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.

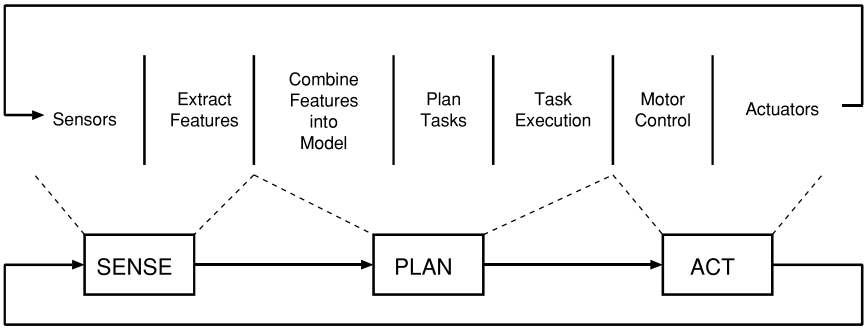

Figure 4.1 Horizontal decomposition of tasks into the S,P,A organization of the Hierarchical Paradigm.

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.

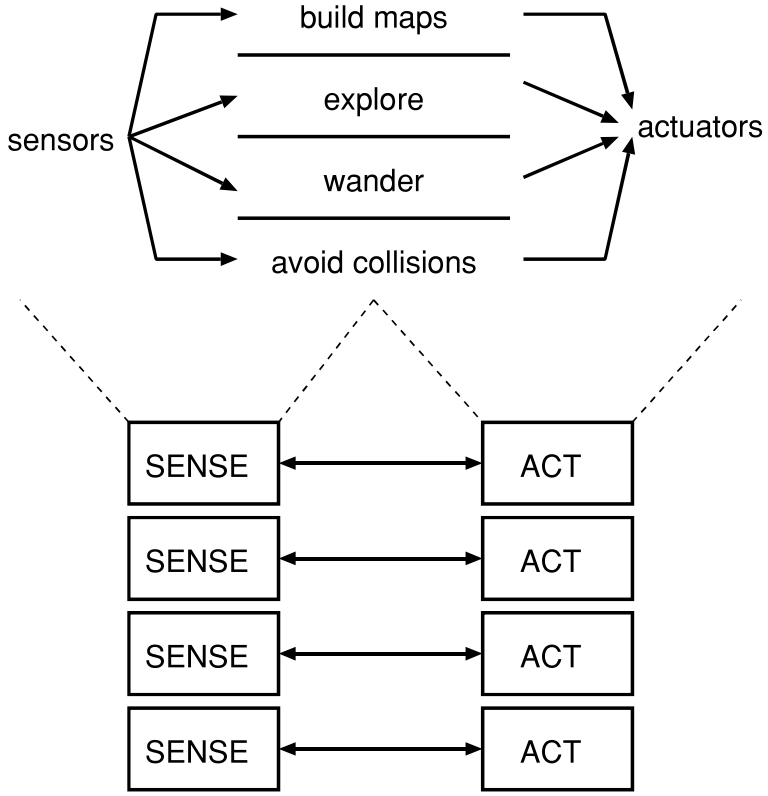

Figure 4.2 Vertical decomposition of tasks into an S-A organization, associated with the Reactive Paradigm.

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-

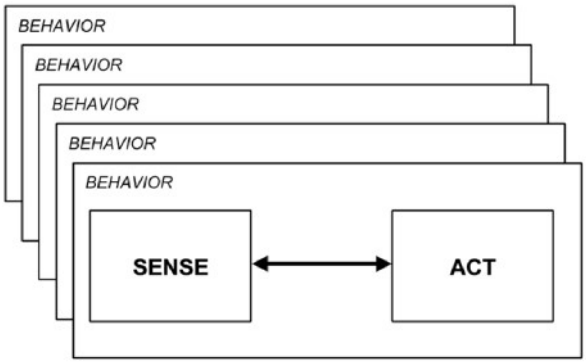

Figure 4.3 S-A organization of the Reactive Paradigm into multiple, concurrent behaviors.

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.

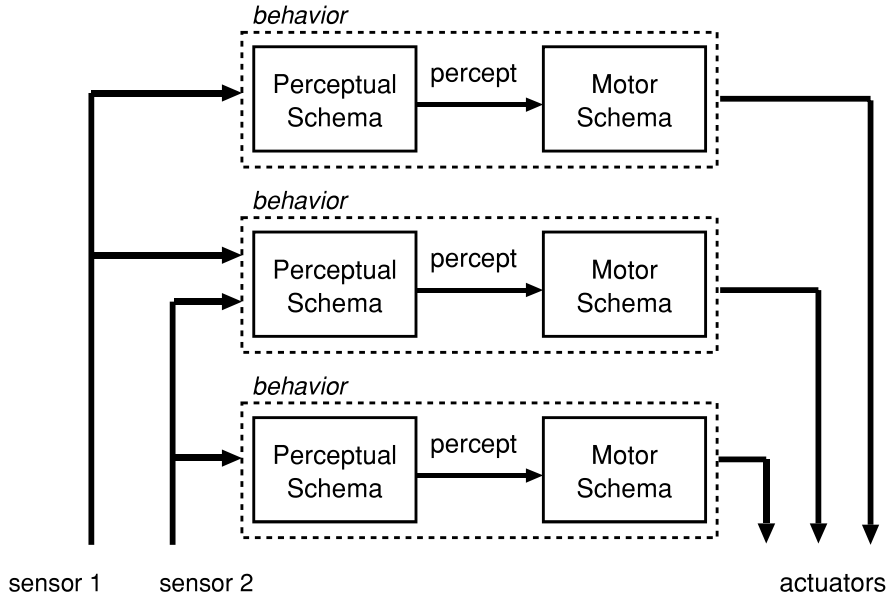

Figure 4.4 Behavior-specific sensing organization in the Reactive Paradigm: sensing is local, sensors can be shared, and sensors can be fused locally by a behavior.

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

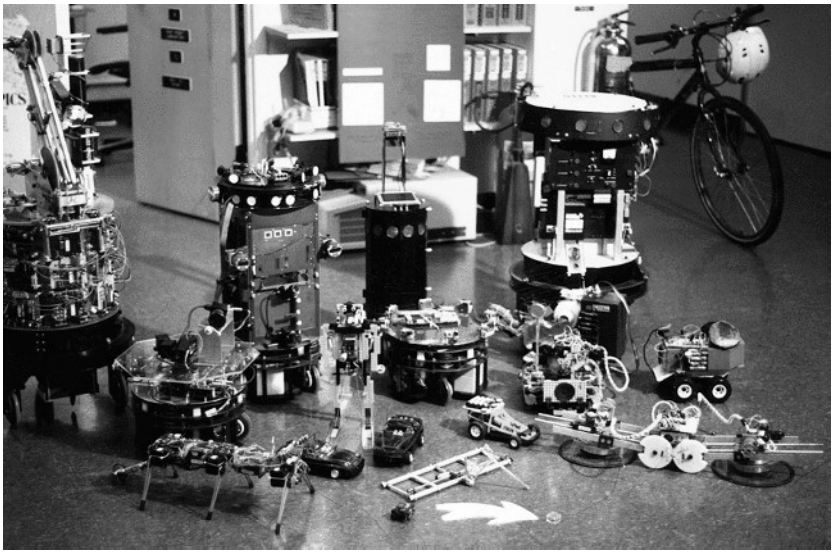

Figure 4.5 "Veteran" robots of the MIT AI Laboratory using the subsumption architecture. (Photograph courtesy of the MIT Artificial Intelligence Laboratory.)

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

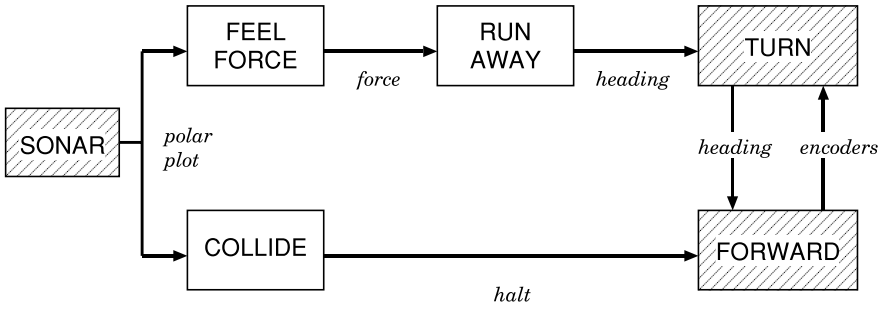

Figure 4.6 Level 0 in the subsumption architecture.

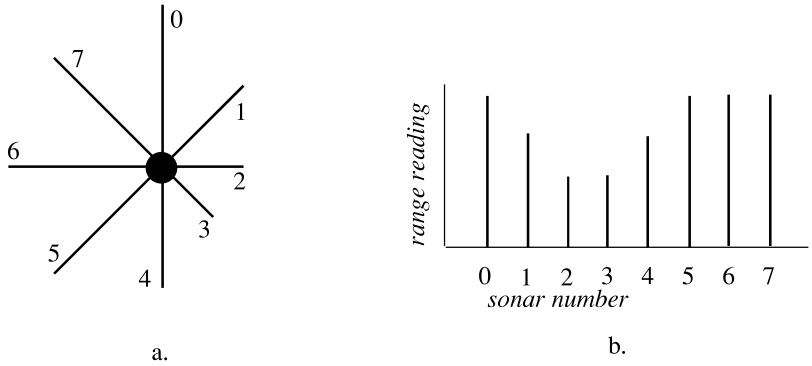

Figure 4.7 Polar plot of eight sonar range readings: a.) "robo-centric" view of range readings along acoustic axes, and b.) unrolled into a plot.

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

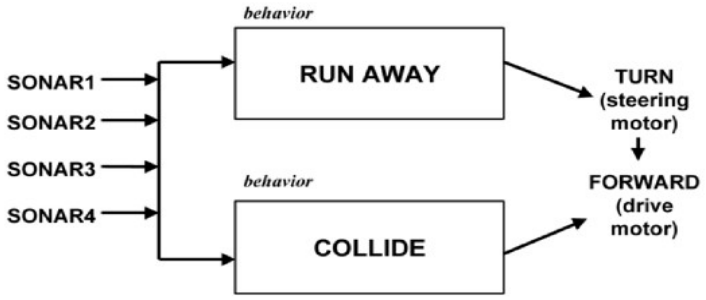

Figure 4.8 Level 0 recast as primitive behaviors.

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

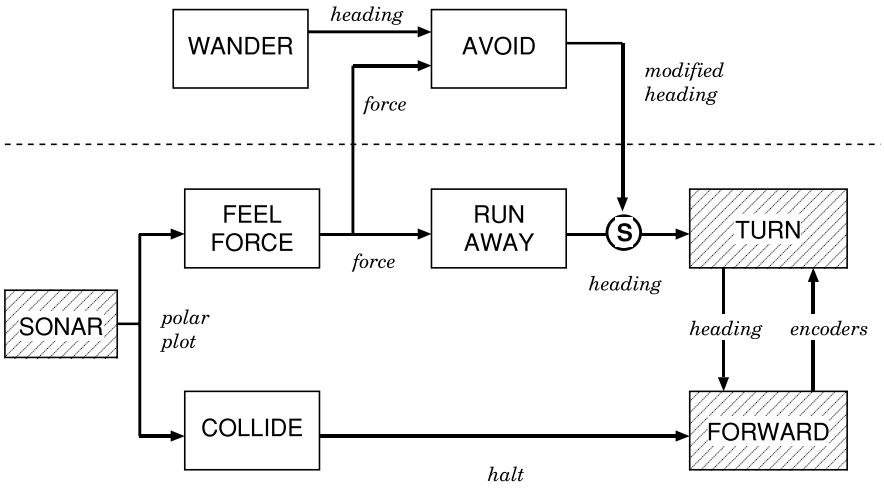

Figure 4.9 Level 1: wander.

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

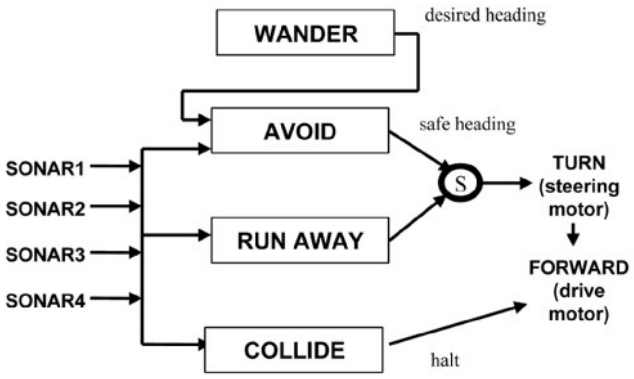

Figure 4.10 Level 1 recast as primitive behaviors.

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-

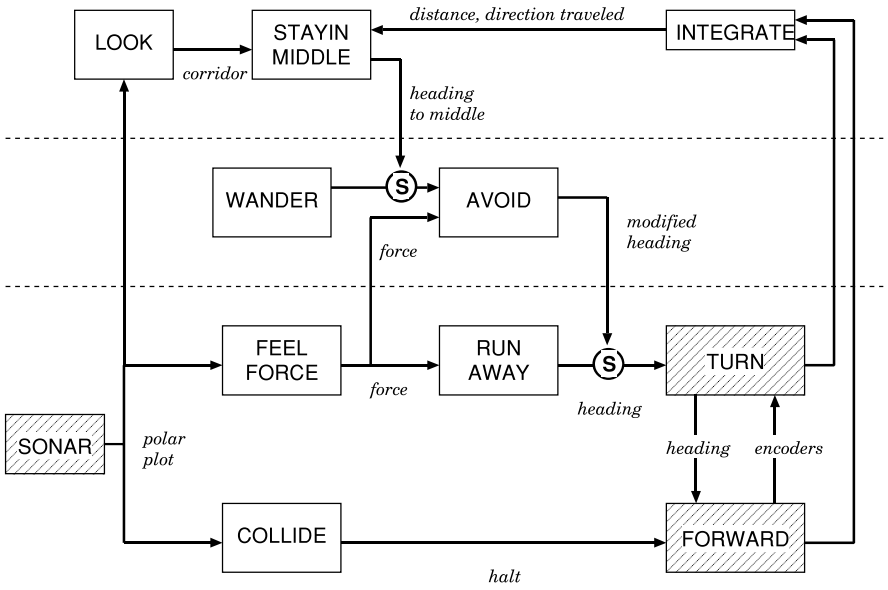

Figure 4.11 Level 2: follow corridors.

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