Conflict resolution

In the previous chapter, we discussed how to combine control policies using guarded parallelism. The combination of a control policy and a guard (trigger, activation level) was called a behavior. These behaviors can then be combined by running them in parallel and letting them freely self-activate. This is what we called the naïve behavior-based system model. On a good day, at least, this allows us to use multiple simple behaviors to solve complicated tasks. Again, the general idea of behavior-based systems is to have a number of locally competent controllers (behaviors) that are each responsible for controlling the robot in a narrow set of circumstances. We think of the behaviors as being self-activating by some kind of internal triggering mechanism. However, some kind of external control of the behaviors is almost always necessary to handle cases where two behaviors self-activate simultaneously. These are variously referred to in the literature as action-selection mechanisms, arbitration mechanisms, or behavior-based architectures. We will refer to them as conflict resolution mechanisms. A number of conflict resolution mechanisms have been proposed over the years.

One important issue is whether the activation level of a behavior is a Boolean quantity or a continuous quantity. In the Boolean case, activation level is just like a switch: it simply represents whether the behavior is in some sense “reasonable” in the current situation, whether it actually runs or not. In the continuous, or graded case, behaviors can have varying degrees of activation and the activation level represents the desirability or ferventness of the behavior.

Prioritization
The simplest technique for conflict resolution is to say that some behaviors are more important than others. This can be accomplished with a simple conditional. If we take the case of the frog feeding behaviors from the previous chapter, we implement prioritization by saying:

```
(define-signal active-behavior-motor-vector
  (cond ((activation-level eat)
         (motor-vector eat))
        ((activation-level face-prey)
         (motor-vector face-prey))
        (else
         do-nothing-motor-vector)))
```

where do-nothing-motor-vector is defined as:

```
(define-signal do-nothing-motor-vector
  (frog-motor-vector sit-still #f))
```

Active-behavior-motor-vector will take the motor vector from whichever behavior it finds active first. It if doesn’t find an active behavior, it does nothing.
While this is nice and simple, we’re computer scientists and we couldn’t hold our heads up in public if we coded the control system in a klugy way like this. We want to have a reusable software abstraction for prioritizing behaviors. We can do this easily using a recursive function that builds a nested conditional from an arbitrary list of behaviors:

;;; Return the motor vector of one of the active behaviors.
;;; If no behavior is active, choose the last one by default.
(define-signal (choose-active . behaviors)
  (if (null? behaviors)
      (do-nothing-motor-vector)
      (let ((next-behavior (car behaviors))
          (rest (cdr behaviors)))
        (if (activation-level next-behavior)
            (motor-vector next-behavior)
            (apply choose-active rest)))))

Now we can just say:

(define-signal active-behavior-motor-vector
  (choose-active eat face-prey))

and this will be expanded by the compiler into:

(if (activation-level eat)
  (motor-vector eat)
  (if (activation-level face-prey)
      (motor-vector face-prey)
      (do-nothing-motor-vector)))

or, written as a cond:

(cond ((activation-level eat)
        (motor-vector eat))
      ((activation-level face-prey)
        (motor-vector face-prey)
        (do-nothing-motor-vector))
      (else
        (do-nothing-motor-vector))

We will assume from here on that drive-base uses this prioritization technique for choosing behaviors to run; it will always choose the leftmost active behavior. If no behavior is active, it will keep the robot motionless.

It will also be useful to write this arbitration technique as a “higher-order operator” over behaviors: one that takes a set of behaviors as inputs and returns a new behaviors whose motor vector is the motor vector of the
leftmost active component behavior and whose activation level is true whenever at least one of the component behaviors is active. We might call this prioritize-behaviors, but we will just call it behavior-or, since it behaves analogously to the or operator in Scheme or the || operator in C/C++:

\[
(\text{define-signal} \ (\text{behavior-or} \ . \ \text{behaviors})
\hspace{1cm}
(\text{behavior} \ (\text{apply} \ \text{or} \ (\text{activation-level} \ \text{behaviors}))
\hspace{1cm}
(\text{apply choose-active} \ \text{behaviors}))
\]

\[\]

\section*{The subsumption suppression operator}

Prioritization is perhaps best known for its use in Brooks' Subsumption Architecture, perhaps the first behavior-based architecture. Although the word "behavior" has a somewhat different meaning in subsumption than we've used it here, the basic idea is that one behavior or collection of behaviors can suppress or inhibit another. To say that behavior \( A \) suppresses \( B \) is to say that when \( A \) is active, it replaces \( B \)'s output with its own. To say that \( A \) inhibits \( B \) is to say that \( B \)'s output stops when \( A \) active. In other words, the suppress and inhibit operators act approximately like this:

\[
(\text{define-signal} \ (\text{suppress} \ a \ b)
\hspace{1cm}
(\text{behavior} \ (\text{or} \ (\text{activation-level} \ a)
\hspace{1cm}
(\text{activation-level} \ b))
\hspace{1cm}
(\text{motor-vector}
\hspace{1cm}
(if \ (\text{activation-level} \ a)
\hspace{1cm}
a
\hspace{1cm}
b)))
\]

\[
(\text{define-signal} \ (\text{inhibit} \ a \ b)
\hspace{1cm}
(\text{behavior} \ (\text{and} \ (\text{not} \ (\text{activation-level} \ a))
\hspace{1cm}
(\text{activation-level} \ b))
\hspace{1cm}
(\text{motor-vector} \ b)))
\]

Note that suppress, as written above, is simply a special case of behavior-or.

\section*{Temporal behavior of prioritization: bouncing and debouncing}

We'll say more about the Subsumption Architecture in the chapter on architecture. However, it is worth mentioning that the true suppress and inhibit operations in subsumption have more complicated temporal behavior. In Subsumption, suppress and inhibit nodes have time constants associated with them. When the node goes into suppress or inhibit mode, it stays in that mode for a certain period of time, even after the suppressing behavior deactivates. We can modify our definitions of suppress and inhibit to have this behavior as follows (changes in \textit{italics}):

\[1\] The real version of behavior-or in the GRL library defaults to generating the motor vector of the lowest priority behavior when no behavior is active, not to generating do-nothing-motor-vector. This shouldn’t matter because the output of behavior-or won’t be active itself unless one of its inputs is active. It’s written this way so we don’t need to have separate copies of behavior-or for each possible kind of motor vector (since we’d need different versions of do-noting-motor-vector).
;;; Return a behavior that's just like B, but that gets replaced
;;; by A when A is active.
(define-signal (suppress a b time-constant)
  (let ((suppressing? (< (true-time (not (activation-level a)))
                      time-constant)))
    (behavior (or suppressing?
               (activation-level b))
               (motor-vector
                (if suppressing?
                   a
                   b)))))

;;; Return a behavior that's just like A, but that's inactive
;;; whenever A is active.
(define-signal (inhibit a b time-constant)
  (behavior (and (> (true-time (not (activation-level a)))
                   time-constant)
                   (activation-level b)
                   (motor-vector b)))

For behaviors as we have defined them, this version of suppress doesn't really make a lot of sense
because it means that the compound behavior is activated for a while after a goes inactive, but it still
outputs a's motor vector. On the other hand, the inhibit operation doesn't have this problem.

Much of the reason why Subsumption has these time constants is because it has a different model of
computation than GRL or conventional programming languages. Systems built in the subsumption
architecture consist of a network of finite-state machines (FSMs) communicating over fixed channels
called "wires". We'll say more about finite-state machines later, but for the moment, they can be thought of
as kind of a particularly simple thread. The basic operation in Subsumption is message passing, in which
an FSM transmits a value over a wire to a receiving register on another FSM. Subsumption FSMs are very
much like GRL transducers, except that Subsumption FSMs have the option of not sending anything at all
over a wire, whereas transducers have to specify new values for their output signals every time they update.

In Subsumption, the suppress and inhibit operations apply to wires, not behaviors. The semantics of
inhibit are that it prevents messages from flowing through one wire when the other wire has messages
flowing through it at the same time. However, FSMs in Subsumption run asynchronously, meaning that
they need not update themselves in lock-step fashion. As a result, "at the same time" isn't really well
defined. The time constant on the node ends up acting as a tolerance factor for deciding when two wires
are active at the same time.

However, there is another advantage of adding time constants to conflict resolution mechanisms which isn't
just an artifact of the message passing model. For example, you can build a left wall follower (a behavior
that drives along, hugging a wall to the left) by driving forward and turning toward the wall whenever the
robot is too far from the wall and driving forward and away from the wall when the robot is too close.

(define-signal turn-away-from-wall
  (behavior (< wall-distance threshold)
            (rt-vector -5 10)))

(define-signal turn-toward-wall
  (behavior #t
            (rt-vector 5 10)))

(define-signal follow-left-wall
  (suppress turn-away-from-wall turn-toward-wall))
This kind of behavior is used a lot in children's toys. It's simple, effective, and implementable using only a mechanical switch, which can also function as the distance sensor (no transistors!). However, it does have the undesirable property that `turn-away-from-wall` never runs for very long - as soon as it starts to run, its own action shuts itself off, triggering `turn-toward-wall`, which almost immediately retriggers `turn-away-from-wall`. This kind of rapid alternation between behaviors is sometimes called **bouncing** or **chattering**. Placing time constants in the suppression operator mean that `turn-away-from-wall` can never run for less than the time constant, thereby reducing the frequency of the switching between the two behaviors. This is sometimes called **debouncing**. Depending on the application, this can reduce wear on the hardware, improve safety, and make the robot look more intelligent.

### Linear combination

One of the major advantages of motor schemas is that their uniform output format, namely Cartesian velocity vectors, makes them easy to combine with weighted summation. If you want to approach a goal while avoiding obstacles, you just sum the approach-goal motor schema with the avoid-obstacle motor schema. Then the agent will naturally follow an obstacle-free path to the goal, provided there are no unwanted local minima.

Given the way we've defined behaviors as `(activation, motor-vector)` pairs, it's convenient to use the (continuous) activation level of a motor-schema behavior as its weight in the linear combination. We can then write motor schema combination as:

```scheme
(define-signal (weighted-motor-vector behavior)
  (* (activation-level behavior)
      (motor-vector behavior)))

(define-signal (weighted-sum . behaviors)
  (apply + (weighted-motor-vector behaviors)))
```

This simply says that the weighted motor vector of a behavior is its motor vector multiplied component-wise by its activation level and the weighted sum of a set of behaviors is the sum of their weighted motor vectors. We have to say `apply` because `weighted-sum` takes a variable number of arguments and so `behaviors` gets bound to a list of all the behaviors passed to `weighted-sum`.

Again, we can write an operator that forms compound behaviors from simpler behaviors, this time using weighted summation for conflict resolution:

```scheme
(define-signal (behavior+ . behaviors)
  (behavior (apply + (activation-level behaviors))
    (apply weighted-sum behaviors)))
```

### Weighted averaging

Weighted averaging is a useful variant of linear combination. The weighted average is simply the weighted sum divided by sum of the weights:

```scheme
(define-signal (weighted-average . behaviors)
  (/ (apply weighted-sum behaviors)
      (apply + (activation-level behaviors))))
```

Again, we have to use `apply` because `behaviors` is really a list. The compound behavior operator is then:

```scheme
(define-signal (behavior++. behaviors)
  (behavior (apply + (activation-level behaviors))
    (apply weighted-average behaviors)))
```
\begin{verbatim}
(define-signal (behavior-average . behaviors)
  (behavior (/ (apply + (activation-level behaviors))
              (length behaviors)))
  (apply weighted-average behaviors)))
\end{verbatim}

**Maximal activation**

Another popular conflict resolution strategy is to choose the behavior with the highest activation level. We can write this as:

\begin{verbatim}
(define-signal (behavior-max . behaviors)
  (list-ref behaviors
              (apply arg-max (activation-level behaviors))))
\end{verbatim}

This, somewhat enigmatic, code fragment starts by making a list of all the activation levels of all the behaviors; it then calls \texttt{arg-max} on it to find out where in the list the largest activation occurs. It then calls \texttt{list-ref} to extract that behavior from the original list of behaviors.\footnote{This may confuse you since GRL doesn’t allow lists as data types at run-time. However, since the list of behaviors is known at compile time, the compiler can compile it to a series of \texttt{if} statements to compute the \texttt{arg-max} and then a \texttt{case} or \texttt{switch} statement to compute the \texttt{list-ref}.}

**Lateral inhibition, spreading activation, and recurrence**

In biological systems, the selection of the strongest behavior is performed by having the different behaviors inhibit one another’s activation levels, a process called **lateral inhibition**:

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{lateral_inhibition.png}
\caption{Lateral inhibition diagram.}
\end{figure}

Here “A stimulus” simply means whatever the original activation of A was, and “A activation level” is the resulting activation level after lateral inhibition. Here’s the idea. Suppose A and B are both active, but A is a little more active than B. Initially, they will both inhibit one another, reducing both their activation levels. However, since A started out higher to begin with, it will inhibit B more than B will inhibit it, thus widening the gap between their activation levels. As this continues over time, the gap widens until B’s activation reaches zero. We could write this in GRL as:

\begin{verbatim}
(define-signal a-activation-level
  (- a-stimulus b-activation-level))

(define-signal b-activation-level
  (- b-stimulus a-activation-level))
\end{verbatim}

It’s important to realize that we’ve glossed over a lot of important computational details here. This network is said to be a **recurrent** or **recursive** network because \texttt{a-activation-level} and \texttt{b-activation-level} are ultimately inputs of themselves. In connecting the network up like this, we’re effectively asking it to solve a set of simultaneous equations. Networks aren’t necessarily any better at that
than you or I, and so they solve those equations iteratively by continually computing their outputs in terms of the values of their outputs from a few moments ago. In order to compute the arg-max function, the biological system is essentially “looping,” computing a series of successive approximations to the answer. One of the things this means is that there has to be some kind of memory or element within the system that’s really making it work. In the case of the GRL code, it’s due to the fact that the signals are being computed in a real loop that first computes one set of solutions and then another. In the case of neural circuits it comes partly from the time it takes for signals to move from one neuron to another (neural signals only move about 50 mph, not at the speed of light) and partly because neurons actually behave more like integrators than like pure adders or subtractors.

We can make a (bad) approximation to the behavior of the biological system by clipping a low-pass filter:

This is a variant of what is often called a winner-take-all or WTA network. We can write it in GRL as:

```glrln
(define-signal a-output
  (clip (low-pass-filter (- a-activation b-output) time-constant)
       0
       1))

(define-signal b-output
  (clip (low-pass-filter (- b-activation a-output) time-constant)
       0
       1))
```

While the general technique of hooking signals up to themselves is called recursion or recurrence, then general technique of using the activation level of one behavior to compute the activation level of another behavior is called spreading activation and can be used for much more complicated computations than lateral inhibition.

**Weighted voting**

One problem with our current definition of behaviors is that they have to propose a specific action. This doesn't always make sense, particularly for avoidance behaviors. If the robot is driving up to an obstacle, the obstacle avoidance behavior doesn't really care whether the robot goes left or right, just so long as it doesn't go straight. However, our current formulation requires it to propose a specific direction to go in. Obviously, some choice has to be make eventually, but it would be nice to allow the collision avoidance system to remain agnostic about which way to turn, so that if some other behavior happens to know that its goal lies to the left, it can resolve the choice intelligently.
This is an example of the principle of least commitment, from artificial intelligence, which states that systems should avoid making choices until they're actually necessary. We can implement least-commitment conflict resolution by allowing the outputs of behaviors to be mappings from actions to preferences. We can then take weighted sums of the preferences rather than actions and choose the action with maximal preference.

Let's suppose we allow the outputs of our behaviors to be preferences on directions rather than velocities. Conceptually, the output of a behavior is then going to be a function from directions (angles) to preferences. In practice, we can't really pass a function over a signal (it's possible to do in Scheme, but it would involve a lot of dynamic allocation and garbage collection, which we are trying to avoid). What we can do instead, however, is to discretely sample the function and pass the sampled version around as a vector. There's a function in the GRL library that does this already, called sample. It takes a function of one argument, minimal and maximal values to sample at, and a number of samples to take. It returns a vector of sample values. To improve clarity and reduce typing, we'll write a front-end for it called angle-preference that just takes the function:

```
(define-signal angle-preference-samples 16)
(define-signal (angle-preference function)
  (sample function
    0 (* 2 pi)
    angle-preference-samples))
```

and the angle with the highest preference will be:

```
(define-signal (maximal-preference-angle angle-preference)
  (* (vector-arg-max angle-preference)
    radians-per-angle-preference-sample))
```

Where vector-arg-max is a procedure that returns the index of the largest element in the vector.

These functions are sufficient to write behaviors that use least-commitment conflict resolution:

```
(define-signal avoid-obstacles
  (behavior 1.0
    (angle-preference sonar-depth))

(define-signal go-forward
  (behavior 1.0
    (angle-preference cos)))
```

which says that avoid-obstacle's preference for going in a given direction is the depth reported by the sonar pointed in that direction (we'll give the code for sonar-depth in a moment). Go-forward's preference for going in a given direction is the cosine of the direction, i.e. +1 for forward, -1 for backward, and 0 for left or right. To get the best direction to go in, we just say:

```
(define-signal chosen-angle
  (maximal-preference-angle (+ avoid-obstacles go-forward)))
```

Since the motors don't implement angle directly, we have to wrap a small control loop around our direction-selection system to drive the motors. Fortunately, this is easy to do using follow-xy-vector:
(define-signal final-motor-vector
  (follow-xy-vector (unit-vector chosen-angle)))

Of course, the sonars don’t report their results in the form of a function from angle to depth, they report
them as a vector indexed by sonar-number. So we need to write a function from angle to depth that just
reads the appropriate entry from the array. A simple version would be (remember quotient is an integer
divide and / is floating-point):

(define (sonar-depth angle)
  (vector-ref sonar-readings
    (quotient angle
      (/ (* 2 pi)
        sonar-count))))

However, it has a round-off error in it. Each sonar covers an angle of 2π/sonar-count. If we assume that
sonar 0 is pointing straight forward, then it’s covering the angle range (-π/sonar-count, π/sonar-count). The
quotient expression in the function doesn’t map this range to 0, it maps the range [0, 2π/sonar-count) to
0. So to get the right behavior, we need to add π/sonar-count before dividing. However, that makes it
possible to get an angle of greater than 2π, which would cause the vector-ref to go off the end of the
array. So we need to wrap large indices back to the beginning of the array using modulo.

(define (sonar-depth angle)
  (vector-ref sonar-readings
    (modulo (quotient (+ angle
                        (/ pi sonar-count))
                        (/ (* 2 pi)
                           sonar-count)))
    sonar-count)))

Of course, the sleazy thing to do would be to make sure that the number of samples is the same as the
number of sonars. Then we could just use the sonar-readings vector (or a rotated version of it) as the
sampled function. However, one would then have to remember to change the number of samples every
time one ported the code to a robot with a different number of sonars. It would also only work for robots
whose sonars lie in a ring configuration.

**Interpolation**

One problem with the weighted voting system we’ve outlined above is that the vector-arg-max
operation always chooses the most preferable direction, even if the next direction in the preference vector is
almost as large (or even identical). Worse yet, the direction next to it might be weighted a little higher on
the next clock cycle and a little less on the one after it, leading to abrupt shifts between directions. The
obvious solution is to choose a direction in-between the two. This is called interpolation. The easiest form
of interpolation to perform is quadratic interpolation, meaning that we fit a quadratic function to every trio
of three adjacent elements in the preference vector and then solve the quadratic for its maximum value. We
can then take the global maximum of all these maxima. While this sounds hard, it’s actually
straightforward.

Let’s let think about fitting the quadratic \( ax^2 + bx + c \) to a function \( f(x) \) at the points -1, 0, and +1. If we can fit
it at -1, 0, and +1, then we can fit it at any three adjacent points, just by shifting it. We know the values of
the quadratic at each of these points, so by plugging in the equation, we get three equations in three
unknowns:

\[
\begin{align*}
  f(-1) &= a - b + c \\
  f(0) &= c \\
  f(1) &= a + b + c
\end{align*}
\]
which we can solve for $a$, $b$, and $c$, since we the values of $f$ are known:

\[
c = f(0)
a = ((a+b) + (a-b))/2 = ((f(1)-f(0))+(f(-1)-f(0)))/2
b = ((a+b) - (a-b))/2 = ((f(1)-f(0))-(f(-1)-f(0)))/2 = (f(1)-f(-1))/2
\]

To find the maximal value, we simply set the derivative of the quadratic to zero and solve:

\[
2ax + b = 0
x = -b/2a
\]

This will be a local maximum provided that $a<0$. Otherwise, it's a local minimum and the quadratic has no maximal value. Now we can simply replace the call to `vector-arg-max` with a call to the following icky, but manageable transducer:

```scheme
(define-transducer (quadratic-arg-max-circular vector)
  (state-variables (max 0.0) ; Interpolated maximal value
                   (arg 0.0) ; Where the maximal value occurred
                   (i 0)) ; Counter
  (set! max (vector-ref vector 0))
  (set! arg 0.0)
  (let ((length (vector-length vector)))
    (for (i 0 l) ; Compute the maximum around $f(i)$
      (let ((f0 (vector-ref vector i)) ; $f(i)$
            (fm1 (vector-ref vector (modulo (- i 1) length)))
            (fp1 (vector-ref vector (modulo (+ i 1) length))))
        ; Fit quadratic
        (let ((a (/ (+ fp1 fm1 (* -2 f0)) 2))
              (b (/ (- fp1 fm1) 2))
              (c f0)) ; Find local maximum
          (let ((xmax (/ (- b)
                        (* 2 a))))
            (let ((fmax (+ (* a xmax xmax)
                            (* b max)
                            c))
                  ;; Update running global maximum
                  (when (> fmax max)
                    (set! max fmax)
                    (set! arg (+ i xmax))))))))
    arg))
```

3 “Icky” is, of course, a technical term.