Building a Hierarchical Finite-State Automaton: A Primer

Sean Luke
sean@cs.gmu.edu


Abstract

This primer introduces concepts in hierarchical finite-state automata (HFA) and their use, gives a rough formalism, and then builds an entire HFA library, with all the trimmings, in Lua. The primer is intended to educate the reader on how HFA can be used, how they are programmed, and how HFA libraries are built.

1 Introduction

A Hierarchical Finite-State Automaton (or HFA) is a finite-state automaton which contains other finite-state automata within it. It’s a relatively common technique for making automata for agents or robots.

Nesting or composition of automata should hardly be alien to software programmers: it’s essentially how we write code. We first write basic functions, then higher-level functions which call the basic functions, and so on until our program is complete. The only difference in the HFA case is that our coding model is automata and not general-purpose programming languages.

A plain finite-state automaton isn’t hard to build: it’s often little more than a loop, a global variable, and a case statement. But HFA can be complicated. In this primer we will build a complete HFA library from scratch, with some additional useful gizmos. Before we get all that, let’s start with what kind of machines we’re going to build. In short, we’re going to assume that our HFA are:

- Hierarchies of Deterministic Finite-State Automata (DFA)
- In the form of Moore machines
- With behaviors (which may themselves be other DFA) assigned to each state
- Which are designed to be pulsed repeatedly, each time progressing the automaton just a little bit.

Such automata are very useful for programming robots or simple agents in fields such as video games, simulation, robotics, and multiagent problems.

If you are unfamiliar with basic notions of finite-state automata (for example, what a DFA is), we won’t be covering that. Furthermore, the description of things here will be breezy and informal, sufficient for purposes of this report but admittedly inexact. Formalists have been dutifully warned.

As mentioned, our HFA are DFA, and thus only one state is active in a given automaton at a given time. Perhaps the most common use of these kinds of automata is determining if strings belong to certain regular languages. Such automata have a distinguished start state and (among other states) one or more accepting states. We are not interested in these kinds of automata: we will not be tokenizing strings, and thus our automata will have no accepting states. Rather our automata will do things while active in a given state. That is, they will have behaviors associated with those states. These behaviors may be either other DFA, or they may be hard-coded basic behaviors we have given the DFA to perform.

2 Policies, Mealy Machines, and Moore Machines

Consider a very simple robot scenario, with three rooms, A, B, and C, as shown at right. The robot is located in room A. When in A, the robot can either go right to room B, or it can go out a door to exit the scenario, but only if the door is open. When in room B, the robot can go left to room A or right to room C. Finally, when in room C, the robot can go to left to room B, or it can flick a switch which opens the door. The robot would like to go out the door.

The robot can sense a few things. First, it knows at all times what room it is in. Second, if it is in room A, it can
sense if the door is open or not. Third, if it is in room C, it can sense if the switch is flicked or not. How might we make a simple program to solve this?

**Policies** Imagine if our program consisted of a table of items of the form *When sensing ... → Do this...* We might construct a program which looks like this:

<table>
<thead>
<tr>
<th>When Sensing ...</th>
<th>Do this ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>In A, Door closed</td>
<td>Right</td>
</tr>
<tr>
<td>In B</td>
<td>Right</td>
</tr>
<tr>
<td>In C, Switch unflicked</td>
<td>Flick</td>
</tr>
<tr>
<td>In C, Switch flicked</td>
<td>Left</td>
</tr>
<tr>
<td>In B</td>
<td>Left</td>
</tr>
<tr>
<td>In A, Door open</td>
<td>Out</td>
</tr>
</tbody>
</table>

In essence, we have written a function \( \pi(f) \rightarrow a \) which takes the current sensor features \( f \) and returns a behavior to perform. This is known as a **policy**. This program looks like it’d do the job, but if you look closely, there’s a bug. This table isn’t describing a sequence of rules to do, but it’s rather supposed to be describing a set of rules for various situations. And we need to have two different rules for the same situation: “In B”. This is called having two situations which are **perceptually aliased**: we can’t distinguish between them, yet we need to do a different action in each of them. And that’s not permitted.

The problem here is that this scenario requires some degree of memory, otherwise known as **internal state**. to do its task. When in room B, the robot needs to remember whether it had flicked the switch or not. But policies, in this basic form, don’t provide for memory. We need a finite-state automaton.

**Mealy Machines** If we assumed that our internal state started at state 0, we could instead write a program like this:

<table>
<thead>
<tr>
<th>When in State ...</th>
<th>and Sensing ...</th>
<th>Do this ...</th>
<th>and set State to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>In A</td>
<td>Right</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>In B</td>
<td>Right</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>In C</td>
<td>Flick</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>In C</td>
<td>Left</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>In B</td>
<td>Left</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>In A</td>
<td>Out</td>
<td>1</td>
</tr>
</tbody>
</table>

Notice that our “In B” confusion is cleared up because each “In B” row has a different internal state value (0 and 1). This is a two-state DFA called a **Mealy machine**.

We can draw this table as a graph. Each edge in the Mealy machine graph connects a **state** to another **state** and is labeled with two labels: the current sensor information and the behavior to perform. Each edge corresponds to one rule (a line) in our table. That is, an edge may be described as:

<table>
<thead>
<tr>
<th>When in State...</th>
<th>and Sensing...</th>
<th>Do this...</th>
<th>and set State to...</th>
</tr>
</thead>
</table>

Thus we can write our Mealy machine in graphical form like this:

![Mealy Machine Graph](image)

Note that now that we have internal memory we don’t need to rely on as much sensor information. This isn’t always the case.

**Moore Machines** Mealy machines are straightforward implementations of standard DFA. But we’re going to focus on a somewhat different formulation which is often more intuitive for programming purposes. In this different formulation, each state will be associated with a **behavior**, though multiple states could be associated with the same behavior. When in a state, we are doing its associated behavior. Transition edges will only be labeled with the sensor situation. If no edges match the current sensor situation, we continue in the current state, and continue doing its behavior. In this model, we might write our program as:

<table>
<thead>
<tr>
<th>When doing ...</th>
<th>and Sensing ...</th>
<th>Now do this ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 [Right]</td>
<td>In B</td>
<td>1 [Right]</td>
</tr>
<tr>
<td>1 [Right]</td>
<td>In C, unflicked</td>
<td>2 [Flick]</td>
</tr>
<tr>
<td>1 [Right]</td>
<td>In C, flicked</td>
<td>3 [Left]</td>
</tr>
<tr>
<td>2 [Flick]</td>
<td>In C, unflicked</td>
<td>3 [Left]</td>
</tr>
<tr>
<td>3 [Left]</td>
<td>In B</td>
<td>4 [Left]</td>
</tr>
<tr>
<td>4 [Left]</td>
<td>In A</td>
<td>5 [Out]</td>
</tr>
</tbody>
</table>

We now have six states, each of which is associated with a behavior (in [square brackets]). In two cases, the same behavior is associated with two different states. It looks more complex, but it’s really not: in fact, we have exactly the same number of edges still. Our simplified edges are still lines in the table, but they take this form:

1. Yes, \( \pi \) is the standard symbol for policies. Don’t ask.
2. One way around this is to do actions with a certain degree of randomness: that is, we might have one rule of the form “In B → Go Right 50% of the time, Left 50% of the time”. This would eventually work! But it might not be a great idea, particularly in robotics, where such randomness might land you in the ditch, breaking your fancy $10,000 robot. We’d like a somewhat more principled approach.
3. As opposed to external state, which is the term sometimes used to describe the current situation of the agent, or its current sensor features. When someone just says state, unfortunately they might refer to internal or external state: it’s not consistent.
4. This program only works right if we start in state 0 and there’s no noise in our state transitions. As an exercise, ask yourself: what program would we need to write if our internal state could start randomly either as 0 or 1? Would it need the additional sensor information we had omitted? How about if the robot started randomly in any room? How about both random initial states and random rooms?
When doing... and Sensing... Now do this...

The resultant DFA program is:

```
(Start) 0 [right] 1 [right] 2 [flick] 3 [left] 4 [left] 5 [out]
```

This is known as a Moore machine. Some items to note. First, a Moore machine, as we’re defining it here, has no self-loops in its edges: they’re unnecessary since we have the rule that if no transition edge is matched by the current situation, we just continue in the current state and continue doing its behavior. Second, note that the Moore machine we constructed feels much more like a sequence of operations than a set of rules. This isn’t really a feature of Moore machines: it’s more a result of our example, which requires a sequence of operations to perform the task, and this feature of our example is just made more clear in the Moore machine configuration.

Consider the following more interesting example. A can-collecting robot repeatedly spins around until it finds a can, acquires the can, looks for another can, moves to the second can, and releases the first can. It then turns to some random angle to prevent it from constantly grabbing the same can over and over again. Then it repeats. Additionally, if the can slips out of its gripper, or if it loses sight of a can, it must recover from this. Here’s a Mealy machine program which performs this task. The machine has six states. We have given them names here rather than mere numbers:

Another way to view this is to separate the behaviors from the states. This makes clear the many-to-one relationship between states and behaviors.
Relationships Between Policies and Automata

Though it may not look like it at first, Mealy and Moore machines are equivalent: anything you implement in one model can be implemented in the other.

Additionally, policies may be viewed simply as degenerate versions of either Mealy and Moore machines. For example, a **policy is a Mealy machine with a single state**. To see this, here’s the previous (broken) policy we developed, shown as a Mealy machine. The invalid rules are shown in bold lines:

Similarly, a **policy is a Moore machine where each state has the exact same outgoing transition edges**. Here’s the same policy shown as a Moore machine, again with the invalid rules shown in bold lines:

3 Hierarchies of Moore Machines

So far the behaviors we’ve attached to states are basic behaviors, that is, hard-coded behaviors such as grab or go forward or go left. But in a Hierarchical Finite-State Automaton, behaviors can also be something else: they can be other automata.

Consider extending our can collecting robot. The can collecting behavior works great. We’ve created a second automaton called Run Away which runs and hides under the bed. Last, we have a basic (hard-coded) behavior called Lurk which stays quiet, in some low-power mode. We have decided to tie these three behaviors in a higher-level finite-state automaton which collects cans in a student’s bedroom when the student is away. But when the student comes in the room, the robot runs away to hide under the bed, then lurks there until the student leaves again, at which time it resumes collecting cans. We could do this with the following automaton:

Notice that we have three states, each associated with a different behavior. But two of these behaviors are unusual: they are other automata. This hierarchy could continue as far as you liked: you could use this new automaton as a behavior in an even higher-level one, and so on.

How would all this work? When we transition into a state which has an automaton as its behavior, the automaton is reset. As long as we stay in that state, we continually pulse the automaton. When we leave the state, the automaton is suspended.

It’s important to note that an HFA could use the same behaviors multiple times in its hierarchy: that is, the hierarchy isn’t a tree so much as a directed acyclic graph (or DAG). For example, consider a robot soccer automaton. At the top level, the Play Soccer automaton iterates between Do Defense and Do Offense, depending on the ball location on the field. Each of these sub-automata might include, somewhere in them, automata behaviors such as Acquire Ball or Get Open. And certainly various basic behaviors will be reused numerous times in many different places in the hierarchy, such as Run or Kick or Turn.

This is the basic advantage of using an HFA: modular reuse. Once you have defined a behavior such as Get Open, you can reuse it multiple times simply by referring to it, rather than reimplementing it over and over again. Of course, you could flatten a hierarchical finite-state automaton into a giant morass of states and basic behaviors, but this could really be a mess. In the same vein, assuming you had no need for recursion, you could flatten a computer program into a single function, a-la BASIC, but it’d be a mess.
Speaking of recursion, it should be clear at this point that you are not permitted to have recursion in your HFA hierarchy. That is, an automaton may not contain itself. Automata are simpler than Turing-complete programming languages. A finite-state automaton has finite state. Recursion requires a stack, thus infinite state.

4 A Basic Model

We’re going to build HFA in the form of Moore machines. Before we do so, let’s get down to brass tacks.

The Start State Rather than having some state designated as a start state, we’ll have a distinguished start state, meaning an initial state literally called Start which is associated with an empty behavior. Our automaton will immediately transition out of this dummy Start state to some other state as appropriate, and begin with that one. This allows us to choose an effective “start state” based on the initial conditions, using a transition function we provided, rather than having an initial state fixed in stone. Generally we won’t ever transition into Start.

Transition Functions So far we’ve described DFA as having transitions in the form of edges. But that’s not really how we’re going to represent things. Instead, each state is associated with a state transition function which, when passed the current world situation, returns a new state to transition to (or perhaps just returns the current state, meaning that the DFA should not transition at all). Essentially a state’s transition function is the collection of all of the state’s outgoing transition edges.

We can abstract this even further, defining the automaton transition function as a function which, when passed the current state in the automaton and the current world situation, returns a new state to transition to. This function can be implemented simply by looking up the state transition function for the passed-in state, then calling it and returning what it returns.

Formal Model A more or less formal model is as follows. A hierarchical finite state automaton (HFA) \( H = \{ S, B, M, F, T \} \in \mathcal{H} \) where:

- \( S = \{ S_0, S_1, ..., S_n \} \) is the set of states. Exactly one state is active at any particular time: we will refer to this as \( S_* \), and call it the current state. Initially (at timestep \( t = 0 \)), a distinguished start state \( S_0 \in S \) is the current state.
- \( B = \{ B_0, B_1, B_2, ..., B_p \} \) is a set of behaviors. A behavior may be a basic behavior, or it may another hierarchical finite-state automaton \( H' \in \mathcal{H} \), with the constraint that recursion is not permitted: automata may not contain themselves (either directly or by transitivity). Multiple different HFA may serve as behaviors in \( H \). The behavior \( B_0 \) is an empty basic behavior and does nothing.
- \( M : S \to B \) is the mapping function which associates states with behaviors. A behavior may be associated with several states. The start state \( S_0 \) is associated with \( B_0 \).
- \( F = \{ F_1, F_2, ... \} \) is a set of situations in the environment, where each \( F_i \) describes the current situation at time \( t \).
- \( T : F \times S \to S - \{ S_0 \} \) is the automaton’s transition function which, given a state \( S_i \in S \) and the current situation \( F_i \), returns a state \( S_j \in S, i \neq 0 \) to transition to. It’s perfectly plausible that \( S_i = S_j \) unless \( i = 0 \). We may view \( T \) as a set of state transition functions \( \{ T_0, T_1, ..., T_n \} \), one per state, where each \( T_i : F \to S \).

Generally we will have one top-level automaton \( H \in \mathcal{H} \) which directly or through transitivity contains all other automata or behaviors of interest, and serves as the entry point to our system. Each timestep \( t, H \) will be pulsed, which causes it to do the following:

1. It first determines a new state \( S_{new} \) to transition to, as \( S_{new} \leftarrow T(F_t, S_*) \).
2. It then sets \( S_{old} = S_* \) and \( S_* \leftarrow S_{new} \).
3. It then determines the behavior \( B_* \) associated with the new current state \( S_* \) as \( B_* \leftarrow M(S_*) \).
4. If \( S_{new} \neq S_{old} \), then it resets \( B_* \). In particular, if \( B_* \) is itself an HFA \( H_* \), then this will cause \( H_* \) to set its own internal state \( S_* \leftarrow S_0 \).
5. Finally, it pulses \( B_* \) (both behaviors and automata may be pulsed).

Note that because the transition is performed before the pulsing, \( S_0 \) can never be pulsed, which is just fine since \( S_0 \) has no associated behavior.

Mapping States to Behaviors In the formal model above, we can map multiple states to the same behavior. But to make coding more straightforward, we won’t actually do that. Instead, we’ll assume that, within a given HFA, each state must be mapped to a unique behavior. This allows us to just treat the behaviors as the states themselves, which is very convenient when writing an HFA library. Plus, in most cases each state will have a unique behavior anyway.

But doesn’t this restrict our FSA’s capabilities? After all, one would think that the three-room robot example earlier would be impossible, since the robot has to go “right” in two different states (0 and 1). Surely this brings up the spectre of perceptually aliased states again.

One would think this, but actually no! Because our HFA is hierarchical, we can get around it with a trick.
Let's say we need to make two states \( S_1 \) and \( S_2 \) which each map to the same behavior \( B_4 \). We do it as follows:

- Map \( S_1 \) to \( B_4 \).
- Map \( S_2 \) to a new HFA \( H_7 \).
- HFA \( H_7 \) has a simple transition function: its start state \( S_0 \) immediately and always transitions to a state \( S'_1 \), whose behavior is \( B_4 \). \( H_7 \) then never transitions out of \( S'_1 \).

I call this trick creating a wrapper HFA: essentially \( H_7 \) is a behavior which does the exact same thing as \( B_4 \) (because it calls \( B_4 \) itself), but is a different behavior than \( B_4 \).

5 Implementing an HFA Library

We’ll implement the HFA library in Lua. It’s a language similar to Python in many respects, and JavaScript in other ways. Some notes to make it easier to follow if you’re not familiar with the language. Comments start with two hyphens (--) . A function of the form \( \text{foo}(x,y) \) is defined using the pattern \( \text{foo} = \text{function}(x,y) \). Local variables must be explicitly declared as local, else they are considered global. A dictionary (hash table) with string keys "a", "b", and "c", pointing to the values stored in the variables \( x, y, \) and \( z \) respectively, could be defined with the literal:

\[
\text{mydict} = \{ \{\text{"a"} = x, \{\text{"b"} = y, \{\text{"c"} = z \} } \}
\]

Thereafter, we can access the value stored with "a" either as \( \text{mydict["a"]} \) or as \( \text{mydict.a} \). I defined a few closures (functions defined inside other functions or scopes which use their local variables) here and there; dynamic programming languages like Lisp or Javascript use these in spades. \( \text{nil} \) is the empty value, equivalent to \( \text{NULL} \) in C or \( \text{null} \) in Java. The rest you can probably figure out.

One more note: for succinctness, the code shown will have no error checking at all.

**Defining a Behavior** To build an HFA, we must define the concept of a behavior. A behavior is some kind of object which contains, among other things, three functions: start, stop, and go. We’ll do this in Lua using a dictionary keyed with the names start, stop, and go. Later we’ll use a slot in the dictionary keyed parent, which will refer to the behavior which is pulsing this one (if any). Last, we’d like to know if the behavior has been pulsed yet, and also we’ll give the behavior a name. Here’s a simple utility function that builds such a behavior.

```lua
makeBehavior = function(name, start, stop, go)
    return { ["start"] = start, ["stop"] = stop, ["go"] = go,
            ["name"] = name, ["parent"] = nil,
            ["pulsed"] = false }
end
```

When we first pulse a behavior, we call its start function, followed by calling its go function once. When we next pulse the behavior, we just call its go function. Finally, when we are finished with a behavior, we call its stop function. Thereafter if we wish to pulse it again, we start from the beginning (calling start, then go, etc.).

It’s up to you to define the start, stop, and go functions for your behavior. Typically you get things set up in the start function, iterate the behavior once or more in the go function, and close up shop in the stop function. You don’t have to implement all (or even any!) of these functions: instead, you could just pass in \text{nil}.

Let’s do an example. Suppose you were writing a behavior which moves a game agent forward by one step. Here you don’t care about starting up or shutting down. You might write:

```lua
forward = makeBehavior("forward", nil, nil,
    -- write the go function
    function(behavior) goForwardOneStep() end)
```

Another example. Let’s say you want to write a robot behavior which starts walking, and when the behavior is exited he needs to clean up and get in a decent pose before he does something else, or otherwise he’ll fall over. Once the robot is walking, there’s no reason to tell the robot to continue to walk, so the go function isn’t relevant. You might write:

```lua
walk = makeBehavior("walk",
    -- write the start function
    function(behavior) startWalking() end,
    -- write the stop function
    function(behavior) stopWalking() end,
    nil)
```

A final example. You want to write a game behavior which causes the agent to fire his laser. This is a one-shot behavior, so you only need to bother writing the start function:

```lua
fire = makeBehavior("fire",
    -- write the start function
    function(behavior) fireLaser() end,
    nil, nil)
```

**Pulsing** Let’s assume that we’re going to only pulse a single behavior — likely an HFA, and as part of that pulsing it’ll recursively pulse sub-behaviors as necessary. We could define this with as the functions:

```lua
-- Pulse a behavior
pulse = function(behavior)
    if (not (behavior.pulsed)) then
        behavior.pulsed = true
        if (not (behavior.start == nil)) then
            behavior.parent = nil
            behavior.start(behavior)
        end
        if (not (behavior.go == nil)) then
            behavior.go(behavior)
        end
    end
end
```
-- Reset the behavior
reset = function(behavior)
  if (behavior.pulsed) then
    if (not (behavior.stop == nil)) then
      behavior.stop(behavior)
    end
    behavior.pulsed = false
  end
end

We’ll discuss the behavior.parent stuff later.

**Defining an HFA**  
An HFA is a special version of a behavior. It has the same slots as a behavior, plus extra slots. Additionally, the start, stop, and go slots of the HFA are filled in with special hard-coded functions which we’ll define in a moment (called startHFA, stopHFA, and goHFA). We also have a few additional slots of interest. First, transition will hold our HFA’s transition function. Second, current will hold the current (active) state. Because we have defined states to have unique behaviors, in this code states will literally be behaviors. The object which current will hold will be a behavior or HFA object.

Every HFA will have a special start state,\(^5\) which is a hard-coded behavior which does nothing at all. We’ll define it as the following constant:

```
start = makeBehavior("start", nil, nil, nil)
```

Now as before, we can make a small utility function which sets these slots for us. We’ll also include some additional slots, to be discussed and used later on:

```
makeHFA = function(name, transition)
  return { 
    ["start"] = startHFA, 
    ["stop"] = stopHFA, 
    ["go"] = goHFA, 
    ["parent"] = nil, 
    ["name"] = name, 
    ["pulsed"] = false, 
    ["transition"] = transition, 
    ["current"] = start,
  }
end
```

-- These is for other gizmos we’ll add -- later, so you can ignore them for now ["counter"] = 0, ["timer"] = 0, ["done"] = false, ["failed"] = false, ["targets"] = nil, ["behaviorTargets"] = nil, ["goReturned"] = nil, ["propagateFlags"] = false, ["events"] = { }, ["returnEvent"] = nil }

We might implement the HFA, and its HFA-wide transition function, as:

```
myTransition = makeTransition(
  {
    [start] = forward,
    [forward] = function(hfa)
      if (frontWallAhead() and noLeftWall()) then
        return turnRight90
      elseif (leftWallTooFar()) then return curveLeft
      elseif (leftWallTooNear()) then return curveRight
      else return forward end
    end,
    [turnRight90] = forward,
    [turnLeft90] = forward,
    [curveRight] = function(hfa)
      if (leftWallCloseEnough()) then return forward
      else return curveRight end
    end,
    [curveLeft] = function(hfa)
      if (leftWallCloseEnough()) then return forward
      else return curveLeft end
  })
```

Notice the three edges labeled “always”: these are unilateral transitions. Suppose we had built five behaviors stored as the global variables turnLeft90, turnRight90, forward, curveLeft, and curveRight. We also have a few sensor functions such as frontWallAhead() or leftWallTooFar(). We might implement the HFA, and its HFA-wide transition function, as:

```
myTransition = makeTransition(
  {
    [start] = forward,
    [forward] = function(hfa)
      if (frontWallAhead() and noLeftWall()) then return turnRight90
      elseif (leftWallTooFar()) then return curveLeft
      elseif (leftWallTooNear()) then return curveRight
      else return forward end
    end,
    [turnRight90] = forward,
    [turnLeft90] = forward,
    [curveRight] = function(hfa)
      if (leftWallCloseEnough()) then return forward
      else return curveRight end
    end,
    [curveLeft] = function(hfa)
      if (leftWallCloseEnough()) then return forward
      else return curveLeft end
  })
```

wallFollow = makeHFA("WallFollow", myTransition)
Notice that the keys are actual behaviors, rather than "strings", like we’ve seen before.

Making a Wrapper HFA  A wrapper HFA has just a single sub-behavior of interest, and a unilateral transition function from start to that behavior. Thus we can define a utility function for wrapper HFAs very simply:

```javascript
makeWrapper = function(name, wrappedBehavior)
  return makeHFA(name,
                  function(hfa)
                    return wrappedBehavior
               end)
end
```

Making a Policy  Recall that a policy is just a Moore machine where all of the states have the same transition function. If that’s the case, we don’t need to do call makeTransition at all. We can just directly provide a single transition function and it'll be used regardless of what state we’re in. For example, take the transition function we defined for [forward] above. Imagine if we wanted to use that transition function for all the states: that is, we would make a policy out of this transition function. We could do it simply as:

```javascript
myTransition = function(hfa)
  if (frontWallAhead() and noLeftWall()) then 
    return turnRight90 
  elseif (frontWallAhead()) then return turnLeft90 
  elseif (leftWallTooFar()) then return curveLeft 
  elseif (leftWallTooNear()) then return curveRight 
  else return forward end
end
wallFollow = makeHFA("WallFollow", myTransition)
```

Policy implemented. Or even simpler: what if regardless of what state we are in, we always transition to forward? That’s a brutally simple policy. Essentially we start in start, then immediately transition to forward and stay there. To do this, we could just say:

```javascript
myTransition = function(hfa) 
  return forward end
wallFollow = makeHFA("WallFollow", myTransition)
```

Hmmm, wait a minute. That’s exactly the same procedure as our makeWrapper function. Ah, right. A wrapper is just a trivial policy which immediately goes to a certain state and stays there.

HFA Functions  Armed with this knowledge, now we’re ready to define the startHFA, stopHFA, and goHFA functions which form the heart of our HFA.

Let’s start with startHFA. This function simply resets the current state to start.\(^6\)

```
startHFA = function(hfa)
  hfa.current = start
end
```

Equally simple is stopHFA. All we have to do is stop our subsidiary behavior. To do this, the function just calls stop on the current state, assuming the stop function was implemented for the state.

```
stopHFA = function(hfa)
  if (not (hfa.current.stop == nil)) then
    hfa.current.stop(hfa.current)
  end
end
```

Done and done. Now the only actually interesting function: goHFA. This function has three parts. First, it must determine what state to transition to. Second, it must perform the transition. If the transition involves changing to a new state, this is done by calling stop on the old state, then calling start on the new state (first setting the new state’s parent backpointer to the HFA so the state knows who its current parent is). The current state is then updated. Third, it must call go on whatever the current state is. The code looks like this:

```javascript
goHFA = function(hfa)
  -- DETERMINE TRANSITION
  local newBehavior = hfa.transition(hfa)
  -- PERFORM TRANSITION
  if (not (newBehavior == nil)) then
    if (not (newBehavior == hfa.current)) then
      if (not (hfa.current.stop == nil)) then
        hfa.current.stop(hfa.current)
      end
      newBehavior.parent = hfa
      if (not (newBehavior.start == nil)) then
        newBehavior.start(newBehavior)
      end
      hfa.current = newBehavior
    end
  end
  -- EXECUTE
  if (not (hfa.current.go == nil)) then
    hfa.current.go(hfa.current)
  end
end
```

And that’s pretty much all there is to it!\(^7\) Perhaps the most surprising part of this implementation is that although an HFA consists of a collection of states (or, in our case, behaviors), in this implementation there is no set or array of states at all: all that exists is a transition function. That’s really all you need for an HFA.

6 Targets

One item I’ve found to be useful to add to an HFA is the ability to add parameters to behaviors or transition

\(\text{some exceptional condition. The straightforward way to implement this would be to not reset the current state to start in startHFA. This is all a nice notion but it has a lot of buggy interactions with other mechanisms, like the parent backpointer, and later utility function such as flags, counters, and timers. So I’d advise against it.}\)

\(^7\)Does the behavior.parent=nil line in the pulse function make sense now?
functions. For historical reasons I have called these targets. The idea is simple. Let’s say you’re writing the HFA for a soccer-playing robot. You might need to write a basic behavior called GoToGoal, and another one called GoToBall, and another one called GoToMidfield, and another one called GoToNearestOpponent. Or you could just write a single behavior called GoTo(X), where X can be specified later. X is a target.

Similarly, you might need to write a transition function for the GoTo(X) state (behavior) which says that if the distance to X is less than 3 meters, transition to Stop, else continue doing GoTo(X). Thus it’d be helpful if your transition function likewise had access to targets.

To keep things simple, targets will be untyped. That is, you cannot create a target designed for (say) positions like “the goal” or “the ball location”, and a different kind of target designed for (say) lengths of time such as “time ball has been in possession” or “time until halftime”. There can only be one kind of target, and the user will have to be careful what he passes in as a target in different situations. This is analogous to languages such as Lua, or Lisp, or Python, which have untyped parameters and variables, as opposed to Java or C, which are typed.

We will add targets to our HFA by changing the start, stop, and go functions of all behaviors and HFA to accept, in addition to the behavior itself, a set of zero or more targets (such as X), each bound to a value (such as an object representing “the ball”). These will take the form of a dictionary where the keys are the target names and the values are the values bound to those targets. For example, if we had a target value object representing the goal stored in goal1, and a target value object representing the ball stored in ball1, and we were intending to pass them into some behavior called getBetween(X, Y), where X and Y were two targets the robot was supposed to position itself between, we might store them bundled like this:

```lua
targetBindings = { ["X"] = goal, ["Y"] = ball }
```

We pass these into the getBetween’s start, stop, and go functions. It’s up to getBetween’s functions to use these bindings as it sees fit (and keep in mind, that since you’re binding these things, the objects you bind to the targets can be whatever you deem appropriate). For example, here’s a possible getBetween(X, Y) behavior:

```lua
getBetween = makeBehavior("getBetween", nil, nil,
  -- write the go function. NOTICE IT NOW TAKES TARGETS
  function(behavior, targets)
    -- we assume X and Y are actually objects that
    -- our made-up function halfWayBetween understands
    local pos = halfWayBetween(targets.X, targets.Y)
    moveTowards(pos)
  end)
```

Now let’s modify the pulse and reset functions to pass in the right bindings:

```lua
-- Pulse a behavior
pulse = function(behavior, targets)
  if (not (behavior.pulsed)) then
    behavior.pulsed = true
  end
  if (not (behavior.start == nil)) then
    behavior.start(behavior, targets)
  end
  if (not (behavior.go == nil)) then
    behavior.go(behavior, targets)
  end
end

-- Reset the behavior
reset = function(behavior, targets)
  if (behavior.pulsed) then
    behavior.pulsed = false
  end
  if (behavior.stop == nil) then
    behavior.stop(behavior, targets)
  end
end
```

Now we need to modify our startHFA, stopHFA, and goHFA functions to accept bindings. Targets pose a special difficulty for HFA. An HFA is a behavior, so it has bindings. But it also uses behaviors as its states, and they might have their own bindings. The HFA may pass some of its targets to its current behavior. Thus we need a way for the HFA to map its targets to the behaviors’ targets. For example, if the HFA’s job is to go to X while constantly starting at Y, then wall follow around X, and to do this it is using a behavior called goTo(Obj, LookingAt) and another behavior called wallFollow(Thing), then it needs to stipulate that X = Obj = Thing, and Y = LookingAt.

We’ll define a mapping as a dictionary whose keys are a sub-behavior’s targets, and whose values are the HFA’s targets meant to be mapped to them. Thus we for goTo we have:

```lua
gotoMap = { ["Obj"] = "X", ["LookingAt"] = "Y" }
```

... and for wallFollow we have:

```lua
wallFollowMap = { ["Thing"] = "X" }
```

For reasons which will become clear in a moment, we also need to indicate the behavior for which these mappings will be used. Let’s store that with a key of 0 (zero):\(^8\)

```lua
-- Map for the goTo behavior
getOtoMap = { [0] = goTo,
  ["Obj"] = "X", ["LookingAt"] = "Y" }

-- Map for the wallFollow behavior
wallFollowMap = { [0] = wallFollow, ["Thing"] = "X" }
```

In addition to binding targets to each other, we can explicitly provide a value for a target. We call this a ground target value. For example, we could say that we should be going to X but looking only at theGoal, a

\(^8\)Why zero? Answer: we need something other than a string. Zero seemed reasonable.
target value object I just made up. To do this we provide the value directly rather than a string:

```lua
gotoMap = { [0] = goTo, 
 ["Obj"] = "X", ["LookingAt"] = theGoal 
}

wallFollowMap = { [0] = wallFollow, ["Thing"] = "X" }
```

We’ll use these mappings in our transition function. The way it works will be as follows: our transition function can either return a new state to transition to (if the state takes no targets), or it can return a mapping. This mapping contains the state to transition to (as the item keyed by “0”), plus all the mappings necessary to convert the HFA’s targets into the behavior’s targets. Besides start we have only goto and wallFollow, which both take targets, so everything in the following example will return a mapping:

```lua
myTransition2 = makeTransition( 
 { 
 [start] = gotoMap,
 [goto] = function(hfa) 
 if (closeEnough()) then 
 return wallFollowMap 
 else 
 return goToMap 
 end, 
 [wallFollow] = wallFollowMap, 
 })
```

Now we need a way of taking the “transition” (a state or a mapping object) returned by the transition function and translating it into a set of targets the underlying behavior will be able to use. The particular translation will depend on the current targets provided by the parent HFA. To do this, we use the following utility function:

```lua
translateTargets = function(targets, mapping)
 mapped = { }
 for mapname, original in pairs(mapping) do
 if (not (mapname == 0)) then
 -- we are mapping a target to another target
 if (type(original) == "string") then
 mapped[mapname] = targets[original]
 else
 -- we assume it's a ground target value
 mapped[mapname] = original
 end
 end
 end
 return mapped
end
```

The Lua function pairs returns key/value pairs from a dictionary (in this case mapping). We’re setting mapname to each key and original to each pair in turn.

The translateTargets function takes a set of targets mapped to values from the HFA, plus a set of mappings from the transition function, and returns a new, properly mapped set of targets and their values to give to an underlying behavior. Armed with this, we can now perform the translation in our goHFA function.

Recall that HFA have a mysterious slot called behaviorTargets. We’ll now use this slot during goHFA: it holds the mapped targets for the current behavior, so we can reuse them on each call to go without rebuilding them. While we’re at it, let’s store the HFA’s own targets (as opposed to the sub-behavior’s mapped targets) in a slot called targets. This probably isn’t that useful in general except for debugging.

```lua
goHFA = function(hfa, targets)
 hfa.targets = targets

 -- DETERMINE TRANSITION
 local newBehavior = hfa.transition(hfa)

 -- EXTRACT TARGETS
 -- the old targets for the behavior we're stopping
 local oldBehaviorTargets = hfa.behaviorTargets

 -- figure the new targets. Both behaviors and mappings
 -- are dictionaries, but we can tell them apart because
 -- a mapping has a 0 (zero) key in it.
 if (not (newBehavior == nil) and
 not (newBehavior[0] == nil)) then
 -- extract the new targets
 hfa.behaviorTargets = translateTargets(targets, newBehavior)

 -- update the underlying behavior
 newBehavior = newBehavior[0]
 else
 hfa.behaviorTargets = nil
 end

 -- PERFORM TRANSITION
 if (not (newBehavior == nil)) then
 if (not (newBehavior == hfa.current)) then
 if (not (hfa.current.stop == nil)) then
 hfa.current.stop(hfa.current, oldBehaviorTargets)
 end
 newBehavior.parent = hfa
 if (not (newBehavior.start == nil)) then
 newBehavior.start(newBehavior, hfa.behaviorTargets)
 end
 hfa.current = newBehavior
 end
 end
 end

 -- EXECUTE
 if (not (hfa.current.go == nil)) then
 hfa.current.go(hfa.current, hfa.behaviorTargets)
 end
 end
```

Once hfa.behaviorTargets is set, we can also use it in our stopHFA function. So we define:

```lua
startHFA = function(hfa, targets)
 hfa.current = start
end

stopHFA = function(hfa, targets)
 hfa.targets = nil
 if (not (hfa.current.stop == nil)) then
 hfa.current.stop(hfa.current, hfa.behaviorTargets)
 end
end
```

... and we’re done! Now so far we’ve been using targets just for behaviors, and have been concentrating on mapping them from HFA to HFA to behavior. But targets can also be used in transitions too. Instead of

---

9Now do you see why we included the behavior in the mappings?
passing targets to a transition, we’ll just directly access the HFA’s current targets in our transition functions. For example, if we want to perform \texttt{goTo} only until we’re close enough to whatever the HFA had called \texttt{X}, we could say:

```lua
myTransition2 = makeTransition({
    [start] = goToMap,
    [goto] = function(hfa)
        if (closeEnough(hfa.behaviorTargets.X) then
            return wallFollowMap
        else
            return goToMap
        end,
    [wallFollow] = wallFollowMap,
})

myHFA = makeHFA("MyHFA", myTransition2)
```

Going way back to our wall-follower provides additional opportunities. Instead of saying \texttt{frontWallAhead()}, we could say \texttt{ahead(\ldots)}, passing in a target which specifies the object of interest to us (perhaps a wall, perhaps not). Recall that the \texttt{WallFollow} behavior took a target called \texttt{Thing}:

```lua
myTransition = makeTransition({
    [start] = forward,
    [forward] = function(hfa)
        if (ahead(hfa.behaviorTargets.Thing) and noLeftWall()) then
            return turnRight90
        elseif (ahead(hfa.behaviorTargets.Thing)) then
            return turnLeft90
        elseif (leftWallTooFar()) then return curveLeft
        elseif (leftWallTooNear()) then return curveRight
        else return forward end
    end,
    [turnRight90] = forward,
    [turnLeft90] = forward,
    [curveRight] = function(hfa)
        if (leftWallCloseEnough()) then return forward
        else return curveRight end
    end,
    [curveLeft] = function(hfa)
        if (leftWallCloseEnough()) then return forward
        else return curveRight end
})

wallFollow = makeHFA("WallFollow", myTransition)
```

Obviously we could have used this target in other functions as well.

7 An Alternative Transition Method

In the HFA we’ve developed so far, we have separate behaviors (states) and transition functions for them. Another approach that’s often done (in plain DFA anyway) is to have \texttt{events} posted to the HFA which may trigger transitions. Events are just strings, such as "finished", or "failedToKick", or what have you. The current state’s \texttt{go} function itself may post such an event to suggest a transition. This is sometimes a more convenient model than the more general transition model we’ve described here.

Recall that our HFA had three special slots, \texttt{events}, \texttt{returnEvent}, and \texttt{goReturned}. The \texttt{events} slot holds a list of strings called \texttt{events}. These events have been posted by various sources. You can post an event yourself using the function:

```lua
-- post an event to the end of the HFA’s event list
postEvent = function(hfa, event)
    table.insert(hfa.events, event)
end
```

In this case, the event will be added to the end of the list. Events are cleared every iteration, immediately after determining the transition but before performing it. You can clear all the events manually if you like:

```lua
-- clear all events on the HFA’s event list
clearEvents = function(hfa)
    hfa.events = {}
end
```

You can have a state’s \texttt{go} function return an event (a string) rather than \texttt{nil}. This is usually done to indicate that your state wants the HFA to transition to somewhere else. If a \texttt{go} function returns an event, it is posted at the very end of the events list.

HFAs are obviously themselves states: but you don’t have any control over what they return, unlike hand-coded states you’ve created. How might you get an HFA to return an event? This is done by setting \texttt{hfa.returnEvent} to the event string. Then the HFA will return this event at the end of its \texttt{go} function. There’s a utility function to do this for you:

```lua
-- set the event the HFA returns after goHFA is done
setReturnEvent = function(hfa, event)
    hfa.returnEvent = event
end
```

We need to modify the \texttt{stopHFA(\ldots)} function to clean up the HFA’s events when it is called.

```lua
stopHFA = function(hfa, targets)
    hfa.targets = nil
    if (not (hfa.current.stop == nil)) then
        hfa.current.stop(hfa.current, hfa.behaviorTargets)
    end
clearEvents(hfa)
end
```

Obviously we’ll need to make some modifications to the \texttt{goHFA} function too. Here’s how it’ll work. When the HFA is calling the \texttt{go(\ldots)} function of the underlying state, the return value of that function will get
stored away in the slot hfa.goReturned. The next iteration, the first thing that goHFA will do is check to see if hfa.goReturned has any event in it. If it does, it will post that event, then clear it.

After goHFA has determined what transition to use, it will clear all the existing events. Then it extracts the targets and performs the transition as usual. It executes the underlying go(...) method, possibly yielding an event to put in hfa.goReturned, and then it finally returns an event of its own if that event had been set in hfa.returnEvent. All in all it looks like this:

```javascript
goHFA = function(hfa, targets)
  hfa.targets = targets
  if (not (hfa.goReturned == nil)) then
    postEvent(hfa, hfa.goReturned)
    hfa.goReturned = nil
  end
  -- DETERMINE TRANSITION
  local newBehavior = hfa.transition(hfa)
  clearEvents(hfa)
  -- EXTRACT TARGETS
  -- the old targets for the behavior we're stopping
  local oldBehaviorTargets = hfa.behaviorTargets
  -- figure the new targets. Both behaviors and mappings
  -- are dictionaries, but we can tell them apart because
  -- a mapping has a 0 (zero) key in it.
  if (not (newBehavior == nil) and
      not (newBehavior[0] == nil)) then
    -- extract the new targets
    hfa.behaviorTargets = translateTargets(targets, newBehavior)
  else
    hfa.behaviorTargets = nil
  end
  -- PERFORM TRANSITION
  if (not (newBehavior == nil)) then
    if (not (newBehavior == hfa.current)) then
      if (not (newBehavior.stop == nil)) then
        newBehavior.stop(hfa.current, oldBehaviorTargets)
      end
      newBehavior.parent = hfa
    end
    newBehavior.start(hfa.current, hfa.behaviorTargets)
    end
    hfa.current = newBehavior
  end
  -- EXECUTE
  if (not (hfa.current.go == nil)) then
    hfa.goReturned = hfa.current.go(hfa.current, hfa.behaviorTargets)
  end
  -- RETURN THE RETURNEVENT IF ANY
  if (not (hfa.returnEvent == nil)) then
    return hfa.returnEvent(hfa, targets)
  else
    return nil
  end
end
```

We’ll also need to tweak startHFA to clear out hfa.goReturned in the first place:

```javascript
startHFA = function(hfa, targets)
  hfa.current = start
  hfa.goReturned = nil
end
```

**Transitioning on Events** We now have a facility for posting and clearing events. How do we transition on them? By creating a new kind of transition function. We’ll make a function called event(...) which builds a transition function which transitions based on events as indicated in its arguments.

```
event(...) takes a table of states or transition function, keyed by events. If an event has been posted, it will return the appropriate state, or call the appropriate transition function and return whatever it had returned. For example, imagine if we had a system which posted the events “leftWallCloseEnough”, and “bumpedIntoWall”. We might revise the curveRight and curveLeft transition functions like this:
```

```
myTransition = makeTransition(
  {
    [start] = forward,
    [forward] = function(hfa)
      if (ahead(hfa.behaviorTargets.Thing)
        and noLeftWall()) then
        return turnRight90
      elseif (ahead(hfa.behaviorTargets.Thing)) then
        return turnLeft90
      elseif (leftWallTooFar()) then return curveLeft
      elseif (leftWallTooNear()) then return curveRight
      else return forward end
    end,
    [turnRight90] = forward,
    [turnLeft90] = forward,
    [curveRight] = event({
      ["leftWallCloseEnough"] = forward,
      ["bumpedIntoWall"] = turnRight90 })
  end,
  [curveLeft] = event({
    ["leftWallCloseEnough"] = forward,
    ["bumpedIntoWall"] = turnLeft90 })
end
)

wallFollow = makeHFA("WallFollow", myTransition)
```

This says that when curving right, if the leftWallCloseEnough event was posted, transition to forward. If the bumpedIntoWall event was posted, transition to turnRight90. Otherwise continue curving right. What if both were posted? Then whichever event was posted first wins. The curveLeft state is similar.

Note that you could have whole transition functions rather than states. That is, instead of ["leftWallCloseEnough"] = forward, you might do ["leftWallCloseEnough"] = function(hfa) ... The event(...) method is pretty straightforward to write:
-- build a transition function based on events
event = function(transitions)
    return function(hfa)
        for event in ipairs(hfa.events) do
            local transition = transitions[event]
            if (not (transition == nil)) then
                if (type(transition) == "function") then
                    return transition(hfa)
                else
                    return transition
                end
            end
        end
        return nil
    end
end

8 Utility Behaviors

I’ve found it useful to have certain utility behaviors built into the library:

- Behaviors which increment or reset **counters** so HFAs can conveniently do things for a set number of times.
- Behaviors which update or reset **timestamps** so HFAs can conveniently do things for a set amount of wall-clock time (useful for HFAs involved in robotics).
- Behaviors which raise **flags** to signal parent HFAs that it might be a good time for the parent HFA to transition to something else now.

Each of these behaviors will need to update some special slots stored in the HFA. Let’s discuss these in turn.

**Counters**  We’ll include one **counter** in each HFA. The counter is just an integer which starts at zero and increments or is reset to zero as requested. At any time, our transition functions can query the current counter value and make transition decisions based on it. This would allow us, for example, to try kicking a ball six times, and if we still have failed, go do something else.

If our HFA were called hfa, the counter in the HFA will be called hfa.counter. We’ll create two behaviors which modify it:

```lua
-- increments the parent HFA’s counter
bumpCounter = makeBehavior("bumpCounter",
    function(behavior, targets)
        if (not (behavior.parent == nil)) then
            behavior.parent.counter = behavior.parent.counter + 1
        end
    end
end)

-- zeros the parent HFA’s counter
resetCounter = makeBehavior("resetCounter",
    function(behavior, targets)
        if (not (behavior.parent == nil)) then
            behavior.parent.counter = 0
        end
    end
end)
```

Notice that we’re also **finally** making use of the parent slot in behaviors. This slot was set in our goHFA function before the behavior was called.

If you’re too spooked about accessing the slot directly, your transition functions can get the current counter value using the following convenience function:

```lua
currentCounter = function(hfa) return hfa.counter end
```

**Timers**  Timers are just like counters, only they store intervals of time. Whereas counters are useful for simulations, timers are particularly useful for real-time stuff like robotics or game agents. We’ll include one single timer, in the HFA slot **timer**.

Unfortunately, Lua doesn’t have a good measure of system time (Google for “lua time milliseconds”): its finest resolution is one second. But it’s better than nothing. Here we’ll make a behavior which sets timer to the current time, and provide a utility function which measures the difference between the current time and the last time the timer had been set.

```lua
-- updates the timer to the current time
resetTimer = makeBehavior("resetTimer",
    function(behavior, targets)
        if (not (behavior.parent == nil)) then
            behavior.parent.timer = os.time()
        end
    end, nil, nil)

-- returns the number of seconds
-- since the timer was last updated
currentTimer = function(hfa)
    return os.time() - hfa.timer
end
```

**Flags**  A flag is a slot in an HFA which is either true or false. Flags are most often used by HFA to inform their parent HFA that they think they have finished their task (or failed at it) and perhaps the parent should now transition to something else.

We’ll define two flags: done and failed. The actual flag slot is stored in the parent HFA, and the parent HFA’s transition functions can test these flags (to determine if they should transition somewhere else) using the following convenience functions:

```lua
-- isDone(hfa) returns the current done flag in the HFA
isDone = function(hfa) return hfa.done end

-- isFailed(hfa) returns the current failed flag in the HFA
isFailed = function(hfa) return hfa.failed end
```

A child HFA sets a flag in its parent HFA by transitioning its current state to a **flag-setting behavior**. There are four such behaviors provided, which differ based on what happens after the flag is set. In some cases you just want to set the flag. In other cases, you set the flag because you’re out of stuff to do: where you should you transition to then? In these other cases, we just automatically transition to the start of the automaton. So we have:
• **sayDone** raises the done flag.

• **sayFailed** raises the failed flag.

• **done** raises the done flag and immediately resets to the start state, so the next transition will be out of the start state. This is often more convenient than sayDone because you don’t have to specify a transition function for this state. More often then not, when your HFA thinks it is “done”, you don’t care about specifying further behaviors: you might as well just start it over again if its parent foolishly decides to continue on.

• **failed** raises the failed flag and immediately resets to the start state, so the next transition will be out of the start state. This is often more useful than sayFailed for the same reason that done (above) is more useful than sayDone.

These behaviors set flags using an internal function called setFlag. We’ll include one last gizmo in setFlag called **flag propagation**. Often we see HFA which look like this:

```
Start always → Do Some Stuff is done → Done
```

Note the transition to done when the flag is done. Ick. In fact often we see whole cascades of HFA and parent HFA with this pattern. Rather than explicitly add all these “done” behaviors into these HFA, we can simply tell the HFA to propagate their done and failed flags to their parents: that is, when their done flag is set, they automatically set their parent’s done flag (similarly failed):

```
myHFA.propagateFlags = true
```

Then the HFA would look like:

```
Start always → Do Some Stuff
```

... because flag propagation is being handled automatically. Thus setFlag will look like this:

```
-- Internal utility function.
-- Sets a flag in the *parent* of the given HFA
setFlag = function(hfa, flag)
  if (not (hfa.parent == nil)) then
    hfa.parent[flag] = true
  end
  if (hfa.parent.propagateFlags == true) then
    setFlag(hfa.parent, flag)
  end
end
```

Armed with this function, we can create our four behaviors:

```
-- done: sets the "done" flag in the HFA's parent, -- and transitions to "start"
done = makeBehavior("done", nil, nil,
  function(behavior, targets)
    if (not (behavior.parent == nil)) then
      behavior.parent.current = start
      setFlag(behavior.parent, "done")
    end
  end)

-- sayDone: sets the "done" flag in the HFA's parent
sayDone = makeBehavior("sayDone",
  function(behavior, targets)
    if (not (behavior.parent == nil)) then
      setFlag(behavior.parent, "done")
    end
  end, nil, nil)

-- failed: sets the "failed" flag in the HFA's parent, -- and transitions to "start"
failed = makeBehavior("failed", nil, nil,
  function(behavior, targets)
    if (not (behavior.parent == nil)) then
      behavior.parent.current = start
      setFlag(behavior.parent, "failed")
    end
  end)

-- sayFailed: sets the "failed" flag in the HFA's parent
sayFailed = makeBehavior("sayFailed",
  function(behavior, targets)
    if (not (behavior.parent == nil)) then
      setFlag(behavior.parent, "failed")
    end
  end, nil, nil)
```

It’s convenient, or sometimes critical, to reset these flags, counters, and timers whenever we start an HFA, or in some cases whenever we iterate the HFA. We’ll do this in startHFA. Specifically, we’ll reset the hfa.done, hfa.failed, hfa.counter, and hfa.timer variables, yielding:

```
startHFA = function(hfa)
  hfa.done = false;
  hfa.failed = false;
  hfa.counter = 0;
  -- maybe this is too costly and we should
  -- restrict it to the resetTimer function?
  hfa.timer = os.time()
  hfa.goReturned = nil;
  hfa.current = start;
end
```

Last but not least, we want to clear the hfa.done and hfa.failed flags every iteration, so we modify goHFA as follows:
goHFA = function(hfa, targets)
    hfa.targets = targets
    if (not (hfa.goReturned == nil)) then
        postEvent(hfa, hfa.goReturned)
        hfa.goReturned = nil
    end
    -- DETERMINE TRANSITION
    local newBehavior = hfa.transition(hfa)
    clearEvents(hfa)
    -- EXTRACT TARGETS
    -- the old targets for the behavior we’re stopping
    local oldBehaviorTargets = hfa.behaviorTargets
    -- figure the new targets. Both behaviors and mappings
    -- are dictionaries, but we can tell them apart because
    -- a mapping has a 0 (zero) key in it.
    if (not (newBehavior == nil) and
        not (newBehavior[0] == nil)) then
        -- extract the new targets
        hfa.behaviorTargets = translateTargets(targets, newBehavior)
        -- update the underlying behavior
        newBehavior = newBehavior[0]
    else
        hfa.behaviorTargets = nil
    end
    -- PERFORM TRANSITION
    if (not (newBehavior == nil)) then
        if (not (newBehavior == hfa.current)) then
            if (not (hfa.current.stop == nil)) then
                hfa.current.stop(hfa.current, oldBehaviorTargets)
            end
            newBehavior.parent = hfa
            if (not (newBehavior.start == nil)) then
                newBehavior.start(newBehavior, hfa.behaviorTargets)
            end
            hfa.current = newBehavior
            hfa.done = false
            hfa.failed = false
        end
    end
    -- EXECUTE
    if (not (hfa.current.go == nil)) then
        hfa.goReturned = hfa.current.go(hfa.current, hfa.behaviorTargets)
    end
    -- RETURN THE RETURNEVENT IF ANY
    if (not (hfa.returnEvent == nil)) then
        return hfa.returnEvent(hfa, targets)
    else
        return nil
    end
end