Hierarchical State Machines in the Automation Process

1. Introduction

Often during the process of supporting, modifying or re-engineering automation software, I have found myself making the same conclusion: “This is a Hierarchical State Machine trapped in the body of a C program, struggling to get out!” It seems that many programmers tackle a problem in the A to Z approach, whereby they say “First, you need to do this, second, this, … and finally this.” This creates a program that is chained to the original programmer’s concept of how the process is to be automated, which often times does not agree with the next person’s view, nor the next, etc… The result is a system that is difficult to support, suffers from instability, and is difficult to add or modify functionality.

My experience has led me to develop a methodology, set of class libraries and finally a sophisticated Interactive Development Environment (IDE) to quickly model an automation process as a Finite State System composed of Hierarchical State Machines. Systems developed this way are very stable, easy for new programmers to support, and are relatively simple to add features and functionality. This article will introduce you to the concept of a Hierarchical State Machine (HSM) and show you how this approach works so well in software engineering for the automation industry.

2. Cost Benefit Analysis

Before investigating the technical details of a HSM based solution, a brief discussion of the cost of implementation vs. investment return is in order. The following lists detail the costs and benefits:

Costs

- Increased initial development time
- Longer learning curve for new developers

Benefits

- Ease of modeling and developing new applications
- Ease of maintaining running system
- Ease of modifying existing system to address new business requirements
- Ease of modifying existing system to address environmental effects

- Reuse of code base

Implementing a HSM solution for the first time involves (naturally) a learning curve. This involves developing an understanding of the class framework, support modules and application structure that would not be present if you were developing from scratch. Of course, if this were the case, you probably would be writing your own framework and set of support modules as development progresses. Once the developers learn the class framework and support modules, development can proceed at a fixed pace, with no unforeseen major code rewrites anticipated.

Once one or two such systems have been deployed, subsequent development efforts are greatly reduced by simply re-using the existing framework and code base, and perhaps reusing code from other running HSM solutions. Furthermore, developers who have supported one such application can easily see and understand HSM based systems written by other developers.

When business requirements or environmental effects dictate that an existing HSM based system be modified, the greatest benefit is obtained. In a monolithic approach, developers are required to “shoehorn” the code supporting the operational changes into the existing structure, often times requiring wide-reaching changes in the application. This brute-force approach, when applied repeatedly, often times results in unsupportable code that eventually will require a rewrite. HSM based systems are easily modifiable to support changing requirements, due to their modular structure and table based logic. Modifications are accomplished by first, modifying the state machine table definitions to address the change requirements, and second, filling in the stub routines.

Here are two hypothetical examples of how a HSM based solution can save time, money and frustration.

New Application

In a new application development, the analysis is performed detailing the input signals, output to controls, timing issues, interactivity among components, etc… The output of this effort can be modeled by generating a series of Finite State Machine definitions, which directly result in code, in the form of header files defining the machines, and stub routines requiring completion. Existing applications can be borrowed from to fill out some of the stubs. Testing can proceed, which focuses on the overall behavior and specifically on the code added to the stub routines. The resulting application can be easily understood by a developer who has supported other HSM based applications.

Modification to Existing Application

Modifications after implementation usually involve unforeseen effects on a system imposed by environment. An example would be a computer located in a “noisy” area, one subject to interference, vibration, dust, etc… Consider a sensor that is used to detect product entering a material handling system. It’s purpose is to detect an object and serve as a starting point for a timer that is used to position the object on the conveyor for electro-mechanical processing, for example, label application.

After implementation, it is found that the system is not behaving as expected. An analysis results in the conclusion that the sensor is returning false clear indications after it returns a block indication. This can be due to electro-magnetic noise, reflective surface on the object, or any unforeseen effect.

The solution to this problem, assuming that the physical cause cannot be addressed, is to add logic to filter out unrealistic signals from the real ones. Here, the HSM structure can be easily modified to implement sanity timers in the process. That is, after a clear indication, if a block occurs in less than 50 milliseconds, the assumption can be made that it is a false clear. This kind of timer can be easily implemented by modifying the state table to add a new timer, and few events and action routines. Supporting this in a monolithic design would clearly be a much greater effort.

3. Example of a Finite State Machine (Garage Door)

Most readers probably have a good idea of what a Finite State Machine (FSM) is. For those who don’t, we’ll look at a simple example that we’ve all encountered at one time or another – the automatic garage door. The garage door in question has a remote opener/closer, a motor to drive the door through its opening and closing states and a photo-eye to detect a foreign object in the path of the closing door. In addition, there are switches that trip when the door is fully opened and fully closed.

A finite state machine consists of four components

- States – A set of states in which the system can exist
- Events – Meaningful occurrences that affect the system
- Actions – Operations taken when an event occurs
- Transitions – Changes from one state to another brought about by events

Let’s define each for this example.

The states of this FSM are Opened, Closed, Closing and Opening.

The events are:

- Signal received from remote
- Open door switch activated
- Closed door switch activated
- Photo-eye blocked

The actions are Open Door, Close Door and Stop Motor.

The transitions are defined in the following table, which rolls all of the components into a complete definition of the finite state machine. This table is referred to as a State Transition Table.

State	**Event**	**Action**	**New State**
Open	Signal Received	Close Door	Closing
Closing	Closed door switch	Stop Motor	Closed
Closing	Photo-eye blocked	Open Door	Opening
Opening	Open door switch	Stop Motor	Open
Closed	Signal Received	Open Door	Opening

Another way to visualize a finite state machine is via a State Transition Diagram. There are many versions of this diagram, featuring different symbols for the states. In our diagram, states are represented by circles, transitions by arrows from one circle to another (or even one circle to itself), and events and actions are indicated by captions within the diagram.

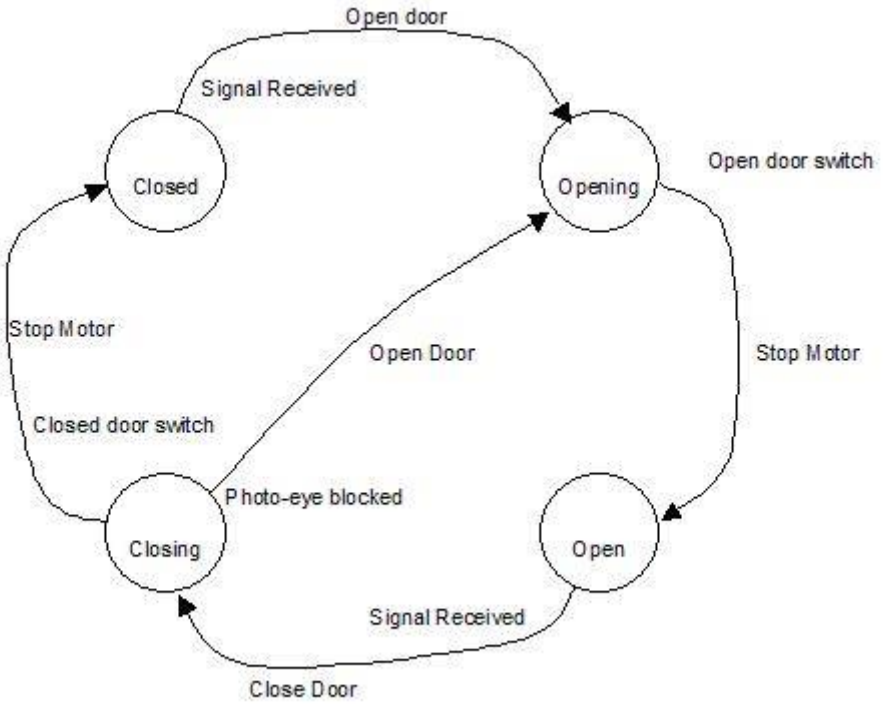

Figure1. Garage Door Finite State Diagram

You can see that this is a very simple Finite State Machine. Automation processes are usually modeled by much more complex state machines than this.

Let's get a bit more complex and consider some exception conditions. For example, what would happen if the close door or open door switch malfunctioned causing no indication to be received that the door was closed or opened. Well I would imagine that the motor would run forever and eventually burn up. We can address this by adding a new event and re-defining some actions. To protect against this, we will define the maximum time that the motor needs to run to open the door, and also to close the door. When we start opening or closing the door, we will start the timer. When a close door switch or open door switch is detected, we not only stop the motor, we also kill the timer. If the timer expires before the switch is tripped, we stop the motor (and to be really responsible, send an alert message so that the switch can be fixed). Here's what our table looks like now:

State	**Event**	**Action**	**New State**
Open	Signal Received	Close Door, Start Timer	Closing
Closing	Closed door switch	Stop Motor, Stop Timer	Closed

Closing	Photo-eye blocked	Open Door, Restart Timer	Opening
Closing	Timer Popped	Stop Motor, Signal Error	Closed
Opening	Open door switch	Stop Motor, Stop Timer	Open
Opening	Timer Popped	Stop Motor, Signal Error	Open
Closed	Signal Received	Open Door, Start Timer	Opening

The lesson here is that using a finite state machine to model the software implementation driving the electric garage door enabled us to quickly address a problem that was not dealt with in the initial implementation. Using a monolithic approach to this project would have rendered this modification far more difficult.

4. The Next Step: Finite State System

Admittedly, the garage door example is simplistic and one that doesn't push the envelope as far as advanced controls are concerned, but it serves us well in illustrating how a state machine approach benefits the development effort. Let's take it to the next level and inject some complexity. If the state machine that models a real-world process gets too complex to fathom for most mortals, it's time to add some structure…

4.1 The Hierarchical Approach

One way to simplify a very complex finite state system is to break it down into reasonable sized chunks, each doing a specific task or set of tasks, and each concerned only with its charter. The relationship among components varies, but invariably there is a hierarchical relationship involved. In such a case, you have a single FSM that controls its "children". Child FSM's may not resemble one another, or they may be multiple instances of the same FSM. The former might include an assembly line process where each child FSM operates on an object, then that object is passed on to the next FSM, and so on. The latter might include a series of identical FSM's, each one acting on its own assigned hardware component. Let's consider how these FSM's work together to create a well-running system.

4.2 Controlling Multiple Identical Processes

Consider an application that directs containers to areas in a warehouse for picking or put-away purposes. The containers are directed from their starting point to their assigned zones by traveling a conveyor that branches out to serve each pick zone. At each split there exists a barcode scanner and a diverter, which when activated directs the container to the left, right or straight ahead. There are many such divert points, each

controlled by a single program. When a container is scanned, a decision must be made within a short amount of time, perhaps 100 milliseconds, based on the following criteria:

- Is the container assigned to this zone
- Has the container already visited this zone
- Is the zone full to capacity

This application can be modeled with a Hierarchical State Machine consisting of one FSM for each diverter, and a master FSM controlling each Diverter FSM. The finite state table for a single diverter might look like this:

State	**Event**	**Action**	**New State**
Waiting for Scan	Scan Received	Ask for Direction, Start Divert Timer 1	Waiting for Direction
Waiting for Direction	Direction Received	Store Direction	Waiting to Divert
Waiting for Direction	Timer Popped	Alert Operator, Log error	Waiting for Scan
Waiting to Divert	Timer Popped	Activate Diverter, Start Divert Timer 2	Diverting
Diverting	Timer Popped	De-activate Diverter	Waiting for Scan
Waiting for Scan	Direction Received	Log Error	Waiting for Scan

This is a fairly simple FSM to control the divert process for a single diverter. Of interest here is the use of timers to control the precise timing of the divert mechanism, preventing a crash or jam on the conveyor. It is not enough to fire the diverter as soon as the directions are received, as that time is not predictable. Also note that the FSM addresses the potential of receiving divert instructions after the container has passed the divert point, handling it as an exception condition.

This program however must control many diverters, not just one. This is where the Master FSM comes in. It has three basic responsibilities – controlling startup, controlling shutdown, and directing events to the proper diverter FSM's. Worker threads will be called upon by the diverter FSM's to start a read operation on the scanners, and to obtain divert instructions from the container data store. When the scanner IO completes, or when the divert instructions are returned, that information, along with the identity of the diverter to which the information is pertinent will be sent to the Master FSM for processing. The Master FSM uses the identity of the diverter to issue an event to the proper diverter FSM.

4.3 Controlling Sequential Processes

The previous example showed how a master FSM could control a series of identical child FSM's to control identical processes. Let's look at an example where child FSM's are all different from one another. This example works like an assembly line. Each item is processed in turn by different stations of the system. Each processing station is independent of the others, performs a single task and releases the object for further processing by the next station. The objects are fed through the system by conveyor belts with photo-eyes positioned to allow precise positioning of the objects for processing by each station.

As in the prior example, the Master FSM is responsible for controlling startup and shutdown and for directing events to the proper child FSM's. An additional responsibility of the Master FSM is to control the movement of objects through the system. It prevents an object from moving from one station to the next, until that station is ready to accept a new object. Thus, an object in station 2 cannot move to station 3 until station 3's photo-eye is unblocked, indicating that there is room for a new object. The final station will similarly hold objects until the out-take conveyor's photo-eye is unblocked, indicating that there is room on the conveyor for more accumulation.

Since we're manufacturing widgets, it makes no sense to show the child FSM's here, however the master FSM is worth looking into. Here is its FSM definition:

State	Event	Action	Ne
Uninitialized	Startup	Start Children	Per
Pending Initialization	Initialization Timer Popped	Check Child FSM Init Status	Per
Pending Initialization	Initialization Complete	Start Operation Timer	Init
Initialized	Process Timer Popped	Check Child FSM Process Status	Init
Initialized	Shutdown	Stop Children	Per
Pending Shutdown	Shutdown Timer Popped	Check Child FSM Shutdown Status	Per
Pending Shutdown	Shutdown Complete	Shutdown Master FSM	Uni

This is fairly simple, but leaves a lot to the imagination. Let's dig a little deeper into some of the action routines. The first action routine *Start Children* does the following:

- Sends a startup event to each child FSM
- Starts the Initialization Timer

At this point, the master FSM is waiting for each child FSM to complete its initialization. When the Initialization timer pops, *Check Child FSM Init Status* does the following:

- Check each child FSM status
- If all are initialized, send a *Initialization Complete* event to itself
- If not all are initialized, restart *Initialization Timer*

Once startup is complete, the master FSM periodically checks the progress of its child FSM's making sure each is busy if there is work to do. This occurs in *Check Child FSM Process Status*, which does the following:

- Check each position, from last to process to first to process for ability to accept work. Each station that is ready for work results in the next (earlier in process) FSM being issued an Eject event, causing that device to activate its out-take motor.
- Start *Process Timer*

Here, it is important to note that each child FSM controls its own photo-eyes, out-take motors, etc… The only information exposed to the master FSM by the child FSM's is whether the child FSM can accept new work. In addition, the master FSM checks the status of the out-take conveyor for the entire system to ensure that processing is stopped when it is not accepting anything, i.e., motor stopped or photo eye blocked.

4.4 Summary

We have seen the flexibility of the two simple examples presented above. In the real world processes are not always that simple. This approach can easily address much more complex models, such as combinations of multiple identical FSM's and sequential FSM's. In addition, we have seen only parent/child relationships so far. This approach can address hierarchies to any level of depth.

5. How it All Fits Together

Up to this point, I've attempted to make a case for using Hierarchical State Machines to develop software solutions for automation processes. If you want to know a bit more about how it all fits together from a software development point of view, read on.

5.1 Distributed Processing

Not all automation processes can be addressed by a single hierarchical state machine. Often, multiple programs on multiple computers are required. An example is a process that requires a database lookup. It might not be feasible to distribute database drivers to each computer in a facility, so addressing all database requests from one or more data managers might be appropriate.

Communication among programs is handled by an interface class that implements socket communication in the following flavors:

- Raw data over sockets
- SOAP encoded messages over sockets

In addition, the socket class can communicate using Secure Socket Layer (SSL) to eliminate the potential of exposing sensitive data to prying eyes. Messages arriving from external programs can be handled as events that drive the state machine hierarchy. For example, a command from a Human Machine Interface requesting the application to shutdown would be delivered to the highest-level FSM in the hierarchy as a Shutdown event.

5.2 Real-time Solution

Since automation processes are typically very time-sensitive, this approach must function as a real-time system. A real-time system is loosely defined as a system where violation of time constraints produces catastrophic results. In the conveyor diverter example described earlier, missing a time constraint results in a container being sent to the wrong place, possibly affecting shipping schedules. Worse, it could cause an entire shipping route to be delayed while the container is filled. Imagine the result if this were not just a one-time occurrence but a common occurrence. This approach guarantees real-time performance.

FSM Behavior

All Finite State Machines are serviced by a single processing thread. This is possible because looking up an event in an event table, executing a well-written action routine and transitioning to a new state is a very rapid process. Thousands of events can be handled in a second if the action routines are well written. For that to occur, action routines must never block on a resource. If an action results in a database lookup, file I/O, or printing of a label, for example, that should be handed off to a worker thread for completion in an asynchronous manner. While the worker thread is processing the request, the FSM is free to process remaining events. Once the worker thread completes its request, it issues an event to the top-level FSM, containing the identification of the requesting FSM, the disposition of the request and any requested data. Event processing logic within the hierarchy directs the event to the proper FSM.

Events issued to a FSM are queued in a First In, First Out basis. The FSM implementation processes events repetitively until there are none to process. It then waits on a semaphore that is signaled the next time an event is added to the stack. Additionally,

events can be placed on the front of the stack by specifying priority. This might be used to handle an Emergency Stop event.

Structures

Finite state tables and event queues are implemented using the Standard Template Library (STL), which is an open-source, platform independent set of template classes. These classes are used to implement ordered lists in a variety of ways. STL classes are invaluable in implementing volatile storage of relational data, providing the fastest access possible to list elements. Since STL is based on the C++ standard library, it is platform independent, supporting the ability to implement FSM solutions on all platforms.

5.3 Class Library Approach

We have implemented a set of class libraries for use in Hierarchical State Machine applications, which include the following:

- Finite State Machine – This is the ancestor class for all FSM subclasses implementing a hierarchy. This class incorporates all finite state table management, event processing, and error and status reporting from lowest to highest in the hierarchy enabling one call to obtain all status and error messages from all state machines.
- Thread Object – This class encapsulates a worker thread, including synchronization objects for controlling startup and shutdown processing sequence. The thread object is used to implement worker threads used by the FSM's to perform blocking and non-real-time processes.
- Communication Classes – Both secure and non-secure socket classes are provided, with Simple Object Access Protocol (SOAP) capability. Additionally, a worker thread pool class allows a limited set of threads to service I/O completion on many socket connections. Since both sockets and SOAP are platform independent, using the communication classes enables components written on disparate platforms to communicate with each other.

5.4 IDE Eases Development Process

After putting a few of these under our belts, it became obvious that the initial setup time for developing a new HSM based project was one of the most time consuming aspects of the job. This step involves generating a C++ header file to define each finite state table and each finite state machine. Next, generating the implementation files for each finite state machine follows, with stubs for each action routine. Finally, putting all these together in a hierarchy to relate all machines completes the setup. At this point, it's time to fill in the stubs to complete the implementation.

To speed up the setup process, we realized that it would be natural to define a hierarchical state machine system using XML, which is well suited for specifying data in hierarchical relationships. By writing an Interactive Development Environment (IDE) to allow definition of hierarchy and FSM's via a graphical user interface, the setup time is minimized. FSM definitions and relationships are stored in a single XML file, and a post-processor is utilized to convert the XML to actual C++ definition and implementation files.

6. Conclusions

Time has proven that the Hierarchical State Machine model has provided the following benefits to the development, maintenance and support of automation applications:

- The development cycle is shortened.
- Maintenance is simplified
- Stability is enhanced
- Changing business requirements are easy to address

Understanding this approach, one can see how these benefits are gained. Development starts with a visual process. The process is defined in terms of manageable chunks of logic, closely modeling the real world. State machines are defined and created using the IDE, which allows the developer to easily see the state transitions in a tabular display. Development continues with the filling out of stub routines. Testing is simplified via the extensive logging capabilities, allowing state transitions to be viewed after the fact.

Maintenance is simplified by allowing failing logic to be identified and isolated. Changes to address issues are limited to small, easily understandable chunks of code. New programmers can easily understand the process by viewing the state machine in the IDE, rather than having to pour through thousands of lines of code. With an easy to maintain application also comes stability, which is high on the priority list of any automation facility.

Business requirements can be addressed without causing a major ripple effect on the application. States, events and action routines can be added easily, without causing impacting other portions of the code, simply by modifying the state machine definition and filling in new stubs.

Maintaining code libraries for multiple clients on a wide variety of applications would be a nightmare without a unified, logical approach. "Necessity is the mother of invention",

and in our case, the necessity of a sane way to support our customers has led us to adopt the Hierarchical State Machine approach. Doing so has paid off, with each successful implementation a validation of this approach.

****

This article was written by Steve Elliot of Copperline Consulting Services. Founded in 1991, Copperline has provided reliable, fast and scalable solutions to automate product manufacturing and distribution processes for its client companies. Where others focus on one area of the automation process, they have been successful in addressing automation needs across the entire spectrum of technologies encountered in an automated facility. Their integration specialists have the experience and knowledge to integrate the business applications with the production facilities, to connect the back-office with the factory floor. For more information on Copperline Consulting Services, please visit their web site at www.copperlineinc.com

Source:

http://www.automation.com/library/articles-white-papers/process-control-process-monitoring/hierarchical-state-machines-in-the-automation-process