Using State Machines as a Design and Coding Tool

The wide acceptance of real-time extensions to structured analysis techniques have led to the use of state machine descriptions for the specification of systems in which state or sequence is a vital part. However, the techniques for implementing these specifications have remained poorly understood and haphazard, leading to implementations that are difficult to verify against the specification. This paper examines different approaches to the use of state machines and explores their advantages and disadvantages.

by Mark J. Simms

In the theory of state machines, two types of state machine model are defined: the Mealy Model and the Moore model. In the Mealy model, outputs are associated with transitions. When a transition happens, the output is generated. The format of this type of state machine as implemented in Cadre's Teamwork software is shown in Fig. 1. In the Moore model, outputs are associated with states. When a state is entered, the output is generated. The format of this type of state machine as implemented in Teamwork is shown in Fig. 2.

Traditionally, the Mealy state machine model has been used for software systems. Teamwork versions prior to 4.0 did not support the Moore state machine model. This is because software state-based systems typically only take action in response to an event. An output is set on the transition, although it may remain set indefinitely. This is sometimes not the optimum way of approaching a state-based model, but tends to reflect the way software engineers think.

As a result, this paper will concentrate on the Mealy model and references to state machines may be taken to assume this model unless there is a specific reference to the contrary. All of the concepts can be applied to Moore-model state machines because any Moore state machine can be implemented as a Mealy state machine, although the converse is not true.

When the original concepts of structured analysis were proposed by Tom DeMarco,[1] no concepts of state or sequence were used. This led to difficulty in modeling a large class of problems, including real-time control systems that are largely based around state and sequence and have little data flow

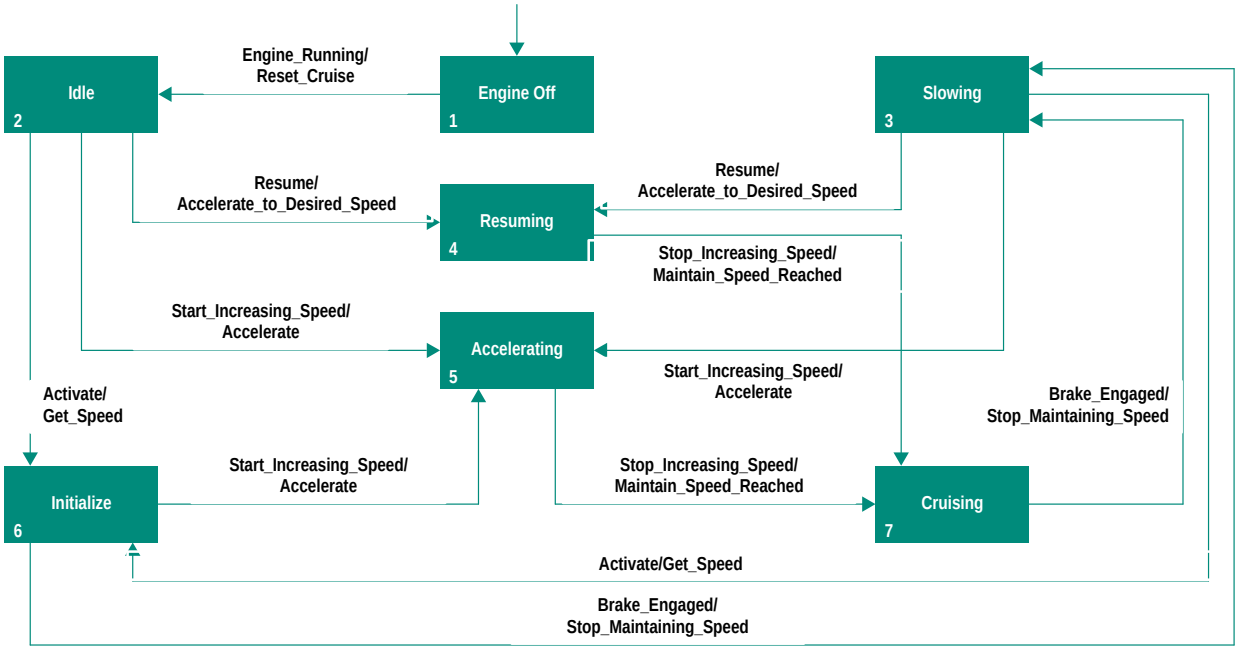

Fig. 1. Mealy state machine.

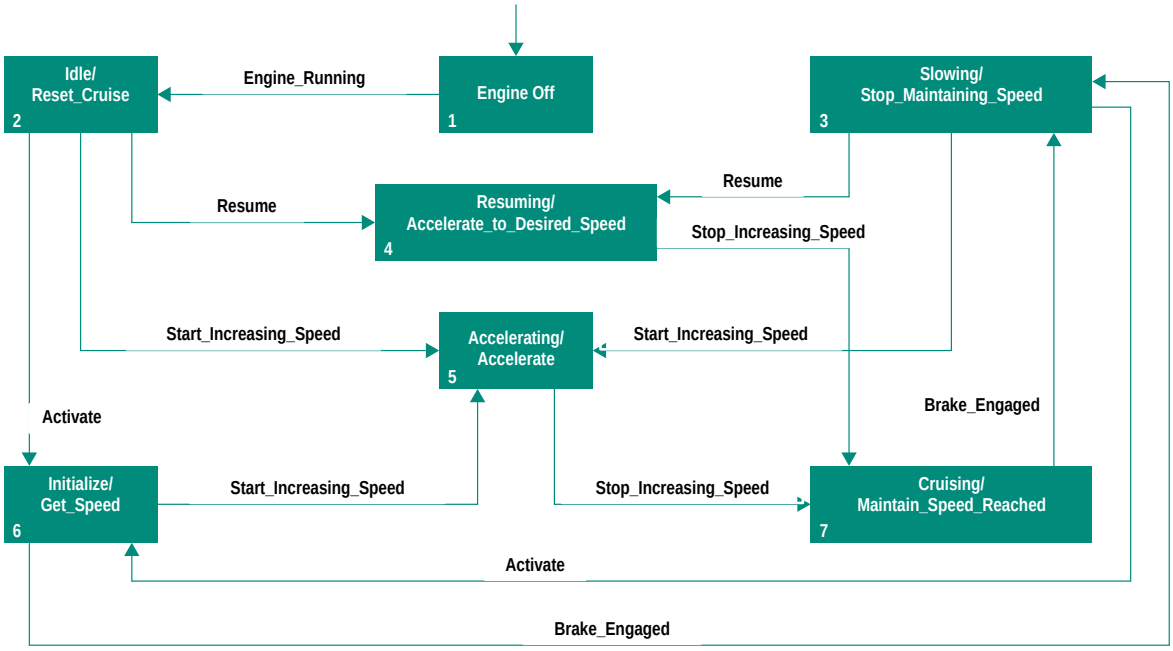

Fig. 2. Moore state machine.

content. As a result, two proposals were made for extensions to the structured analysis notation that would enable this type of problem to be modeled.

The first proposal came from Paul Ward and Steve Mellor,[2] who introduced a concept of signals to the structured analysis notation. Signals differ from data flows in that they carry timing information but no data. There are two types of signals: events and prompts. Events are generated by state machines and by data transformations in response to changes in the environment and cause state transitions within the state machine. Prompts are generated by state machines to control data transformations. Ward and Mellor proposed three types of control that could be exercised on data transformations: enable, disable, and trigger. Teamwork/SIM adds the kill prompt. The format of a Ward-Mellor state machine is shown in Fig. 3.

The second proposal came from Derek Hatley and Imtiaz Pirbhai,[3] who use the concept of a control flow. Control flows have the same properties as data flows, but are used for control purposes. They carry data and are continuously valid. The only way to pass timing information is to change the value of a control flow in response to an external event. Logical expressions involving input control flow values are used to determine when transitions of a state machine take

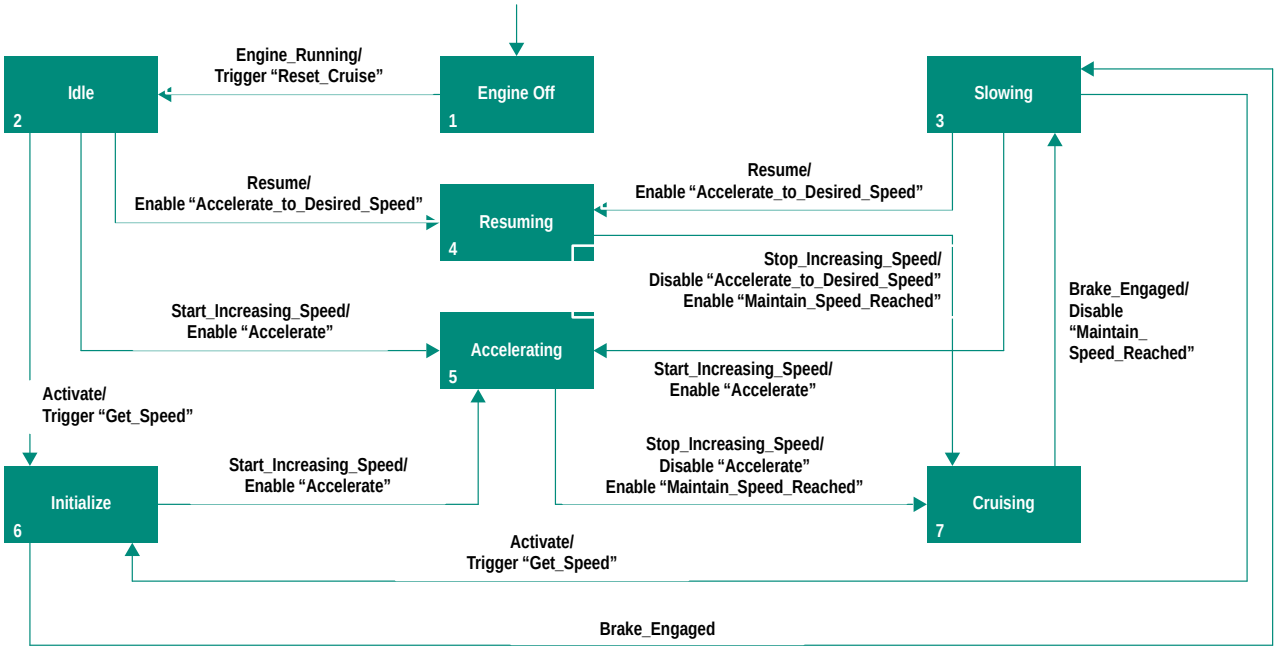

Fig. 3. Ward-Mellor state machine.

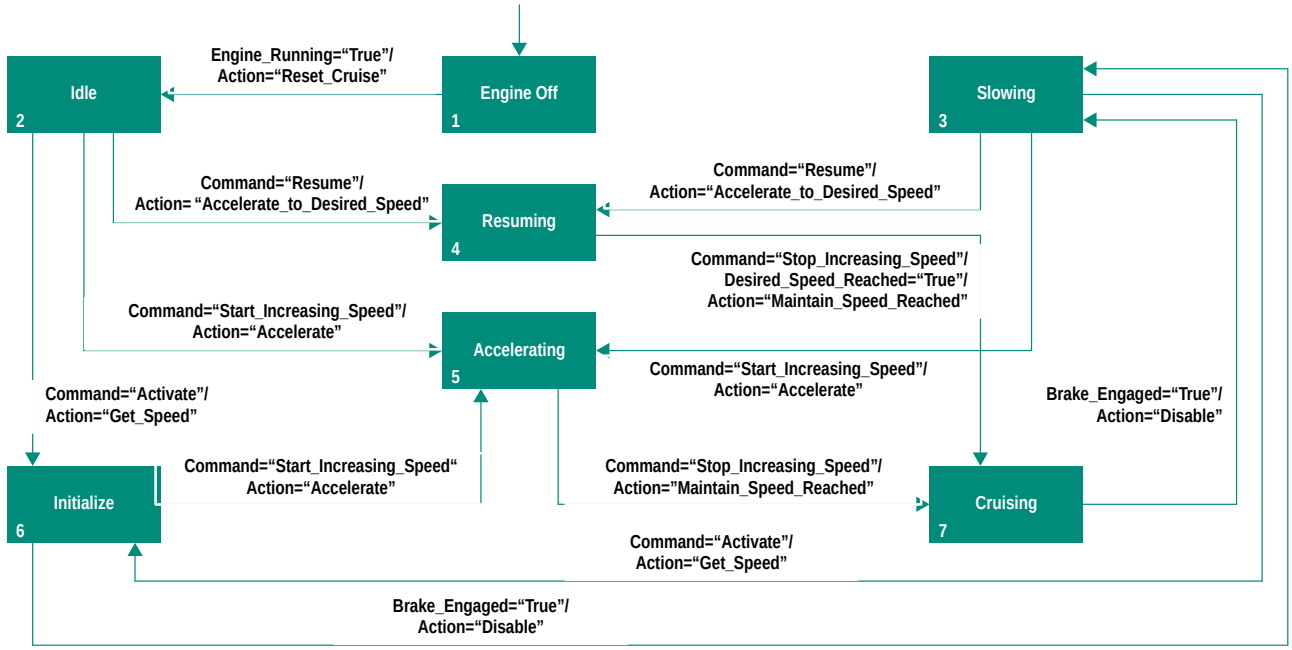

Fig. 4. Hatley-Pirbhai state machine.

place. Assignments are used in a state machine to control the values of output control flows. The format of a Hatley-Pirbhai state machine is shown in Fig. 4.

Ward and Mellor's signals can only be used in a Mealy state machine since they have an instantaneous quality and it makes little sense to have an instantaneous signal associated with a state. This could be interpreted as the signal being sent whenever the state is entered, but this is really associating it with all the transitions into the state.

Hatley and Pirbhai's control flows can be used with either state machine model. If the setting of control flows is associated with transitions in a Mealy state machine, they are assumed to be set until actively reset on another transition. If the setting of control flows is associated with states in a Moore state machine, then the control flow is deemed to be undefined if the state machine is in a state that does not actively output that control flow. This leads to a clearer definition of what is happening with a Mealy state machine and therefore a tendency to use this model.

Design Criteria

When using state machines as a design tool rather than an analysis tool, the method of implementing the state machine must be considered to give the design a rigid definition. In particular, the means of passing the external inputs to the state machine and the way the state machine interfaces to the procedural flow of the rest of the code must be well-defined.

Ward and Mellor recommend that the analysis be partitioned into tasks according to the number of state machines in the system. In addition to the state machine, the data transformations that it prompts and the event recognizers that supply it with events may be in the same task. This leads to a very simple interface to the rest of the code. The state machine is the top-level module of the task. It calls an event recognizer to get an event from the outside world. The event recognizer returns an event to the state machine if one is available or returns a code indicating no event if no event is available. The state machine then executes the transition by flagging data transformations as enabled, disabled, or triggered and changing to the next state. It executes any enabled or triggered data transformations by calling the relevant procedures. Finally, it sets the state of triggered data transformations to disabled before calling the event recognizer again.

This implementation approach has a few drawbacks. First, if there are a large number of state machines, then the number of tasks can become very large. For systems that have to be implemented on hardware with limited power, this can be very wasteful. To get around this problem, a system by which multiple state machines can be implemented in a single task is required. This can be done by making the state machine return control to the calling procedure after each loop. This allows multiple state machines to be called in sequence. This also allows other data transformation modules to be called in the same sequence, imitating the parallel nature of the analysis.

Secondly, in systems that are capable of suspending tasks, the Ward-Mellor approach is very wasteful of processor time. In this type of system, the event passing mechanism should be built into the operating system in such a manner that tasks can block until an event is generated. This works well if triggers are the only prompts required. If data transformations must be enabled and disabled, the operating system must be used to do this. However, this might produce excessive operating system overhead.

The most efficient implementations based on this type of design require that only triggers be used, but multiple state machines can be executed in the same task. With these restrictions, the Ward-Mellor design approach can be highly efficient for large and medium-sized systems. A real-time operating system handles the scheduling of the tasks and can block a task waiting for a semaphore to be set. The semaphore can be set either by another task or by an interrupt service routine responding to an external event. A counting semaphore can be used to allow multiple events to

activate a single task. Alternatively, a message queue mechanism can be used to implement a similar technique. If events are passed between state machines in the same task, then this can be implemented by flags that are checked by the event recognizer before suspending.

The top-level module in each task is responsible for calling the event recognizer routine. This blocks until the task's semaphore is set. It then determines what event occurred and returns the corresponding event code to the top-level module. The appropriate state machine is called based on the event code and passed the code as a parameter. The state machine determines the data transformation modules to be triggered and executes them in turn. It assigns the new state and returns. The event recognizer is then called again.

Hatley and Pirbhai offer far less guidance than Ward and Mellor on how to convert structured analysis state machines into designs. However, the meaning of the state machine appears more obvious at first sight, leading to a fairly obvious design.

Since the control flows are simple data values, these are easily implemented as static variables. The state machine needs only to read the variables, evaluate the logical expressions on the transitions in sequence until one is found to be true, execute the assignments on that transition and assign the new state. This continues with the transitions from the new state.

This approach has two problems. First, the order in which the transitions are processed is arbitrary. This means that the actions performed by the implementation are not necessarily uniquely determined by the state machine description. This is, however, a problem with the underlying analysis method. It is the responsibility of the analyst to ensure either that no two transitions become active simultaneously or that the system will operate within specification even if they do.

Second and more serious, the state machine offers no obvious way of interacting with the environment other than through the static variables. This means that the state machine would ideally run in its own task interfacing with the rest of the system via the variables representing the control flows. This is a highly inefficient implementation. This can be improved by implementing the state machine in such a manner that it returns control to the module that called it after each cycle. This allows several state machine modules and associated data transformation modules to be placed in the same task. This improves efficiency somewhat, but still uses processing time when nothing is being done.

Because of the inefficiency of this sort of implementation it can only be used where there is sufficient processor time to spare. However, it does have advantages when there is no operating system or where the operating system only offers basic functionality. Since the state machine operates on global variables, some simple data transformations can be incorporated into the state machines. This can produce a design that is very easy to learn and to follow and that can be implemented very easily.

It is sometimes advantageous to add some of the features of control-flow-based state machines to signal-based state machines. This involves adding states that control the flow of the state machine based on the value of variables. There should always be at least one exit transition active whenever such a state is entered. These states are not true states since the state machine never waits in the state. Instead they are merely decision points that affect the future flow through the state machine. Under certain circumstances, this can greatly simplify the state machine.

Implementation Techniques

There are two major approaches to implementing state machines in software. The first is to generate inline code that executes the state machine logic directly. This is the faster solution, but uses a large amount of code space. This approach is typically used on large systems where there is a lot of memory and on systems where response time is very important.

The second approach is to generate a table that encodes the state machine logic in a compact manner that is then interpreted by a separate state machine interpreter. This produces very compact code that is suitable for systems where code space is low and response time is not critical.

Both signal-driven and control-flow-driven state machines can be implemented either as inline code or as tables. Various different coding schemes can be used depending on the complexity of the syntax used for the state machine. In systems where the state machine module never returns, the program counter can be used to determine the state. More usually, however, a static state variable is used to maintain the state between calls of the state machine module.

For a Hatley-Pirbhai type state machine, this leads to an implementation of the form shown in Fig. 5. The state machine is implemented as a single C function. A static variable within the function holds the state. This variable is initialized to the initial state of the state machine when the task is started. The function is structured as a switch statement in which each case limb is the processing for a given state. The

```
void state_machine( void )
{
  static state = Engine_Off;
  switch ( state ) {
  case Engine_Off:
       if ( Engine_Running == True ) {
            Action = Reset_Cruise;
            state = Idle;
       }
       break;
  case Idle:
       if ( Command == Resume ) {
            Action = Accelerate_to_Desired_Speed;
            state = resuming;
       } else if ( Command == Start_Increasing_Speed ) {
            Action = Accelerate;
            state = accelerating;
       } else if ( Command == Activate ) {
            Action = Get_Speed;
            state = Initialize;
       }
       break;
  case ( Slowing ):
       .
       .
       .
  break;
  }
}
```

Fig. 5. Code for Hatley-Pirbhai state machine.

```
#define number_of_enabled_functions 3
#define Accelerate_to_Desired_Speed_INDEX 0
#define Accelerate_INDEX 1
#define Maintain_Speed_Reached_INDEX 2

void (*function_array)()[ number_of_enabled_functions ] =
  {   accelerate_to_desired_speed,
      accelerate,
      maintain_speed_reached
  };

int enabled_array[ number_of_enabled_functions ]
  = { FALSE, FALSE, FALSE };

void state_machine( int event )
{
  static int state = Engine_Off;
  int i
  switch ( state ) {
  case Engine_Off:
      if ( event == Engine_Running ) {
          Reset_Cruise();
          state = Idle;
      }
      break;
  case Idle:
      if ( event == Resume ) {
          enabled_array[
              Accelerate_to_Desired_Speed_INDEX ] =
                  TRUE;
          state = Accelerating;
      } else if ( event == Start_Increasing_Speed ) {
          enabled_array[ Accelerate_INDEX ] = TRUE;
          state = Accelerating;
      } else if ( event == Activate ) {
          Get_Speed();
          state = Initialize;
      }
      break;
  case Slowing:
      .
      .
      .
  break;
  }
  for ( i = 0; i < number_of_enabled_functions; i++ ) {
      if ( enabled_array[ i ] ) {
          function_array[ i ]();
      }
  }
}
```

Fig. 6. Ward-Mellor state machine code.

case limb corresponding to the currently active state is executed when the function is called. This tests the exit conditions for the state in a series of if statements. If one of these is qualified, then the actions are carried out in the statements attached to the if statement and the new state is assigned. The function is then exited to allow other processing to be carried out in the same task.

For a Ward-Mellor type state machine, this approach leads to an implementation of the form given in Fig. 6. Two arrays are used. The first holds pointers to all the functions that are enabled and disabled. The second holds a flag indicating whether the function is currently enabled or disabled. The function that implements the state machine has many similarities to that for the Hatley-Pirbhai state machine. It is structured as a switch statement with a case limb for each state. Each case limb consists of if statements that determine if the corresponding actions should be executed. The conditions used are far more restrictive than in the Hatley-Pirbhai case. They check whether the event code passed to the function as a parameter is the value corresponding to the required event. The actions are either triggers, which simply call the function that corresponds to the required action, or enables and disables, which set and clear flags in the array.

At the end of the function, the array of flags is searched and the corresponding functions are called via the pointers in the second array.

This design makes a few assumptions about the calling environment. First, the event recognizer functionality is not in the state machine itself. The event recognizer is executed in the calling environment and the event code is passed as a parameter to the function. Secondly, the calling environment must not block because this would prevent the enabled modules from being executed. Since the enabled modules are called as functions from the state machine, the state machine function must be executed at least as often as the enabled modules need to be called.

Since the content of these state machines is fairly simple and well-defined, machine code is a somewhat inefficient way of storing it. As a result, in systems where code space is at a premium, it may be advantageous to implement the description of the state machine as a table that is interpreted by a separate routine. The following paragraphs describe one possible way of doing this for a Hatley-Pirbhai state machine.

The state machine consists of an array of pointers and a state variable. The state variable is used as an index into the array to get the address of an array of structures containing a pointer and a value. Each transition consists of a single condition structure followed by a series of action structures and then a structure with a null pointer and the destination state indicating the end of the transition. The end of the transitions for a given state is indicated by another structure with a null pointer. This is depicted in Fig. 7.

To execute this state machine, an interpreter function is given the state variable and the list of pointers. Using the state variable as an index into the table, it uses the corresponding pointer to find the first structure corresponding to

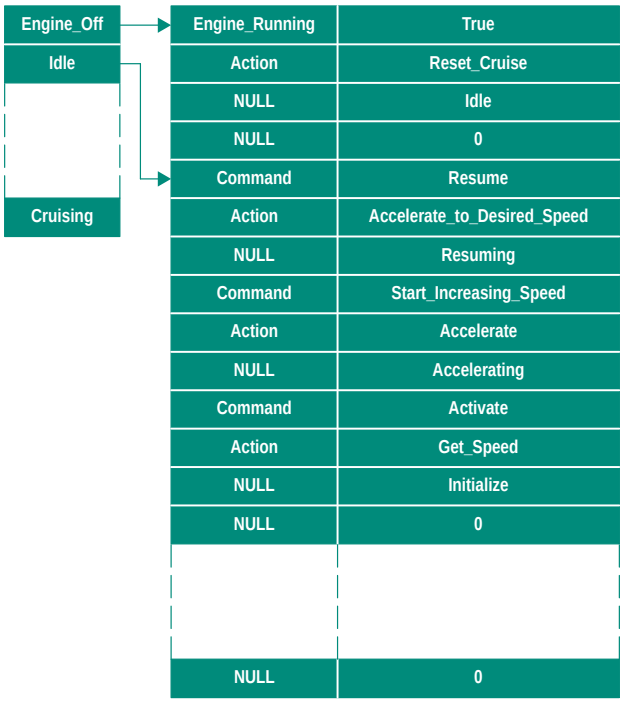

Fig. 7. State tables for a table-driven state machine.

a transition from the current state. It then checks to see if the condition is true by comparing the variable pointed to by the pointer with the value stored in the structure. If they are different, the table is scanned until a structure with a null pointer is found indicating the end of that transition. The procedure is then repeated until either a true condition is found or a condition with a null pointer is found. If a condition with a null pointer is found, all the conditions have been tested and the interpreter function returns to its calling environment.

If a condition is found to be true, the interpreter function scans the subsequent structures and assigns each value in the structure to the variable indicated by the corresponding pointer. It continues to do this until a structure with a null pointer is encountered, indicating the end of the transition. It assigns the value in this structure to the state variable to cause a change of state. It then returns to its calling environment.

A similar approach can be used for Ward-Mellor state machines. Event codes and pointers to functions to be enabled or disabled are encoded in the tables. This requires a slightly more flexible table format, but the principles are the same.

Automatic Generation

Once a rigorous mapping has been defined between the state machine design and the code to be produced from it, it is theoretically possible to design tools that can translate state machine descriptions directly to the source code for the final software. With the extensive use of graphical state machine editors for analysis, this gives the potential for a graphical form of source code that is easy to follow and easy to modify, removing some of the major problems of software maintenance.

Analysis tools are not designed with direct code generation in mind. As a result, the mappings from state machine description to code must be defined by the engineers on the project using the tools. This allows a lot of flexibility for experienced engineers to produce mappings that are highly tuned to the application concerned. It does mean that there is a requirement for anyone wishing to learn about the code to determine what the mapping is and why it was chosen. Once this is understood, the functionality of the code can be followed from the state machine diagrams.

For a structured analysis tool to be able to fulfill this role, it must have a number of features. The following are some of the most important:

- The tool must support the state machine features required by the implementers. This includes Mealy and Moore state machines, types of conditions that cause transitions, types of actions that can be placed on transitions or in states, size and complexity of diagrams, and so on.
- It must be possible to integrate the tool into the configuration management system. Since the state machine diagrams are now source code, it is vital that they be treated with a high degree of care and attention.
- The tool must be able to access the diagrams. If the data is stored in any sort of database system, the appropriate access routines must be supplied.
- The tool must keep diagrams in a documented format that does not change between revisions. Continually modifying code generation programs to track the format of the diagrams is unacceptable.

Once these features have been established, code generation becomes a simple task of defining the mapping between the state machines and the code and then designing the translator program and any interpreter routines for table-driven state machines. The rest of the code can then be designed from the remainder of the structured analysis with the state machine implementation in mind.

There have been a number of dedicated code generator programs on the market for some time, many of which use state machines as part or all of their input tools. These systems come with rigorously defined semantics for their diagrams so that users of the system can rapidly understand designs with which they are unfamiliar. These programs also come with a defined mapping of the diagrams to code.

The biggest problem with this sort of system is that the features of the state machines and the mapping to code supplied by the vendor may not be ideal for the implementation that is required for a given problem. Few systems currently available offer any ways of tuning the implementation for a given set of design criteria. This is one of the major features to look for in such a system.

A hybrid approach is possible in which a code generator tool is used, but a custom front end is included to tune the resulting code from the generator. This might be done either by treating the tool in the same way as a generic structured analysis tool and accessing the diagrams and generating code directly from them or by postprocessing the resultant code to optimize it for the given situation.

The hybrid approach is probably harder than designing a generator for a generic structured analysis tool, but the results could be better. The added rigor of code generators means that it is easier to use standardized semantics for the system. The semantics are more likely to be complete than those of a structured analysis tool.

Summary

The usefulness of state machines for specifying control applications has been well-proven. Their use in design and implementation is also showing a great deal of promise. It has shown major advantages in the following areas:

- Rapid code implementation because of the very close mapping of analysis, design, and code
- Ease of maintenance because of the availability of easy-to-read code in the form of state machine diagrams
- Compact implementation of a large proportion of the functionality of a problem because of the use of table-based state machines.

These advantages make the investment in tools for this technique well worth the effort and expense involved.

References

1. T. DeMarco, *Structured Analysis and System Specification,* Prentice Hall, 1978.

2. P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems,* Prentice Hall, 1985.

3. D.J. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House Publishing, 1987.