Everything You Always Wanted To Know About Programming Behaviors But Were Afraid To Ask

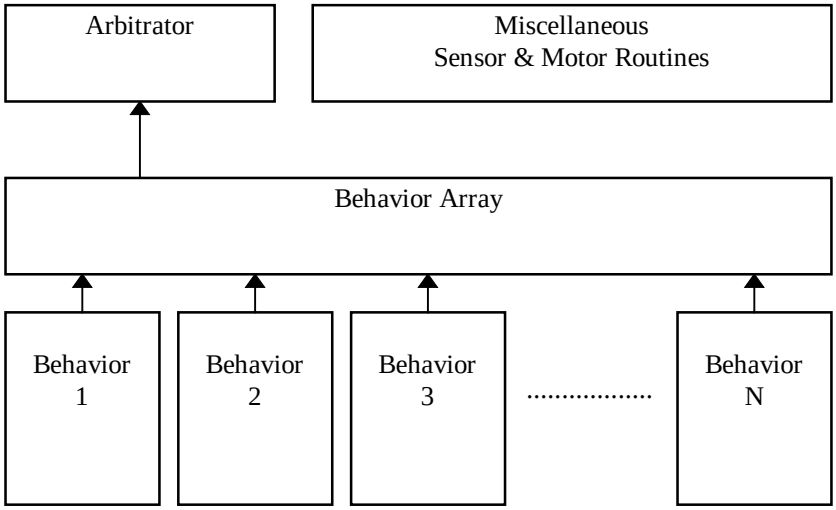
By Kevin Harrelson

Machine Intelligence Lab
University of Florida
Spring, 1995

Overview

Programming multiple behaviors in IC may seem at first to be a daunting task, but it can be easily broken up into several steps. The multitasking capabilities of IC even facilitate this process. This paper will attempt to explain the steps necessary to create these behaviors.

General System Design



In my experience, this is a design for the behaviors that I highly recommend. It consists of several separate, but interrelated parts.

The approach outlined here offers many advantages. Perhaps the greatest is that the behaviors are completely independent routines. The suggestions outlined here provide a framework upon which to hang behaviors. This will make it easy to add, remove, or change behaviors. It also provides a method to allow the behaviors to interact with each other. Since the behaviors can work out for themselves which one should be most important at any time, it will not be necessary to change the logic of the arbitrator whenever a new behavior is added.

Low-Level Routines

A class of low-level routines are needed to perform various housekeeping functions within the robot. The purpose of these routines is not to provide any sort of logic or decision-making capabilities, but to perform some of the basic tasks that are needed in order to make the processes work properly.

Sensor Routines

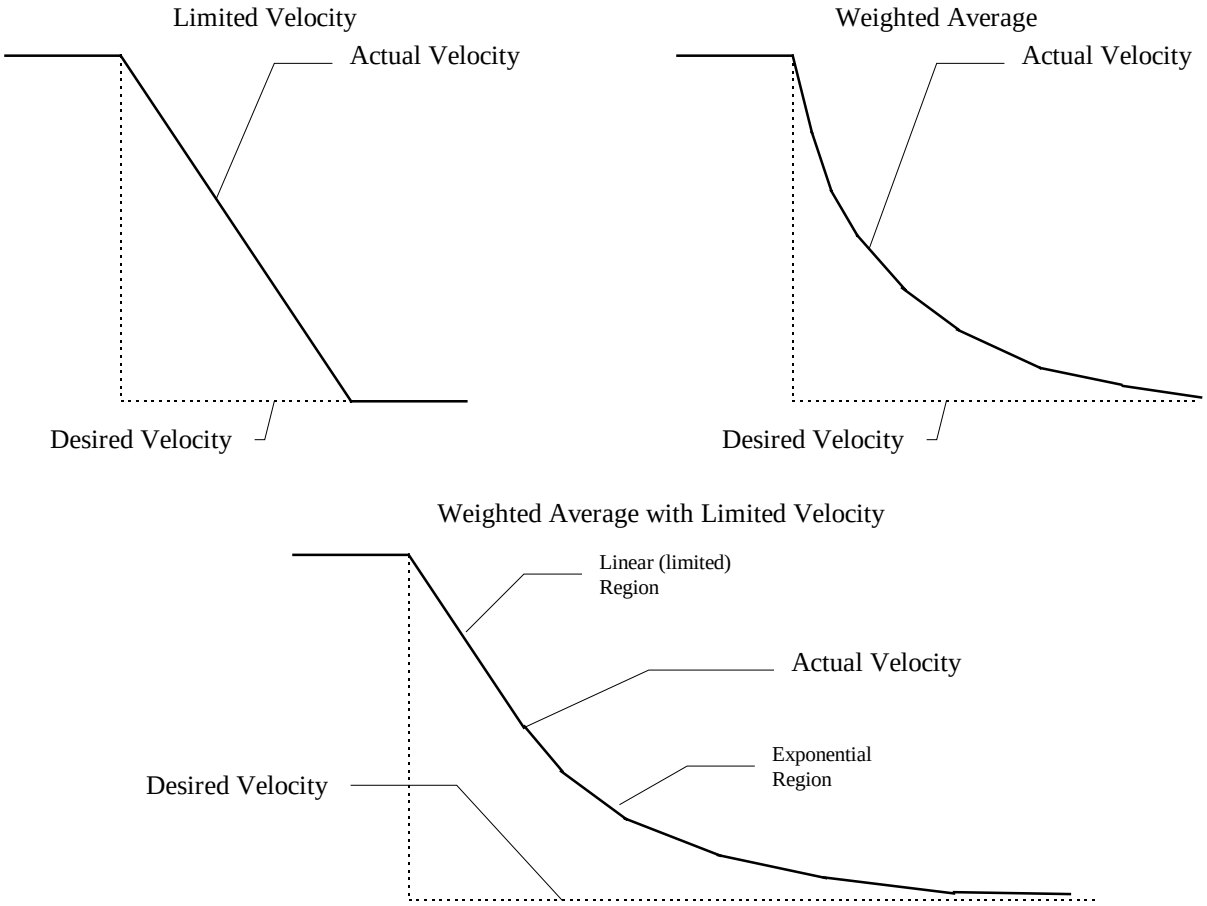
At the lowest level, some routines will be necessary to control the sensors. The first job of these routines is to take all data from the sensors, and place them in descriptive global variables, such as IR_COLLISION_RIGHT, or PYRO_ANGLE. This will allow all behaviors that depend on these sensor readings to be much more readable and easier to debug. This will also allow others to use the same code with a minimal amount of modification.

It will also probably be necessary to have some sensor control routines. Many sensors require some sort of service. There may be several different IR systems, which should not operate at the same time, or there may be a sensor mounted on a servo, which needs to be controlled. The control and scheduling of sensor operations should not be left up to the behaviors. Instead, separate routines should control these functions. This way, there will not be any conflicts arising from two different behaviors trying to do two different things to the same sensor at the same time. If it is necessary, the behaviors can communicate with the sensor routine by using global variables.

Motor Routines

The motor control routines should be placed in an independent routine. No behaviors should ever control the motors. Instead, the behaviors should send their requested motor valued to variables. This will eliminate any conflict caused by behaviors fighting over the motors. This will also make it easier to create smooth motor control. One problem that may be encountered while controlling the motors is a sudden change of the velocity. This may be caused by a behavior, or by different behaviors being active. In any case, this sudden, jerky motion is undesirable not only for aesthetics, but also because the motor current will increase substantially, and may cause the board to reset if the batteries begin to run down.

One way to do this may be to limit the acceleration of to motors to a certain value. This may be done by recording the last known speed sent to the motor. Then, if the new speed is different, change the current motor speed by some small value. Another way might be to use a weighted average method. For example, for a 1/20 weighting, the formula might be $\text{Speed} = \frac{19 \times \text{Old} + \text{Desired}}{20}$. This approach will create a smooth transition as the current speed approaches the desired speed, but may be jerky initially. An improved solution may be to use averaging, and limit the acceleration afterwards.



These are only suggestions, and not necessarily the best ones. Perhaps a routine could be written that would limit the change in acceleration ("jerk"). The key is in experimenting and trying to determine what works best.

Behavior Control

At the heart of the robot are several behaviors. These can be extremely simple behaviors, such as one to simply avoid an obstacle, or follow a wall, or a complex one that can calculate the direction to a goal. One of the more difficult problems in programming a robot is determining which behavior should be active at various times. If the guidelines in this paper are followed, then adding new behaviors and integrating them into the existing ones should be relatively straightforward.

In simple terms, each behavior runs at the same time. Each behavior will act upon the sensor data, and produce an output. This output will then be placed in the behavior array, along with the outputs of all of the other behaviors. Then a master program, called the arbitrator, will examine all of the outputs and decide which one should have control of the robot.

The Behaviors

A behavior is a program that is designed to have the robot perform one simple action. Several examples are programs that might follow a wall, follow a light, avoid a light, seek out other robots, circle an object, or get out of a tight spot. In essence, a behavior is just an algorithm that will take the sensor inputs, and produce the outputs corresponding to the desired motor speeds. In addition, a behavior should produce a priority output. This output should correspond to how important a behavior thinks that it should be. A collision avoidance behavior should have a high priority if a collision is imminent, but should have a low priority if no obstacles are detected.

The Behavior Array

The behavior array is, in essence, the heart of the system. The purpose of the array is to hold all of the output from the behaviors, so that the arbitrator can examine them and determine which one should be active. The array is configured to be N×M, where N is the number of different behaviors present on the robot, and M is the number of parameters and outputs associated with each robot. A sample behavior array is shown below.

A Sample Behavior Array

Behavior	1: Collision	2: Wall Follow	.....	N: Find Beacon
Motor Left	-20	40	.....	60
Motor Right	80	40	.....	20
Priority	100	22	.....	8
Strength	0	5	.....	12
Turns	3	0	.....	0

In this simple example, the first two rows consists of the actual output of the behaviors. In this example, the output is motor left and motor right. It might also have been forward velocity and rotational velocity, or any other output.

The priority is a number that the behavior itself generates. The collision behavior gives itself a priority of 100, which means that it must think that what it has to say is important. This must be because of an imminent collision. The wall follow has a priority of 22, which must mean that it has found a wall. The find goal routine has a priority of 8. This could mean that it has found a goal, but that it is far away. Please note that the range of priorities does not necessarily have to be the same. Perhaps the find beacon behavior can only have a maximum priority of 15, while wall follow could have a maximum priority of 40. Also, the numbers chosen for this example are completely arbitrary. A real robot might have priorities from 0-255, or from 0-10,000.

The next line is the strength. This line allows behaviors to modify the relative value of other behaviors. In effect, the strength becomes a type of multiplier for the

priority. This will allow a wall following routine to, in effect, turn down the priority of the collision behavior. Also, a wall following behavior could find a sudden end to it's wall, and promote a turn corner behavior so that it has an almost certain chance of becoming dominant.

Since integer math is far easier to do than floating point, some kind of allowances need to be made. Simply having the strength as a multiplier may not be the most effective method. This would not allow a behavior to promote another one by 10%. It would only allow promotion or demotion in 100% increments. Also, a simple demoting should probably not be done linearly. It would seem far more intuitive to do a "divide by" operation.

One good compromise would be to have the initial strength as zero. Each point would then be 10% of the priority. If the strength were a positive number, then the priority could be increased by that percentage. For a negative number, a decrease in priority would result. For positive numbers, the equation would be $\text{new\_priority} = \text{old\_priority} \times \left(1 + \frac{\text{strength}}{10}\right)$. For a negative number, the equation would be $\text{new\_priority} = \frac{\text{old\_Priority}}{\left(1 + \frac{|\text{strength}|}{10}\right)}$.

The strength value could also be used to implement a subsumption type architecture. Using this, behaviors could be created that would, upon command, suppress or strengthen other behaviors upon command.

The last row in the sample behavior array is turns. Each time the arbitrator determines the dominant behavior, it would increment the turns counter of the winner, and reset the turns counter of the losers to zero. This would result in a counter of the number of consecutive turns that a behavior has been dominant. This would allow another behavior to reduce the strength of a behavior that has been dominant too long.

The previous example was just a sample, and was not intended to be the "correct" solution. There are many more things that could be done. Perhaps, in addition to a counter for the number of turns, adding a counter for the number of turns since a behavior was last dominant, or even a counter that counts the total number of turn it has been dominant since the robot was turned on.

Since the behavior array is a two dimensional array, which IC cannot handle, it will be necessary to simulate a two dimensional array. Please refer to the *Creating Virtual Multi-Dimensional Arrays in IC* section.

The Arbitrator

The job of the arbitrator is extremely easy. After all of the behaviors have calculated their output, the arbitrator simply steps through the output of each behavior, and calculates the modified priority, based on that behavior's priority and strength values. Then, it determines the dominant behavior. It may also need to modify any additional counters, such as the Turns counter. It should then place the number of the dominant behavior in a global variable, such as dominant_behavior. Then, the motor routines would know which behavior to grab the output from.

Another possibility would be to have the arbitrator calculate an adjusted priority, based on the priority and strength values. Then, this adjusted priority could be stored in the behavior array. That would allow the motor routine to take a weighted average, based on the adjusted priority. This might result in smoother operation.

Programming Advice

Perhaps the most important thing is to keep the behaviors relocatable. This means that the behavior's own behavior number should be stored in a variable at the beginning of the routine. It that behavior is removed, then the other behaviors could be easily renumbered accordingly. That would also allow a behavior to be transferred from robot to robot with a minimum of difficulty. This rule also applied to behaviors that modify the strengths of other behaviors. All behavior number should be in variables at the beginning of the routine.

It is also possible that one or more behaviors might be extremely process intensive to the point of slowing all other processes down. In a case like this, such behaviors can examine the dominant_behavior output of the arbitrator. The behavior could generate its strength, and then only generate its actual output if it is dominant. Such a scheme would result in a small, one processing cycle lag when it is activated, but may be the only option when dealing with extremely complex behaviors.

Creating Virtual Multi-dimensional Arrays in IC

While creating behaviors and programs in IC, a multi-dimensional array may be necessary. Unfortunately, IC does not have this capability, but it can be added without too much work. The trick is to create a one-dimensional array that is large enough, and map it to a virtual multi-dimensional array.

In the following examples, all uppercase letters represent the maximum dimension size of an array. Lowercase letters represent a specific element. An array of A×B×C is a three-dimensional array. Also, a×b×c is an element of the array A×B×C. Also, all uppercase letters assume that a 1 is the lowest allowed number. All lowercase numbers assume that 0 is the lowest allowed number. For example, an array of 5×5 has the

smallest element of 0×0, and the largest element of 4×4. Also, the array X[] will be the one-dimensional array that holds the virtual array.

Creating a virtual multi-dimensional array simply requires some multiplication and addition. For example, the array A×B×C×D can be found be represented by an array of size X[A*B*C*D]. To access the element a×b×c×d in this virtual array, access X[((a*B+b)*C+c)*D+d]. The pattern in this formula should be readily observable, and may be extended to any number of dimensions.

Genesis of These Ideas

These ideas were formed from the construction of my robot, Medusa. My robot only had a couple of behaviors. At the end of the project, my behaviors had begun to be organized in a manner similar to the one presented here. Upon completion of the project, I deliberated long and hard over what I would have done differently if I could do the project over again. I will, in fact, use these same ideas for my next project.