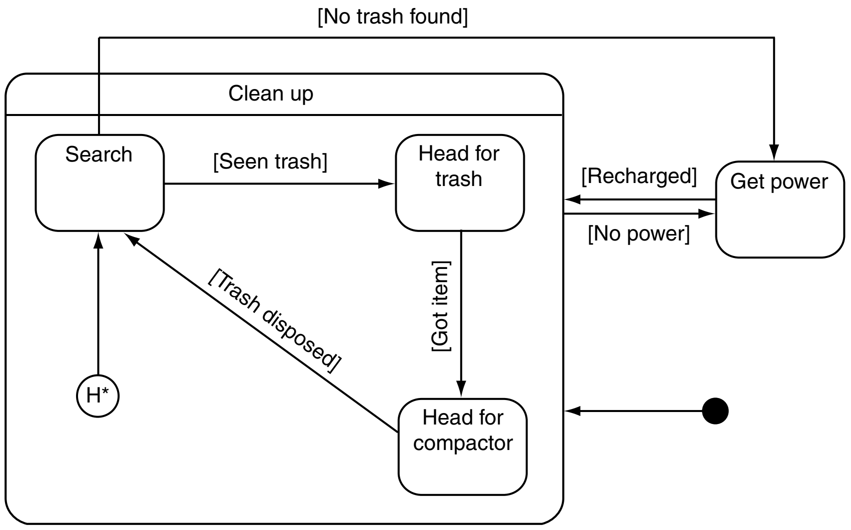

Figure 5.17 A hierarchical state machine with a cross-hierarchy transition

directly out of this state, the inner state machine no longer has any state. When the robot has refueled and the alarm system transitions back to cleaning, the robot will not have a record of where to pick up from, so it must start the state machine again from its initial node ("Search").

The Problem

We'd like an implementation of a state machine system that supports hierarchical state machines. We'd also like transitions that pass between different layers of the machine.

The Algorithm

In a hierarchical state machine, each state can be a complete state machine in its own right. We therefore rely on recursive algorithms to process the whole hierarchy. As with most recursive algorithms, this can be pretty tricky to follow. The simplest implementation covered here is doubly tricky because it recurses up and down the hierarchy at different points. We'd encourage you to use the informal discussion and examples in this section alongside the pseudo-code in the next section and play with the Hierarchical State Machine program that is available on the website to get a feel for how it is all working.

WEB PROGRAM

The first part of the system returns the current state. The result is a list of states, from highest to lowest in the hierarchy. The state machine asks its current state to return its hierarchy. If the

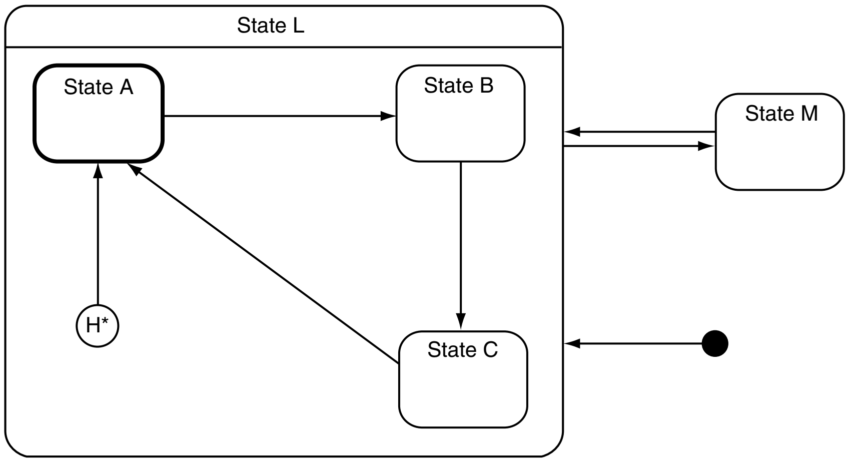

Figure 5.18 Current state in a hierarchy

state is a terminal state, it returns itself; otherwise, it returns itself and adds to it the hierarchy of state from its own current state.

In Figure 5.18 the current state is [State L, State A].

The second part of the hierarchical state machine is its update. In the original state machine we assumed that each state machine started off in its initial state. Because the state machine always transitioned from one state to another, there was never any need to check if there was no state. State machines in a hierarchy can be in no state; they may have a cross-hierarchy transition. The first stage of the update, then, is to check if the state machine has a state. If not, it should enter its initial state.

Next, we check if the current state has a transition it wants to execute. Transitions at higher levels in the hierarchy always take priority, and the transitions of sub-states will not be considered if the super-state has one that triggers.

A triggered transition may be one of three types: it might be a transition to another state at the current level of the hierarchy, it might be a transition to a state higher up in the hierarchy, or it might be a transition to a state lower in the hierarchy. Clearly, the transition needs to provide more data than just a target state. We allow it to return a relative level; how many steps up or down the hierarchy the target state is.

We could simply search the hierarchy for the target state and not require an explicit level. While this would be more flexible (we wouldn't have to worry about the level values being wrong), it would be considerably more time consuming. A hybrid, but fully automatic, extension could search the hierarchy once offline and store all appropriate level values.

So the triggered transition has a level of zero (state is at the same level), a level greater than zero (state is higher in the hierarchy), or a level less than zero (state is lower in the hierarchy). It acts differently depending on which category the level falls into.

If the level is zero, then the transition is a normal state machine transition and can be performed at the current level, using the same algorithm used in the finite state machine.

If the level is greater than zero, then the current state needs to be exited and nothing else needs to be done at this level. The exit action is returned, along with an indication to whomever called the update function that the transition hasn't been completed. We will return the exit action, the transition outstanding, and the number of levels higher to pass the transition. This level value is decreased by one as it is returned. As we will see, the update function will be returning to the next highest state machine in the hierarchy.

If the level is less than zero, then the current state needs to transition to the ancestor of the target state on the current level in the hierarchy. In addition, each of the children of that state also needs to do the same, down to the level of the final destination state. To achieve this we use a separate function, `updateDown`, that recursively performs this transition from the level of the target state back up to the current level and returns any exit and entry actions along the way. The transition is then complete and doesn't need to be passed on up. All the accumulated actions can be returned.

So we've covered all possibilities if the current state has a transition that triggers. If it does not have a transition that triggers, then its action depends on whether the current state is a state machine itself. If not, and if the current state is a plain state, then we can return the actions associated with being in that state, just as before.

If the current state is a state machine, then we need to give it the opportunity to trigger any transitions. We can do this by calling its update function. The update function will handle any triggers and transitions automatically. As we saw above, a lower level transition that fires may have its target state at a higher level. The update function will return a list of actions, but it may also return a transition that it is passing up the hierarchy and that hasn't yet been fired.

If such a transition is received, its level is checked. If the level is zero, then the transition should be acted on at this level. The transition is honored, just as if it were a regular transition for the current state. If the level is still greater than zero (it should never be less than zero, because we are passing up the hierarchy at this point), then the state machine should keep passing it up. It does this, as before, by exiting the current state and returning the following pieces of information: the exit action, any actions provided by the current state's update function, the transition that is still pending, and the transition's level, less one.

If no transition is returned from the current state's update function, then we can simply return its list of actions. If we are at the top level of the hierarchy, the list alone is fine. If we are lower down, then we are also within a state, so we need to add the action for the state we're in to the list we return.

Fortunately, this algorithm is at least as difficult to explain as it is to implement. To see how and why it works, let's work through an example.

Examples

Figure 5.19 shows a hierarchical state machine that we will use as an example.

To clarify the actions returned for each example, we will say S-entry is the set of entry actions for state S and similarly S-active and S-exit for active and exit actions. In transitions, we use the same format: 1-actions for the actions associated with transition 1.

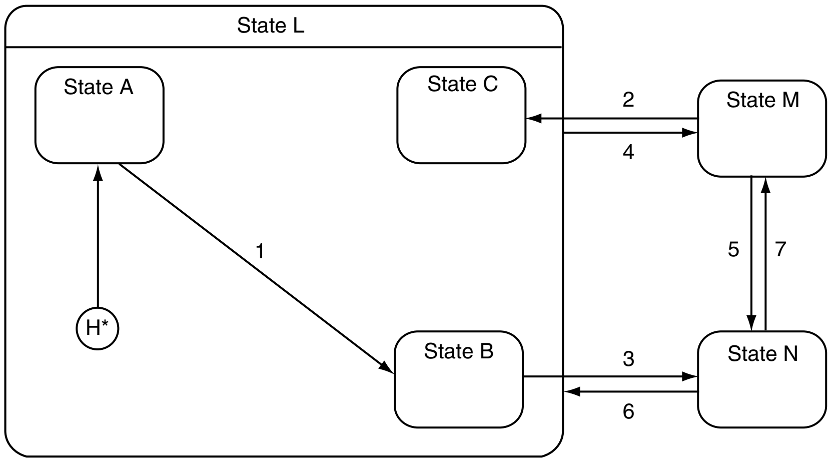

Figure 5.19 Hierarchical state machine example

These examples can appear confusing if you skim through them. If you're having trouble with the algorithm, we urge you to follow through step by step with both the diagram above and the pseudo-code from the next section.

Suppose we start just in State L, and no transition triggers. We will transition into State [L, A], because L's initial state is A. The update function will return L-active and A-entry, because we are staying in L and just entering A.

Now suppose transition 1 is the only one that triggers. The top-level state machine will detect no valid transitions, so it will call state machine L to see if it has any. L finds that its current state (A) has a triggered transition. Transition 1 is a transition at the current level, so it is handled within L and not passed anywhere. A transitions to B, and L's update function returns A-exit, 1-actions, B-entry. The top-level state machine accepts these actions and adds its own active action. Because we have stayed in State L throughout, the final set of actions is A-exit, 1-actions, B-entry, L-active. The current state is [L, B].

From this state, transition 4 triggers. The top-level state machine sees that transition 4 triggers, and because it is a top-level transition it can be honored immediately. The transition leads to State M, and the corresponding actions are L-exit, 4-actions, M-entry. The current state is [M]. Note that L is still keeping a record of being in State B, but because the top-level state machine is in State M, this record isn't used at the moment.

We'll go from State M to State N in the normal way through transition 5. The procedure is exactly the same as for the previous example and the non-hierarchical state machine. Now transition 6 triggers. Because it is a level zero transition, the top-level state machine can honor it immediately. It transitions into State L and returns the actions N-exit, 6-actions, L-entry. But now L's record of being in State B is important; we end up in State [L, B] again. In our implementation we don't return the B-entry action, because we didn't return the B-exit action when we left State L previously. This is a personal preference on our part and isn't fixed in stone. If you want to exit and reenter State B, then you can modify your algorithm to return these extra actions at the appropriate time.

Now suppose from State [L, B] transition 3 triggers. The top-level state machine finds no triggers, so it will call state machine L to see if it has any. L finds that State B has a triggered transition. This transition has a level of one; its target is one level higher in the hierarchy. This means that State B is being exited, and it means that we can't honor the transition at this level. We return B-exit, along with the uncompleted transition, and the level minus one (i.e., zero, indicating that the next level up needs to handle the transition). So, control returns to the top-level update function. It sees that L returned an outstanding transition, with zero level, so it honors it, transitioning in the normal way to State N. It combines the actions that L returned (namely, B-exit) with the normal transition actions to give a final set of actions: B-exit, L-exit, 3-actions, N-entry. Note that, unlike in our third example, L is no longer keeping track of the fact that it is in State B, because we transitioned out of that state. If we fire transition 6 to return to State L, then State L's initial state (A) would be entered, just like in the first example.

Our final example covers transitions with level less than zero. Suppose we moved from State N to State M via transition 7. Now we make transition 2 trigger. The top-level state machine looks at its current state (M) and finds transition 2 triggered. It has a level of minus one, because it is descending one level in the hierarchy. Because it has a level of minus one, the state machine calls the `updateDown` function to perform the recursive transition. The `updateDown` function starts at the state machine (L) that contains the final target state (C), asking it to perform the transition at its level. State machine L, in turn, asks the top-level state machine to perform the transition at its level. The top-level state machine changes from State M to State L, returning M-exit, L-entry as the appropriate actions. Control returns to state machine L's `updateDown` function. State machine L checks if it is currently in any state (it isn't, since we left State B in the last example). It adds its action (C-entry) to those returned by the top-level machine. Control then returns to the top-level state machine's update function: the descending transition has been honored; it adds the transition's actions to the result and returns M-exit, C-entry, L-entry, 2-actions.

If state machine L had still been in State B, then when L's `updateDown` function was called it would transition out of B and into C. It would add B-exit and C-entry to the actions that it received from the top-level state machine.

Pseudo-Code

The hierarchical state machine implementation is made up of five classes and forms one of the longest algorithms in this book. The `State` and `Transition` classes are similar to those in the regular finite state machine. The `HierarchicalStateMachine` class runs state transitions, and `SubMachineState` combines the functionality of the state machine and a state. It is used for state machines that aren't at the top level of the hierarchy. All classes but `Transition` inherit from a `HSMBase` class, which simplifies the algorithm by allowing functions to treat anything in the hierarchy in the same way.

The HSMBase has the following form:

```
class HSMBase:
  # The structure returned by update
  struct UpdateResult:
    actions
    transition
    level

  def getAction(): return []

  def update():
    UpdateResult result
    result.actions = getAction()
    result.transition = None
    result.level = 0
    return result

  def getStates() # unimplemented function
```

The HierarchicalStateMachine class has the following implementation:

```
class HierarchicalStateMachine (HSMBase):

  # List of states at this level of the hierarchy
  states

  # The initial state for when the machine has no
  # current state.
  initialState

  # The current state of the machine.
  currentState = initialState

  # Gets the current state stack
  def getStates():
    if currentState: return currentState.getStates()
    else: return []

  # Recursively updates the machine.
  def update():

    # If we're in no state, use the initial state
    if not currentState:
      currentState = initialState
      return currentState.getEntryAction()

    # Try to find a transition in the current state
    triggeredTransition = None
    for transition in currentState.getTransitions():
      if transition.isTriggered():
        triggeredTransition = transition
        break

    # If we've found one, make a result structure for it
    if triggeredTransition:
      result = UpdateResult()
      result.actions = []
      result.transition = triggeredTransition
      result.level = triggeredTransition.getLevel()

    # Otherwise recurse down for a result
    else:
      result = currentState.update()

    # Check if the result contains a transition
    if result.transition:

      # Act based on its level
      if result.level == 0:

        # Its on our level: honor it
        targetState = result.transition.getTargetState()
        result.actions += currentState.getExitAction()
        result.actions += result.transition.getAction()
        result.actions += targetState.getEntryAction()

        # Set our current state
        currentState = targetState

        # Add our normal action (we may be a state)
        result.actions += getAction()

        # Clear the transition, so nobody else does it
        result.transition = None

      else if result.level > 0:

        # Its destined for a higher level
        # Exit our current state
        result.actions += currentState.getExitAction()
        currentState = None

        # Decrease the number of levels to go
        result.level -= 1

      else:

        # It needs to be passed down
        targetState = result.transition.getTargetState()
        targetMachine = targetState.parent
        result.actions += result.transition.getAction()
        result.actions += targetMachine.updateDown(
          targetState, -result.level
          )

        # Clear the transition, so nobody else does it
        result.transition = None

    # If we didn't get a transition
    else:

      # We can simply do our normal action
      result.action += getAction()

    # Return the accumulated result
    return result

  # Recurses up the parent hierarchy, transitioning into
  # each state in turn for the given number of levels
  def updateDown(state, level):

    # If we're not at top level, continue recursing
    if level > 0:
      # Pass ourself as the transition state to our parent
      actions = parent.updateDown(this, level-1)

    # Otherwise we have no actions to add to
    else: actions = []

    # If we have a current state, exit it
    if currentState:
      actions += currentState.getExitAction()

    # Move to the new state, and return all the actions
    currentState = state
    actions += state.getEntryAction()
    return actions
```

The State class is substantially the same as before, but adds an implementation for getStates:

```
class State (HSMBase):

  def getStates():
    # If we're just a state, then the stack is just us
    return [this]

  # As before...
  def getAction()
  def getEntryAction()
  def getExitAction()
  def getTransitions()
```

Similarly, the Transition class is the same but adds a method to retrieve the level of the transition:

```
class Transition:

  # Returns the difference in levels of the hierarchy from
  # the source to the target of the transition.
  def getLevel()

  # As before...
  def isTriggered()
  def getTargetState()
  def getAction()
```

Finally, the SubMachineState class merges the functionality of a state and a state machine:

```
class SubMachineState (State, HierarchicalStateMachine):

  # Route get action to the state
  def getAction(): return State::getAction()

  # Route update to the state machine
  def update(): return HierarchicalStateMachine::update()

  # We get states by adding ourself to our active children
  def getStates():
    if currentState:
      return [this] + currentState.getStates()
    else:
      return [this]
```

Implementation Notes

We've used multiple inheritance to implement `SubMachineState`. For languages (or programmers) that don't support multiple inheritance, there are two options. The `SubMachineState` could encapsulate `HierarchicalStateMachine`, or the `HierarchicalStateMachine` can be converted so that it is a sub-class of `State`. The downside with the latter approach is that the top-level state machine will always return its active action from the update function, and `getStates` will always have it as the head of the list.

WEB LIBRARY

We've elected to use a polymorphic structure for the state machine again. It is possible to implement the same algorithm without any polymorphic method calls. Given that it is complex enough already, however, we'll leave that as an exercise. Our experience deploying a hierarchical state machine involved an implementation using polymorphic method calls (provided on the website). In-game profiling on both PC and PS2 showed that the method call overhead was not a bottleneck in the algorithm. In a system with hundreds or thousands of states, it may well be, as cache efficiency issues come into play.

Some implementations of hierarchical state machines are significantly simpler than this by making it a requirement that transitions can only occur between states at the same level. With this requirement, all the recursion code can be eliminated. If you don't need cross-hierarchy transitions, then the simpler version will be easier to implement. It is unlikely to be any faster, however. Because the recursion isn't used when the transition is at the same level, the code above will run about as fast if all the transitions have a zero level.

Performance

The algorithm is $O(n)$ in memory, where n is the number of layers in the hierarchy. It requires temporary storage for actions when it recurses down and up the hierarchy.

Similarly, it is $O(nt)$ in time, where t is the number of transitions per state. To find the correct transition to fire, it potentially needs to search each transition at each level of the hierarchy and $O(nt)$ process. The recursion, both for a transition level <0 and for a level >0 is $O(n)$, so it does not affect the $O(nt)$ for the whole algorithm.