- 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.*

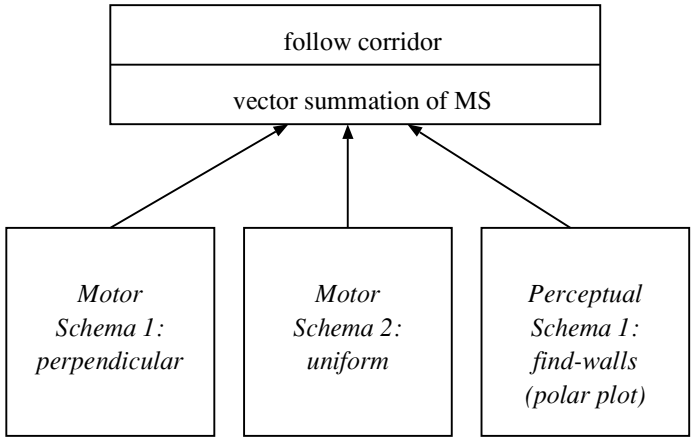
- Behaviors can (and should) use libraries of schemas. The pfields suffix on the `pfields.attraction()` meant that attraction was a method within another object identified as pfields. The five primitive potential fields could be encapsulated into one class called `PFields`, which any motor schema could 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.

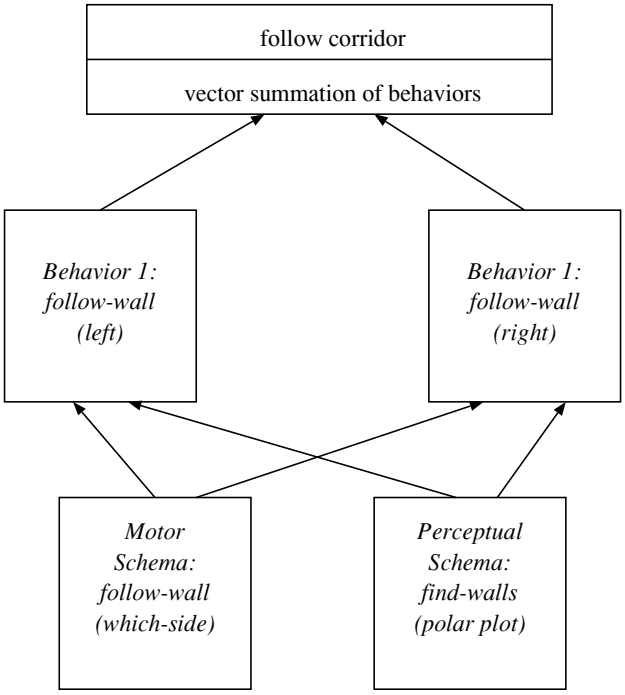
5.2.2 Example: An abstract follow-corridor behavior

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_-`



a.



b.

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,

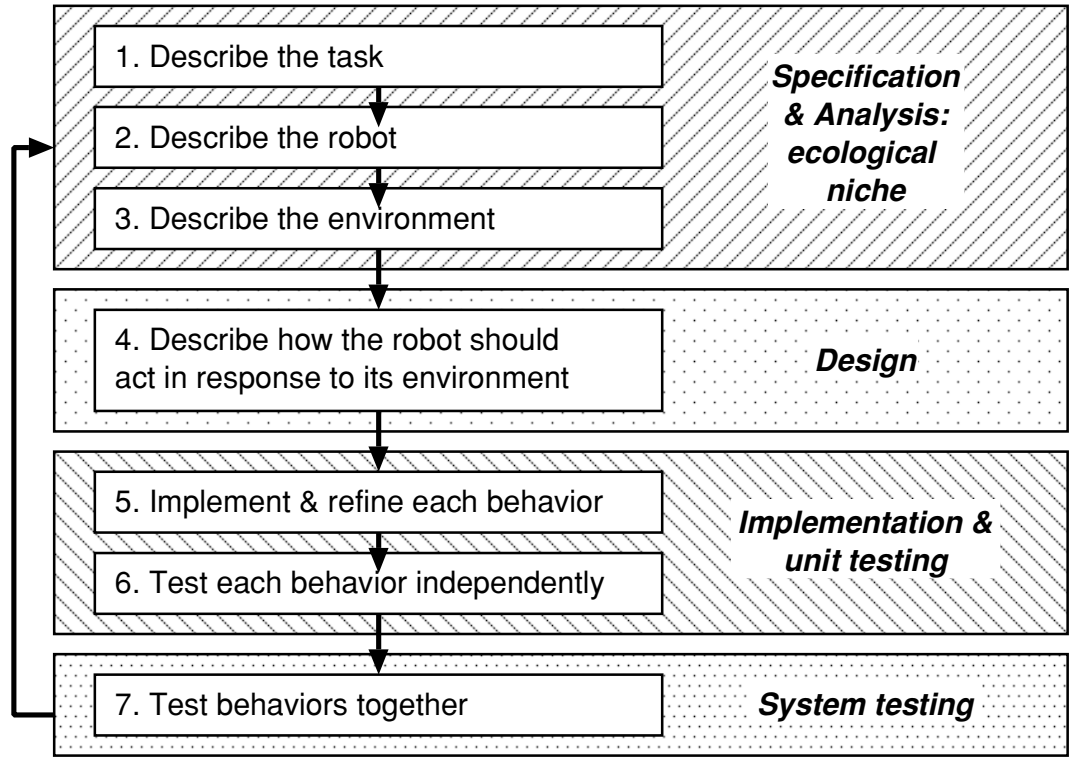


Figure 5.3 Steps in designing a reactive behavioral system, following basic software engineering phases.

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

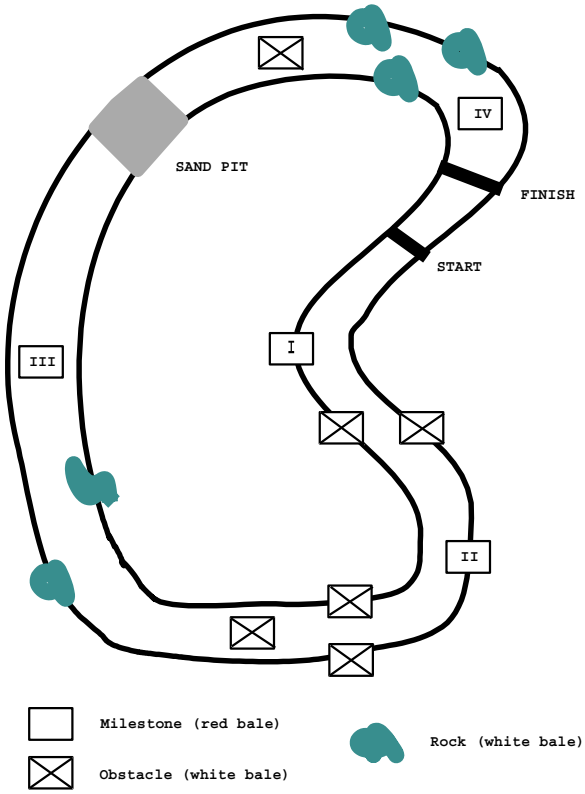


Figure 5.5 The course for the 1994 Ground Robotics Competition.

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,

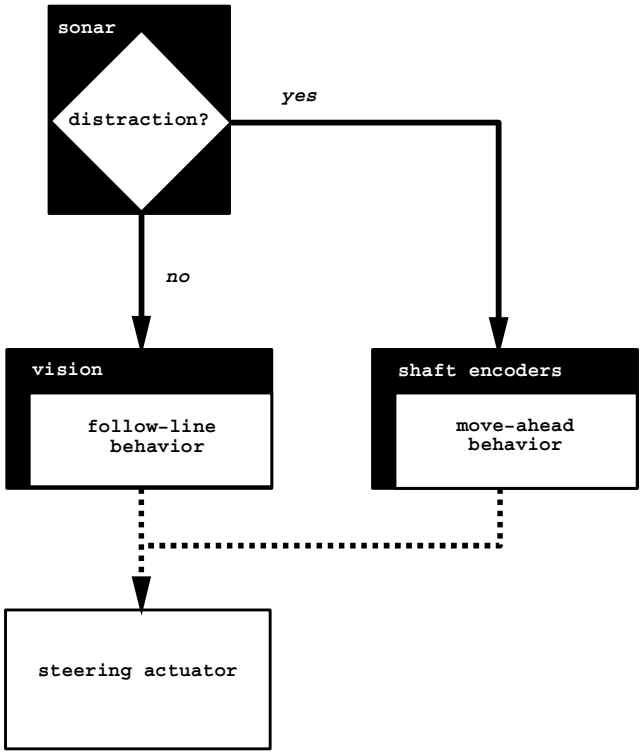


Figure 5.6 The behavioral layout of the CSM entry in the 1994 Unmanned Ground Vehicle Competition.

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).

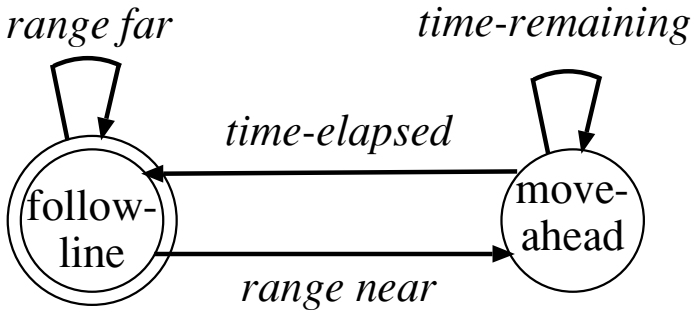
competition required 34 skills and 80 RAPs to do the same task, in part because of the complexity of the arm.[53;54] Since in general skills lead to a more complex implementation than FSA and scripts, they will not be covered here. The most important point to remember in assembling behaviors is to try 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.

$M : K =$ {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 5.8 A FSA representation of the coordination and control of behaviors in the UGV competition: a.) a diagram and b.) the table.

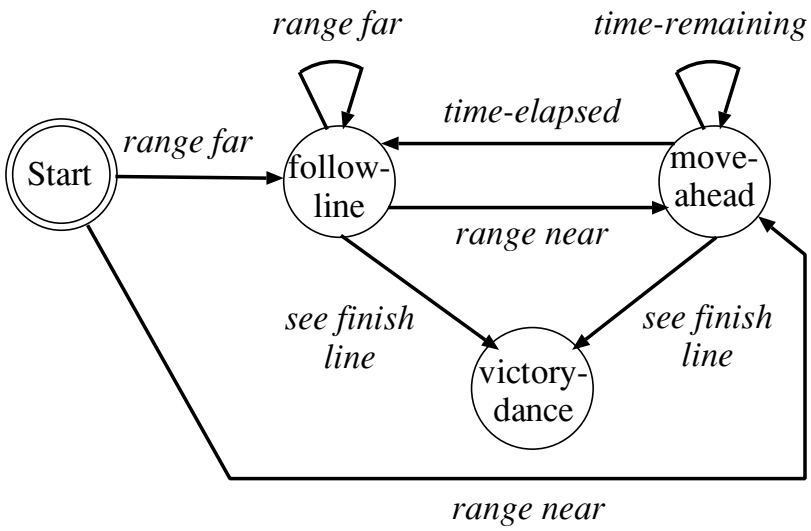
FSA, but each works about the same way. This section follows the notation used in algorithm development.[86]

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()

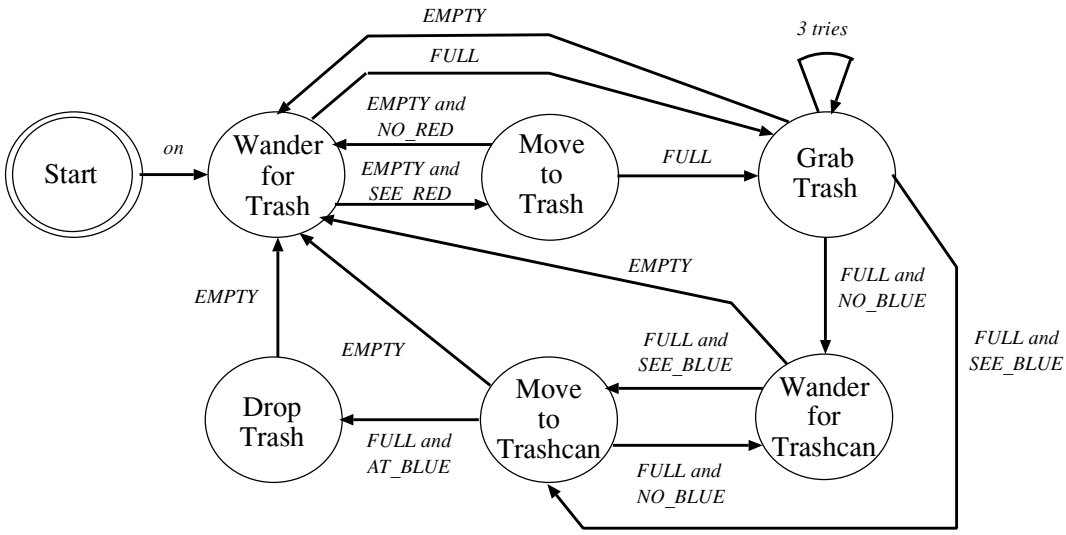
The function calls in the table only show the relevant arguments for brevity. The avoid behavior is interesting. The robot backs up either to the right or left (using a NaT) when it bumps something. It may bump an arena wall at several locations, but eventually a new wander direction will be set. If 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.