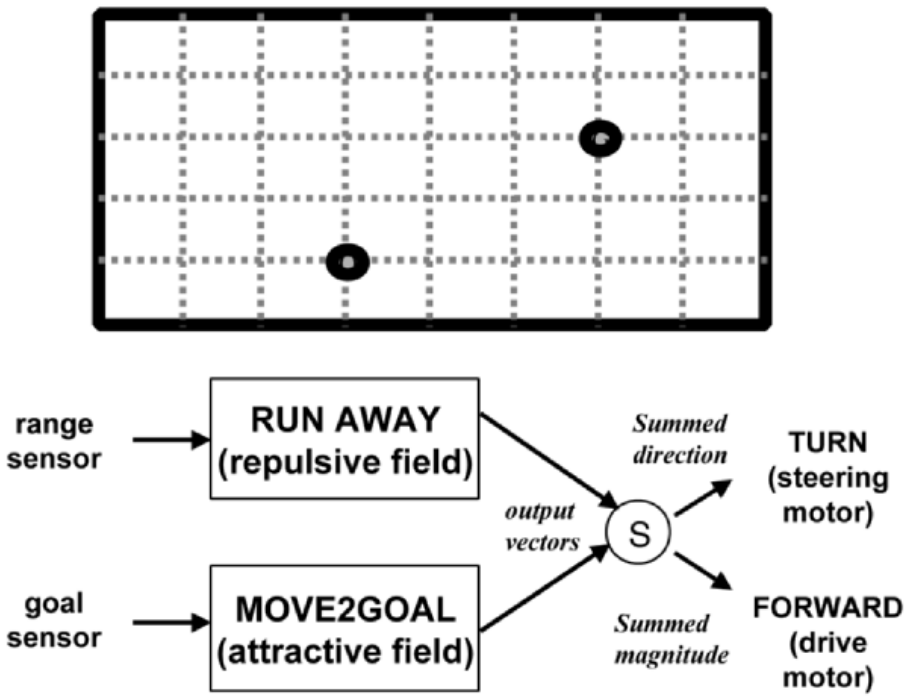


Figure 4.15 A bird's eye view of a world with a goal and obstacle, and the two active behaviors for the robot who will inhabit this world.

schema of the `move2goal` behavior is represented with an attractive potential field, which uses the shaft encoders on the robot to tell if it has reached the goal position. The `runaway` behavior is a repulsive field and uses a range sensor to detect if something is in front of it. Fig. 4.15 shows a bird's eye view of an area, and shows a visualization of the potential field. The `move2goal` behavior in Fig. 4.16b exerts an attractive field over the entire space; where ever the robot is, it will feel a force from the goal. The `runaway` behavior in Fig. 4.15 exerts a repulsive field in a radius around the obstacle (technically the repulsive field extends over all of the space as does the `move2goal`, but the magnitude of the repulsion is 0.0 beyond the radius). The combined field is shown in Fig. 4.16c.

Now consider the emergent behavior of the robot in the field if it starts in the lower right corner, shown in Fig. 4.17. At time t_0, the robot senses the world. It can only perceive the goal and cannot perceive the obstacle, so the only vector it feels is attraction (`runaway` returns a vector with magnitude

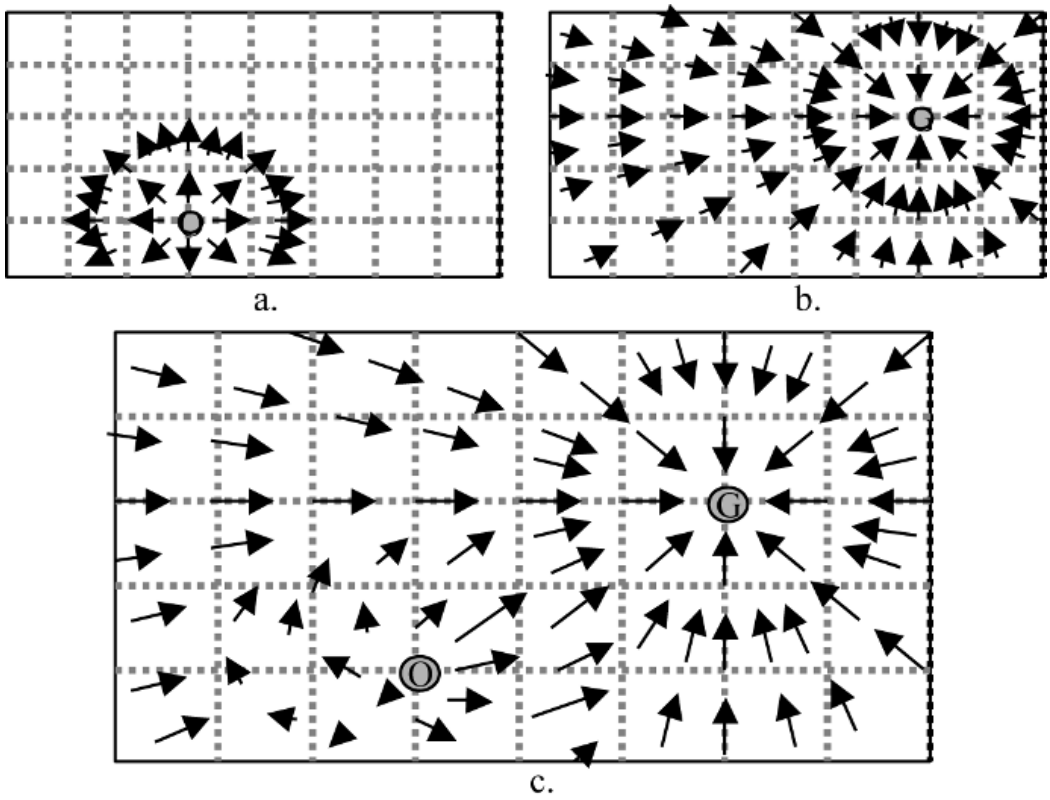


Figure 4.16 Potential fields from the world in Fig. 4.15: a.) repulsive from the obstacle, b.) attractive from the goal, and c.) combined.

of 0.0). It moves on a straight line for the goal. At t_2, it updates its sensors and now perceives both the goal and the obstacle. Both behaviors contribute a vector; the vectors are summed and the robot now moves off course. At t_3, the robot has almost moved beyond the obstacle and the goal is exerting the stronger force. At t_4, it resumes course and reaches the goal.

The example illustrates other points about potential fields methods: the impact of update rates, holonomicity, and local minima. Notice that the distance (length of the arrows) between updates is different. That is due to the changes in the magnitude of the output vector, which controls the robot velocity. If the robot has a "shorter" vector, it travels more slowly and, therefore, covers less distance in the same amount of time. It can also "overshoot" as seen between t_3 and t_4 where the robot actually goes farther without turning and has to turn back to go to the goal. As a result, the path is jagged with sharp lines. The resulting path will be smoother if the robot has a faster

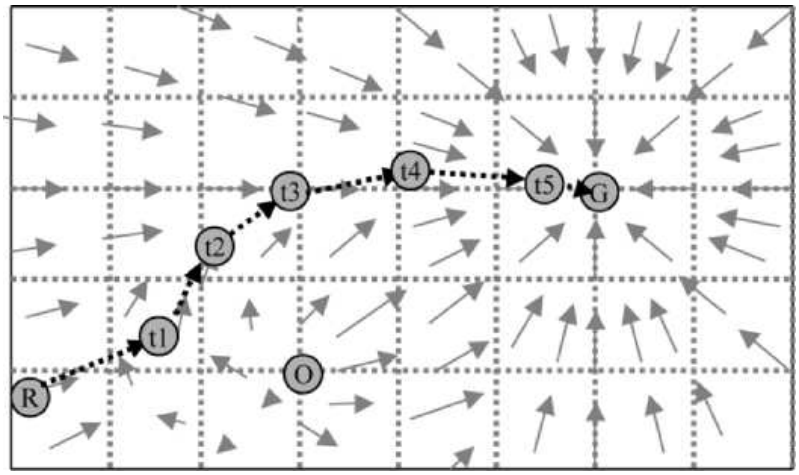


Figure 4.17 Path taken by the robot.

update interval. Another aspect of the update rate is that the robot can overshoot the goal, especially if it is using shaft encoders (the goal is 10.3 meters from where the robot started). Sometimes designers use attractive fields with a magnitude that drops off as the robot approaches, slowing it down so the that the robot can tell with it has reached the goal. (Programmers usually put a tolerance around the goal location, for example instead of 10.3m, the goal is 10.3m+/- 0.5m.)

Potential fields treat the robot as if it were a particle that could change velocity and direction instantaneously. This isn't true for real robots. Research robots such as Kheperas (shown in Fig. 4.18) can turn in any direction in place, but they have to be stopped and there is a measurable amount of error due to the contact between the wheels and the surface. Many robots have Ackerman, or automobile, steering, and anyone who has tried to parallel park an automobile knows that a car can go only in certain directions.

A third problem is that the fields may sum to 0.0. Returning to Fig. 4.16, draw a line between the Goal and the Obstacle. Along that line behind the Obstacle, the vectors have only a head (direction of the arrow) and no body (length of the arrow). This means that the magnitude is 0.0 and that if the robot reaches that spot, it will stop and not move again. This is called the local minima problem, because the potential field has a minima, or valley, that traps the robot. Solutions to the local minima problem will be described at the end of the chapter.

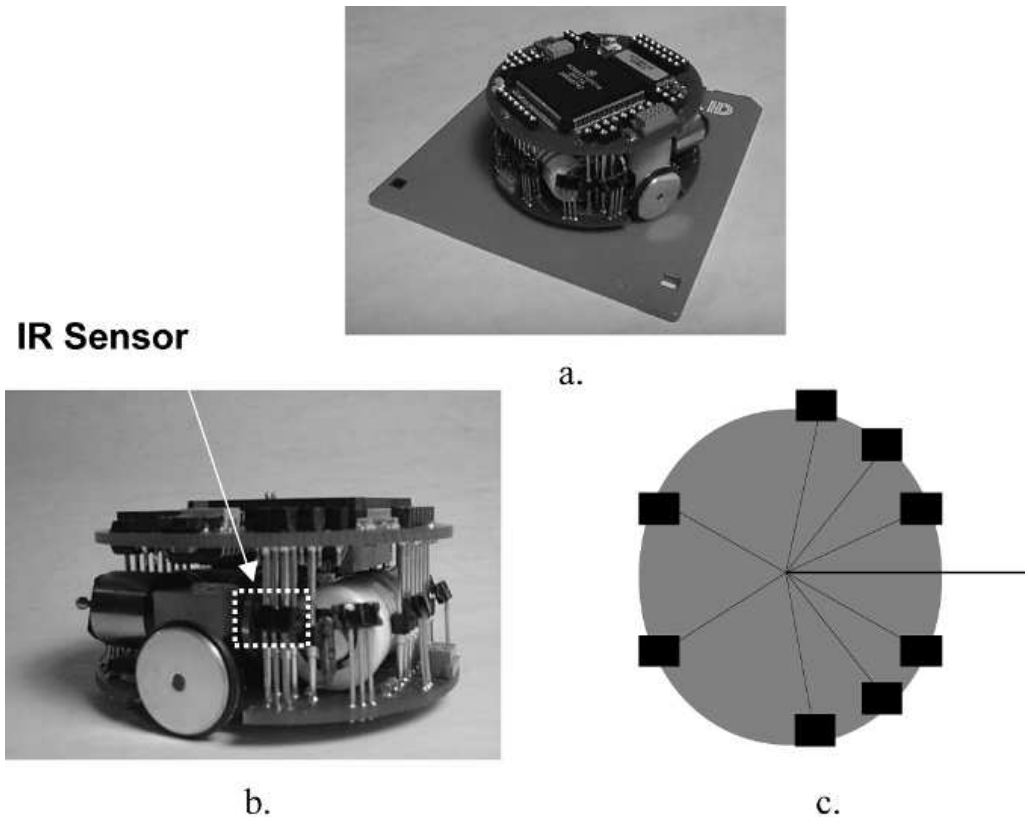


Figure 4.18 Khepera miniature robot: a.) a khepera on a floppy disk, b.) IR sensors (small black squares) along the "waistline," and c.) orientation of the IR sensors.

4.4.6 Example using one behavior per sensor

As another example of how powerful this idea of vector summation is, it is useful to consider how obstacle avoidance runaway is commonly implemented on real robots. Fig. 4.18 shows a layout of the IR range sensors on a Khepera robot. Since the sensors are permanently mounted on the platform, their angle α_i relative to the front is known. If a sensor receives a range reading, something is in front of that sensor. Under a repulsive field RUNAWAY, the output vector will be 180° opposite α_i. The IR sensor isn't capable of telling that the obstacle may be a little off the sensor axis, so a reading is treated as if an obstacle was straight in front of that sensor and perpendicular to it.

If this sensor were the only sensor on the robot, the `RUNAWAY` behavior is very straightforward. But what if, as in the case of a Khepera, the robot has

multiple range sensors? Bigger obstacles will be detected by multiple sensors at the same time. The common way is to have a RUNAWAY behavior for each sensor. This called multiple instantiations of the same behavior. Below is a code fragment showing multiple instantiations; all that had to be done is add a for loop to poll each sensor. This takes advantage of two properties of vector addition: it is associative (a+b+c+d can be performed as $((a+b)+c)+d$), and it is commutative (doesn't matter what order the vectors are summed).

```
while (robot==ON) {
   vector.mag=vector.dir=0.0; //initialize to 0
   for (i=0; i<=numberIR; i++) {
     vectorCurrent=Runaway(i); // accept a sensor number
     vectorOutput = VectorSum(tempVector,vectorCurrent);
   }
   turn(vector.direction);
   forward(vector.magnitude*MAX-VELOCITY);
}
```

BOX CANYON

As seen in Fig. 4.19, the robot is able to get out of the cave-like trap called a *box canyon* without building a model of the wall. Each instance contributes a vector, some of which have a X or Y component that cancels out.

From an ethological perspective, the above program is elegant because it is equivalent to behavioral instantiations in animals. Recall from Ch. 3 the model of *rana computrix* and its real-life toad counterpart where each eye sees and responds to a fly independently of the other eye. In this case, the program is treating the robot as if it had 8 independent eyes!

From a robotics standpoint, the example illustrates two important points. First, the direct coupling of sensing to action works. Second, behavioral programming is consistent with good software engineering practices. The RUNAWAY function exhibits *functional cohesion*, where the function does one thing well and every statement in the function has something directly to do with the function's purpose.[122] Functional cohesion is desirable, because it means the function is unlikely to introduce side effects into the main program or be dependent on another function. The overall organization shows *data coupling*, where each function call takes a simple argument.[122] Data coupling is good, because it means all the functions are independent; for example, the program can be easily changed to accommodate more IRs sensors.

FUNCTIONAL COHESION

DATA COUPLING

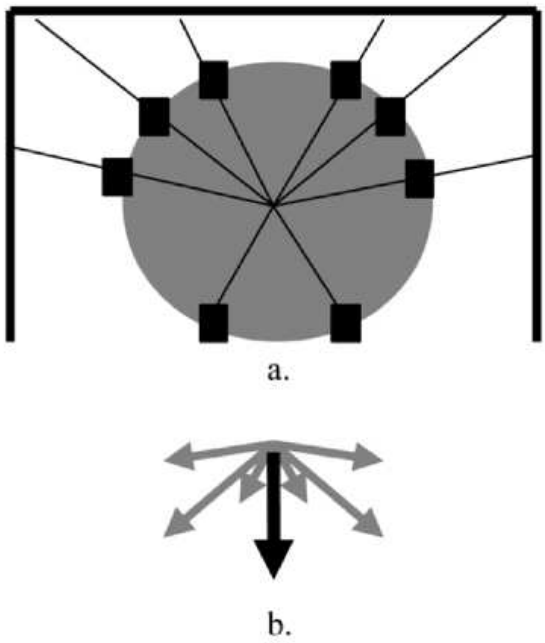


Figure 4.19 Khepera in a box canyon a.) range readings and b.) vectors from each instance of Runaway (gray) and the summed output vector.

The alternative to multiple instantiation is to have the perceptual schema for `RUNAWAY` process all 8 range readings. One approach is to sum all 8 vectors internally. (As an exercise show that the resulting vector is the same.) This is not as elegant from a software engineering perspective because the code is now specific to the robot (the function is said to have *procedural cohesion*),[122] and can be used only with a robot that has eight range sensors at those locations. Another approach, which produces a different emergent behavior, is to have the perceptual schema return the direction and distance of the single largest range reading. This makes the behavior more selective.

PROCEDURAL COHESION

4.4.7 Pfields compared with subsumption

How can simple behaviors form a more complicated emergent behavior? The same way more complicated potential fields are constructed from primitive fields: combining multiple instantiations of primitive behaviors. This can be

seen by revisiting the example behaviors used to describe the subsumption architecture. In the case of Level 0 in subsumption, if there are no obstacles within range, the robot feels no repulsive force and is motionless. If an obstacle comes within range and is detected by more than one sonar, each of the sonar readings create a vector, pointing the robot in the opposite direction. In the subsumption example, it could be imagined that these vectors were summed in the RUNAWAY module as shown in Fig. 4.20. In a potential fields system, each sonar reading would release an instance of the RUNAWAYpf behavior (the "pf" will be used to make it clear which runaway is being referred to). The RUNAWAYpf behavior uses a repulsive potential field. The output vectors would then be summed, and then the resultant vector would be used to guide the turn and forward motors.

```
while (robot==ON)
{
   vector.magnitude=vector.direction=0;
   for (i=0; i<=numberSonars; i++) {
      reading=readSonar();            //perceptual schema
      currentVector=runaway(reading); // motor schema
      vector = vectorSum(vector, currentVector);
   }
   turn(vector.direction);
   forward(vector.magnitude*MAX-VELOCITY);
}
```

The COLLIDE module in subsumption does not map over to a behavior in a potential fields methodology. Recall that the purpose of COLLIDE is to stop the robot if it touches an obstacle; in effect, if the RUNAWAY behavior has failed. This fits the definition of a behavior: it has a sensory input (range to obstacle = 0) and a recognizable pattern of motor activity (stop). But it doesn't produce a potential field, unless a uniform field of vectors with 0 magnitude is permissible. If it were treated as a behavior, the vector it contributes to would be summed with any other vectors contributed by other behaviors. But a vector with 0 magnitude is the identity function for vector addition, so a COLLISION vector would have no impact. Instead, collisions are often treated as "panic" situations, triggering an emergency response outside the potential field framework.

Some of the subtle differences between potential fields and subsumption appear when the case of Level 2 is considered. The same functionality can be accomplished by adding only a single instance of the WANDER behavior, as shown in Fig. 4.21. As before, the behavior generates a new direction

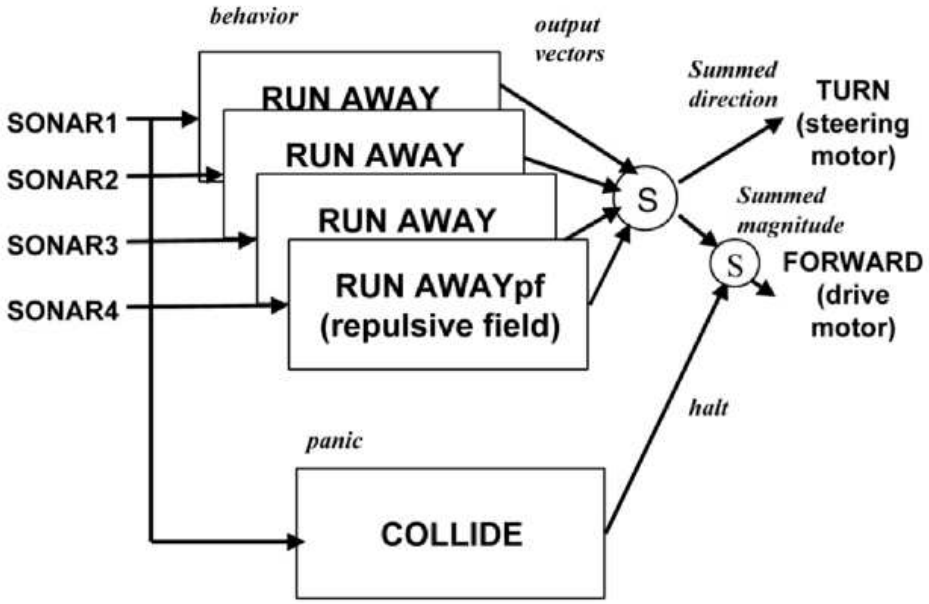


Figure 4.20 Level 0 redone as Potential Fields Methodology.

to move every n seconds. This would be represented by a uniform field where the robot felt the same attraction to go a certain direction, regardless of location, for n seconds. However, by combining the output of `WANDER` with the output vectors from `RUNAWAYpf`, the need for a new `AVOID` behavior is eliminated. The `WANDER` vector is summed with the repulsive vectors, and as a result, the robot moves both away from the obstacles and towards the desired direction. This is shown in Fig. 4.22. The primary differences in this example are that potential fields explicitly encapsulate sensing and acting into primitive behaviors, and it did not have to subsume any lower behaviors. As with subsumption, the robot became more intelligent when the `WANDERpf` behavior was added to the `RUNAWAYpf` behavior.

Now consider how Level 3, corridor following, would be implemented in a potential field system. This further illustrates the conceptual differences between the two approaches. The robot would have two concurrent behaviors: `RUNAWAYpf` and `follow-corridor`. `RUNAWAYpf` would remain the same as before, but `WANDER` would be discarded. In the parlance of potential fields, the task of following a corridor requires only two behaviors, while the task of wandering requires two different behaviors.

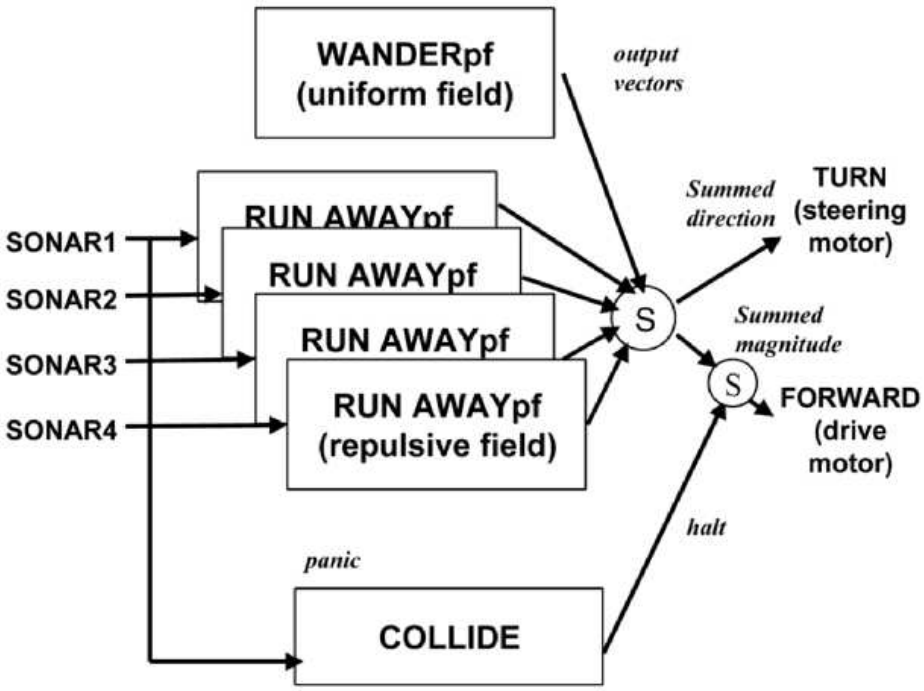


Figure 4.21 Level 1 redone with Potential Fields Methodology.

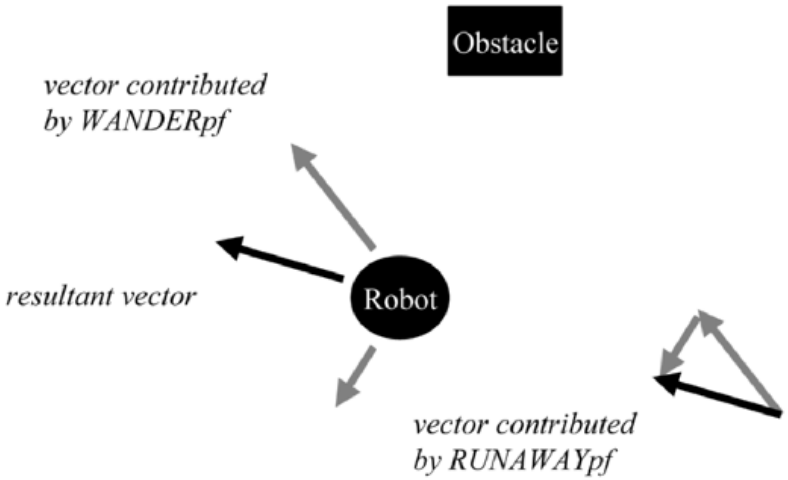


Figure 4.22 Example resultant vector of `WANDERpf` and `RUNAWAYpf`.

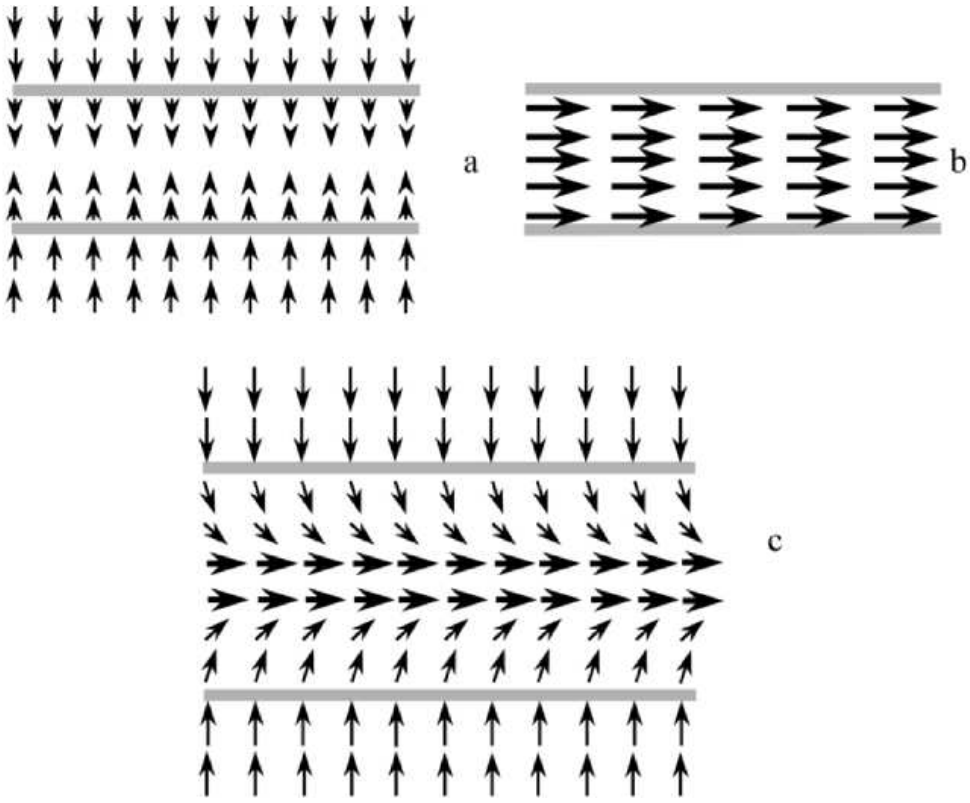


Figure 4.23 The a.) perpendical and b.) uniform fields combining into c.) a `follow-corridor` field.

The `follow-corridor` behavior is interesting, because it requires a more complex potential field. As shown in Fig. 4.23, it would be desirable for the robot to stay in the middle of the corridor. This can be accomplished using two potential fields: a uniform field perpendicular to the left boundary and pointing to the middle, and a uniform field perpendicular to the right boundary and pointing to the middle. Notice that both fields have a linear decrease in magnitude as the field nears the center of the corridor. In practice, this taper prevents the robot from see-sawing in the middle.

Also notice that the two uniform fields are not sufficient because they do not permit the robot to move forward; the robot would move to the middle of the corridor and just sit there. Therefore, a third uniform field is added which is parallel to the corridor. All three fields combined yield a smooth field which sharply pushes the robot back to the middle of the corridor as a function of its proximity to a wall. In the meantime, the robot is constantly making forward progress. The figure below shows the fields involved. Re-

member, in this example, the robot is not projecting past or future boundaries of the corridor; the visualization of the field makes it appear that way.

The `follow-corridor` behavior is using the same sonar data as `avoid`; therefore, walls will produce a repulsive field, which would generally push the robot onto the middle of the corridor. Why not just use a single uniform parallel field for `follow-corridor`? First, behaviors are independent. If there is a corridor following behavior, it must be able to follow halls without depending on side effects from other behaviors. Second, the polar symmetry of the repulsive fields may cause see-sawing, so there is a practical advantage to having a separate behavior.

The use of behavior-specific domain knowledge (supplied at instantiation time as an optional initialization parameter) can further improve the robot's overall behavior. If the robot knows the width of the hall *a priori*, `follow-corridor` can suppress instances of avoid which are using obstacles it decides form the boundary of the wall. Then it will only avoid obstacles that are in the hall. If there are no obstacles, `follow-corridor` will produce a smooth trajectory. If the obstacles are next to a wall, the `follow-corridor` will treat the profile of the obstacle as a wall and move closer to the center.

SEQUENCING AND PARAMETERIZING POTENTIAL FIELDS

The motor schemas for a behavior may be sequenced. One example of this is the `docking` behavior.[12] Docking is when a robot moves to a specific location and orientation relative to a docking station. This is useful for robots performing materials handling in industry. In order to accept a piece of material to carry, the robot has to be close enough to the correct side of the end of a conveyor and facing the right way. Because docking requires a specific position and orientation, it can't be done with an attraction motor schema. That field would have the robot make a bee-line for the dock, even if it was coming from behind; the robot would stop at the back in the wrong position and orientation. Instead, a *selective attraction* field is appropriate. Here the robot only "feels" the attractive force when it is within a certain angular range of the docking station, as shown in Fig. 4.24.

SELECTIVE ATTRACTION

Unfortunately selective attraction does not cover the case of when the robot approaches from behind or to the side. How does the robot move to an area where the selective attraction field can take effect? One way to do this is to have tangential field, which makes the robot orbit the dock until it gets into the selective attraction area. The combination of the two motor schema produces a very smooth field which funnels the robot into the correct position and orientation, as shown in Fig. 4.25.

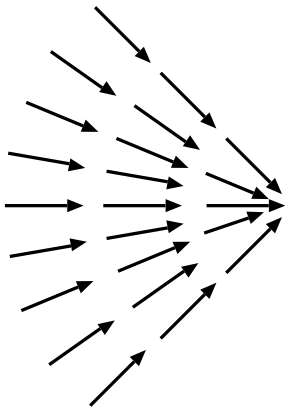


Figure 4.24 Selective attraction field, width of $\pm 45°$.

An interesting aspect of the docking behavior is that the robot is operating in close proximity to the dock. The dock will also release an instance of avoid, which would prevent the robot from getting near the desired position. In this case, the docking behavior would lower the magnitude (gain) of the output vector from an avoid instantiated in the dock area. Essentially, this partially inhibits the avoid behavior in selected regions. Also, the magnitude or gain can define the correct distance: the robot stops where the selective attraction of the dock balances with the repulsion.

The selective and tangential fields are not sufficient in practice, because of the limitations of perception. If the robot can't see the dock, it can't deploy the fields. But an industrial robot might know the relative direction of a dock, much as a bee recalls the direction to its hive. Therefore an attractive force attracts the robot to the vicinity of the dock, and then when the robot sees the dock, it begins the funnel effect into the correct position and orientation, even in the presence of obstacles, as seen in Fig. 4.26.

At least three perceptual schemas are needed for the docking behavior. One is needed to extract the relative direction of the dock for the regular attraction. Another is a perceptual schema capable of recognizing the dock in general, even from behind or the sides, in order to support the tangential field. The third perceptual schema is needed for the selective attention field; it has to be able to respond to the front of the dock and extract the robot's relative distance and orientation.

The docking behavior is now defined as having three perceptual schemas and three motor schemas (they could be grouped into 3 primitive behaviors). A schema-theoretic representation indicates that the behavior has some co-

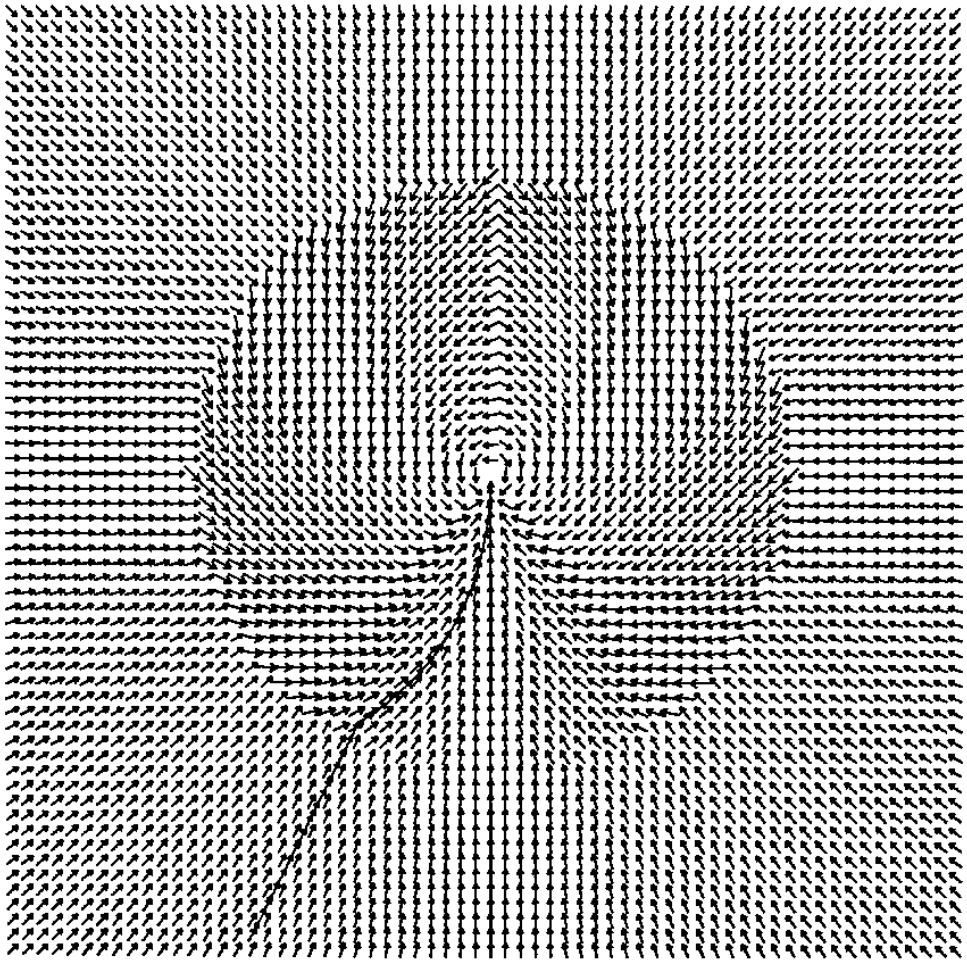


Figure 4.25 Docking potential field showing path of robot entering from slightly off course.

ordinated control program to coordinate and control these schemas. In the case of the docking behavior, a type of finite state machine is a reasonable choice for coordinating the sequence of perceptual and motor schemas. It provides a formal means of representing the sequence, and also reminds the designer to consider state transitions out of the ordinary. For instance, the robot could be moving toward the dock under the tangential plus selective attention fields when a person walks in front. This would occlude the view

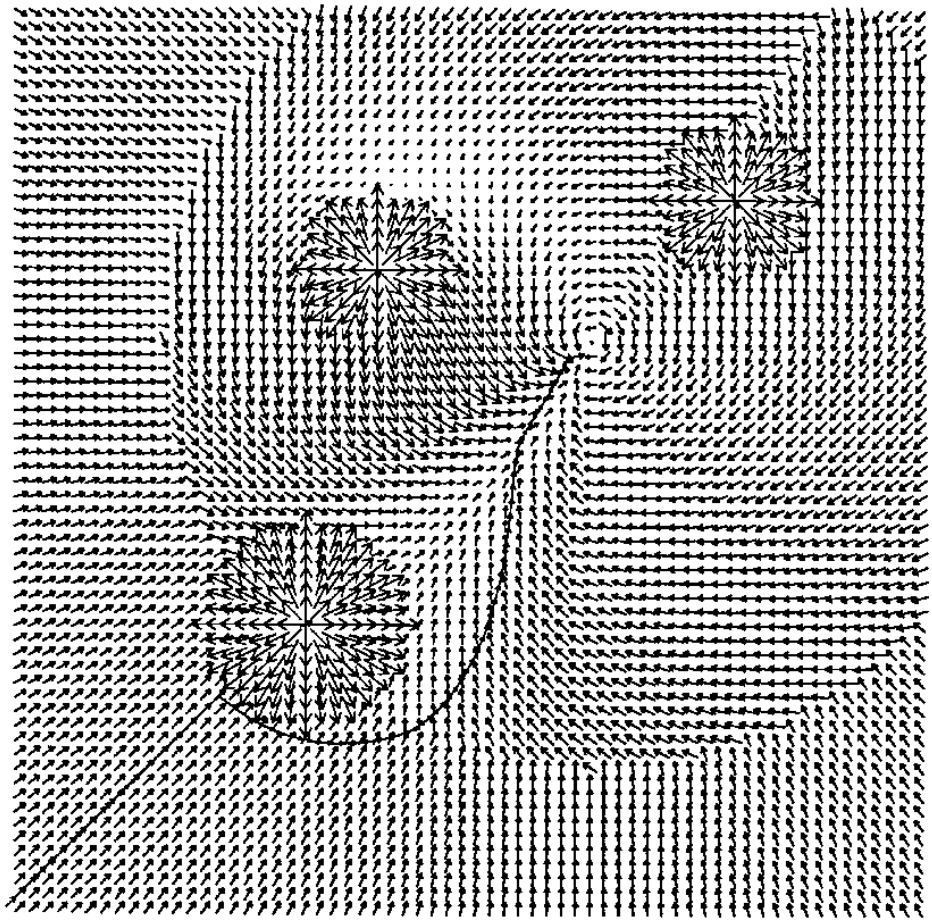


Figure 4.26 Visualization of the docking behavior with obstacles.

of the dock. The general attraction motor schema would then be reactivate, along with the associated perceptual schema to estimate the direction of the dock. This vector would allow the robot to avoid the human in a direction favorable as well as turn back towards the dock and try to re-acquire it.

The docking behavior also illustrates how the sensing capabilities of the robot impact the parameters of the motor schemas. Note that the angular size of the selective attention field would be determined by the angles at which the third perceptual schema could identify the dock. Likewise the radius of the tangential and selective attraction fields is determined by the distance at which the robot can perceive the dock.

4.4.8 Advantages and disadvantages

Potential field styles of architectures have many advantages. The potential field is a continuous representation that is easy to visualize over a large region of space. As a result, it is easier for the designer to visualize the robot's overall behavior. It is also easy to combine fields, and languages such as C++ support making behavioral libraries. The potential fields can be parameterized: their range of influence can be limited and any continuous function can express the change in magnitude over distance (linear, exponential, etc.). Furthermore, a two-dimensional field can usually be extended into a three-dimensional field, and so behaviors developed for 2D will work for 3D.

Building a reactive system with potential fields is not without disadvantages. The most commonly cited problem with potential fields is that multiple fields can sum to a vector with 0 magnitude; this is called the local minima problem. Return to Fig. 4.19, the box canyon. If the robot was being attracted to a point behind the box canyon, the attractive vector would cancel the repulsive vector and the robot would remain stationary because all forces would cancel out. The box canyon problem is an example of reaching a local minima. In practice, there are many elegant solutions to this problem. One of the earliest was to always have a motor schema producing vectors with a small magnitude from random noise.[12] The noise in the motor schema would serve to bump the robot off the local minima.

NAVIGATION TEMPLATES

Another solution is that of *navigation templates* (NaTs), as implemented by Marc Slack for JPL. The motivation is that the local minima problem most often arises because of interactions between the avoid behavior's repulsive field and other behaviors, such as move-to-goal's attractive field. The minima problem would go away if the avoid potential field was somehow smarter. In NaTs, the avoid behavior receives as input the vector summed from the other behaviors. This vector represents the direction the robot would go if there were no obstacles nearby. For the purposes of this book, this will be referred to as the strategic vector the robot wants to go. If the robot has a strategic vector, that vector gives a clue as to whether an obstacle should be passed on the right or the left. For example, if the robot is crossing a bridge (see Fig. 4.27), it will want to pass to the left of obstacles on its right in order to stay in the middle. Note that the strategic vector defines what is left and what is right.

NaTs implement this simple heuristic in the potential field for `RUNAWAY`, promoting it to a true `AVOID`. The repulsion field is now supplemented with a tangential orbit field. The direction of the orbit (clockwise or counter-

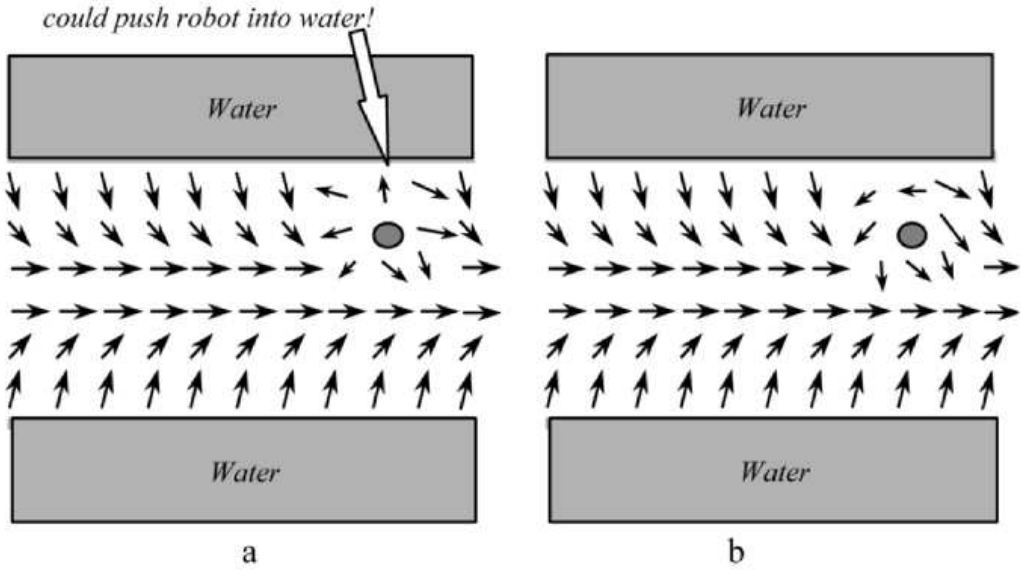


Figure 4.27 Problem with potential fields: a.) an example, and b.) the use of NaTs to eliminate the problem.

clockwise) is determined by whether the robot is to the right or left of the strategic vector. The output of the avoid behavior can be called a tactical vector, because it carries out the strategic goal of the robot in the face of immediate challenges, just as a military general might tell a captain to capture a fort but not specify each footstep.

A more recent solution to the local minima problem has been to express the fields as harmonic functions.[40] Potential fields implemented as harmonic functions are guaranteed not to have a local minima of 0. The disadvantage of this technique is that it is computationally expensive, and has to be implemented on a VLSI chip in order to run in real-time for large areas.

To summarize the major points about potential fields architectures:

- Behaviors are defined as consisting of one or more of both motor and perceptual schemas and/or behaviors. The motor schema(s) for a behavior must be a potential field.

- All behaviors operate concurrently and the output vectors are summed. Behaviors are treated equally and are not layered, although as will be seen in Ch. 5, there may be abstract behaviors which internally sequence be-

haviors. The coordinated control program is not specified; the designer can use logic, finite state machines, whatever is deemed appropriate. Sequencing is usually controlled by perceived cues or affordances in the environment, which are releasers.

- Although all behaviors are treated equally, behaviors may make varying contributions to the overall action of the robot. A behavior can change the gains on another behavior, thereby reducing or increasing the magnitude of its output. This means that behaviors can inhibit or excite other behaviors, although this is rarely used in practice.

- Perception is usually handled by direct perception or affordances.

- Perception can be shared by multiple behaviors. *A priori* knowledge can be supplied to the perceptual schemas, to emulate a specialized sensor being more receptive to events such as hall boundary spacing.

4.5 Evaluation of Reactive Architectures

As seen by the follow-corridor example, the two styles of architectures are very similar in philosophy and the types of results that they can achieve. Essentially, they are equivalent.

In terms of support for modularity, both decompose the actions and perceptions needed to perform a task into behaviors, although there is some disagreement over the level of abstraction of a behavior. Subsumption seems to favor a composition suited for a hardware implementation, while potential fields methods have nice properties for a software-oriented system.

The niche targetability is also high for both, assuming that the task can be performed by reflexive behaviors. Indeed, the use of direct perception emphasizes that reactive robots are truly constructed to fill a niche.

The issue of whether these architectures show an ease of portability to other domains is more open. Reactive systems are limited to applications which can be accomplished with reflexive behaviors. They cannot be transferred to domains where the robot needs to do planning, reasoning about resource allocation, etc. (this led to the Hybrid Paradigm to be described in Ch. 7). In practice, very few of the subsumption levels can be ported to new applications of navigating in an environment without some changes. The different applications create layers which need to subsume the lower layers differently. The potential fields methodology performs a bit better in that the

designer can create a library of behaviors and schemas to choose from, with no implicit reliance on a lower layer.

Neither architecture presents systems which could be called genuinely robust. The layering of subsumption imparts some graceful degradation if an upper level is destroyed, but it has no mechanisms to notice that a degradation has occurred. The finite state mechanisms in the docking behavior show some resilience, but again, only for situations which can be anticipated and incorporated into the state diagram. As with animals, a reactive robot will also do something consistent with its perception of the world, but not always the right thing.

4.6 Summary

Under the Reactive Paradigm, systems are composed of behaviors, which tightly couple sensing and acting. The organization of the Reactive Paradigm is SENSE-ACT or S-A, with no PLAN component. Sensing in the Reactive Paradigm is local to each behavior, or behavior-specific. Each behavior has direct access to one or more sensors independently of the other behaviors. A behavior may create and use its own internal world representation, but there is no global world model as with the Hierarchical Paradigm. As a result, reactive systems are the fastest executing robotic systems possible.

There are four major characteristics of robots constructed under the Reactive Paradigm. Behaviors serve as the basic building blocks for robot actions, even though different designers may have different definitions of what a behavior entails. As a consequence of using behaviors, the overall behavior of the robot is emergent. Only local, behavior-specific sensing is permitted. The use of explicit representations in perceptual processing, even locally, is avoided in most reactive systems. Explicit representations of the world are often referred to as maintaining the state of the world internally, or internal state. Instead, reactive behaviors rely on the world to maintain state (as exemplified by the gripper controlling whether the robot was looking for soda cans or for the recycling bin). Animal models are often cited as a basis for a behavior or the architecture. Behaviors and groups of behaviors which were inspired by or simulate animal behavior are often considered desirable and more interesting than hacks. Finally, reactive systems exhibit good software engineering principles due to the "programming by behavior" approach. Reactive systems are inherently modular from a software design perspective. Behaviors can be tested independently, since the overall behav-

ior is emergent. More complex behaviors may be constructed from primitive behaviors, or from mixing and matching perceptual and motor components. This supports good software engineering practices, especially low coupling and high cohesion.

The subsumption architecture is a popular reactive system. Behaviors are purely reflexive and may not use memory. Behaviors are arranged in layers of competence, where the lower levels encapsulate more general abilities. The coordination of layers is done by higher layers, which have more specific goal-directed behaviors, subsuming lower layers. Behaviors within a layer are coordinated by finite state automata, and can be readily implemented in hardware.

Potential fields methodologies are another popular reactive system. Behaviors in potential field systems must be implemented as potential fields. All active behaviors contribute a vector; the vectors are summed to produce a resultant direction and magnitude for travel. Pfields provide a continuous representation, which is easier to visualize than rule encoding, and are continuous. The fields can be readily implemented in software, and parameterized for flexibility and reuse. The vector summation effect formalizes how to combine behaviors, eliminating issues in how to design behaviors for subsumption. The fields are often extensible to three dimensions, adding to the re-usability and portability. In the example in this chapter, behaviors using potential fields were able to encapsulate several layers in subsumption into a set of concurrent peer behaviors with no layers. Ch. 5 will give examples of how to sequence, or assemble, behaviors into more abstract behaviors.

Despite the differences, subsumption and potential fields appear to be largely equivalent in practice. Both provide support for modularity and niche targetability. The ease of portability to other domains is relative to the complexity of the changes in the task and environment. Neither style of architecture explicitly addresses robustness, although in theory, if only a higher layer of a subsumption system failed, the lower layers should ensure robot survivability.

4.7 Exercises

Exercise 4.1

Define the reactive paradigm in terms of a.) the SENSE, PLAN, and ACT primitives, and b.) sensing organization.

Exercise 4.2

Describe the difference between robot control using a horizontal decomposition and a vertical decomposition.

Exercise 4.3

List the characteristics of a reactive robotic system.

Exercise 4.4

Describe the differences between two dominant methods for combining behaviors in a reactive architecture, subsumption and potential field summation.

Exercise 4.5

Evaluate the subsumption architecture in terms of: support for modularity, niche targetability, ease of portability to other domains, robustness.

Exercise 4.6

Evaluate potential field methodologies in terms of: support for modularity, niche targetability, ease of portability to other domains, robustness.

Exercise 4.7

What is the difference between the way the term "internal state" was used in ethology and the way "internal state" means in behavioral robotics?

Exercise 4.8

Diagram Level 2 in the subsumption example in terms of behaviors.

Exercise 4.9

When would an exponentially increasing repulsive field be preferable over a linear increasing repulsive field?

Exercise 4.10

Suppose you were to construct a library of potential fields of the five primitives. What parameters would you include as arguments to allow a user to customize the fields?

Exercise 4.11

Use a spreadsheet, such as Microsoft Excel, to compute various magnitude profiles.

Exercise 4.12

Return to Fig. 4.17. Plot the path of the robot if it started in the upper left corner.

Exercise 4.13

Consider the Khepera robot and its IR sensors with the `RUNAWAY` behavior instantiated for each sensor as in the example in Fig. 4.19. What happens if an IR breaks and always returns a range reading of N, meaning an obstacle is N cm away? What will be the emergent behavior? and so on. Can a reactive robot notice that it is malfunctioning? Why or why not?

Exercise 4.14

How does the Reactive Paradigm handle the frame problem and the open world assumption?

Exercise 4.15

An alternative `RUNAWAY` behavior is to turn 90° (either left or right, depending on whether its "left handed" or "right handed" robot) rather than 180°. Can this be represented by a potential field?

Exercise 4.16

Using rules, or if-then statements, is a method for representing and combining programming units which are often called behaviors; for example "if OBSTACLE-ON-LEFT and HEADING-RIGHT, then IGNORE." Can the layers in subsumption for hall-following be written as a series of rules? Can the potential fields? Are rules equivalent to these two methods? Do you think rules are more amenable to good software engineering practices?

Exercise 4.17

Some researchers consider random wandering as a primitive potential field. Recall that random wandering causes the robot to periodically swap to a new vector with a random direction and magnitude. How can a wander field be represented? Does the array of the field represent a physical area or time? Unlike regular potential fields, the vector is computed as a function of time, every n minutes, rather than on the robot's relationship to something perceivable in the world.

Exercise 4.18 [*Programming*]

Design and implement potential fields:

a. Construct a potential field to represent a "move through door" behavior from primitive potential fields. Why won't a simple attractive field work? ANS: if the robot is coming from a side, it will graze the door frame because the robot is not a point, it has width and limited turning radius.

b. What happens if a person is exiting the door as the robot enters? Design an appropriate "avoid" potential field, and show the emergent potential field when `AVOID` and MOVE-THRU-DOOR are activated at the same time.

c. Simulate this using the Khepera simulator for Unix systems found at: http://www.k-team.com.

d. Run this on a real khepera.

Exercise 4.19 [*Programming*]

Program two versions of a phototropic behavior using the Khepera simulator. Both versions should use the same motor schema, an attractive field, but different perceptual schemas. In one version, the perceptual schema processes light from a single sensor and the behavior is instantiated 8 times. In the second version, the perceptual schema processes light from all sensors and returns the brightest. Set up five interesting "worlds" in the simulator with different placements of lights. Compare the emergent behavior for each world.

Exercise 4.20 [*Digital Circuits*]

For readers with a background in digital circuits, build one or more of the simple creatures in Flynn and Jones' *Mobile Robots: Inspiration to Implementation*[76] using a Rug Warrior kit.

4.8 End Notes

For a roboticist's bookshelf.
The subsumption architecture favors a hardware implementation using inexpensive hardware. Part of the rapid acceptance of the reactive paradigm and the subsumption architecture was due to *Mobile Robots: Inspiration to Perspiration*[76] by students in the MIT AI Lab. The straightforward circuits allowed any hobbyist to produce intelligent robots. On a more theoretical note, Rodney Brooks has collected his seminal papers on subsumption into a volume entitled *Cambrian Intelligence*,[28] a nice play on the period in evolution and on Brooks' location in Cambridge, Massachusetts.

About Rodney Brooks.
Rodney Brooks is perhaps the best known roboticist, with his insect-like (Genghis, Attila, etc.) and anthropomorphic robots (Cog, Kismet) frequently appearing in the media. Brooks was one of four "obsessed" people profiled in a documentary by Errol Morris, *Fast, Cheap, and Out of Control*. The documentary is well worth watching, and there are some gorgeous shots of robots walking over broken glass giving the terrain a luminous quality. Brooks' reactive philosophy appears in an episode of *The X-Files* on robotic cockroaches called "War of the Corophages." The roboticist from the Massachusetts Institute of Robotics is a combination of Steven Hawking (the character is disabled), Joe Engleberger (the bow tie), Marvin Minsky (stern, professorial manner), and Rodney Brooks (the character says almost direct quotes from Brooks' interviews in science magazines).

About the "s" in subsumption.
Rodney Brooks' 1986 paper never officially named his architecture. Most papers refer to it as "Subsumption," sometimes without the capital "s" and sometimes with a capital.

Building a robot with petty cash.
Erann Gat and his then boss at NASA's Jet Propulsion Laboratory, David Miller, cite another advantage of behavioral robotics: you can build a mobile robot very cheaply. In the late 1980's, Miller's group wanted to build a small reactive robot to compare with the traditional Hierarchical vehicles currently used for research and to get experience with subsumption. Request after request was turned down. But they realized that the individual electronics were cheap. So they bought the parts for their small

mobile robot in batches of $50 or less, which could be reimbursed out of petty cash. The resulting robot was about the size and capability of a Lego Mindstorms kit with a total cost around $500.

Robot name trivia.
The MIT robots used in the early days of subsumption were all named for conquerors: Attila, Genghis, etc. Ron Arkin did his initial work on potential fields methodology while pursuing his PhD at the University of Massachusetts. The robot he used was a Denning MRV called HARV (pronounced "Harvey"). HARV stood for Hardly Autonomous Robot Vehicle morphing into Harmful Autonomous Robot Vehicle, when military research sponsors were present. HARV was also considered a short version of "Harvey Wallbanger," both a description of the robot's emergent behavior and the name of a cocktail.

5 *Designing a Reactive Implementation*

Chapter objectives:

- Use schema theory to design and program behaviors using object-oriented programming principles.
- Design a complete behavioral system, including coordinating and controlling multiple concurrent behaviors.
- For a given behavioral system, draw a *behavioral table* specifying the releasers, perceptual schemas, and motor schemas for each behavior.
- Describe the two methods for assembling and coordinating primitive behaviors within an *abstract behavior: finite state automata* and *scripts*. Be able to represent a sequence of behaviors with a *state diagram* or with a pseudocode *script*.

5.1 Overview

By this point, the sheer simplicity and elegance of reactive behaviors tend to incite people to start designing and implementing their own robots. Kits such as Lego Mindstorms and the Rug Warrior permit the rapid coupling of sensors and actuators, enabling users to build reactive behaviors. However, the new issues are how to program more intelligence into the software and how to exploit better sensors than come with kits. Unfortunately, good intentions in robot programming are often frustrated by two deficits. First, designing behaviors tends to be an art, not a science. Novice roboticists often are uncertain as to how to even start the design process, much less how to know when they've got a reasonable system. The second deficit is more subtle. Once the designer has a few well-designed and tested behaviors, how

are they integrated into a system? The Reactive movement and early work described in Ch. 4 was characterized by robots running a very small set of behaviors which were combined internally to produce an overall *emergent behavior*. The key components of a reactive architecture were shown to be the behaviors plus the mechanism for merging the output of the concurrent behaviors.

EMERGENT BEHAVIOR

SERIES OF BEHAVIORS

However, many applications are best thought of as a *series of behaviors*, each operating in a recognizable sequence. One of the first popular applications that many roboticists looked at was picking up and disposing of an empty soft drink can. This involved search for a can, move toward the can when it is found, pick up the can, search for the recycle bin, move toward the recycle bin, drop the can.[39;129;66] It's counterintuitive to think of these behaviors as being concurrent or merged. (There is certainly the possibility of concurrency, for example avoiding obstacles while moving to the soda can or recycle bin.) Therefore, new techniques had to be introduced for controlling sequences of behaviors. Most of these techniques are conceptually the equivalent of constructing a macro behavior, where the schema structure is used recursively to simplify programming the control program.

This chapter attempts to aid the novice designer in constructing a reactive robot system by addressing each of these two deficits. First, an object-oriented programming approach to designing behaviors is introduced. This approach is based on schema theory, which was introduced in Ch. 3. The "art" of design is presented in a flow chart following the waterfall method of software engineering, along with a case study of a winning entry in the 1994 International Association for Unmanned Systems' Unmanned Ground Robotics Competition. The case study emphasizes the importance of establishing the ecological niche of a reactive robot. Second, two techniques for managing sequences of behaviors are introduced: *finite-state automata* and *scripts*. As could be expected from the material presented in Ch. 3, both of these techniques will look very similar to the Innate Releasing Mechanisms from Tinbergen and Lorenz. Finally, the chapter shows how these techniques were applied to entries in the "Pick Up the Trash" events of the 1994 AAAI and 1995 IJCAI Mobile Robot Competitions. The focus of these examples is how the logic for coordinating a sequence of behaviors is developed, represented, and implemented. The use of schemas to "factor out" and program a small set of generic behaviors rather than designing a large set of specialized behaviors is emphasized throughout the chapter.

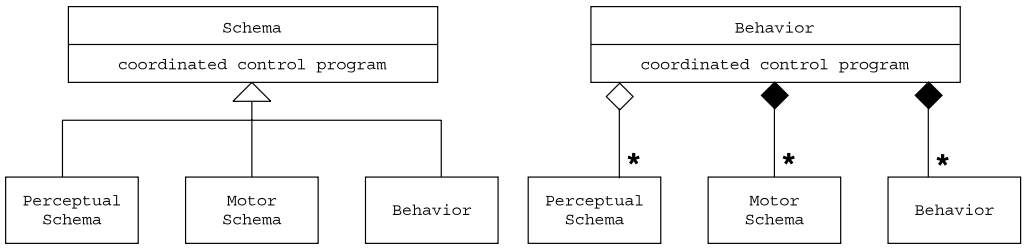


Figure 5.1 Classes: a.) schema and b.) behavior.

5.2 Behaviors as Objects in OOP

Although Object-Oriented Programming (OOP) had not become popular during the time that the Reactive Paradigm was developed, it is useful to cast behaviors in OOP terms. Schema theory is well suited for transferring theoretical concepts to OOP. Furthermore, schema theory will be used as a bridge between concepts in biological intelligence and robotics, enabling a practical implementation of reactivity exploiting innate releasing mechanisms and affordances.

Recall from software engineering that an object consists of data and methods, also called attributes and operations. As noted in Ch. 3, schemas contain specific knowledge and local data structures and other schemas. Fig. 5.1 shows how a schema might be defined. Following Arbib,[6] a schema as a programming object will be a class. The class will have an optional method called a *coordinated control program*. The coordinated control program is a function that coordinates any methods or schemas in the derived class.

COORDINATED CONTROL PROGRAM

Three classes are derived from the Schema Class: Behavior, Motor Schema, and Perceptual Schema. Behaviors are composed of at least one Perceptual Schema and one Motor Schema; these schemas act as the methods for the Behavior class. A Perceptual Schema has at least one method; that method takes sensor input and transforms it into a data structure called a *percept*. A Motor Schema has at least one method which transforms the percept into a vector or other form of representing an action. Since schemas are independent, the Behavior object acts as a place holder or local storage area for the percept. The Perceptual Schema is linked to the sensor(s), while the Motor Schema is linked to the robot's actuators. The sensors and actuators can be represented by their own software classes if needed; this is useful when working with software drivers for the hardware.

PERCEPT

Using the Unified Modeling Language representation,[55] the Schema and Behavior classes look like Fig. 5.1. The OOP organization allows a behavior to be composed of multiple perceptual schema and motor schema and even behaviors. Another way of stating this is that the definition of a behavior is recursive. Why is it useful to have multiple perceptual schema and motor schema? In some cases, it might be helpful to have two perceptual schema, one for say daylight conditions using a TV camera and one for nighttime using infra-red. Sec. 5.2.2 provides a more detailed example of why multiple schemas in a behavior can be helpful.

PRIMITIVE BEHAVIOR

Recall that a *primitive behavior* is composed of only one perceptual schema and one motor schema; there is no need to have any coordinated control program. Primitive behaviors can be thought of being *monolithic*, where they do only one ("mono") thing. Because they are usually a simple mapping from stimulus to response, they are often programmed as a single method, not composed from multiple methods or objects. The concept of Perceptual and Motor Schema is there, but hidden for the sake of implementation.

ABSTRACT BEHAVIORS

Behaviors which are assembled from other behaviors or have multiple perceptual schema and motor schema will be referred to as *abstract behaviors*, because they are farther removed from the sensors and actuators than a primitive behavior. The use of the term "abstract behavior" should not be confused with an abstract class in OOP.

5.2.1 Example: A primitive move-to-goal behavior

This example shows how a primitive behavior can be designed using OOP principles. In 1994, the annual AAAI Mobile Robot Competitions had a "Pick Up the Trash" event, which was repeated in 1995 at the joint AAAI-IJCAI Mobile Robot Competition.[129;66] The basic idea was for a robot to be placed in an empty arena about the size of an office. The arena would have Coca-Cola cans and white Styrofoam cups placed at random locations. In two of the four corners, there would be a blue recycling bin; in the other two, a different colored trash bin. The robot who picked up the most trash and placed them in the correct bin in the allotted time was the winner. In most years, the strategy was to find and recycle the Coca-Cola cans first, because it was easier for the robot's vision processing algorithms to perceive red and blue.

One of the most basic behaviors needed for picking up a red soda can and moving to a blue bin is `move_to_goal`. When the robot sees a red can, it must move to it. When it has a can, it must find and then move to a blue bin.

It is better software engineering to write a general move to goal behavior, where only what is the goal—a red region or a blue region—varies. The goal for the current instance can be passed in at instantiation through the object constructor.

Writing a single generic behavior for `move_to_goal(color)` is more desirable than writing a `move_to_red` and a `move_to_blue` behaviors. From a software engineering perspective, writing two behaviors which do the same thing is an opportunity to introduce a programming bug in one of them and not notice because they are supposed to be the same. Generic behaviors also share the same philosophy as factoring in mathematics. Consider simplifying the equation $45x^2 + 90x + 45$. The first step is to factor out any common term to simplify the equation. In this case, 45 can be factored and the equation rewritten as $45(x + 1)^2$. The color of the goal, red or blue, was like the common coefficient of 45; it is important, but tends to hide that the key to the solution was the move-to-goal part, or x.

Modular, generic code can be handled nicely by schemas as shown in Fig. 5.2. The behavior `move_to_goal` would consist of a perceptual schema, which will be called `extract-goal`, and a motor schema, which uses an attractive field called `pfields.attraction`. `extract-goal` uses the affordance of color to extract where the goal is in the image, and then computes the angle to the center of the colored region and the size of the region. This information forms the percept of the goal; the affordance of the Coke can is the color, while the information extracted from the perception is the angle and size. The attraction motor schema takes that percept and is responsible for using it to turn the robot to center on the region and move forward. It can do this easily by using an attractive field, where the larger the region, the stronger the attraction and the faster the robot moves.

AFFORDANCE

The `move_to_goal` behavior can be implemented as a primitive behavior, where `goal_color` is a numerical means of representing different colors such as `red` and `blue`:

`move_to_goal(goal_color)`:

Object	Behavioral Analog	Identifier
Data	percept	`goal_angle` `goal_strength`
Methods	`perceptual_schema` `motor_schema`	`extract_goal(goal_color)` `pfields.attraction(goal_angle, goal_strength)`

The above table implies some very important points about programming with behaviors:

- The behavior is the "glue" between the perceptual and motor schemas. The schemas don't communicate in the sense that both are independent entities; the perceptual schema doesn't know that the motor schema exists. Instead, *the behavior puts the percept created by the perceptual schema in a local place where the motor schema can get it.*

- Behaviors can (and should) use libraries of schemas. The pfields suffix on the `pfields.attraction()` meant that attraction was a method within another object identified as pfields. The five primitive potential fields could be encapsulated into one class called `PFields`, which any motor schema could use. `PFields` would serve as a library. Once the potential fields in `PFields` were written and debugged, the designer doesn't ever have to code them again.

- Behaviors can be reused if written properly. In this example, the move to goal behavior was written to accept a structure (or object) defining a color and then moving to a region of that color. This means the behavior can be used with both red Coke cans and blue trash cans.

5.2.2 Example: An abstract follow-corridor behavior

The move to goal example used a single motor schema with a single perceptual schema. This example shows how a potential fields methodology can be implemented using schemas. In the corridor following example in Ch. 4, the `follow_corridor` potential field consisted of two primitive fields: two instances of `perpendicular` to the walls and one `uniform` parallel to the walls. The `follow_corridor` field could be implemented in schemas in at least two different ways as shown in Fig. 5.2. In one way, each of the primitive fields would be a separate motor schema. The follow corridor motor schema would consist of the three primitives and the coordinated control program. The coordinated control program would be the function that knows that one field is perpendicular from the left wall going towards the center of the corridor, which way is forward, etc. They were summed together by the coordinated control program in the behavioral schema to produce a single output vector. The perceptual schema for the follow corridor would examine the sonar polar plot and extract the relative location of the corridor walls. The perceptual schema would return the distance to the left wall and the right wall.

Another way to have achieved the same overall behavior is to have `follow_wall` composed of two instances of a follow wall behavior: `follow_-`

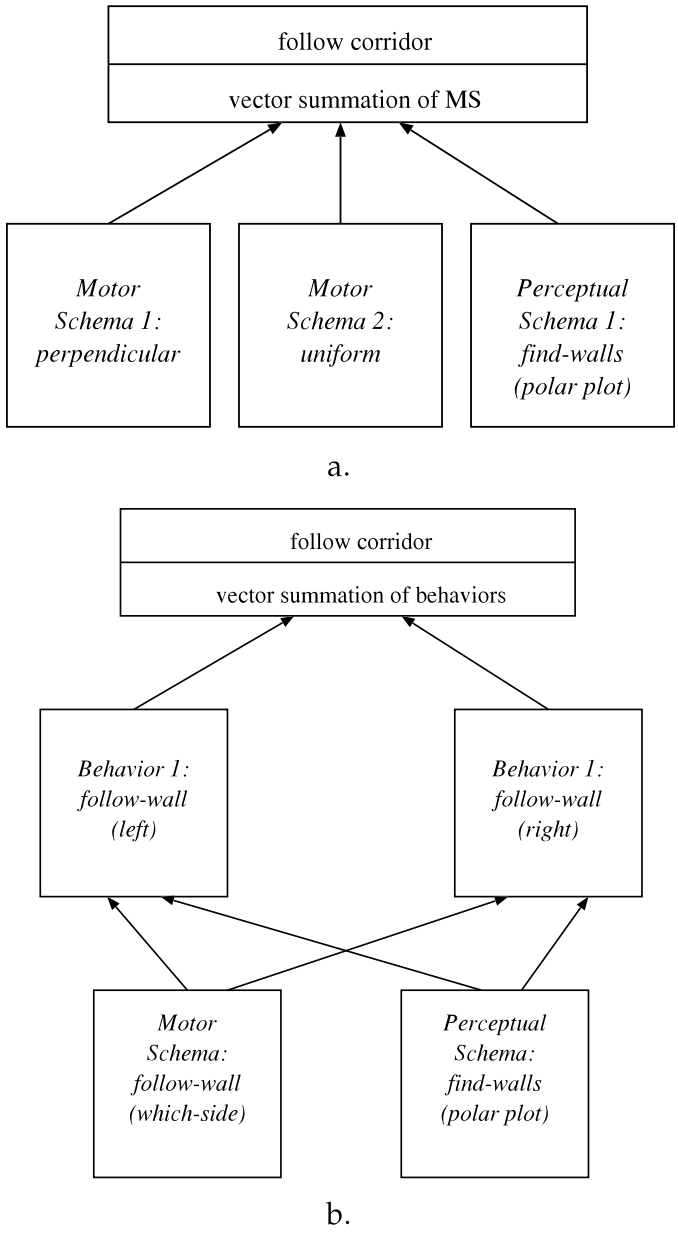


Figure 5.2 Class diagrams for two different implementations of `follow_corridor`: a.) use of primitive fields, and b.) reuse of fields grouped into a `follow_wall` behavior.

`wall (left)` and `follow_wall(right)`. Each instance of follow wall would receive the sonar polar plot and extract the relevant wall. The associated class diagram is shown on the right in Fig. 5.2.

In both implementations, the motor schema schemas ran continuously and the vectors were summed internally in order to produce a single output vector. Since there were multiple motor schemas, the coordinated control program for follow-corridor is not null as it was for move-to-goal. The vector summation and the concurrency form the conceptual coordinated control program in this case.

5.2.3 Where do releasers go in OOP?

TWO PURPOSES OF PERCEPTION

The previous examples showed how behaviors can be implemented using OOP constructs, such as classes. Another important part of a behavior is how it is activated. As was discussed in Ch. 3, perception serves two purposes: *to release a behavior* and *to guide it.* Perceptual schemas are clearly used for guiding the behavior, either moving toward a distinctively colored goal or following a wall. But what object or construct contains the releaser and how is it "attached" to the behavior?

The answer to the first part of the question is that *the releaser is itself a perceptual schema.* It can execute independently of whatever else is happening with the robot; it is a perceptual schema not bound to a motor schema. For example, the robot is looking for red Coke cans with the `extract_color` perceptual schema. One way to implement this is when the schema sees red, it can signal the main program that there is red. The main program can determine that that is the releaser for the move to goal behavior has been satisfied, and instantiate `move_to_goal` with `goal=red`. `move_to_goal` can instantiate a new instance of `extract_color` or the main program can pass a pointer to the currently active `extract_color`. Regardless, `move_to_goal` has to instantiate `pfield.attraction`, since the attraction motor schema wouldn't be running. In this approach, the main program is responsible for calling the right objects at the right time; the releaser is attached to the behavior by the designer with little formal mechanisms to make sure it is correct. This is awkward to program.

Another, more common programming approach is to have the releaser be part of the behavior: the single perceptual schema does double duty. This programming style requires a coordinated control program. The behavior is always active, but if the releaser isn't satisfied, the coordinated control program short-circuits processing. The behavior returns an identity function,

in the case of potential fields, a vector of (0.0,0.0), which is the same as if the behavior wasn't active at all. This style of programming can tie up some resources, but is generally a simple, effective way to program. Fig. 5.2 shows the two approaches.

Either way, once the robot saw red, the observable aspect of move to goal (e.g., moving directly toward the goal) would commence. The extract goal schema would update the percept data (relative angle of the goal and size of red region) every time it was called. This percept would then be available to the motor schema, which would in turn produce a vector.

As will be covered in Sec. 5.5, the releaser must be designed to support the correct sequence. Depending where the robot was in the sequence of activities, the robot uses move to goal to move to a red Coke can or a blue recycling bin. Otherwise, the robot could pursue a red Coke can and a blue recycling bin simultaneously. There is nothing in the OOP design to prevent that from happening—in fact, OOP makes it easy. In this situation, there would be two move to goal objects, one instantiated with goal of "red" and the other with goal of "blue." Notice that the move to goal behavior can use any perceptual schema that can produce a goal angle and goal strength. If the robot needed to move to a bright light (phototropism), only the perceptual schema would need to be changed. This is an example of software reusability.

5.3 Steps in Designing a Reactive Behavioral System

Fig. 5.3 shows the steps in designing a reactive behavioral system, which are taken from *Behavior-Based Robotics*[10] and a case study by Murphy.[98] This section will first give a broad discussion of the design process, then work through each step using the winning approach taken in the 1994 Unmanned Ground Vehicle Competition.

The methodology in Fig. 5.3 assumes that a designer is given a task for the robot to do, and a robot platform (or some constraints, if only budgetary). The goal is to design a robot as a situated agent. Therefore, the first three steps serve to remind the designer to specify the *ecological niche* of the robot.

The fourth step begins the iterative process of identifying and refining the set of behaviors for the task. It asks the question: *what does the robot do?* Defining the ecological niche defines constraints and opportunities but doesn't necessarily introduce major insights into the *situatedness* of the robot: *how it acts and reacts to the range of variability in its ecological niche.* This step is where a novice begins to recognize that designing behaviors is an art. Sometimes,

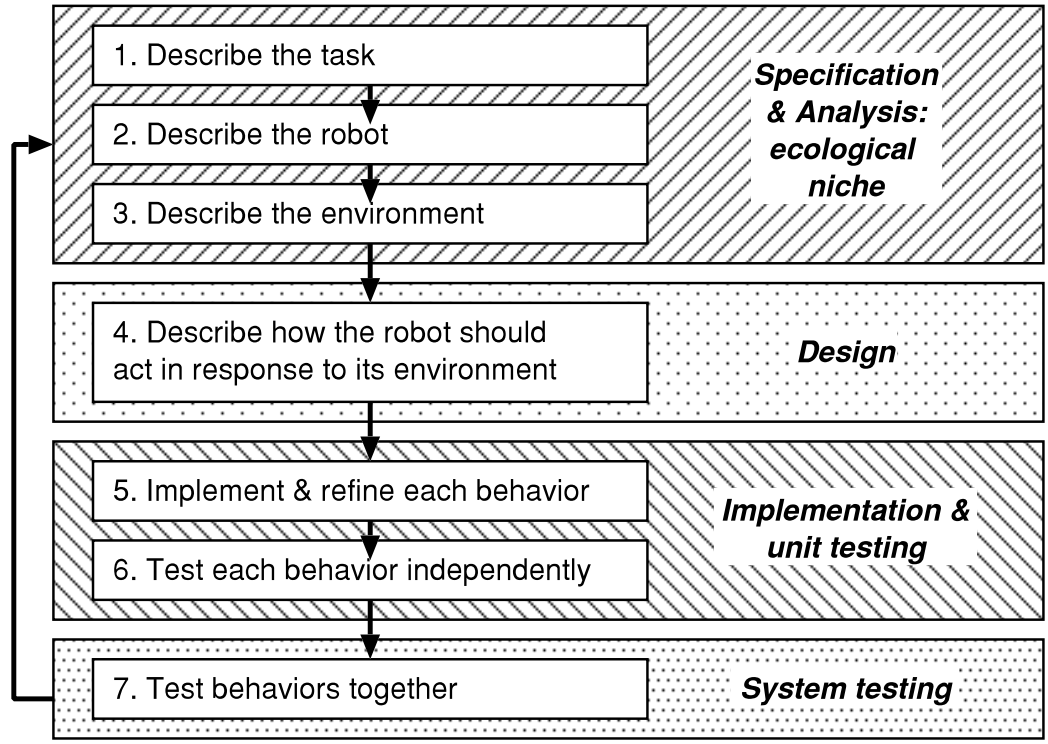


Figure 5.3 Steps in designing a reactive behavioral system, following basic software engineering phases.

a behavioral decomposition appears obvious to a roboticist after thinking about the ecological niche. For example, in the 1994 and 1995 Pick Up the Trash events, most of the teams used a partitioning along the lines of: random search until see red, move to red, pick up can, random search until see blue, move to blue, drop can.

Roboticists often attempt to find an analogy to a task accomplished by an animal or a human, then study the ethological or cognitive literature for more information on how the animal accomplishes that class of tasks. This, of course, sidesteps the question of how the roboticist knew what class of animal tasks the robot task is similar to, as well as implies a very linear thinking process by roboticists. In practice, roboticists who use biological and cognitive insights tend to read and try to stay current with the ethological literature so that they can notice a connection later on.

Steps 5-7 are less abstract. Once the candidate set of behaviors has been proposed, the designer works on designing each individual behavior, spec-

ifying its motor and perceptual schemas. This is where the designer has to write the algorithm for finding red blobs in a camera image for the random search until find red and move to red behaviors. The designer usually programs each schema independently, then integrates them into a behavior and tests the behavior thoroughly in isolation before integrating all behaviors. This style of testing is consistent with good software engineering principles, and emphasizes the practical advantages of the Reactive Paradigm.

The list of steps in implementing a reactive system can be misleading. Despite the feedback arrows, the overall process in Fig. 5.3 appears to be linear. In practice, it is iterative. For example, a supposed affordance may be impossible to detect reliably with the robot's sensors, or an affordance which was missed in the first analysis of the ecological niche suddenly surfaces. The single source of iteration may be testing all the behaviors together in the "real world." Software that worked perfectly in simulation often fails in the real world.

5.4 Case Study: Unmanned Ground Robotics Competition

This case study is based on the approach taken by the Colorado School of Mines team to the 1994 Unmanned Ground Robotics Competition.[98] The objective of the competition was to have a small unmanned vehicle (no larger than a golf cart) autonomously navigate around an outdoor course of white lines painted on grass. The CSM entry won first place and a $5,000 prize. Each design step is first presented in boldface and discussed. What was actually done by the CSM team follows in italics. This case study illustrates the effective use of only a few behaviors, incrementally developed, and the use of affordances combined with an understanding of the ecological niche. It also highlights how even a simple design may take many iterations to be workable.

Step 1: Describe the task. The purpose of this step is to specify what the robot has to do to be successful.

The task was for the robot vehicle to follow a path with hair pin turns, stationary obstacles in the path, and a sand pit. The robot which went the furthest without going completely out of bounds was the winner, unless two or more robots went the same distance or completed the course, then the winner was whoever went the fastest. The maximum speed was 5 mph. If the robot went partially out of bounds (one wheel or a portion of a tread remained inside), a distance penalty was subtracted. If the robot hit an obstacle enough to move it, another distance penalty was levied. Therefore,

the competition favored an entry which could complete the course without accruing any penalties over a faster entry which might drift over a boundary line or bump an obstacle. Entrants were given three runs on one day and two days to prepare and test on a track near the course; the times of the heats were determined by lottery.

Step 2: Describe the robot. The purpose of this step is to determine the basic physical abilities of the robot and any limitations. In theory, it might be expected that the designer would have control over the robot itself, what it could do, what sensors it carries, etc. In practice, most roboticists work with either a commercially available research platform which may have limitations on what hardware and sensors can be added, or with relatively inexpensive kit type of platform where weight and power restrictions may impact what it can reasonably do. Therefore, the designer is usually handed some fixed constraints on the robot platform which will impact the design.

In this case, the competition stated that the robot vehicle had to have a footprint of at least 3ft by 3.5ft but no bigger than a golf cart. Furthermore, the robot had to carry its own power supply and do all computing on-board (no radio communication with an off-board processor was permitted), plus carry a 20 pound payload.

The CSM team was donated the materials for a robot platform by Omnitech Robotics, Inc. Fig. 5.4 shows Omnibot. The vehicle base was a Power Wheels battery powered children's jeep purchased from a toy store. The base met the minimum footprint exactly. It used Ackerman (car-like) steering, with a drive motor powering the wheels in the rear and a steering motor in the front. The vehicle had a 22° turning angle. The on-board computing was handled by a 33MHz 486 PC using Omnitech CANAMP motor controllers. The sensor suite consisted of three devices: shaft encoders on the drive and steer motors for dead reckoning, a video camcorder mounted on a mast near the center of the vehicle and a panning sonar mounted below the grille on the front. The output from the video camcorder was digitized by a black and white framegrabber. The sonar was a Polaroid lab grade ultrasonic transducer. The panning device could sweep 180°. All coding was done in C++.

Due to the motors and gearing, Omnibot could only go 1.5 mph. This limitation meant that it could only win if it went farther with less penalty points than any other entry. It also meant that the steering had to have at least a 150ms update rate or the robot could veer out of bounds without ever perceiving it was going off course. The black and white framegrabber eliminated the use of color. Worse yet, the update rate of the framegrabber was almost 150ms; any vision processing algorithm would have to be very fast or else the robot would be moving faster than it could react. The reflections from uneven grass reduced the standard range of the sonar from 25.5 ft to about 10 ft.

Figure 5.4 Omnibot, a mobile robot built from a Power Wheels battery-powered toy jeep by students and Omnitech Robotics, Inc.

Step 3: Describe the Environment. This step is critical for two reasons. First, it is a key factor in determining the situatedness of the robot. Second, it identifies perceptual opportunities for the behaviors, both in how a perceptual event will instantiate a new behavior, and in how the perceptual schema for a behavior will function. Recall from Chapter 4 that the Reactive Paradigm favors direct perception or affordance-based perception because it has a rapid execution time and involves no reasoning or memory.

The course was laid out on a grassy field with gentle slopes. The course consisted of a 10 foot wide lane marked in US Department of Transportation white paint, roughly in the shape of a kidney (see Fig. 5.5). The exact length of the course and layout of obstacles of the course were not known until the day of the competition, and teams were not permitted to measure the course or run trials on it. Obstacles were all stationary and consisted of bales of hay wrapped in either white or red plastic. The bales were approximately 2 ft by 4 ft and never extended more than 3 feet into the lane. The sonar was able to reliably detect the plastic covered bales at most angles of approach at 8 feet away. The vehicles were scheduled to run between 9am and 5pm on May 22, regardless of weather or cloud cover. In addition to the visual challenges of changing lighting due to clouds, the bales introduced shadows on the white lines between 9–11am and 3–5pm. The sand pit was only 4 feet long and placed on a straight segment of the course.

The analysis of the environment offered a simplification of the task. The placing of the obstacles left a 4 ft wide open area. Since Omnibot was only 3 ft wide, the course could be treated as having no obstacles if the robot could stay in the center of the lane

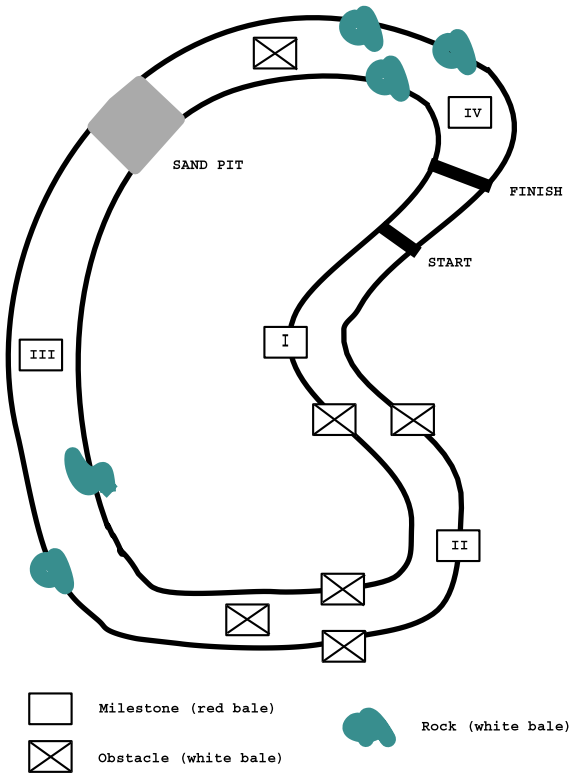


Figure 5.5 The course for the 1994 Ground Robotics Competition.

with a 0.5 ft tolerance. This eliminated the need for an avoid obstacle behavior.

The analysis of the environment also identified an affordance for controlling the robot. The only object of interest to the robot was the white line, which should have a high contrast to the green (dark gray) grass. But the exact lighting value of the white line changed with the weather. However, if the camera was pointed directly at one line, instead of trying to see both lines, the majority of the brightest points in the image would belong to the line (this is a reduction in the signal to noise ratio because more of the image has the line in it). Some of the bright points would be due to reflections, but these were assumed to be randomly distributed. Therefore, if the robot tried to keep the centroid of the white points in the center of the image, it would stay in the center of the lane.

Step 4: Describe how the robot should act in response to its environment. The purpose of this step is to identify the set of one or more candidate primitive behaviors; these candidates will be refined or eliminated later. As the designer describes how the robot should act, behaviors usually become apparent. It should be emphasized that the point of this step is to concen-

trate on what the robot should do, not how it will do it, although often the designer sees both the what and the how at the same time.

In the case of the CSM entry, only one behavior was initially proposed: follow-line. The perceptual schema would use the white line to compute the difference between where the centroid of the white line was versus where it should be, while the motor schema would convert that to a command to the steer motor.

BEHAVIOR TABLE

In terms of expressing the behaviors for a task, it is often advantageous to construct a *behavior table* as one way of at least getting all the behaviors on a single sheet of paper. The releaser for each behavior is helpful for confirming that the behaviors will operate correctly without conflict (remember, accidently programming the robotic equivalent of male sticklebacks from Ch. 3 is undesirable). It is often useful for the designer to classify the motor schema and the percept. For example, consider what happens if an implementation has a purely reflexive move-to-goal motor schema and an avoid-obstacle behavior. What happens if the avoid-obstacle behavior causes the robot to lose perception of the goal? Oops, the perceptual schema returns no goal and the move-to-goal behavior is terminated! Probably what the designer assumed was that the behavior would be a fixed-action pattern and thereby the robot would persist in moving toward the last known location of the goal.

Behavior Table

Releaser	Behavior	Motor Schema	Percept	Perceptual Schema
always on	follow-line()	stay-on-path(c_x)	c_x	compute-centroid(image,white)

As seen from the behavior table above, the CSM team initially proposed only one behavior, follow-line. The follow-line behavior consisted of a motor schema, stay-on-path(centroid), which was reflexive (stimulus-response) and taxis (it oriented the robot relative to the stimulus). The perceptual schema, compute-centroid(image,white), extracted an affordance of the centroid of white from the image as being the line. Only the x component, or horizontal location, of the centroid was used, c_x.

Step 5: Refine each behavior. By this point, the designer has an overall idea of the organization of the reactive system and what the activities are. This step concentrates on the design of each individual behavior. As the designer constructs the underlying algorithms for the motor and perceptual schemas, it is important to be sure to consider both the normal range of environmental conditions the robot is expected to operate in (e.g., the steady-state case) and when the behavior will fail.

The follow-line behavior was based on the analysis that the only white things in the environment were lines and plastic covered bales of hay. While this was a good assumption, it led to a humorous event during the second heat of the competition. As the robot was following the white line down the course, one of the judges

stepped into view of the camera. Unfortunately, the judge was wearing white shoes, and Omnibot turned in a direction roughly in-between the shoes and the line. The CSM team captain, Floyd Henning, realized what was happening and shouted at the judge to move. Too late, the robot's front wheels had already crossed over the line; its camera was now facing outside the line and there was no chance of recovering. Suddenly, right before the leftmost rear wheel was about to leave the boundary, Omnibot straightened up and began going parallel to the line! The path turned to the right, Omnibot crossed back into the path and re-acquired the line. She eventually went out of bounds on a hair pin further down the course. The crowd went wild, while the CSM team exchanged confused looks.

What happened to make Omnibot drive back in bounds? The perceptual schema was using the 20% brightest pixels in the image for computing the centroid. When it wandered onto the grass, Omnibot went straight because the reflection on the grass was largely random and the positions cancelled out, leaving the centroid always in the center of the image. The groundskeepers had cut the grass only in the areas where the path was to be. Next to the path was a parallel swatch of uncut grass loaded with dandelion seed puffs. The row of white puffs acted just as a white line, and once in viewing range Omnibot obligingly corrected her course to be parallel to them. It was sheer luck that the path curved so that when the dandelions ran out, Omnibot continued straight and intersected with the path. While Omnibot wasn't programmed to react to shoes and dandelions, it did react correctly considering its ecological niche.

Step 6: Test each behavior independently. As in any software engineering project, modules or behaviors are tested individually. Ideally, testing occurs in simulation prior to testing on the robot in its environment. Many commercially available robots such as Khepera and Nomads come with impressive simulators. However, it is important to remember that simulators often only model the mechanics of the robot, not the perceptual abilities. This is useful for confirming that the motor schema code is correct, but often the only way to verify the perceptual schema is to try it in the real world.

Step 7: Test with other behaviors. The final step of designing and implementing a reactive system is to perform integration testing, where the behaviors are combined. This also includes testing the behaviors in the actual environment.

Although the follow-line behavior worked well when tested with white lines, it did not perform well when tested with white lines and obstacles. The obstacles, shiny plastic-wrapped bales of hay sitting near the line, were often brighter than the grass. Therefore the perceptual schema for follow-line included pixels belonging to the bale in computing the centroid. Invariably the robot became fixated on the bale,

and followed its perimeter rather than the line. The bales were referred to as "visual distractions."

Fortunately, the bales were relatively small. If the robot could "close its eyes" for about two seconds and just drive in a straight line, it would stay mostly on course. This was called the move-ahead behavior. It used the direction of the robot (steering angle, dir) to essentially produce a uniform field. The issue became how to know when to ignore the vision input and deploy move-ahead.

The approach to the issue of when to ignore vision was to use the sonar as a releaser for move-ahead. The sonar was pointed at the line and whenever it returned a range reading, move-ahead took over for two seconds. Due to the difficulties in working with DOS, the CSM entry had to use a fixed schedule for all processes. It was easier and more reliable if every process ran every update cycle, even if the results were discarded. As a result the sonar releaser for move-ahead essentially inhibited follow-line, while the lack of a sonar releaser inhibited move-ahead. Both behaviors ran all the time, but only one had any influence on what the robot did. Fig. 5.6 shows this inhibition, while the new behavioral table is shown below.

New Behavior Table

Releaser	Inhibited by	Behavior	Motor Schema	Percept	Perceptual Schema
always on	near=read_sonar()	follow_line()	stay-on-path(c_x)	c_x	compute_centroid(image,white)
always on	far=read_sonar()	move_ahead(dir)	uniform(dir)	dir	dead_reckon(shaft-encoders)

The final version worked well enough for the CSM team to take first place. It went all the way around the track until about 10 yards from the finish line. The judges had placed a shallow sand pit to test the traction. The sand pit was of some concern since sand is a light color, and might be interpreted as part of the line. Since the sand was at ground level, the range reading could not be used as an inhibitor. In the end, the team decided that since the sand pit was only half the length of a bale, it wouldn't have enough effect on the robot to be worth changing the delicate schedule of existing processes.

The team was correct that the sand pit was too small to be a significant visual distraction. However, they forgot about the issue of traction. In order to get more traction, the team slipped real tires over the slick plastic wheels, but forgot to attach them. Once in the sand, the robot spun its wheels inside the tires. After the time limit was up, the team was permitted to nudge the robot along (done with a frustrated kick by the lead programmer) to see if it would have completed the entire course. Indeed it did. No other team made it as far as the sand pit.

It is clear that a reactive system was sufficient for this application. The use of primitive reactive behaviors was extremely computationally inexpensive,

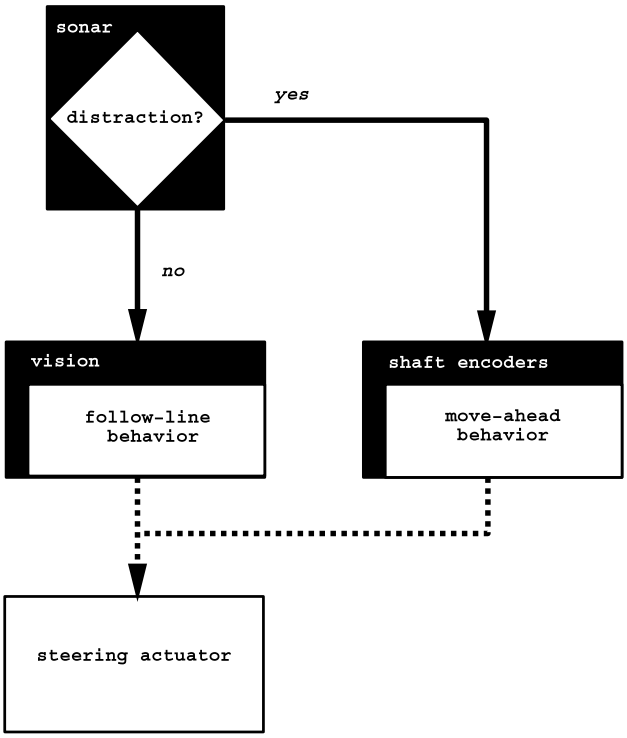


Figure 5.6 The behavioral layout of the CSM entry in the 1994 Unmanned Ground Vehicle Competition.

allowing the robot to update the actuators almost at the update rate of the vision framegrabber.

There are several important lessons that can be extracted from this case study:

- The CSM team evolved a robot which fit its ecological niche. However, it was a very narrow niche. The behaviors would not work for a similar domain, such as following sidewalks, or even a path of white lines with an intersection. Indeed, the robot reacted to unanticipated objects in the environment such as the judge's white shoes. The robot behaved "correctly" to features of the open world, but not in a desirable fashion.

- This example is a case where the releaser or inhibitor stimulus for a behavior was not the same perception as the percept needed to accomplish the task. The sonar was used to inhibit the behaviors. Follow-line used vision, while move-ahead integrated shaft encoder data to continue to move in the last good direction.

- This example also illustrates the tendency among purely reactive motor schema to assign one sensor per behavior.

5.5 Assemblages of Behaviors

The UGV competition case study illustrated the basic principles of the design of reactive behaviors. In that case, there were a trivial number of behaviors. What happens when there are several behaviors, some of which must run concurrently and some that run in a sequence? Clearly there are releasers somewhere in the system which determine the sequence. The issue is how to formally represent the releasers and their interactions into some sort of sequencing logic. If a set of behaviors form a prototypical pattern of action, they can be considered a "meta" or "macro" behavior, where a behavior is assembled from several other primitive behaviors into an *abstract behavior*. This raises the issue of how to encapsulate the set of behaviors and their sequencing logic into a separate module.

The latter issue of encapsulation is straightforward. The same OOP schema structure used to collect a perceptual schema and a motor schema into a behavior can be used to collect behaviors into an abstract behavior, as shown by a behavior being composed of behaviors in Fig. 5.1. The *coordinated control program* member of the abstract behavior expresses the releasers for the component behaviors.

This leaves the issue of how to formally represent the releasers in a way that both the robot can execute and the human designer can visualize and troubleshoot. There are three common ways of representing how a sequence of behaviors should unfold: *finite state automata*, *scripts* and *skills*. Finite state automata and scripts are logically equivalent, but result in slightly different ways about thinking about the implementation. Skills collect behavior-like primitives called Reaction-Action Packages (RAPs) into a "sketchy plan" which can be filled in as the robot executes. FSA-type of behavioral coordination and control were used successfully by the winning Georgia Tech team[19] in the 1994 AAAI Pick Up the Trash event, and the winning *LOLA* team in the 1995 IJCAI competition for the Pick Up the Trash event. Scripts were used by the Colorado School of Mines team in the 1995 competition; that entry performed behaviorally as well as the winning teams but did not place due to a penalty for not having a manipulator. Those three teams used at most eight behaviors, even though LOLA had a more sophisticated gripper than the Georgia Tech team. In contrast, *CHIP* the second place team in the IJCAI

SKILLS

a. b.

Figure 5.7 Two award-winning Pick Up the Trash competitors: a.) Georgia Tech's robots with the trash gripper (photograph courtesy of Tucker Balch and AAAI), and b.) University of North Carolina's LOLA (photograph courtesy of Rich LeGrand and AAAI).

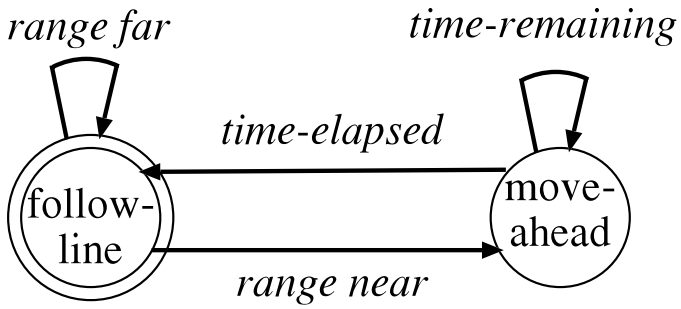
competition required 34 skills and 80 RAPs to do the same task, in part because of the complexity of the arm.[53;54] Since in general skills lead to a more complex implementation than FSA and scripts, they will not be covered here. The most important point to remember in assembling behaviors is to try to make the world trigger, or *release*, the next step in the sequence, rather than rely on an internal model of what the robot has done recently.

5.5.1 Finite state automata

FINITE STATE AUTOMATA

STATE DIAGRAM

Finite state automata (FSA) are a set of popular mechanisms for specifying what a program should be doing at a given time or circumstance. The FSA can be written as a table or drawn as a *state diagram*, giving the designer a visual representation. Most designers do both. There are many variants of



a.

$M : K = \{\text{follow-line, move-ahead}\}, \Sigma = \{\text{range near, range far}\},$
$s =$follow-line, $F = \{\text{follow-line, move-ahead}\}$

q	σ	$\delta(q, \sigma)$
follow-line	range near	move-ahead
follow-line	range far	follow-line
move-ahead	time remaining	move-ahead
move-ahead	time elapsed	follow-line

b.

Figure 5.8 A FSA representation of the coordination and control of behaviors in the UGV competition: a.) a diagram and b.) the table.

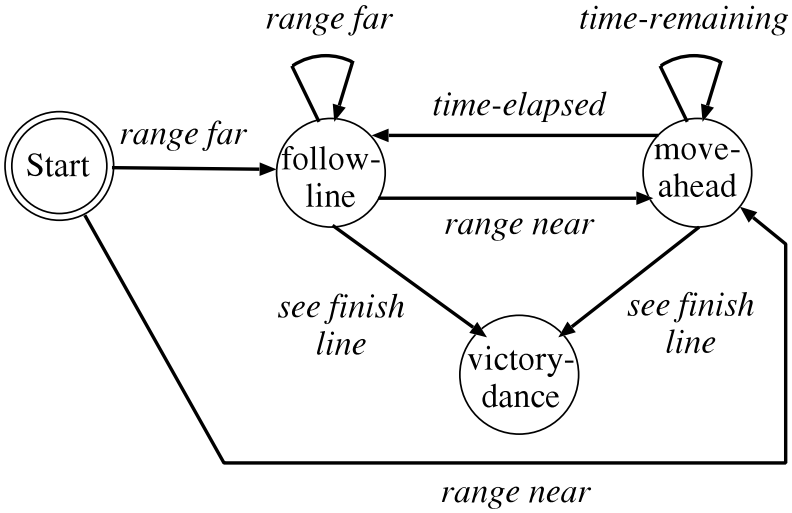
FSA, but each works about the same way. This section follows the notation used in algorithm development.[86]

STATES

To begin with, the designer has to be able to specify a finite number of discrete *states* that the robot should be in. The set of states is represented by K, and each state, $q \in K$ is a listing of the behaviors that should be active at the same time. In the case of the UGR competition, there were only two states: following the line and moving forward. States are represented in a table under the heading q, and by circles in a graph. (See Fig. 5.8.) By convention, there is always a *Start state*, and the robot would always start there. The Start state is often written as s or q_o and drawn with a double circle. In the case of the UGR entry, the following-line state was the start state since the robot always starts with the follow-line behavior active and not suppressed.

START STATE

The next part of the FSA is the inputs (also called the alphabet). Inputs are the behavioral releasers, and appear under the column heading σ. Unlike the IRM diagrams, the FSM table considers what happens to each state q for



a.

M: $K = \{$follow-line, move-ahead$\}$, $\Sigma = \{$range near, range far$\}$,
$s =$follow-line, $F = \{$follow-line,move-ahead$\}$

q	σ	$\delta(q,\sigma)$
follow-line	range near	move-ahead
follow-line	range far	follow-line
move-ahead	time remaining	move-ahead
move-ahead	time elapsed	follow-line

b.

Figure 5.9 An alternative FSA representation of the coordination and control of behaviors in the UGV competition: a.) a diagram and b.) the table.

all possible inputs. As shown in Fig. 5.8, there are only two releasers for the UGR example, so the table doesn't have many rows.

TRANSITION FUNCTION

The third part of the FSM is the *transition function*, called δ, which specifies what state the robot is in after it encounters an input stimulus, σ. The set of stimulus or affordances σ that can be recognized by the robot is represented by Σ (a capital σ). These stimuli are represented by arrows. Each arrow represents the *releaser* for a behavior. The new behavior triggered by the releaser depends on the state the robot is in. This is the same as with Innate Releasing Mechanisms, where the animal literally ignores releasers that aren't relevant to its current state.

Recall also in the serial program implementations of IRMs in Ch. 3 that the agent "noticed" releasers every second. At one iteration through the loop, it might be hungry and "enter" the state of feeding. In the next iteration, it might still be hungry and re-enter the state of feeding. This can be represented by having arrows starting at the feeding state and pointing back to the feeding state.

For the example in Fig. 5.8, the robot starts in the state of following a line. This means that the robot is not prepared to handle a visual distraction (range near) until it has started following a line. If it does, the program may fail because the FSA clearly shows it won't respond to the distraction for at least one update cycle. By that time, the robot may be heading in the wrong direction. Starting in the following-line state fine for the UGR competition, where it was known in advance that there were no bales of hay near the starting line. A more general case is shown in Fig. 5.9, where the robot can start either on a clear path or in the presence of a bale. The FSA doesn't make it clear that if the robot starts by a bale, it better be pointed straight down the path!

The fourth piece of information that a designer needs to know is when the robot has completed the task. Each state that the robot can reach that terminates the task is a member of the set of *final state*, F. In the UGR example, the robot is never done and there is no final state—the robot runs until it is turned off manually or runs out of power. Thus, both states are final states. If the robot could recognize the finish line, then it could have a finish state. The finish state could just be stopped or it could be another behavior, such as a victory wag of the camera. Notice that this adds more rows to the FSA table, since there must be one row for each unique state.

FINAL STATE

In many regards, the FSA table is an extension of the behavioral table. The resulting table is known as a finite state machine and abbreviated M for machine. The notation:

$$M = \{K, \Sigma, \delta, s, F\}$$

is used as reminder that in order to use a FSA, the designer must know all the q states (K), the inputs σ the transitions between the states δ, the special starting state q_0, and the terminating state(s) (F). There must be one arrow in the state diagram for every row in the table. The table below summarizes the relationship of FSA to behaviors:

FSA	Behavioral analog
K	all the behaviors for a task
Σ	the releasers for each behavior in K
δ	the function that computes the new state
s	the behavior the robot starts in when turned on
F	the behavior(s) the robot must reach to terminate

In more complex domains, robots need to avoid obstacles (especially people). Avoidance should always be active, so it is often implicit in the FSA. move-to-goal often is shorthand for move-to-goal *and* avoid. This implicit grouping of "interesting task-related behaviors" and "those other behaviors which protect the robot" will be revisited in Ch. 7 as strategic and tactical behaviors.

Another important point about using FSA is that they describe the overall behavior of a system, but the implementation may vary. Fig. 5.8 is an accurate description of the state changes in the UGV entry. The control for the behaviors could have been implemented exactly as indicated by the FSA: if following-line is active and range returns range near, then move-ahead. However, due to timing considerations, the entry was programmed differently, though with the same result. The following examples will show how the FSA concept can be implemented with subsumption and schema-theoretic systems.

5.5.2 A Pick Up the Trash FSA

As another example of how to construct and apply an FSA, consider the Pick Up the Trash task. Assume that the robot is a small vehicle, such as the ones used by Georgia Tech shown in Fig. 5.7 or the Pioneer shown in Fig. 5.10, with a camera, and a bumper switch to tell when the robot has collided with something. In either case, the robot has a simple gripper. Assume that the robot can tell if the gripper is empty or full. One way to do this is to put an IR sensor across the jaw of the gripper. When the IR returns a short range, the gripper is full and it can immediately close, with a grasping reflex. One problem with grippers is that they are not as good as a human hand. As such, there is always the possibility that the can will slip or fall out of the gripper. Therefore the robot should respond appropriately when it is carrying a can or trash and loses it.

The Pick Up the Trash environment is visually simple, and there are obvious affordances. Coca-cola cans are the only red objects in the environment,

and trash cans are the only blue objects. Therefore visually extracting red and blue blobs should be sufficient. All objects are on the floor, so the robot only has to worry about where the objects are in the x axis. A basic scenario is for the robot to start wandering around the arena looking for red blobs. It should head straight for the center of the largest red blob until it scoops the can in the forklift. Then it should try three times to grab the can, and if successful it should begin to wander around looking for a blue blob. There should only be one blue blob in the image at a time because the two trash cans are placed in diagonal corners of the arena. Once it sees a blue blob, the robot should move straight to the center of the blob until the blob gets a certain size in the image (looming). The robot should stop, let go of the can, turn around to a random direction and resume the cycle. The robot should avoid obstacles, so moving to a red or blue blob should be a fixed pattern action, rather than have the robot immediately forget where it was heading.

The behavior table is:

Releaser	Behavior	Motor Schema	Percept	Perceptual Schema
always on	avoid()	pfields.nat(goal_dir)	bumper_on	read_bumper()
EMPTY=gripper_status()	wander()	pfields.random()	time-remaining	countdown()
EMPTY=gripper_status() AND SEE_RED=extract_color(red)	move-to-goal(red)	pfields.attraction(c_x)	c_x	extract-color(red, c_x)
FULL=gripper_status()	grab-trash()	close_gripper()	status	gripper_status()
FULL=gripper_status() AND NO_BLUE=extract_color(blue)	wander()	pfields.random()	time_remaining	countdown()
FULL=gripper_status() AND SEE_BLUE=extract_color(blue)	move-to-goal(blue)	pfields.attraction(c_x)	c_x	extract-color(blue)
FULL=gripper_status() AND AT_BLUE=looming(blue, size=N)	drop_trash()	open_gripper() turn_new_dir(curr_dir)	curr_dir	read_encoders()

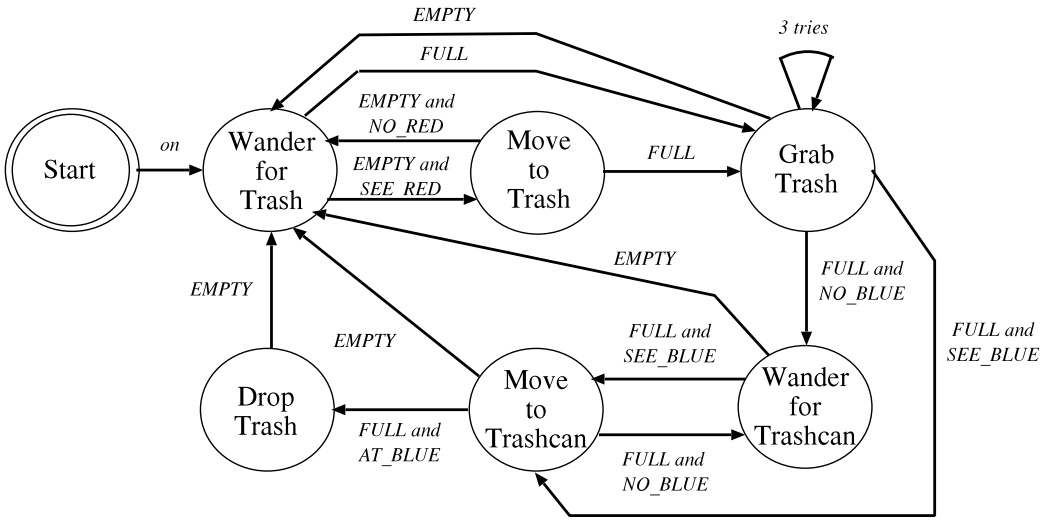
The function calls in the table only show the relevant arguments for brevity. The avoid behavior is interesting. The robot backs up either to the right or left (using a NaT) when it bumps something. It may bump an arena wall at several locations, but eventually a new wander direction will be set. If the robot bumps a can (as opposed to captures it in its gripper), backing up gives the robot a second chance. This table shows that the design relies on the gripper to maintain where the robot is in the nominal sequence. An empty gripper means the robot should be in the collecting the trash phase, either looking for a can or moving toward one. A full gripper means the robot is in the deposit phase. The looming releaser extracts the size of the blue region in pixels and compares it to the size N. If the region is greater than or equal

Figure 5.10 A Pioneer P2-AT with a forklift arm suitable for picking up soda cans. (Photograph courtesy of ActivMedia, Incorporated.)

to N, then the robot is "close enough" to the trash can and the robot can drop the can.

There are two problems with the behavior table. The first is that it doesn't show the sequence, or flow of control, clearly. The second is how did the designer come up with those behaviors? This is where a FSA is particularly helpful. It allows the designer to tinker with the sequence and represent the behavioral design graphically.

Fig. 5.11 shows a FSA that is equivalent to the behavior table. The FSA may be clearer because it expresses the sequence. It does so at the cost of not showing precisely how the sequence would be implemented and encouraging the designer to create internal states. A programmer might implement two wander behaviors, one which is instantiated by different releasers and terminates under different conditions, and two move-to-goal behaviors. Many designers draw and interpret FSA as carrying forward previous releasers. For example, the correct transition from Grab Trash to Wander For Trash can is FULL and NO_BLUE, but a designer may be tempted to label the arrow as only NO_BLUE, since to get that state, the gripper had to be FULL. This is a very dangerous mistake because it assumes that the implementation will be keeping up with what internal state the robot is in (by setting a vari-



a.

$K = \{$wander for trash, move to trash, grab trash, wander for trash can, move to trash can, drop trash $\}$, $\Sigma = \{$on, EMPTY, FULL, SEE_RED, NO_BLUE, SEE_BLUE, AT_BLUE$\}$, $s = Start$, $F = K$

q	σ	$\delta(q, \sigma)$
start	on	wander for trash
wander for trash	EMPTY, SEE_RED	move to trash
wander for trash	FULL	grab trash
move to trash	FULL	grab trash
move to trash	EMPTY, NO_RED	wander for trash
grab trash	FULL, NO_BLUE	wander for trash can
grab trash	FULL, SEE_BLUE	move to trash can
grab trash	EMPTY	wander for trash
wander for trash can	EMPTY	wander for trash
wander for trash can	FULL, SEE_BLUE	move to trash can
move to trash can	EMPTY	wander for trash
move to trash can	FULL, AT_BLUE	drop trash
drop trash	EMPTY	wander for trash

b.

Figure 5.11 A FSA for picking up and recycling Coke cans: a.) state diagram, and b.) table showing state transitions.

able), instead of letting directly perceivable attributes of the world inform the robot as to what state it is in. Internal state is incompatible with reactivity.

The FSA also hid the role of the avoid behavior. The avoid behavior was always running, while the other behaviors were asynchronously being instantiated and de-instantiated. It is difficult to show behavioral concurrency with an FSA. Other techniques, most especially Petri Nets, are used in software engineering but have not been commonly used in robotics. The avoid behavior was not a problem in this case. It was always running and the output of the avoid potential field vector could be summed with the output of whatever other behavior was active.

5.5.3 Implementation examples

In a schema-theoretic implementation, the FSA logic would exist in one of two places. If the robot's sole task was to recycle soda cans, the controlling logic could be placed in the main program. If the robot had many tasks that it could do, the ability to recycle trash would be an *abstract behavior*, called by the main program whenever the robot needed to recycle trash. In that case, the FSA logic would be placed in the coordinated control program slot of the behavior schema.

Although the current discussion is on where the FSA goes, it might be useful to spend some time on the overall implementation. While the wander-to-goal and move-to-goal behaviors can be easily implemented with a potential fields methodology, drop-trash cannot. Drop-trash really isn't a navigation behavior. It fits the overall profile of a behavioral schema: it has an obvious motor schema (open the gripper, turn the wheels), a perceptual schema (read gripper encoders and wheel encoders), a coordinated control program (open THEN turn), and a releaser (at trash can). While schema-theoretic implementations use potential field methodologies and vector summation for effector control, not every behavior will generate a vector based on a potential field.

One advantage of FSA is that they are abstract, and can be implemented in a number of ways. The behavior table illustrated one way the FSA could be implemented with a schema-theoretic system. Fig. 5.12 shows one way it could be implemented in subsumption. This example shows the power of inhibition and suppression which is not well represented by FSA state diagrams.

In keeping with the idea of modularity and incrementally adding behaviors, the system starts with an explicit avoid behavior running on top of Level 0 (not shown). At the next level the robot wanders until it sees red. Then

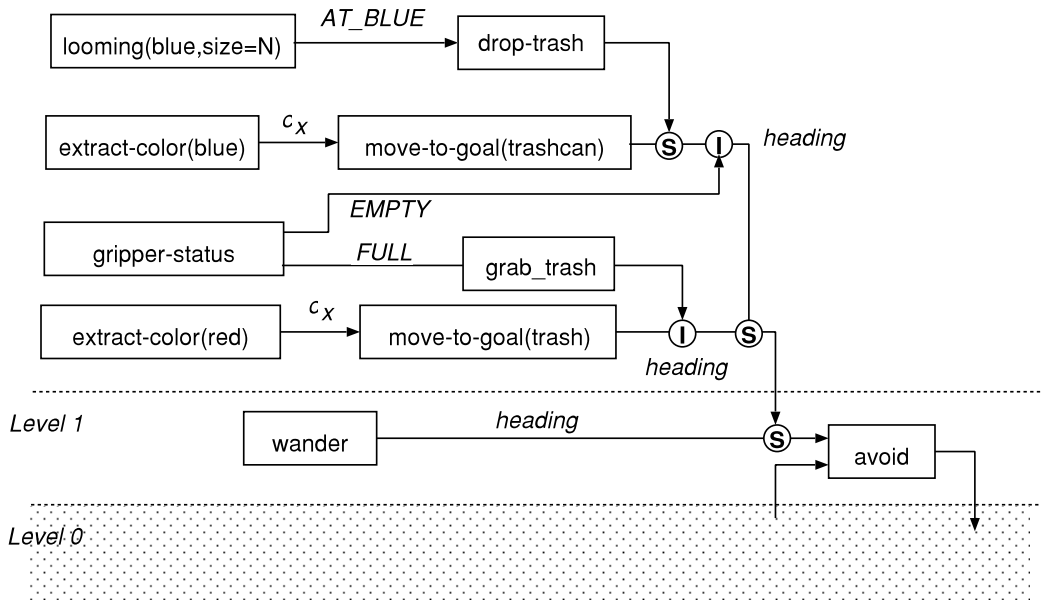


Figure 5.12 One possible Pick Up the Trash abstract behavior using subsumption.

move-to-trash suppresses wandering and replaces the heading with the direction to the trash. The move-to-trash behavior continues until the can is in the gripper. Once the gripper is full, the gripper closes and grabs the trash. Then the move-to-trash behavior is inhibited from executing. This prevents move-to-trash from sending any directions out, so the wandering direction is now again the output.

The next level of competency is to deposit trash in the trash can. When it sees the blue trash can, move-to-trash can begins to suppress wander and direct the robot to the trash can. If the gripper is empty, the output for move-to-trash can is inhibited. The robot is simultaneously looking for red and blue blobs, but as long as the gripper is empty, it only responds to red blobs.

Dropping trash also is constantly executing. If the robot happens to wander next to a blue trash can, it will signal to drop trash and turn around. The new heading suppresses any heading direction from move-to-trash can. But the robot will not open its gripper and turn around if the gripper is empty, because empty inhibits that whole line. The subsumption example produces a much less complex system than a direct FSA mapping.

5.5.4 Abstract behaviors

Finite state automata are a useful tool for expressing the coordinated control program of an abstract behavior. They fall short as a programming tool for abstract behaviors in a number of ways. First, in many cases, the assemblage of behaviors represents a prototypical sequence of events that should be slightly adapted to different situations, in essence, a *template* or *abstract behavior*. In the Pick Up the Trash event, recycling Coke cans was only part of the task; the robots were also supposed to find white Styrofoam cups and deposit them in yellow trash cans. The behaviors represented by the FSA could be collected into an abstract behavior: `pick-up-the-trash(trash-color, trash can-color, size-trash can)`.

TEMPLATE
ABSTRACT BEHAVIOR

Second, templates need to handle different initialization conditions. Initialization wasn't a big problem for the Pick Up the Trash task, but it could be one for other applications. For example, the emergent behavior of the robot described in the Unmanned Ground Vehicle competition in Sec. 5.4 could be described as an abstract "follow-path" behavior. Recall that the robot's program assumed that it started facing the line. A more general purpose, reusable follow-path behavior would need to handle a broader range of starting conditions, such as starting facing a bale or not perfectly lined up with the line. Another common initialization behavior is *imprinting*, where the robot is presented with an object and then records the perceived color (or other attribute) of the object for use with the nominal behavior. In the Pick Up the Trash competitions, several teams literally showed the robot the Coke can and let it determine the best values of "red" for the current lighting conditions. Likewise, some abstract behaviors would need special termination behavior. In the case of the UGR competition, the termination behaviors was NULL, but it could have been the victory dance.

IMPRINTING

Third, some times robots fail in their task; these events are often called *exceptions*. An exception might be when the robot does not pick up a soda can in over 10 minutes. Another behavior can be substituted (do a raster scan rather than a random wander) or alternative set of parameters (use different values for red).

EXCEPTIONS

5.5.5 Scripts

Abstract behaviors often use *scripts*,[49;50;87;100] or a related construct called *skills*,[53;54] to create generic templates of assemblages of behaviors. Scripts provide a different way of generating the logic for an assemblage of behav-

SCRIPTS
SKILLS

Script	*Collection of Behaviors*	*Example*
Goal	Task	pick up and throw away a Coca-Cola can
Places	Applicability	an empty arena
Actors	Behaviors	WANDER_FOR_GOAL, MOVE_TO_GOAL, GRAB_TRASH, DROP_TRASH
Props, Cues	Percepts	red, blue
Causal Chain	Sequence of Behaviors	WANDER_FOR_GOAL(TRASH), MOVE_TO_GOAL(TRASH), GRAB_TRASH,WANDER_FOR_GOAL(TRASH CAN), MOVE_TO_GOAL(TRASH CAN), DROP_TRASH
Subscripts	Exception Handling	if have trash and drop, try GRAB_TRASH three times

Figure 5.13 Comparison of script structures to behaviors.

iors. They encourage the designer to think of the robot and the task literally in terms of a screenplay. Scripts were originally used in natural language processing (NLP) to help the audience (a computer) understand actors (people talking to the computer or writing summaries of what they did).[123] In the case of robots, scripts can be used more literally, where the actors are robots reading the script. The script has more room for improvization though, if the robot encounters an unexpected condition (an exception), the robot begins following a *sub-script*.

SUB-SCRIPT

Fig. 5.13 shows how elements of an actor's script compares to a robot script. The main sequence of events is called a *causal chain*. The causal chain is critical, because it embodies the coordination control program logic just as a FSA does. It can be implemented in the same way. In NLP, scripts allow the computer to keep up with a conversation that may be abbreviated. For example, consider a computer trying to read and translate a book where the main character has stopped in a restaurant. Good writers often eliminate all the details of an event to concentrate on the ones that matter. This missing, but implied, information is easy to extract. Suppose the book started with "John ordered lobster." This is a clue that serves as an index into the current or relevant event of the script (eating at a restaurant), skipping over past events (John arrived at the restaurant, John got a menu, etc.). They also focus the system's attention on the next likely event (look for a phrase that indicates John has placed an order), so the computer can instantiate the function which looks for this event. If the next sentence is "Armand brought out the lobster and refilled the white wine," the computer can infer that Armand is the waiter and that John had previously ordered and received white wine, without having been explicitly told.

CAUSAL CHAIN

INDEXING
FOCUS-OF-ATTENTION

In programming robots, people often like to abbreviate the routine portions of control and concentrate on representing and debugging the important sequence of events. Finite state automata force the designer to consider and enumerate every possible transition, while scripts simplify the specification. The concepts of *indexing* and *focus-of-attention* are extremely valuable for coordinating behaviors in robots in an efficient and intuitive manner. Effective implementations require asynchronous processing, so the implementation is beyond the scope of this book. For example, suppose a Pick Up the Trash robot boots up. The first action on the causal chain is to look for the Coke cans. The designer though realizes that this behavior could generate a random direction and move the robot, missing a can right in front of it. Therefore, the designer wants the code to permit the robot to skip searching the arena if it immediately sees a Coke can, and begin to pick up the can without even calling the wander-for-goal(red) behavior. The designer also knows that the next releaser after grab-trash exits to look for is blue, because the cue for moving to the trash can and dropping off trash is blue.

The resulting script for an abstract behavior to accomplish a task is usually the same as the programming logic derived from an FSA. In the case of Pick Up the Trash, the script might look like:

```
for each update...
\\ look for props and cues first: cans, trash cans, gripper
rStatus=extract_color(red, rcx, rSize); \\ ignore rSize
if (rStatus==TRUE)
  SEE_RED=TRUE;
else
  SEE_RED=FALSE;
bStatus=extract_color(blue, bcx, bSize);
if (bStatus==TRUE){
  SEE_BLUE=TRUE; NO_BLUE=FALSE;
} else {
  SEE_BLUE=FALSE; NO_BLUE=TRUE;
}
AT_BLUE=looming(size, bSize);
gStatus=gripper_status();
if (gStatus==TRUE) {
  FULL=TRUE; EMPTY=FALSE;
} else {
  FULL=FALSE; EMPTY=TRUE;
}
```

```
\\index into the correct step in the causal chain
if (EMPTY){
  if (SEE_RED){
    move_to_goal(red);
  else
    wander();
} else{
  grab_trash();
  if (NO_BLUE)
     wander();
  else if (AT_BLUE)
     drop_trash();
  else if (SEE_BLUE)
     move_to_goal(blue);
}
```

5.6 Summary

As defined in Ch. 4, a reactive implementation consists of one or more behaviors, and a mechanism for combining the output of concurrent behaviors. While an architectural style (subsumption, potential fields) may specify the structure of the implementation, the designer must invest significant effort into developing individual behaviors and into assembling them into a sequence or an abstract behavior.

Schema theory is highly compatible with Object Oriented Programming. A behavior is derived from the schema class; it is a schema that uses at least one perceptual and motor schema. If a behavior is composed of multiple schema, it must have a coordinated control program to coordinate them. Finite State Automata offer a formal representation of the coordination logic needed to control a sequence of behaviors. Scripts are an equivalent mechanism with a more natural story-like flow of control.

The steps in designing robot intelligence under the Reactive Paradigm are:

1. Describe the task,
2. Describe the robot,
3. Describe the environment,
4. Describe how the robot should act in response to its environment,

5. Refine each behavior,

6. Test each behavior independently,

7. Test with other behaviors,

and repeat the process as needed. A *behavior table* serves as a means of representing the component schemas and functions of the behavioral system. For each behavior, it shows the releasers, the motor schemas, the percept, and the perceptual schemas.

These steps emphasize the need to fully specify the ecological niche of the robot in order to design useful behaviors. Since the idea of behaviors in the Reactive Paradigm is derived from biology, it is not strange that the idea of a robot being evolved to fit its environment should also be part and parcel of the design process. Regardless of the implementation of the coordination program , the control should rely on the world to inform the robot as to what to do next, rather than rely on the program to remember and maintain state.

5.7 Exercises

Exercise 5.1

What is the difference between a *primitive* and an *abstract* behavior?

Exercise 5.2

Define:

a. behavior table

b. causal chain

c. coordinated control program†

Exercise 5.3

Can the perceptual schema and the motor schema for a behavior execute asynchronously, that is, have different update rates?

Exercise 5.4

Fig. 5.2 shows two methods of implementing a potential fields-based follow-corridor behavior. A third way is to have two instances of a move-away-from-wall (perpendicular) behavior with a move-parallel-to-wall behavior. What are the advantages and disadvantages of such an implementation?

Exercise 5.5

List and describe the steps in designing a reactive system.

Exercise 5.6

Consider the Pick Up the Trash example in Sec. 5.5.2. The example assumed the arena was empty except for walls, cans, and trash cans. What would happen if there were chairs and tables present? Could the gripper accidently scoop a chair or table leg? How would the system react? What changes, if any, would need to be made to the behavior table and FSA?

Exercise 5.7

Solve the 1994 International Association for Unmanned Systems Unmanned Ground Vehicle problem using STRIPS (Ch. 2). Be sure to include a world model and a difference table.†

Exercise 5.8

Solve the 1994 AAAI Pick Up the Trash problem using STRIPS (Ch. 2). Be sure to include a world model and a difference table.

Exercise 5.9

How is defining a robot's behaviors linked to the robot's ecological niche?†

Exercise 5.10

What is special about a primitive behavior in terms of perceptual and motor schema?†

Exercise 5.11

Construct a Behavioral Table of the behaviors needed for a robot to move from room to room.†

Exercise 5.12

What are the two main deficits encountered when designing and implementing reactive robotic systems?†

Exercise 5.13

Make up a task for a robot to complete. Describe what each of the 7 steps of the design methodology would require you to do to complete the task.†

Exercise 5.14

Describe the two methods for assembling primitive behaviors into abstract behaviors: finite state automata and scripts.†

Exercise 5.15

Assume that the CSM robot had been wider and needed to use an avoid-obstacle behavior. Make a Behavior Table to include this new behavior.†

Exercise 5.16

Assuming the state of the robot in question 1, describe how the coordinated control program should handle concurrent behaviors.†

Exercise 5.17

Recall how some mammals freeze when they see motion (an affordance for a predator) in an attempt to become invisible, and persist in this behavior until the predator is very close, then flee. Define this behavior using Behavior Tables.†

Exercise 5.18

Suppose the competition course described in Section 5.4 has been modified so that a hay bale can be placed directly on the white line. The robot must move around the bale and rejoin the line on the other side. Create a behavior table that not only follows the line but correctly responds to the bale so that it can continue following the line.†

Exercise 5.19 [*World Wide Web*]

Search the web for the International Association for Unmanned Systems competition home page and briefly describe three unmanned vehicle projects linked to the IAUS site.†

Exercise 5.20 [*World Wide Web*]

Identify at least 5 robot competitions, and for each describe the basic task. Can the task be accomplished with a reactive robot? Why or why not?

Exercise 5.21 [*Programming*]

Using Rug Warrior kits, Lego Mindstorms kits, or similar robotics tools, implement your own schools version of an IAUS Unmanned Ground Robotics Competition. Your class should decide on the course, the course objectives, rules, and prizes (if any). Groups should not begin construction of their robot without first describing the steps involved in designing a reactive behavioral system for the task at hand.†

Exercise 5.22 [*Programming*]

Write a script in pseudo-code for following a hallway. Consider that the robot may start in a hallway intersection facing away from the hallway it is supposed to take. Also the robot might start in the middle of a hall facing a wall, rather than having forward point to the long axis of the hall.

Exercise 5.23 [*Programming*]

The Pick Up the Trash competitions were a bit different than actually presented in this book. For example, the robots were actually permitted to cache up to three cans before going to the trash can, and the robots could go after white Styrofoam cups as trash. How would you integrate this into:

a. the FSA and

b. the script described in the book?

Exercise 5.24 [*Programming*]

Could the exception handling sub-scripts for picking up trash be implemented with the exception handling functionality provided by Ada or C++?

Exercise 5.25 [*Advanced Reading*]

Read Rodney Brooks' one paragraph introduction to Chapter 1 in *Cambrian Intelligence*[28] on the lack of mathematics in behavioral robotics. Now consider the behavior of the CSM robot around white shoes and dandelions. Certainly it would be useful to have theories which can prove when a behavioral system will fail. Is it possible?

Questions marked with a † were suggested by students from the University of Wisconsin Lacrosse.

5.8 End Notes

For the roboticist's bookshelf.
Artificial Intelligence and Mobile Robots[14] is a collection of case studies from top mobile robotics researchers, detailing successful entries in various robot competitions. The book provides many insights into putting theory into practice.

The IAUS Ground Robotics Competition.
The International Association for Unmanned Systems (formerly known as the Association for Unmanned Vehicle Systems) sponsors three annual robot competitions for student teams: Ground Robotics, Aerial Vehicle, and Underwater. The Ground Robotics competition is the oldest, having started in 1992. As noted in a Robotics Competition Corner column,[101] the competitions tend to reward teams for building platforms rather than programming them.

Reusability principle of software engineering and beer cans.
While researchers worried about picking up and disposing of soda cans, a popular magazine began discussing what it would take to have a robot go to the refrigerator and bring the user a beer. In an apparent acknowledgment of the lifestyle of someone who would have such a robot, the robot was expected to be able to recognize and avoid dirty underwear lying at random locations on the floor.

The 1995 IJCAI Mobile Robot Competition.
The competition was held in Montreal, Canada, and many competitors experienced delays shipping their robots due to delays at Customs. The University of New Mexico team spent the preparatory days of the competition frantically trying to locate where their robot was. Almost at midnight the evening before the preliminary rounds were to begin, a forklift drove up with the crate for their robot. All the teams looked up from their frantic testing and cheered in support of the New Mexicans. Until the fork lift came close enough and everyone could clearly see that the "This Side Up" arrow was pointing down... The New Mexicans didn't even bother to un-crate the robot to catalog the damage; they instead sought solace in Montreal's wonderful night life.

The teams with robots properly aligned with gravity went back to programming and fine tuning their entries.

Cheering at robot competitions.
The chant "Core Dump. Core Dump. Segmentation Fault!" has a nice cadence and is especially appropriate to yell at competitors using Unix systems.

Subsumption and Soda Cans.
Jon Connell addressed this task in 1989 in his thesis work at MIT,[39] applying subsumption to a robot arm, not a set of flappers. He used a special type of FSM called an Augmented Finite State Machines (AFSMs), and over 40 behaviors to accomplish the task.

Being in a Pick Up the Trash competition without a manipulator.
As often happens, robot competitions often pose problems that are a step beyond the capability of current hardware and software technology. In 1995, arms on mobile robots were rare; indeed Nomad introduced a forklift arm just in time for the competition. Participants, such as the Colorado School of Mines with an older robot and no arm, could have a "virtual manipulator" with a point deduction. The robot would move to within an agreed tolerance of the object, then either play a sound file or make a noise. The virtual manipulator—a human team member, either Tyler Devore, Dale Hawkins, or Jake Sprouse—would then physically either pick up the trash and place it on the robot or remove the trash. It made for an odd reversal of roles: the robot appeared to be the master, and the student, the servant!

About grippers maintaining the state of world.
The Pick Up the Trash event mutated in 1996 to picking up tennis balls in an empty arena, and in 1997 into a variation on sampling Mars. For the 1997 Find Life on Mars event, the sponsors brought in real rocks, painted black to contrast with the gray concrete floor and blue, green, red, and yellow "martians" or toy blocks. Because of weight considerations in shipping the rocks, the rocks were about the size of a couple of textbooks and not that much bigger than the martians. One team's purely reactive robot had trouble distinguishing colors during preliminary rounds. It would misidentify a rock as a martian during a random search, navigate to it, grip it, and then attempt to lift it. Since the rock was heavy, the gripper could not reach the full extension and trigger the next behavior. It would stay there, clutching the rock. Sometimes, the robot would grip a rock and the gripper would slip. The robot would then try to grip the rock two more times. Each time it would slip. The robot would give up, then resume a random search. Unfortunately, the search seemed to invariably direct the robot back to the same rock, where the cycle would repeat itself. Eventually the judges would go over and move the robot to another location.

Documentaries.
Scientific American Frontiers did an excellent special on robot competitions called "Robots Alive!" The special covered the AUVS Aerial Vehicle Competition (take away lesson: try your robot outdoors before you show up at an outdoor robot competition) and the 1996 AAAI Mobile Robot Competition where the robots picked up orange tennis balls instead of coca-cola cans.