3. Behaviors and Fuzzy Control

3.1. Divide and Conquer

Getting a mobile robot to perform what seems to be a simple task can be an enormously complicated job. A robot's movements along a path must satisfy conflicting demands made upon it by its overall goal(s) and the various maintenance policies that guide and protect it along the way. For example, in navigating from one room to another in an office environment, the programmer usually will include milestones along the path at corridor intersections or other choice points which the robot should achieve as quickly as possible, subject to safety and power considerations. And as the robot works to achieve those larger goals, it should avoid obstacles and respond to contingencies, such as closed doors or blocked corridors, along the way. At any given point, what *control action* the robot takes— how it moves its wheels, for instance—is a complex function of the task and the internal state of the robot.

One way to reduce the complexity of robot motion is to break down its control function into sets of simpler actions that are more easily programmed and debugged. This *divide-and-conquer* strategy is effective in many AI applications that deal with complex systems. In mobile robotics, the primitive units of robotics action are called *behaviors*. However, put three roboticists together and you'll likely end up with four different definitions of what a behavior is. Here's a general definition that covers most of the main ideas:

Definition1. Behavior

A *behavior* is a mapping from the internal state to control actions of a robot, that

- Accomplishes a single goal
- Within a restricted context

This definition is very abstract: it says that behaviors characterize robot movements in terms of how they accomplish some goal. It's not the movements themselves that are important, but the end result. The definition leaves open exactly how the robot is to be controlled to accomplish the goal. In this chapter we'll look in detail at one method, *fuzzy control*.

The biggest challenge in defining behaviors is relating the internal state of the robot to the external goal the robot is supposed to accomplish. The internal state reflects the actual world, because it contains information gathered by the robot's sensors. If the robot is trying to go through a doorway, but it's sensors can't pinpoint the exact location of the doorway, or even if there *is* a doorway.

The second part of the definition implements the divide-and-conquer strategy. As long as we look at a restricted context, it is usually possible to formulate a movement strategy that accomplishes a goal. We'll get to some examples of this shortly: going to particular point, avoiding obstacles. The trick is to be able to *combine* behaviors to produce more complex actions that satisfy multiple goals: going to a goal position *and* avoiding obstacles.

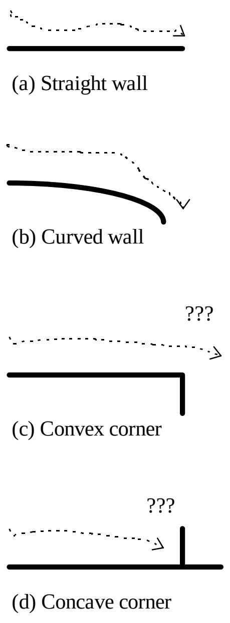


Figure 3-1 A wall-following behavior

3.1.1. Simple Behavior Example

Let's apply our definition of *behavior* to the example of wall-following: The desired goal is for the robot to traverse along a wall on its right. The robot should move near, but not too close to the wall, and if the wall curves, the robot should follow the curve.. Figure 3-1 illustrates four different types of walls the robot might be asked to follow. The robot's wall-following trajectory goal for each is the dotted line. The arrow in the Figure 3-1 points in the direction of the robot's movement.

Assume that the robot has some sensor that can tell how far away a surface is on the right-hand side of the robot, and that this sensor's results are reported in the robot's internal state. Then, the *mapping* from internal state to control actions would keep the robot moving forward in a straight line as long as the sensor reports that the wall is within some reasonable distance. If the distance to the wall grows larger than the prescribed separation, the behavior acts to turn the robot to the right. If the distance grows too small, the robot's wall-following behavior turns it away to the left. In keeping with the idea that a behavior is a simple form of action, the mapping is a simple one, and works well for smooth walls like those in Figure 3-1 *a* and *b*.

However, what happens when the robot encounters a sharp discontinuity, such as the end of a wall in Figure 3-1*c*? You can imagine many real-world cases in which it would not be appropriate for the robot to continue moving forward. e.g., if the robot is supposed to be following along a corridor, and the discontinuity is caused by a junction. In this case, the wall-following behavior is not in an appropriate context for accomplishing its goal. We could define some default action for when the robot encounters a junction (and for each and every other possible discontinuity, for that matter). But remember that our "divide and conquer" strategy is to make a behavior as *simple* as possible, in the interests of debugging and reliability. Accordingly, we just leave the behavior undefined here, by saying that its context demands that there be a smooth wall on the right of the robot.

What happens when the robot encounters a protrusion (someone standing in the hallway or the end of a hallway, for instance) as it follows along the wall (Figure 3-1*d*)? In this case, the context of being next to a wall is satisfied, but there's also the implicit notion that the robot should not collide with the obstacle. Again, in keeping with the simplicity of individual behaviors, we won't demand that the wall-following behavior deal with this situation.

3.1.2. Fuzzy Theory in Saphira

To keep behaviors simple, we've got to rely on some mechanism to combine behaviors for more complicated situations. There are two different possibilities here. The first is to allow several behaviors to operate at the same time, and merge their control outputs so that all of the behavior goals are satisfied, at least to some extent. We'll call this *behavior blending*. The second possibility is to invoke behaviors only when they are appropriate, that is, when their context is satisfied. Here we need a mechanism that decides when to switch in an appropriate behavior, depending on the situation at hand. We'll call this second method *behavior sequencing*.

To implement behavior blending, we turn to the elegant and useful methods of *fuzzy control theory*. Within Saphira, this theory forms the basis for the implementation of behaviors as modular building blocks for complex control. By using the techniques of fuzzy control, we'll be able to program simple behaviors in a natural way, then blend them together to accomplish more complex goals in less restrictive contexts.

This Chapter and its accompanying exercises are a sufficient tutorial for programming simple and compound behaviors in Saphira. They also introduce the notion of the context of applicability of a behavior, along with some simple ways of sequencing behaviors. But several key components will still be missing: the connection between perception and behavior, and the ability to put together sequences of behaviors based on their goals. We'll look at these issues in future chapters.

3.2. Fuzzy Control and Saphira

Fuzzy control is a method of controlling a system that is similar to classical *feedback control* approaches, but differs in that it substitutes *imprecise, symbolic* notions for the precise numeric measures of control theory. Figure 3-2 illustrates a typical feedback controller (inside dashed box) for a system. It's job is to calculate an estimated state of the system from sensor input, compare that estimated state with some desired state, and output control commands to the system so that, in time, the estimated state approaches or equals the desired state. Accordingly, the controller "maps" its internal state to control actions

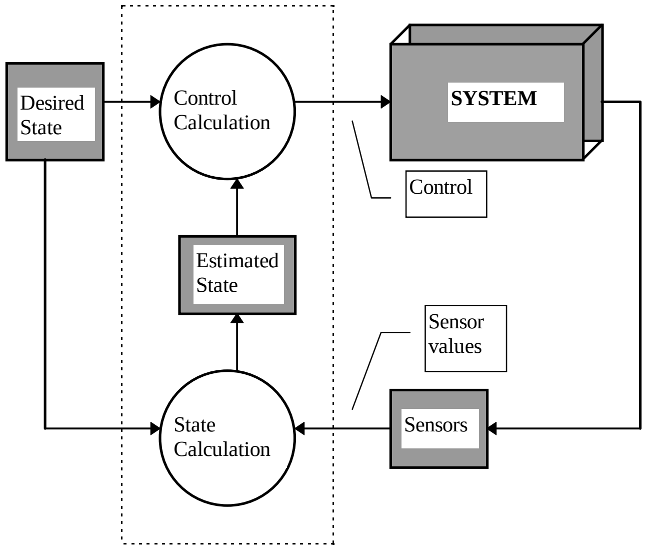


Figure 3-2 Feedback Controller Diagram

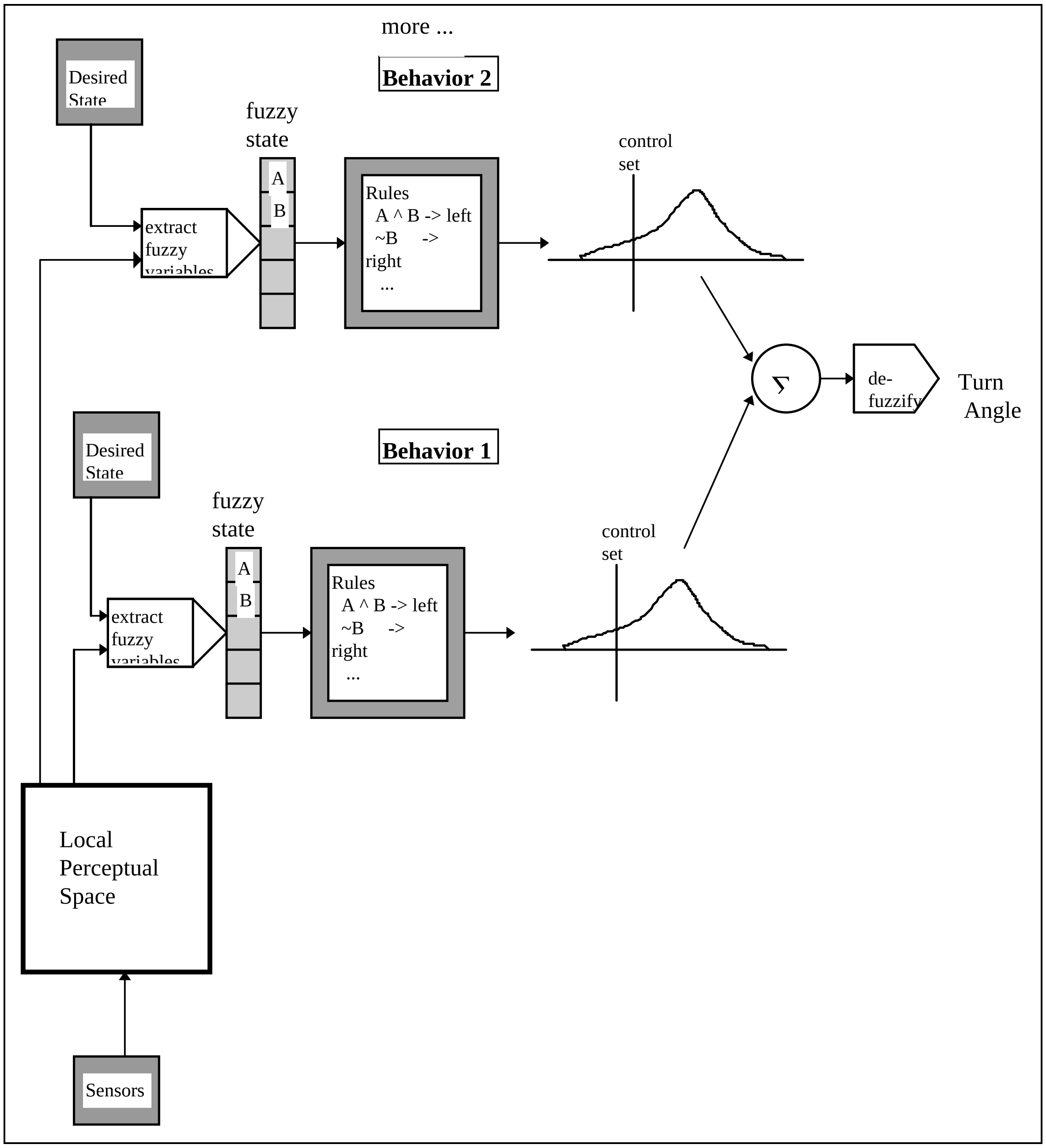


Figure 3-3 A Partitioned Fuzzy Controller

Although similar in overall structure, fuzzy control differs from classic feedback in that it substitutes uses *imprecise, symbolic* notions for the precise numeric measures of control theory. In fuzzy control, the controller has the same function, inputs, and outputs as in feedback control, but internally the control values are computed using techniques from fuzzy logic.

Figure 3-3 illustrates the fuzzy control processes employed by Saphira for robotics applications. From the bottom up: The system collects the robot's sensor information—wheel positions, sonar echoes, and on—into a data pool called Local Perceptual Space (LPS). Saphira then uses the LPS to compute a set of *fuzzy variables*—one set for each behavior based on the desired state the behavior is trying to achieve. In effect, the fuzzy variables summarize the state of the robot with respect to each particular behavior implemented for the robot.

The fuzzy controller then takes the fuzzy state variables and, by applying sets of *fuzzy rules*, produces a set of *fuzzy control values*. Like fuzzy variables, fuzzy control values are not precise numbers, but rather represent a range of possible values with different weights.

Eventually, a decision must be made as to what control value to give to the system. In standard fuzzy control theory, there is just one set of rules (one behavior), and a *defuzzification method* that uses a function, such as the centroid of the set, to pick a control value from the control set. In Saphira, we have decomposed the control problem into a set of modular behaviors, so the output of each active behavior must be *blended* into an overall control set, which is then defuzzified.

Figure 3-3 might seem like a complicated way control a robot, particularly if all you want to do is to move it around a bit. For that, you don't need all the complexity of fuzzy control, and you may as well program the robot controller using simpler, classical techniques. But as the complexity of the robot tasks increase, you'll soon find it difficult if not impossible to accommodate all the competing goals, and it won't be easy to find the correct control action to take. Saphira's partitioned fuzzy controller gives you a nice method of breaking the problem down to size.

Don't worry if you don't understand all of the concepts just mentioned: the following sections will explain everything in detail, and help you get started writing your own behaviors with some simple examples. And after getting familiar with the concepts of fuzzy controls, you'll find it natural to decompose a complex control problem into a set of easily debugged behaviors, which you can then combine to produce sophisticated action.

Of course, if you don't want to worry about how behaviors work, you can always use the predefined behaviors given in the Saphira API. But you'll be better equipped to figure out what's going on, and how to correct behavioral problems in your robot, if you read this Chapter carefully.

3.3. Fuzzy Behavior Components

3.3.1. Fuzzy Variables

Fuzzy variables summarize the state of the robot for a behavior. Each behavior has its own set of fuzzy variables, designed to abstract just that information necessary for the behavior to operate. A fuzzy variable can have any value between 0.0 and 1.0 (not just 0 *or* 1, as for a boolean variable), which indicates the *degree of membership* of the robot's state in the concept expressed by the variable. For example, for obstacle avoidance one of the key properties of the robot's state is whether the path is clear in front of the robot or not. Call this property *front-clear*. What should the value of *front-clear* be? Or put another way, how is the membership of the robot state in *front-clear* determined?

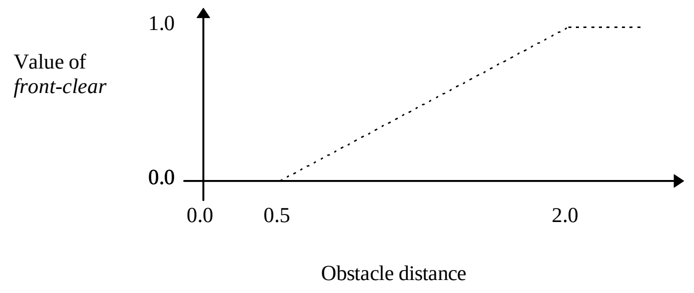


Figure 3-4 A membership function for the *front-clear* variable

Suppose the robot has a sensor that indicates how far the nearest obstacle is in front. If the sensor indicates an object very close to the robot, the value of *front-clear* should be 0.0. Conversely, value of *front-clear* should be 1.0 if there's nothing in front of the robot (within the sensors' range, of course). As the robot nears an obstacle, the value of *front-clear* should fall between 0.0 and 1.0 to indicate the changing state of the obstacle in front of the robot.

3.3.2. Membership Functions

A *membership function* defines how the state information collected by a robot's sensors gets transformed into a fuzzy variable value. For the *front-clear* variable, the input to the membership function might be the distance to the nearest obstacle detected by the sensors, and its membership function might look like the one shown in Figure 3-4. This function is a linear interpolation between the two distances 0.5 meters and 2.0 meters. Accordingly, if the sensors detect an obstacle at 0.5 feet in front, the computed value for the *front-clear* variable based in the membership function shown in Figure 3-4 is 0.0. At 1.5 meters, the variable's value is around 0.75.

Membership functions don't have to be linear, or even monotonic. For instance, an *at-one-meter* membership function like that shown in Figure 3-5 might be used to compute a corresponding fuzzy variable value for obstacles at a distance of a meter from the robot. However, developers favor piecewise linear functions as membership functions because they are easy to compute..

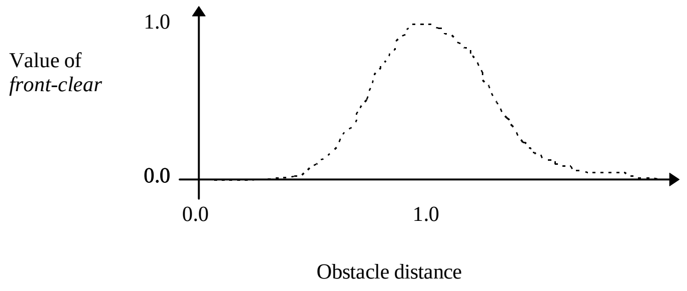


Figure 3-5 Membership function for *at-one-meter*

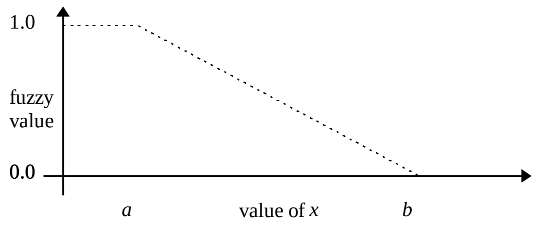
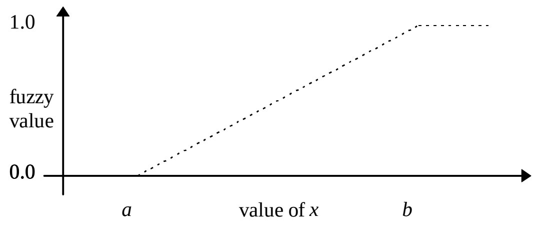
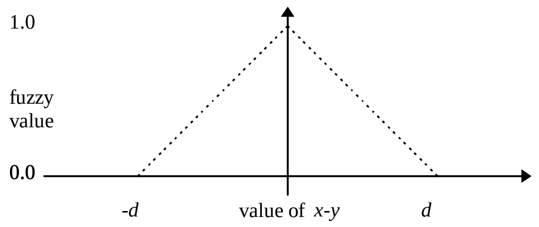
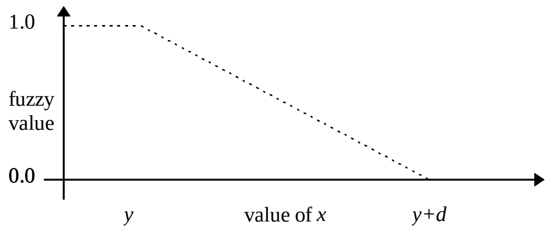
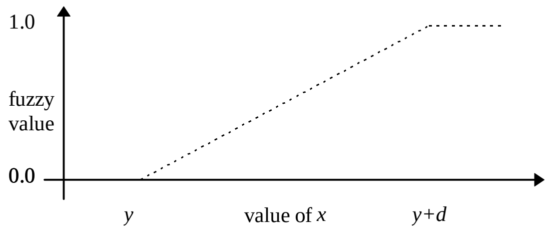
Saphira provides a set of pre-defined piecewise linear membership functions (Table 3-1). All of these functions take floating-point number arguments and return a floating-point fuzzy value between 0.0 and 1.0. Saphira's comparison functions equal, greater-than, and less-than also accept a *tolerance* value that defines how lenient the comparison will be. By setting the tolerance very low, you can make a steep transition between values of the variable; a high setting gives a more gradual transition. In controlling the robot, the tolerance of fuzzy variables will play a big role in determining the robot's actions. Too high a tolerance will make the robot respond sluggishly to inputs; too low, and it will over-react.

Exercise 3-1

Show how to define the membership function *f_eq* in terms of *up_straight* or *straight_down*.
[*f_eq(x,y,d) = up_straight(abs(x-y), 0.0, d)*]

Notice that all the Saphira membership functions, except *f_eq*, have the form of *straight_up* or *straight_down*. By looking at the graphs in Table 3-1, it is easy to see, for example, that *f_greater(x,y,d)* can be defined as *up_straight(x,y,y+d)*.

Table 3-1 Piecewise linear membership functions in Saphira

Function Name	Function Graph
straight_down(x,a,b)	
up_straight(x,a,b)	
f_eq(x,y,d)	
f_smaller(x,y,d)	
f_greater(x,y,d)	

3.3.3. Combining Membership Functions

What if you need to define another membership function for a new behavior, because the ones in Table 3-1 aren't quite right? The good news is you don't have to: fuzzy variables come with their own method for creating new, more complex variables from primitive ones, thanks to *fuzzy logic*. Just as in propositional logic where we define boolean combinations of primitive propositions, so too in fuzzy logic we define boolean combinations of primitive fuzzy variables. Of course, since fuzzy variables take on values between 0.0 and 1.0, the combination functions will be different from boolean functions, since they

Table 3-2 Boolean operators for fuzzy variables

	x AND *y*	*x* OR *y*	NOT *x*
(a)	*min(x,y)*	*max(x,y)*	*1-x*
(b)	*xy*	*x + y - xy*	*1-x*
(c)	*max(x+y-1, 0)*	*min(x+y,1)*	*1-x*

have to account for intermediate values, as well as 0 and 1. To make everything consistent, the combination function AND must be a *triangular norm*, and OR must be a *triangular conorm.*[1]

Table 3-2 shows some examples of valid combination functions. Because they give good performance in behaviors and are easy to compute (among other reasons)e, we have chosen to use the *min* and *max* functions for AND and OR, respectively, in Saphira:.

The implemented combination functions in Saphira are summarized in Table 3-3. They all take floating-point arguments and return a floating-point fuzzy value.

Table 3-3 Combination functions in Saphira

Name	Function
f_and(x,y)	*min(x,y)*
f_or(x,y)	*max(x,y)*
f_not(x)	*1 - x*

Arbitrary piecewise linear functions can be formed by applying the combination functions to an initial set of fuzzy variables. For instance, to determine when the robot is too close *or* too far from a goal point, you might first define two fuzzy variables: *too-close*, which the valu*e* 1.0 when the robot is within 0.1 meters of a fixed point, and *too-far*, which equals 1.0 when the robot is more than 0.5 meters away. Then, the combination *too-close* OR *too-far* describes the function shown in Figure 3-6. So, as the robot gets to the just the right distance away from the goal—not too close or too far—the value of the combination function *too-close* OR *too-far* approaches 0.0. Otherwise, the combined value approaches or equals 1.0. Similarly, the inverse of that same combination, NOT(*too-close* OR *too-far*), yields a region with a hump between the 0.1 and 0.5 meters, and its value approaches 1.0 as the robot nears the prescribed distance from its goal, and approaches or is equal to 0.0 elsewhere.

Note that the depth of these combination function wells increase and decrease as you change the tolerances of the initial fuzzy variables. And, if you make the tolerances too large, the hump or well disappears.

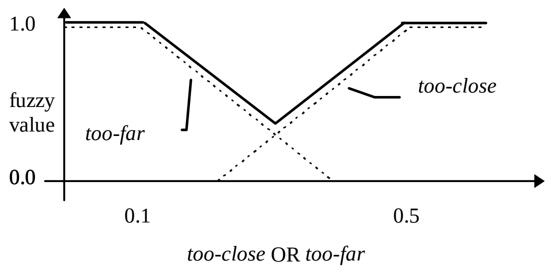


Figure 3-6 A fuzzy OR combination

By now, those of you familiar with propositional logic will be wondering if there is a correspondence between NOT(*a* OR *b*) and (NOT *a*) AND (NOT *b*), the well-known DeMorgan's Law which holds for boolean functions. Yes, there is a correspondence, at least for the *max* and *min* functions we've picked to implement combinations in Saphira. So it doesn't matter which way you write a fuzzy combination, you will get the same result.

Exercise 3-2

Show that DeMorgan's Law holds by showing that *1-max(a,b) = min(1-a, 1-b)*, and that *1-min(a,b) = max(1-a,1-b)*. Does this theorem hold for all of the norms and conorms in Table 3-2?

[1]A triangular norm is a binary operator on [0,1] that is commutative, associative, and non-decreasing in each argument, and has 1 as unit. A triangular conorm is similar, but has 0 as its unit. With a little thought, it's easy to see that these properties make sense for AND and OR. For example, cummutativity means that the result doesn't depend on the order of the arguments.

3.3.4. Fuzzy Control Rules

The central component of a fuzzy behavior is its fuzzy ruleset. This is where the action is: rules define how fuzzy variables get transformed into fuzzy control values, which ultimately get combined (“defuzzified”) into a control command for the robot. All rules have the form

IF *fuzzy value* THEN *control set* .

What is a control set? In keeping with the idea of fuzzy set membership, a *control set* is just a mapping from control values to the interval [0,1]. It’s easiest to think of it as a *fuzzy control variable*, that is, a fuzzy variable that represents a control value. To make this more concrete, consider the case where we want to have the robot move at 100 mm/sec (a very slow walk). In PID control we would define the target velocity as precisely 100 mm/sec. In the fuzzy world we relax this precision and define a fuzzy control variable *move-slowly* by a range of possible velocities centered on 100 mm/sec, as in Figure 3-7. Note the similarity of this to Figure 3-5: the only difference is that here the fuzzy variable is a control variable, and refers to the value of one of the robot’s control channels. The control variable is satisfied (close to 1.0) for velocities near 100 mm/sec, and is unsatisfied (close to 0.0) elsewhere.

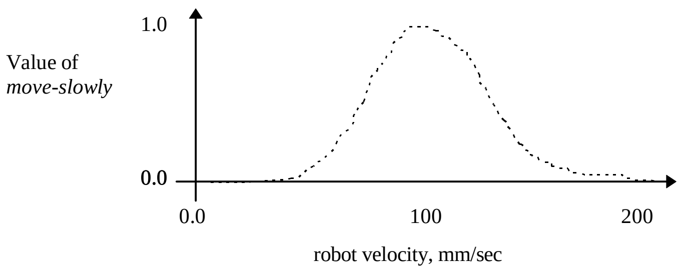


Figure 3-7 A fuzzy control variable for robot velocity

3.3.5. Rule Antecedents

Let’s bypass for the moment how we specify control sets and concentrate on how the antecedent (IF fuzzy variables) of the fuzzy rule is involved in the result. At the extremes, the antecedent value of 1.0 returns an unmodified control set; and ifan antecedent of 0.0 shuts the control off completely, that is, makes it zero everywhere.. In between, the antecedent should weaken the control set by an appropriate amount, lowering its value. An appropriate method is to use the AND function already defined, that is, to make the result of the rule the *min* of the antecedent and the control set. Since the antecedent is a single fuzzy value, this has the effect of clipping the control set to the value of the antecedent, as in **Error! Reference source not found.**.

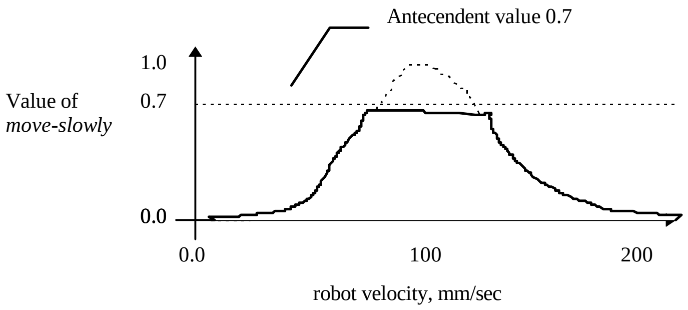


Figure 3-8 Clipping by the rule antecedent

After clipping by rule antecedents, the control sets of all the rules in a behavior are combined into a summed rule set. Since each rule can contribute to the outcome, the combination function is an OR, that is, the *max* function. The control set for the behavior looks something like a string of large and small volcanoes, with their tops truncated by antecedent clipping.

3.3.6. Defuzzification

The final step in the fuzzy control process is to compute a single control value from the control set—the robot's controllers cannot work with a fuzzy set; they need neat, hard directions. This is the process of *defuzzification*, an ugly if descriptive term.

How is defuzzification done? A little reflection will show that there are many different possibilities. The simplest way to defuzzify a fuzzy control set is to pick the highest value in the control set, the one with greatest membership value. But there can be serious drawbacks to this method. First, when several competing peaks have almost the same fuzzy value, the behavior can switch rapidly back and forth between them as the inputs change by just small amount, perhaps leading to wild oscillations. Second, a narrow, high peak has a small set of control values that are good, but is surrounded by bad ones, so it may not be very robust. A typical example here is the robot going through a narrow opening, where turning even a little bit will lead to a collision. In this case it's probably better to design a strategy that gives the controller a little more leeway, since it may not be able to accurately track the high-precision demands of the behavior.

Another defuzzification strategy is to compute the *centroid* of the control set, which is the average of the controls weighted by their fuzzy values. This strategy has the advantage of picking large, broad peaks when they exist, which leads to smooth, robust behavior. However, the computational demands can be somewhat expensive for arbitrary control sets: you have to divide the control area into a set of discrete values v_i, and take the weighted sum $\Sigma f_i v_i$, where f_i is the fuzzy value for that interval. But if the form of the control set is restricted enough, there are cheap approximations (see below).

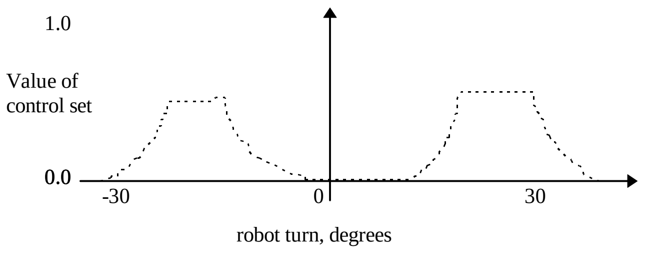


Figure 3-9 A schizophrenic control set for head-on collision

Another disadvantage of the centroid defuzzification strategy is that it doesn't do very well with schizophrenic rule sets, where two rules give equally weighted, but very different controls. A typical case is obstacle avoidance: As the robot approaches an obstacle head-on, one rule might want to turn right, and another to turn left, giving a control set that looks like Figure 3-9. The centroid method would choose something in the middle, which is not a very good idea: Like Buridan's ass, the robot wouldn't know which of the two good ways to go, and so would wind up doing nothing.

A more complicated defuzzification scheme that could deal with schizophrenic control sets would be to find two or more significant peaks, and choose the one with the largest area. In the example case, the robot would wind up going either right or left. The disadvantage to this method is that it requires expensive analysis of the control set, and like the highest peak method, could wind up switching rapidly between controls unless some hysteresis were included [ref].

In Saphira, we take the centroid approach—slightly modified—mostly because of its good computational properties and its robustness. We modified the scheme to deal with yet another problem (one common to almost all defuzzification methods): What to do when the control set offers only very weak advice about the control value?

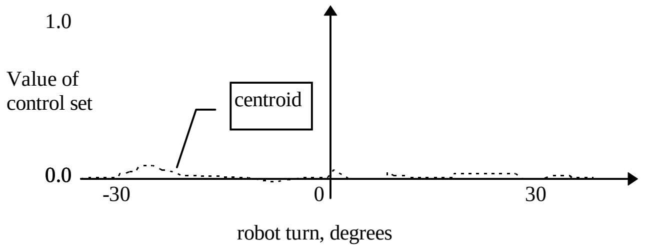


Figure 3-10 A weak control set with a large control value

Consider the control set shown in Figure 3-10, where all fuzzy values are close to 0. Just a small blip somewhere in the set can cause large variations in the defuzzified value. To deal with this situation, the Saphira method always adds in a moderately-sized peak at the zero or neutral value of the control set. When the control set proposed by rules is strong, this added peak has little effect; when it's weak, it tends to bring the defuzzified control to the neutral position.

3.3.7. Saphira Control Sets

Before we end this section on control rules, we have to deal with two more issues. The first is the form of the control set provided for the fuzzy rules. In Saphira, we have made the simplifying assumption that all control sets for fuzzy rules are rectangular with a standard width. So, you only have to specify a single number in the consequent, namely the center value of the control set.

Also, when the antecedent clips the control set, the effect is to keep the same rectangular form, but just lower the fuzzy value. So the fuzzy rule interpreter only has to keep track of the center value and the height of the control set for each rule, and can use simple computations for the centroid value. Standard symbolic names for most control sets are already part of Saphira (see the next section), and can be used as the consequent in control rules.

For example, a typical control rule would be

IF *front-blocked* THEN *Turn Left Sharply*

where *Turn Left Sharply* is a control set with a peak at an angle of about -30 degrees.

What do the control sets actually refer to in Saphira? In standard fuzzy control theory, the controls are *accelerations* or *forces*, just as in classical control theory. There are two basic motion control channels for Saphira servers: forward/back velocity and angular position. How are the control set accelerations converted into these Saphira controls? Let's look at turning first.

The turn controller uses a parallelogram velocity profile to achieve a given turn angle setpoint (see Section XXX???). It always accelerates and decelerates at a constant rate. However, if the setpoint is not very different from the current robot angle, then the turn velocity will be small, since the controller plans to decelerate after just a short time. On the other hand, if the difference is large, the turn velocity will be greater. So, in effect, Saphira controls the change in velocity by keeping the setpoint a larger or smaller distance from the current turn angle. A control set name such as *turn left sharply* is translated into an appropriate differential in the turn-angle setpoint.

For forward/back velocity, we could use the same idea with velocity setpoints. However, we normally want our control rules to set the velocity, rather than an acceleration. Take the example of following a wall: We'd like the robot to maintain a certain speed as long as it is keeping the wall in view on one side. So the control rule for the forward/back motion channel could specify a velocity setpoint directly. This is indeed what Saphira does. Typically a velocity rule will be of the form

IF *front-blocked* THEN *speed 50*

where the speed is expressed in mm/sec.

We've done a lot of exposition of the basics of fuzzy control theory. It's time to apply it all to some simple behaviors for the robot.

3.4. Using Saphira Behaviors

Let's get down to the practical matter of writing behaviors for Saphira clients. It's actually easy, at least for simple behaviors, thanks to a syntactic front-end that eliminates most of the nastiness in C syntax necessary to represent fuzzy variable manipulations and fuzzy control rules.

There are four basic parts to a Saphira behavior (we'll add on a few extras in subsequent sections, but for now these four will suffice).

- Input parameters
- Fuzzy variable calculation (update function)
- Fuzzy rule set
- Activity

Example 3-1 Constant Velocity Behavior Definition

```
/*
 * Define the behavior constant_vel
 *   with one parameter, the desired speed in mm/sec
*/

BeginBehavior sfConstantVel    /* behavior name */
     Params sfFLOAT sval       /* input parameter, desired speed */
     Rules                     /* just one rule */
         If  too_fast Or too_slow  Then Speed sval
     Update                    /* calculate fuzzy variables here */
         too_fast = up_straight(current_vel(), sval, 50.0);
         too_slow = straight_down(current_vel(), sval, 50.0);
     Activity
         Speed 1.0
EndBehavior
```

When you invoke a behavior in a client, Saphira first binds the input parameters to the values given in the invocation, and then adds the resultant *behavior instance* to a list of executing behaviors. Thereafter, on each Saphira cycle (every 100 ms), the behavior's update function is evaluated to update the value of all fuzzy variables in the behavior. Then the fuzzy rules are evaluated to produce control sets, the control sets are combined with OR, and the result is defuzzified to produce a control value that is sent to the robot server. When there is more than one active behavior, an activity section in each controls the interaction between behaviors.

3.4.1. Constant Velocity Behavior Example

Let's begin with an example (Example 3-1). It is a behavior that will keep the robot moving at a constant velocity. It is written in a combination of behavior syntax and C code. The keyword *BeginBehavior* indicates the start of the behavior definition; *EndBehavior* closes the definition and ends the behavior syntax mode.

Within the behavior definition segment, you specify—in order—the parameters, rules, update functions, and activity that comprise the four basic components of a Saphira behavior. In the example, there is just a single parameter—the desired speed, *sval*, which is a floating-point number. There is also just a single rule, whose antecedent is the disjunction of two fuzzy variables, indicating that the robot is going either too fast or too slow. The control set is the desired Speed..

Saphira calculates our two fuzzy state values, *too_fast* and *too_slow*, in the behavior's update function, as expressed in standard C code. Also note the use of the Saphira membership functions (Table 3-1) to derive the fuzzy values. We set the speed tolerance to 50 mm/sec, which means the controller will try and speed up or slow down the robot so that its velocity will come to and stay within *sval* ± 50 mm/sec. The function . *current_vel()* in the expression returns the current velocity of the robot.

Of course, you know from reading the previous sections that the OR of the two fuzzy variables in the example describe the well shape in Figure 3-6. So, when the robot's speed is too far on either side of the desired velocity, the antecedent's value will be high, and the control set produced by the behavior will be at the desired speed.

The Activity section determines which control channels the behavior affects, and by how much. We'll describe it in much more detail below. For now, the specification "Speed 1.0" means that this behavior is fully engaged in controlling the robot's speed.

The example isn't the most compact or even the best way to write a constant velocity rule. But it does illustrate the basic ideas behind defining fuzzy variables in the update function, combining their values, and producing control sets with the rules. If you invoke the *sfConstantVel* behavior (see the next subsection), you'll find that the robot consistently achieves a velocity of almost 30 mm/sec less than the desired velocity. The reason is the bias towards neutral (zero velocity) inherent in Saphira's defuzzification process. When the robot gets near the desired velocity, the antecedent of the rule becomes weaker, until it just balances with the neutral bias, at a speed somewhat less than the goal.

Exercise 3-3

Write a simple velocity-control behavior that will achieve a desired velocity setpoint. (Don't cheat by looking at *sfConstantVelocity* in handler/src/behave/behavior.c.).

3.4.2. Behavior Compilation and Invocation

To get Saphira to execute our constant velocity behavior with the robot, we have to include it in a Saphira client's source code (Example 3-2), compile it, and finally invoke it from within the Saphira application.

The compilation process has two steps: First, you use a special Saphira behavior-syntax compiler to convert *BeginBehavior/EndBehavior* sequences—

Example 3-3. Makefile for Compiling and Linking a Saphira Client Application

```
#
# Makefile for behavior compilation
#   copy source file to "test.c"
#   creates the executable file "client"
#
client:    test.beh
       bgram test.beh test.c
       cc test.c -o client -lSf -lXm -lXt -lX11 -lm -lc
```

```
 *    with one parameter, the desired speed in mm/sec
*/

BeginBehavior sfConstantVel    /* behavior name */
     Params sfFLOAT sval       /* input parameter, desired speed */
     Rules                     /* just one rule */
         If  too_fast Or too_slow  Then Speed sval
     Update                    /* calculate fuzzy variables here */
         too_fast = up_straight(current_vel(), sval, 50.0);
         too_slow = straight_down(current_vel(), sval, 50.0);
     Activity
         Speed 1.0
EndBehavior


void
myConnectFn(void)
{
   sfInitBehavior(sfConstantVel, "constant velocity", 0, 1,
                     sfFLOAT, 200.0, sfEND);
}

void
main(int argc, char **argv)
{
   sfOnConnectFn = myConnectFn;
   sfStartup(0);
}
```

behavior definitions—in the Saphira client source into standard C source code.. The program *bgram*, included with the Saphira 5.1 distribution, does this conversion. It takes an input file as its first argument, and writes its result to the file that is its second argument. Then, in the second step, you compile that converted file into an executable Saphira client with a standard C compiler.

The makefile in Example 3-3 automates these two compilation steps in creating *test.c* from the Saphira client source containing a behavior definition (*test.beh),* and then compiling it into *client*.

We'll use this makefile for all our example programs. To do so for our current example, just copy *forward.beh* to *test.beh*, then call *make* (which uses the makefile), and the executable will be found in the file *client*.

3.4.3. The Saphira Client

In the client code, you invoke a Saphira behavior via the *sfInitBehavior* function, which forms the behavior instance and places it in Saphira's behavior execution list. This function has a lot to do, and you have to help it out by providing a number of arguments. As shown in Example 3-2, we need to provide::

1. A pointer to the behavior *(sfConstantVel).*
2. A name for the behavior instance ("constant velocity"). This is a string that is displayed in the behavior window of the client where you check on the state of the behavior and turn it on or off.
3. A *priority* for the behavior. In the example, the priority is set to 0. Low priority numbers indicate high-priority behaviors. (In subsequent sections, we'll discuss how Saphira uses the behavior priorities to trade off between competing behaviors.)
4. A boolean digit indicating whether the initial state of the behavior: enabled (1) or disabled (0). The user also may enable or disable the behavior from the Saphira client's Behaviors window.
5. A list of parameters pairs for the behavior, consisting of a datatype followed by the actual parameter value, and terminated by the sfEND constant. In the example, there is one parameter of type floating-point (sfFLOAT), the desired speed of 200 mm/sec.

When you start up the *client* executable, and connect to the robot server—either the robot itself or its simulator—the robot should immediately begin to move forward at about 200 mm/sec. That's because we put the *sfInitBehavior* invocation for *the sfConstantVel* behavior into the *sfOnConnectFn* function section of the client, which gets called as soon as a connection to the robot server is opened.

With the *forward.beh* client, the robot will keep moving until you turn the behavior off by quitting the *client* program. However, have you tried disabling the behavior in the client's Behaviors window? If so, you may have been surprised to find that the robot keeps moving!

What's happening is that the robot server velocity setpoint is at 200 mm/sec, as set by the constant velocity behavior. Turning the behavior off means that no new setpoint commands are sent to the server, so it just keeps happily moving along. You can get the robot to stop by pressing spacebar, which resets the setpoint to zero, or by adding another behavior that stops the robot. We'll get to that shortly; first, let's look at a simple behavior that turns the robot.

3.4.4. Turn, Turn, Turn

Like the constant velocity behavior, the robot-turning behavior of Example 3-2 is simple: it just keeps the robot turning in one direction. The parameters to the behavior are the direction and speed of the turn. Both are integers, and Saphira has symbolic names for the turn direction and turn control sets.

Table 3-4 lists the turn-parameter terms you should use in C code and in the behavior grammar. Use C code terms in ordinary C functions; for instance , when invoking a behavior with *sfInitBehavior*. Use the behavior grammar terms within the *Rules* section of a behavior definition: *If x Then Turn Left Slowly*, for example.

Table 3-4 Turn Parameters for Grammar and C Code

	C Code	Rule Grammar
Turn directions	*sfLEFTTURN*	*Turn Left*
	sfRIGHTTURN	*Turn Right*
Turn values	*sfVSLOWLY*	*Very Slowly*
	sfSLOWLY	*Slowly*
	sfMODERATELY	*Moderately* [default]
	sfSHARPLY	*Sharply*
	sfVSHARPLY	*Very Sharply*

If you look closely at the turn behavior definition, you'll discover it has some peculiarities. The update function is empty and we don't calculate any fuzzy variables! That's because the only rule has an antecedent of 1.0. It's always full on, telling the robot to keep turning. We specified the rule's control set using the parameters passed in to the behavior (turn direction and strength), so we just use the parameter names in the rule. And finally, this behavior controls turning, so in the activity section we simply specify *Turn 1.0*.

As before, to compile the *turn.beh* file, copy it to *test.beh* and call *make*, which will run the *bgram* parser and then compile the output into the *client* executable. After connecting the executing client to the simulator or actual robot, you should see the robot rotate at a constant velocity counterclockwise (the robot's left).

Try different values for the turn strength to see how quickly or slowly the robot turns. Also, try disabling the behavior by going to the behavior window and turning its radio button ("constant turn") off. Unlike the forward/back control channel which controls a velocity setpoint, the robot's turning channel sets an angular position goal which is a difference value from the robot's current angle. When the behavior stop changing that angular position goal, the robot stops settles at the goal position and stops turning..

Having the robot spin around and around normally isn't very useful, and just gets the robot dizzy. Programming the robot to achieve a particular angle, say 20 degrees left from its current heading, is not hard to do, but we'll have to wait until we learn how to introduce the robot's internal dead-reckoned position into the fuzzy variables, later in this Chapter.

Example 3-3 The Turn behavior

```
/*
 * Define the behavior “sfTurn”
 *  with two parameters, the desired direction and speed
 *
 * File: handler/src/samples/turn.beh
 */

#include “saphira.h”

BeginBehavior sfTurn         /* behavior name */
   Params sfINT tdir         /* sfLEFTTURN or sfRIGHTTURN */
         sfINT tval          /* sfSLOWLY, sfMODERATELY, etc */
   Rules                     /* just one rule */
       If 1.0 Then Turn tdir tval
   Update                    /* calculate fuzzy variables here */
   Activity
       Turn 1.0
EndBehavior


void
myConnectFn(void)
{
   sfInitBehavior(sfTurn, “constant turn”, 0, 1,
                  sfINT, sfLEFTTURN,
                  sfINT, sfSHARPLY,
                  sfEND);
}

void
main(int argc, char **argv)
{
   sfOnConnectFn = myConnectFn;
   sfStartup(0);
}
```

3.5. Behavior Definition Grammar

So far we’ve given a few examples of behavior definitions. Now it’s time to exhibit the behavior grammar in all its glory.

We use the *Backus-Naur Form* for the behavior grammar (Table 3-5). Each rule in the table defines a phrase in the grammar. A nonterminal symbol on the left-hand side of the rule is defined in terms of a mix of terminal and nonterminal symbols on the right. By convention, nonterminals are in uppercase and terminals are in lowercase or quotes.

At the top level, a BEHAVIOR form is defineas a sequence of fixed words, a terminal symbol that is the name of the behavior, and a set of phrases. The elements enclosed by square brackets are optional; you may include these phrases or not. All of the other elements are required, or the parser generates an error. Don’t forget to name your behavior, this is a common source of parsing errors.

As an aside, the error reporting of *bgram*—the parsing program that uses the behavior grammar—is sparse. It will tell you the source file line number where the error occurred, but it’s up to you to decide what the error is. If you’re confused by the error and can’t find an obvious fix, try separating your input so that only one word appears on each line, and then it will at least be clear where the parser choked.

The PARAM_STMTS phrase uses a special notation to define alternative parameters: each is placed between curly braces and separated by a bar. Exactly one of them must be present in a phrase. Note that this rule is *right-recursive*, since the head of the rule reappears on the right-hand side. This is a common trick to indicate any number of occurrences in the body of a phrase. Parameters must be typed so that the update function knows how to treat them. We've seen integer and floating-point parameters. In Section 0 we'll see examples of pointer parameters.

Rule statements have an optional name at the beginning of the rule. This name is the one that gets displayed in Saphira in the information window of the behavior, so you can tell what the rule is doing. Fuzzy expressions are built up from fuzzy variable symbols, floating-point numbers, and the boolean operators. Be sure to use floating-point 0.0 and 1.0 here, rather than integers 0 and 1, since the parser won't accept them. (We've already described most of the control set syntax in the two example behaviors above.)

Activity statements give the turn or forward/back activity of the behavior in terms of a fuzzy expression. Don't forget to include at least one of these, or the behavior will have no effect at all! Saphira also has *Goal* and *Progress* activities. We'll describe them in detail in the next few sections.

The update part of the behavior is a set of C statements. Since these statements are put into a C function, you can define local variables at the beginning. All of the parameters and fuzzy variables in the rules are automatically declared as static global variables, so you can access them in the update function.

There is one optional element at the top level of a behavior definition: the *Init* phrase. Like the update function, the *Init* function is a set of C statements that are evaluated by the behavior executive. The initialization only occurs once—when the behavior is first invoked, not when it may be enabled from the Behaviors window. It's used to set up global variables that may be required by the behavior.

Sometimes the *bgram* parser generates a C file, but then the C compiler complains of an error. If the error is in code generated by *bgram*, then you have to go back and decide what caused it. Most problems are going to be in the C code that you write for the update or *Init* functions, and typically are caused by forgetting to put local variable declarations at the beginning of the Update clauses. Of course, *bgram* won't protect you from making mistakes in writing C expressions.

Table 3-5 Behavior Grammar in BNF

```
/* Behavior definition: name params rules update activity */

BEHAVIOR:=
     “BeginBehavior” symbol
     “Params”   [PARAM_STMTS]
     “Rules”    [RULE_STMTS]
    [“Init”      C_STMTS]
     “Update”   [C_STMTS]
     “Activity” [ACT_STMTS]
     “EndBehavior”

/* behavior parameters */
PARAM_STMTS:=
       {“sfINT” | “sfFLOAT” | “sfPTR”} symbol [PARAM_STMTS]

/* Rule definition: name fuzzy-var action mod */
RULE_STMTS:=
      [SYMBOL] “If” FUZZY_EXP “Then” CONTROL [RULE_STMTS]

/* fuzzy expression */
FUZZY_EXP:=
       symbol
     | float
     | “Not” FUZZY_EXP
     | FUZZY_EXP “And” FUZZY_EXP
     | FUZZY_EXP “Or”  FUZZY_EXP
     | “(“ FUZZY_EXP “)”

/* rule actions and modifiers */
CONTROL:=
       “Turn Left”  [MOD]
     | “Turn Right” [MOD]
     | “Turn” symbol [MOD]
     | “Speed” MVAL

MOD:=
       “Very Slowly”
     | “Slowly”
     | “Moderately”
     | “Sharply”
     | “Very Sharply”
     | symbol

MVAL:=
       symbol | int | float

/* activity statements */
ACT_STMTS:=
      {“Turn” | “Speed” | “Goal” | “Progress”} FUZZY_EXP [ACT_STMTS]
```

3.6. Blending Behaviors

Now it's time to put our divide-and-conquer strategy to the test by building more complicated control strategies out of the simple behaviors we've already defined. Obviously, invoking two behaviors at the same time may have unforeseen effects, especially if they compete with each other for control of the robot. There are several ways of *blending* behaviors, combining their outputs so they produce a coordinated action. The key to blending is in defining an *activation level* or *activity* for each behavior. The activity is a fuzzy expression that is ANDed with the control set output of a behavior. As you know, fuzzy expressions are based on the state of the robot, and so the activation level for a behavior is a way of making the behavior *context-sensitive*, so that it is more or less active depending on the situation the robot finds itself in.

3.6.1. Behavior Activation

Let's consider the example of coordinating turning and moving forward in sequence. We want the robot to turn left 90 degrees, then move forward. We could do this by "hard-coding" the sequence of behaviors, so that the turn behavior happens first, followed by the constant velocity behavior. But it's more interesting if we start up both at the same time, and let the activation manalev take care of blending their controls.

To do this, we need two things: a way of deciding when the robot has reached 90 degrees from its original position, and a way of making this information change the activation level of two behaviors. Anticipating the discussion in the next section, we'll make use of the robot's dead-reckoning ability to measure how far it's turned. As given in Example 3-4, we first define a *turn90* behavior that turns the robot 90 degrees from its current heading. The behavior starts by calculating the goal angle in the absolute coordinate system of the robot. In the update function, the fuzzy variable *near_angle* is satisfied when the robot is facing the desired angle. Note the use of a global variable *turn90_angle* to preserve the goal angle across calls to the update function.

The rule turns the robot moderately in the direction of the input parameter, but only when it is not close to the goal angle. Hence, this behavior will stop turning the robot as it approaches the desired direction.

In the example, we invoke a constant velocity behavior at the same time as the turn behavior. The constant velocity behavior is nearly identical to our original example, but with the speed activity changed to be dependent of the fuzzy variable *near_angle*. (Note that *near_angle* is a global variable defined implicitly in the rules of *turn90*. It's okay to use such variables within the same file, but since they're defined statically, they won't be accessible in other files.) Consequently, the activation of constant velocity depends on how close the turning behavior brings the robot to the goal angle. The net result is that, when the robot is started up, it will turn left until it is almost 90 degrees from its starting position, then start to move forward. The two behaviors are *blended* to form a sequence, in which turning gradually gives way to forward motion.

While this use of activation achieves the desired result of sequencing the behaviors, it's not necessarily the best way to do it. At the end of this section we'll give some guidelines on when to use different forms of blending, and when direct sequencing of behaviors may be more efficient and easier to understand and debug.

Example 3-4 Turning and Moving

```
/* File: handler/src/samples/turn_go.c */

#include “saphira.h”

/* Define the behavior “sfTurn90”
 *  with one parameter, the desired direction */

float turn90_angle;          /* goal angle, in radians */

BeginBehavior sfTurn90           /* behavior name */
   Params sfINT tdir         /* sfLEFTTURN or sfRIGHTTURN */
   Rules                     /* just one rule */
       If Not near_angle Then Turn tdir Moderately
   Init                      /* calculate desired angle */
      if (tdir == sfLEFTTURN)
         turn90_angle = sfAddAngle(sfRobot.ath, ANG90);
      else
         turn90_angle = sfSubAngle(sfRobot.ath, ANG90);
   Update                    /* calculate fuzzy variables here */
       near_angle = f_eq(sfRobot.ath, turn90_angle, ANG90/10.0);
   Activity
       Turn Not near_angle
EndBehavior


BeginBehavior sfConstantVel  /* behavior name */
   Params sfFLOAT sval       /* input parameter, speed in mm/sec */
   Rules                     /* just one rule */
       If  too_fast Or too_slow  Then Speed sval
   Update                    /* calculate fuzzy variables here */
       too_fast = up_straight(current_vel(), sval, 50.0);
       too_slow = straight_down(current_val(), sval, 50.0);
   Activity
       Speed near_angle
EndBehavior


void
myConnectFn(void)
{
   sfInitBehavior(sfTurn90, “turn 90”, 0, 1, sfINT, sfLEFTTURN, sfEND);
   sfInitBehavior(sfConstantVel, “move”, 0, 1, sfFLOAT, 200.0, sfEND);
}

void
main(int argc, char **argv)
{
   sfOnConnectFn = myConnectFn;
   sfStartup(0);
}
```

3.6.2. Behavior Priority

Remember how, after invoking th constant velocity behavior, the robot continues to move forward even when the behavior is disabled from the Behaviors window?. To stop the robot, we introduce another behavior called *sfStopIt*, and invoke it at the same time as the constant velocity behavior in the Example 3-5. Create the client executable and connect it to the simulator or Pioneer. What did the robot do?

Example 3-5 The Stop Behavior

```
/*
 * Define the behavior “sfStopIt”
 *  no parameters
 *
 * File: handler/src/samples/stop.c
 */

#include “saphira.h”

BeginBehavior sfStopIt           /* behavior name */
    Params                   /* none */
    Rules                    /* just one rule, please */
         If 1.0 Then Speed 0.0
    Update                   /* nothing here */
    Activity
         Speed 1.0           /* affect the speed */
EndBehavior


!!! Include the definition of sfConstantVel here (see Example 3-4) !!!

void
myConnectFn(void)
{
    sfInitBehavior(sfConstantVel, “constant velocity”, 0, 1,
                      sfFLOAT, 200.0, sfEND);
    sfInitBehavior(sfStopIt, “stop”, 1, 1, sfEND);
}

void
main(int argc, char **argv)
{
    sfOnConnectFn = myConnectFn;
    sfStartup(0);
}
```

The robot should start up and move forward, just as if the stop behavior were absent. So what’s going on? What happened to the *sfStopIt* behavior? Take a closer look at the Behaviors window. It should indicate that the constant velocity behavior is active, but not the stop behavior. How did that happen, particularly since we invoked both at the same time?

Examine the Example 3-5 code. Notice that we gave the constant velocity behavior a priority of 0 (third argument of *sfInitBehavior*), while *sfStopIt* has priority 1. Behaviors with lower priority numbers get attention before behaviors with higher priority numbers.. All behaviors at each priority level have their control sets combined with OR, but to form an overall control level, activities at higher priority levels (lower priority number) cancel the activity of behaviors at higher levels. The action of *sfStopIt* is effectively suppressed by the higher priority constant velocity behavior.

If you disable constant velocity from the Behavior window, the stop behavior should become active immediately, because it is no longer being suppressed by a higher priority behavior , and the robot stops. Turn constant velocity back on, and the robot starts up again.

Priorities are one way of blending behaviors so that they produce useful actions. If we invoke the constant velocity and stop behaviors at the same priority level, they’ll compete and interfere with each other, so that the robot goes at a slower speed, or worse, oscillates between stopping and going forward (try it!).

Here are some things to keep in mind when deciding priority levels for your behaviors:

1. Priorities are non-negative integers.
2.

Be careful with the terminology vs. implementation: Behaviors with *lower priority numbers* have *higher priority* and may suppress the actions of those behaviors with higher priority numbers.

1. Priority suppression happens independently on each control channel. Hence, just because the constant velocity behavior may be active at priority level 0 doesn't mean that Saphira will suppress lower priority turning behaviors. This feature can be used to good advantage in many situations; for example, in a crowded area you may want to slow down the robot by suppressing the velocity, while leaving the turning behavior of the robot unaffected.
2. For obvious safety reasons, set obstacle avoidance and other emergency/safety behaviors to the highest priorities (priority numbers of 0 or 1) and set goal-achieving behaviors to lower priorities.

3.6.3. Blending Guidelines

1. Plan on a fixed priority scheme; for example, levels 0-2 for emergency maneuvers, 3-6 for goal acquisition, and so on.
2. Use internal activation levels, i.e., each behavior knows its own context of applicability.
3. Use multiple active behaviors with activation levels when a situation can occur repeatedly, e.g., obstacle avoidance; otherwise sequencing might be simpler to understand and debug.

3.7. Sequencing

Sequencing behaviors, as opposed to blending, is often the most direct and efficient way of forming complex robotic actions. We devote all of Chapter 6 to this important subject, but in anticipation we'll describe how to do basic sequencing from within a Saphira process.

Recall that a process is just a finite state machine that makes a transition every Saphira cycle (100 ms). The basic idea is to start up a behavior, and then continually monitor the state of the behavior from within the process until it succeeds (or fails), then start the next behavior in the sequence. The code for doing this isn't pretty, but the Procedural Reasoning System we describe in detail in Chapter 6 has a very nice interface for behavior sequencing.

3.7.1. Accessing Behavior Instances

The key piece of information we need for sequencing is how to access the internal state of an invoked behavior, to discover when it has succeeded or failed. In other words, we need access to the behavior instance. The function *sfInitBehavior* returns a pointer to this closure, so we can just set a variable to keep track of it: *p = sfInitBehavior(..).* Every time Saphira evaluates the behavior instance, all of its activities are kept as part of the behavior state, and Saphira gives us functions to access the state (Table 3-6). Each function returns the fuzzy value of the appropriate behavior instance variable. So, a behavior can communicate with other processes by setting its state, and letting these other processes read it.

Table 3-6 Behavior State Access Functions

float sfGoalActivity(BEHCLOSURE p)
float sfProgressActivitys(BEHCLOSURE p)
float sfTurnActivity(BEHCLOSURE p)
float sfSpeedActivity(BEHCLOSURE p)

3.7.2. A Sequencing Example

Now let's redefine the turning and moving action of Example 3-4, which is a blending of behavior activities, into a sequence of the two behaviors (Example 3-6). We' let you insert the turn behavior's definition—it's exactly the same as in the original example. The only change in the moving forward behavior definition is its activity: make it 1.0 in the sequencing example.

The salient feature of the sequencing approach to the turning and moving example is in *seq* process: It starts by invoking the turning behavior, then goes to a state in which it monitors this behavior until its turn activity is low. When that happens, the *seq* process disables the turn behavior, and invokes the move forward behavior.

Example 3-6 A Turn-and-Move Sequence

```
/* File: handler/src/samples/turn-seq.c */

#include “saphira.h”

BeginBehavior myConstantVel  /* behavior name */
   Params sfFLOAT sval       /* input parameter, speed in mm/sec */
   Rules                     /* just one rule, please */
       If  too_fast Or too_slow  Then Speed sval
   Update                    /* calculate fuzzy variables here */
       too_fast = up_straight(current_vel(), sval, 50.0);
       too_slow = straight_down(current_val(), sval, 50.0);
   Activity
       Speed 1.0
EndBehavior

void seq(void)               /* this is the sequencing process */
{
  BEHCLOSURE p;
  switch(process_state)
  {
    case INIT:
      p = sfInitBehavior(sfTurn90, “turn 90”, 0, 1, sfINT, sfLEFTTURN, sfEND);
      process_state = 20; break;
    case 20:
      if (sfTurnActivity(p) < 0.2) process_state = 30;    /* next behavior */
      break;
    case 30:
      sfKillBehavior(p);
      sfInitBehavior(sfConstantVel, “move”, 0, 1, sfFLOAT, 200.0, sfEND);
      break;
  }
}

void
myConnectFn(void)
{
   sfInitProcess(seq, “Sequence behaviors”);
}

void
main(int argc, char **argv)
{
   sfOnConnectFn = myConnectFn;
   sfStartup(0);
}
```

3.8. Using Artifacts within Behaviors

You’ll find that most behaviors act to have the robot respond to some object in the world—to avoid, follow it, or go through an object, for example. This is such a broad and important topic that we’ll devote a whole chapter to it (Chapter 5). But as a preview, we introduce the process here by writing a few behaviors that use the internal state of the robot, and in particular, the robot’s internal representation of objects, called *artifacts*.

Artifacts represent all sorts of things for the robot, including objects like chairs or environmental features, such as doorways, corridors, people, and even imaginary lines like the center of a corridor Saphira maintains two coordinate systems for all artifacts: a global coordinate system in which the position of the artifacts doesn't change (but the robot's position does), and a robot-centric system in which the relative position of the artifacts changes as the robot moves. (See ???.) The robot-centric system in which artifacts' positions change with respect to the robot (those in front get nearer as the robot moves forward, for example) is very useful for behaviors, since we can tell the robot how to move with robot-centered commands (*turn to your right* or *back up,* for example).

Another reason why artifacts are useful to behaviors is that they can give a graphic representation of the behavior the robot is supposed to achieve, making it much easier to debug. For example, if the robot is supposed to achieve a goal position, using a point artifact as the goal and displaying it on the screen with the robot gives immediate visual feedback as to how well the behavior is doing.

In this section we'll use just the simplest artifacts, point artifacts. These were introduced in Section ??, along with some functions for manipulating them. Point artifacts have a location and direction, although for now we'll ignore their direction, since it isn't important for the examples.

The two Saphira functions which create point artifacts are:

```
point *sfCreateGlobalPoint(float x, float y, float th)
point *sfCreateLocalPoint(float x, float y, float th)
```

The first uses Saphira's global ("world-centric") coordinate system, and the second uses Saphira's robot-centric coordinate system. All points have both sets of coordinates, and you can refer to whichever is the handiest.

There are several useful functions for computing geometric values from points. In our examples, we use *sfPointPhi(point *p)*, which gives the angle (in radians) from the center of the robot to the point, and *sfPointDist(point *p)*, which obtains the distance of the point from the center of the robot.

3.8.1. Example of Achieving a Goal Position

One of the most ubiquitous robot movements is to move to a goal position. A simple behavior which directs the robot to a goal position first turns it until it directly faces the goal, and then has the robot move forward. (This assumes, of course, that there's nothing between the robot and the goal. In keeping with our philosophy of simple behaviors, we make this assumption, with the understanding that another behavior will take care of obstacles.)

Our *go-to* behavior (Example 3-7) does just that: it turns the robot and moves it to a goal position. The Example has two notable features. First, like the Example 3-2, it blends its turning and moving forward behaviors, so that as the robot nears directly facing the goal it will also start to move. Second, the example behavior is able to *track* the goal, so that if the goal position moves, the robot will turn to follow it. This kind of behavior is useful when a higher priority behavior momentarily moves the robot away from the goal or if the goal position moves away, such as when the goal point is a person being tracked by a vision system and chased by the robot (humans rarely stand still when being approached by a robot).. Our *goto* Saphira example is our most complicated behavior to this point, with four rules: two for turning and two for velocity. The turning rules are simple: if the robot is pointing too far left of the goal, turn right, and *vice versa*. Velocity control is a little trickier, since we have to take care of three cases: one when the robot is pointing directly at the goal, but is still too far away; another for when the robot has achieved the goal; an a third case for when the robot is too far away, but not facing the goal directly. It's easy to get the right combinations using the fuzzy variables *near_goal*, *too_right*, and *too_left*.

The update function for calculating the fuzzy variables is straightforward, using the provided Saphira functions. Since *sfPointPhi* returns a negative angle (in radians) if the point is to the right of the robot, and a positive one if it's on the left, defining *too_left* and *too_right* are no problem (0.1 radian is about 6 degrees). The fuzzy variable *near_goal* uses the distance to the goal and the success radius, so that it's value is 1.0 when the robot is within the goal radius, and falls off to 0.0 when the robot is beyond twice the goal radius.

Finally, both the turning and velocity activities are high when the robot has not achieved the goal position. We also introduced the *Goal* activity, which is simply a way of reporting when the behavior has reached its goal. The goal activity becomes part of the behavior closure's state, and is a very useful way of reporting the behavior's success or failure to other procedures. We won't use it here, but it's a good idea to get in the habit of giving goal-achieving behaviors a *Goal* activity.

Copy the *goto.beh* file to *test.beh* and compile it (*make*) as in previous examples. Start up *client*, and connect it to the simulator or to the robot. You should see the robot icon in the Saphira Main window, with a goal position 1 meter in front and to the left. The robot will start by turning toward the goal, smoothly accelerate towards it, and then slow to a stop as it achieves the goal. Switch the display to global view (*Display* menu) and watch the robot move instead of the goal.

Even after the robot achieves the goal, the behavior is still present, but its activity is low. Click the left mouse button somewhere in the Saphira Main window. The *myButtonFn* function handles mouse clicks, so that the goal position jump to the mouse's cursor position. The behavior should reactivate and the robot will once again turn and move towards the goal. Imagine some perpetual process that updates the goal position, rather than your mouse, and you can see how to effectively implement a general tracking procedure.

Example 3-7 Moving to a Goal Position

```
/*
 * Define the behavior “goto”
 *   parameters: sfPTR goal point
 *               sfFLOAT success radius
 *
 * File: handler/src/samples/goto.beh
 */

#include “saphira.h”

BeginBehavior myGoto         /* behavior name */
    Params
        sfPTR goal_pt        /* pointer to goal point */
        sfFLOAT radius       /* how close we come, in mm */
    Rules
         If too_left Then Turn Right
         If too_right Then Turn Left
         If Not (near_goal Or too_left Or too_right) Then Speed 200.0
         If near_goal Or too_left Or too_right Then Speed 0.0
    Update
         float phi = sfPointPhi(goal_pt);
         float dist = sfPointDist(goal_pt);
         too_left = up_straight(phi, 0.1, 0.6);
         too_right = straight_down(phi, -0.6, -0.1);
         near_goal = straight_down(dist, radius, radius*2);
    Activity
         Turn Not near_goal
         Speed Not near_goal
         Goal near_goal
EndBehavior


static point *goal;          /* this is the goal position */

void
myConnectFn(void)
{
    goal = sfCreateLocalPoint(1000.0, 1000.0, 0.0);
    sfInitBehavior(myGoto, “go to point”, 0, 1,
                       sfPTR, goal,
                       sfFLOAT, 200.0, sfEND);
}

void
myButtonFn(int x, int y, int which)
{
    goal.x = sfButtonToRobotX(x,y);
    goal.y = sfButtonToRobotY(x,y);
    sfSetGlobalCoords(goal);
    return 1;
}

void
main(int argc, char **argv)
{
    sfOnConnectFn = myConnectFn;
    sfProcessButtonFn = myButtonFn;
    sfStartup(0);
}
```

3.9. A Complex Behavior Example

Here's a final example that exercises all the main points we've made about behaviors. The problem is to get the robot to move in a square, 1.5 meters on a side. The best way to do that might be to sequence behaviors for moving from one goal position to another, and one of the Exercises below asks that you do this. But, as a testament to the versatility of Saphira behaviors, let's alternatively solve the problem by invoking four behaviors simultaneously, and let the activation level take care of which behavior is in charge. Each behavior will guide the robot to one of the goal positions in the square, which we've set up beforehand. Of course, there has to be some sequencing mechanism for changing the activation, but we'll let each individual behavior decide when to switch, and which goal to switch to.

This example requires very little modification from the *goto* Example 3-7. We simply add another parameter, the next goal position, to the behavior, and keep track of the current goal in the global variable *current_goal* (Example 3-8). Check out the behavior *myGotoNext*, too. The behavior definition uses another fuzzy variable called *mygoal*, which is set to 1.0 when the behavior's goal is the current one, and 0.0 otherwise. The activity of the behavior is ANDed with *mygoal*, so that the behavior is only active when the current goal is its own. Also, the behavior is responsible for switching *current_goal* to the next goal, as soon as it approaches its own goal closely enough.

In the connection function, we first create all the goals of the square and give them names. We also set the current goal to be the one directly in front of the robot. Then we start up all four behaviors at once, each with its own goal and the goal of the next position.

That's all there is to our last example. When connected to the robot or simulator, the Saphira client has the robot move towards the first position. When it gets close, the robot's goal position switches to the second one, and so on. You can watch the behavior switching in the Saphira Behaviors window. Again, it's probably best to switch to global view in the *Display* menu, so that the robot moves among the goals positions on the display.

Exercise 3-4 Interesting Behaviors

(a) Write a behavior in which the robot moves in the square of Example 3-8, but use sequencing of behaviors in a process.

(b) Write a set of behaviors that will move the robot forward and randomly turn it in a different direction from time to time.

(c) The state of Pioneer's motors can be queried with the function *sfStalledMotor(n)*, where *n* is *sfLEFT* or *sfRIGHT*. This function returns 1 if the corresponding motor is stalled, and 0 if it is not. Using this function, write a behavior or set of behaviors that will move the robot forward, and if the robot bumps into something, will back up, turn appropriately, and continue on its way.

Example 3-8 Moving in a square

```
                                   /*
 * Define the behavior “myGotoNext”
 *   parameters: sfPTR goal point
 *               sfPTR next goal
 *               sfFLOAT success radius
 *
 * File: handler/src/samples/square.beh
 */

#include “saphira.h”

static point *current_goal;  /* this is the goal position */

BeginBehavior myGotoNext     /* behavior name */
    Params
        sfPTR goal_pt         /* pointer to goal point */
        sfPTR next_pt         /* pointer to next goal */
        sfFLOAT radius        /* how close we come, in mm */
    Rules
         If too_left Then Turn Right
         If too_right Then Turn Left
         If Not (near_goal Or too_left Or too_right) Then Speed 200.0
         If near_goal Or too_left Or too_right Then Speed 0.0
    Update
         float phi = sfPointPhi(goal_pt);
         float dist = sfPointDist(goal_pt);
         too_left = up_straight(phi, 0.1, 0.6);
         too_right = straight_down(phi, -0.6, -0.1);
         near_goal = straight_down(dist, radius, radius*2);
         if (current_goal == goal_pt)
            if (near_goal > 0.8)     /* switch goals on success */
                current_goal = next_goal;
         mygoal = current_goal == goal_pt ? 1.0 : 0.0;
    Activity
         Turn  mygoal And Not near_goal
         Speed mygoal And Not near_goal
         Goal  near_goal
EndBehavior

void
myConnectFn(void)
{
    point *p1, *p2, *p3, *p4;/* positions on the square */
    p1 = sfCreateLocalPoint(0.0, 0.0, 0.0);
    p2 = sfCreateLocalPoint(1500.0, 0.0, 0.0);
    p3 = sfCreateLocalPoint(1500.0, 1500.0, 0.0);
    p4 = sfCreateLocalPoint(0.0, 1500.0, 0.0);
    current_goal = p2;       /* start by going to p2 */
    sfInitBehavior(myGotoNext, “go to 1”, 0, 1, sfPTR, p1, sfPTR, p2,
                      sfFLOAT, 200.0, sfEND);
    sfInitBehavior(myGotoNext, “go to 2”, 0, 1, sfPTR, p2, sfPTR, p3,
                      sfFLOAT, 200.0, sfEND);
    sfInitBehavior(myGotoNext, “go to 3”, 0, 1, sfPTR, p3, sfPTR, p4,
                      sfFLOAT, 200.0, sfEND);
    sfInitBehavior(myGotoNext, “go to 4”, 0, 1, sfPTR, p4, sfPTR, p1,
                      sfFLOAT, 200.0, sfEND);
}

void
main(int argc, char **argv)
{
    sfOnConnectFn = myConnectFn;
    sfStartup(0);
}
```

3.10. Multivalued Logic and Fuzzy Control Theory

This section describes the theory behind our implementation of fuzzy control. The treatment here is brief, and lists the main points; for a more thorough exposition see [Saffiotti et al 1996].

We use the framework of multivalued logics [lukasiewicz.30,rescher.book69] . By representing degrees of truth on a numerical scale, multivalued logics provide an ideal framework to merge notions taken from the world of planning, typically expressed in symbolic terms, with notions taken from the world of control, typically expressed in numerical terms.

Multivalued logics can be viewed as logics of graded preference [rescher.67] where we interpret the truth value of a proposition in a world as the utility, or *desirability* , of being in that world from the point of view of the proposition [bellman.80,ruspini.ijar91,ruspini.uai91] . Accordingly, we use propositions to represent control strategies and goals, and logical connectives to combine them.

Let P be a propositional language and *S* a set of states, and further let *f* be a function that assigns a truth value in [0,1] to each atomic proposition in each state. We extend *f* to non-atomic propositions by the equations

(P, s) = & 1 - (P,s)
(P Q, s) = & (P,s) (Q,s)
(P Q, s) = & (P,s) (Q,s)
(P Q, s) = & (Q,s) (P,s)

Equation 3` 1

$$f(\neg P, s) = 1 - f(P, s)$$
$$f(P \wedge Q), s) = f(P, s) \otimes f(Q, s)$$
$$f(P \vee Q), s) = f(P, s) \oplus f(Q, s)$$
$$f(P \supset Q, s) = f(Q, s)$$

where is any continuous triangular norm, or t-norm , with quasi-inverse , and is any continuous t-conorm .

T-norms and t-conorms are used as a generalization of logical conjunction and disjunction, respectively [schweizer.book,weber.fss83,valverde.fss85] . Mathematically, a t-norm is any binary operator on that is commutative, associative, non-decreasing in each argument, and has as unit; a t-conorm has the same properties but has as unit.

Given a continuous t-norm , its quasi-inverse is defined by
x y = [0,1] y x .
Note that, for any , implies and implies .

The following are examples of t-norms and t-conorms.

The operators in (c) are the ones originally used by Lukasiewicz in its logic [lukasiewicz.30] . Those in (a) are the ones we use in Saphira. Notice that if we restrict truth values to be either 0 or 1, the conditions (mov.tv.eqn) reduce to classical logic for any choice of , and .

Given a proposition , we are often interested in the set of states of where holds, denoted by . As truth values are numbers in , the membership of any state to is a matter of degree. We measure this degree by the membership function defined by
f_ P (s) = (P,s) .

Graded sets of this type are commonly known as fuzzy sets [zadeh.65] . For notational simplicity, and when there is no risk of ambiguity, we will denote the fuzzy set simply by , and write
for .

We define complement, intersection and union of fuzzy sets based on the equations (mov.tv.eqn) above as follows:
equation mov.fs.eqn
P (s) &=& 1 - P(s)
(P Q)(s) &=& P(s) Q(s)
(P Q)(s) &=& P(s) Q(s)
We also define set inclusion by
P Q iff
P(s) Q(s) for all in .

If we use the operators (a) in Example mov.tnorm.ex , these operations correspond to those originally defined by Zadeh [zadeh.65] .

In what follows, we will sometimes refer to a proposition as a desirability function on to emphasize the interpretation of as the desirability of being in state from the viewpoint . We will use routinely the , and operators to combine desirability functions. The reader should keep in mind that is meant to capture the notion of conjunction, the notion of disjunction, and the notion of implication (right to left). We will also use the sup (least upper bound) and inf (greatest lower bound) operators to capture the notions of existential and universal quantification, respectively.

Consider an agent with a set of possible internal states, and a set of (atomic) control actions. A control strategy for this agent is generally defined by a function that produces, in each state, a control action. To define generic types of movements --- movements that can be instantiated in several ways depending on the circumstances of execution --- we need to be less rigid. To fix the ideas, consider the type of movement ``take a step." We may well have an archetype of the ideal step, but when we take a step on a muddy road we may perform a movement that only vaguely meets this archetype; still, this inelegant maneuver is probably the best choice available in that situation, and we want our definition of ``take a step" to account for it. We extend the notion of a control strategy to produce, in each state, a value of desirability of each possible control.

defn mov.schema.def

Let be a set of states, and a set of control actions. A control schema on and is any function

$$D: S \to A\,[0,1]$$

Intuitively, the value of measures how much executing in state is desirable from the viewpoint of performing that type of movement. Figure mov.des.fig illustrates two control schemas. In (a), we consider a control schema, called Follow , for proceeding within a given ``lane" (represented by the double lines); (b) refers to the Keep-Off control schema, intended to stay away from a given spot. The picture is a representation of the current state , with the agent being located at the small dot. Each vector indicates a possible turning action : the length of the vector is proportional to the desirability .

Note that the control actions in are meant to be directly executable low-level commands. In the case of the robot, the elements of are the turning and velocity control values to be sent to the wheel effectors at each control cycle (100 msec).

A desirability function is a descriptive device. Given a control schema , any actual implementation of a type of movement described by will have to select, at each state, one control action to send to the effectors. We represent such an implementation by a control function

F_D : S A .

Selecting actions by the function corresponds to choosing and following one of the desired trajectories, , to execute one particular movement of type . Of course, we would like to choose the ``best" trajectory. This is a search problem, and is an intractable computation in general, as the size of the state space grows exponential with its dimensionality. Similar problems occur with potential field methods (see Section mov.fields.sec). There are many different approximate search methods that could be employed to find candidate trajectories.

We have built an implementation of control schemas for our mobile robot Flakey by using techniques based on fuzzy control [ruspini.iizuka90,saffiotti.ieee93,saffiotti.sri93a]. A control schema is encoded by a set of fuzzy rules of the form

IF P_i THEN A_i , i = 1, ,n ,

where each is a proposition in multivalued logic, and each is a fuzzy set of control actions. From these fuzzy rules, a desirability function can be computed by

mov.ruleset.eqn

D_R(s,a) = 1 i n (P_i(s), A_i(a)) .

Intuitively, says that is a desirable control in if there is some rule in that supports and whose antecedent is true in . The process of choosing one action from this desirability function is called defuzzification . We have used the centroid method, which computes a desirability-weighted average control

mov.defuzz.eqn

F_R(s) = a D(s,a) d a

D(s,a) d a .

For averaging to make sense, the rules should not suggest dramatically opposite actions in the same state. Our coding heuristic has been to make sure that rules with conflicting consequents have disjoint antecedents. Other authors have preferred to use more involved choice functions (, yen.92).

3.11. Historical Notes and Related Work

The field of planning and control has burgeoned in the last few years, and many new ideas have emerged, especially in the area of reactive planning. The work we have presented owes much to previous work, and we have been influenced by methodologies and specific systems. In this section we give an overview of the points of contact, and draw attention to the distinct features of the multivalued approach to complex controllers.

3.11.1. Methodologies

Both the subsumption architecture of Brooks and his students [brooks86,connell90] and the situated automata of Rosenschein and Kaelbling [rosenschein87,kaelbling90] are methodologies for producing embedded agents that perform complex tasks. In part we have borrowed from these in developing the complex behavior methodology, especially the subsumption idea of decomposing complex behavior into the composition of simple behaviors. In part we are in conflict with the spirit of these methodologies, in preferring explicit model-based perception and analogical representations of the world as part of embedding the controller.

The theory of situated automata is a formal methodology for constructing embedded agents by representing the environment of the agent, its task, and its capabilities. An automata is constructed to perform a task by considering these specifications. The general form of the automata is a finite state machine consisting of an update function and an action function with bounded execution times [kaelbling.aaai88] . Situated automata theory is an abstraction away from the traditional planning approach in that it makes no commitment to an internal state that represents the world in an analogical fashion, i.e., that attempts to model surfaces, recognize objects, and so forth. The key observation of situated automata theory is that the state of the agent should contain just enough information to accomplish the specified tasks of the agent in its environment; and this information need not be in an analogical form.

Situated automata theory is not incompa ible with our multivalued logic (MVL) approach to complex controllers; it just makes no commitment to any kind of internal architecture or representation. Ideally, we would like to be able to prove, using the techniques of situated automata, that any particular complex controller we design actually accomplishes its task in the intended environments. Further, we would like to be able to synthesize complex controllers that provably accomplish their goals. But the current state of situated automata theory is not developed enough to satisfy such an ambitious program. Its main practical success has been a suite of development tools: REX, to produce bounded-depth combinatorial circuits from low-level descriptions; and GAPPS and RULER [kaelbling.aaai88] for generating circuit descriptions from more abstract goals. GAPPS has been used to generate controllers for mobile robot navigation; we discuss it below.

The subsumption architecture has many points in common with situated automata theory, but without the formal emphasis of the latter. It is a task-oriented methodology for constructing agents. Each task is accomplished by a behavior, which integrates sensing, computation, and acting. The subsumption architecture is even stronger than situated automata theory in that it rejects the idea of a central, analogical representation of the environment. Each behavior is responsible for extracting needed information from the sensors, processing it in a task-dependent manner, and producing control actions. Behaviors are organized hierarchically, with the lowest level behaviors responsible for maintaining the viability of the agent, and the higher levels pursuing more purposeful goals. The idea is that if the higher levels cannot provide guidance, lower levels still cause the agent to act reasonably, e.g., not bump into obstacles. When a higher-level is active, can suppress more primitive behaviors below it. This vertical decomposition by behavior or task is contrasted with the horizontal decomposition of the traditional architecture, with its expensive and nonreactive perceive/plan/execute cycle.

The subsumption architecture has been highly influential in the mobile robotics community, and its ideas have permeated most of the proposed architectures to some extent. We have incorporated the concept of vertical decomposition into the way in which behaviors interact with sensing and perception in : more reactive behaviors can access raw sensor readings, while more purposeful behaviors use more complex perceptual routines. In fact, the very notion of behaviors themselves is in the spirit of subsumption architecture, since each behavior is oriented towards accomplishing a particular goal. But MVL differs in several important respects from the subsumption architecture. The most obvious one is a commitment to embedding and goal abstraction by means of a perceptual subsystem shared by all behaviors. Without such a subsystem, it is difficult to coordinate reactive and purposeful behavior in a general way, or to abstract the goals of behaviors so deliberation processes can make use of them, or to have behaviors whose purpose is to facilitate perceptual processes of recognition and anchoring. We note that there are some impressive results without such representation, e.g., Connell's can-retrieving robot [connell90] and Mataric's naviga ion experiments [mataric90] . The second main difference is in the formal properties of MVL, which we have exploited to show how behaviors can be combined to accomplish more relaxed goals in wider contexts.

3.11.2. Control Systems

Here we discuss some particular types of controllers for complex systems, including artificial potential fields, situation-action controllers, and the "circuit semantics" systems that are inspired by the subsumption architecture.

Potential Fields

Probably the most influential type of complex controller is based on the so-called ``artificial potential fields," first introduced by Khatib [khatib.86] and now extensively used in the robotic domain latombe.book . In the potential field approach, a goal is represented by a potential representing the desirability of each state from that goal's viewpoint. For example, the goal of avoiding obstacles is represented by a potential field having maximum value around the obstacles; and the goal of reaching a given location is represented by a field having minimum value at that location. At each point, the robot responds to a pseudo-force proportional to the vector gradient of the field.

The major technical difference between our MVL approach and potential-field methods is how they combine information. Potential fields are combined by linear superposition: one takes a weighted vector sum of the associated pseudo-forces. Each force is a summary of the preferences that produced that force (, which direction is best to avoid an obstacle), and the combined force is a combination of the summaries. In contrast, when combining control schemas one takes the t-norm of the two desirability functions, obtaining an assignment of utility values to each possible control. We have seen in Section mov.blend.sec that these two forms of combination can produce different results. A second difference is that we express both goals and applicability conditions as formulae in a logical language. Complex goals and constraints can often be described more easily in a logical form than in the analytical form of a potential field function. Moreover, the ability to specify the context as a logical sentence makes integration with classical symbolic planning techniques easier. Potential-field approaches typically do not try to make contact with the symbolic planning methodologies.

A system based on potential fields, but which also has many points of contact with our approach is AuRA, developed by Arkin [arkin.87,arkin.smc90] . Arkin adopts Arbib's notion of motor schema, discussed in the Introduction. As we have noted, motor schemas are similar to our concept of behavior, having the same basic components of control function, activation level, and perceptual parameters. However, AuRA implements control functions by using pseudo-forces instead of desirability functions. Correspondingly, AuRA takes a more limited approach to integrating planning with motor schemas. AuRA's planner is basically a path planner that generates a piece-wise linear path to the goal. This plan is then passed to the execution layer, where motor schemas to follow this path leg by leg are dynamically chosen and instantiated at execution time. In our approach, by contrast, the planner generates complex compositions of behavior schemas, expected to orient the agent toward the achievement of given goals. By contrast, the approach we propose is not limited to navigation tasks.

Situation-action plans

A number of authors have claimed that plans to be executed by agents situated in the real world are better expressed in the form of `` situation action " rules. The triangle tables developed in early robot planning work strips already incorporated the idea to represent, together with each action in the plan, the condition under which that action should be activated. More recently the idea of plans as sets of situation-action rules that specify reactions to situations has been clearly spelled out and extensively developed [suchman.book,schoppers.ijcai87,drummond.kr89,payton.smc90] . Our behavioral plans belong to this category.

Most of the current agent architectures based on situation-action rules put these rules inside the control loop. This implies that the evaluation of the situation should be performed in bounded time, and that the actions should be elementary action steps. By considering this, Nilsson has recently extended the idea of triangle tables to develop a new formalism for plans, called teleo-reactive trees . A teleo-reactive tree encodes a set of situation-action rules, ordered by a subgoal relation: the activation of each rule in the tree is expected to eventually produce the conditions for the activation of its parent rule. Teleo-reactive trees resemble the behavioral plans that we have built in Section planning.sec . In fact, a behavioral plan can be seen as a generalization of a TR-tree where contexts can take intermediate degree of truth, desirability functions are multivalued, and we can have concurrent activation of behaviors.

Drummond [drummond.kr89] has proposed a form of situation-action rules called Situated Control Rules (SCR). SCRs are similar to our control schemas in that they are treated as constraints that limit, rather than determine, the behavior produced by the execution module. However, SCRs are not blended into trade-offs, and conflicts are resolved by completely deactivating some SCRs. Drummond's architecture incorporates an interesting runtime projection mechanism to simulate future executions and prevent undesired effects. It would be interesting to add a similar mechanism to our multivalued controllers.

Schoppers [schoppers.ijcai87] views planning as the task of partitioning (part of) the set of possible environmental situations according to the reaction that the agent should produce from the viewpoint of achieving the given goal. His universal plans specify a reaction to each possible situation that the agent can encounter, and can be thought of as controllers that provide an input-output mapping specific to the achievement of a given goal. Behavioral plans can be seen as a more control-oriented form of universal plans. The way we generate behavioral plans, by extending the context of an initial behavior to cover more and more situations, is similar to Schopper's notion of space partitioning. It is natural to ask how much we should enlarge this context. Two extreme solutions are to stop as soon as the context includes the current situation, or to continue until we cover the entire state space, as proposed by Schoppers.

A common feature of the approaches based on situation-action rules is that situation-action plans can be compiled into executable structures. Synthesis is a convenient property, because it automates the process of producing complex controllers. Another method for synthesizing complex controllers by compiling plans is GAPPS [kaelbling.aaai88] . Given a top-level goal, the GAPPS compiler reduces it using the goal-reduction rules into a set of condition-action rules for primitive control actions. Later development of GAPPS incorporated goal-regression compilation using operator descriptions [kaelbling.tr90] , which gives GAPPS a more abstract way of representing action, and enables it to do predictive planning. The two synthesis methods we explored for , goal-regression planning and run-time deliberation, are similar to GAPPS goal-regression and goal-reduction methods, respectively. Where the two systems differ is in the level of detail of complex action specification. We have concentrated on how control schemas combine to produce complex behavior. GAPPS control actions are simple effector commands, and conjoining controls is only possible if they produce the same control action, or one of the control actions is unspecified. Although it is conceivable that GAPPS could be used to program multivalued logic control schemas, it would require the same development as we have presented here.