Proceedings of the 2nd CIRP International Conference on Reconfigurable Manufacturing,
Ann Arbor, MI, August 20 - 21, 2003.

Expanding the Role of Finite State Machine Technology in Open Architecture Control

J.L. Michaloski, F.M. Proctor, and W. G. Rippey
National Institute of Standards and Technology
Gaithersburg, MD, USA

S.R. Kolla
Department of Technology Systems
Bowling Green State University
Bowling Green, Ohio USA

ABSTRACT
Modern machine control systems rely heavily on software to achieve system functionality. Until recently, control software's primary purpose was to provide logic and sequencing of machine execution. With computer hardware now providing power and memory to spare, control software is broadening its horizon from a focus on execution control to multiple phases within a control systems lifetime, such as design, testing, and maintenance. This paper reviews an object-oriented Finite State Machine (FSM) framework developed at National Institute of Standards and Technology (NIST) that exploits this potential and expands the software utility and functionality of machine control logic. The public domain FSM framework handles additional control requirements such as reusability, extensibility, modularity, testing, diagnostic troubleshooting, reporting, and maintenance. Please send any comments or questions by e-mail to: john.michaloski@nist.gov.

KEYWORDS: control, Finite State Machine, open architecture

1 INTRODUCTION

Sensor-based motion controllers in manufacturing applications rely more and more on software to achieve system functionality. From a behavioral standpoint, the control logic provides the means by which certain actions are taken in response to different situations. Control logic is predicated on the fact that controller behavior must operate safely in an unpredictable environment where components or communications can fail at random. In this realm, control systems fall under the domain of state machines, in which digital and analog devices and sensors interact over time and incorporate both feedforward planning and feedback error compensation. Examples of these systems abound in manufacturing such as Computer Numerical Control (CNC) machines, robots, and Program Logic Controllers (PLC). In this paper, we focus on improving the reuse and expanding the role of Finite State Machines (FSM) in machine control.

As part of our work at National Institute of Standards and Technology (NIST), a General Motion Control (GMC) Testbed has been developed with one goal being to validate the Open Modular Architecture Controller (OMAC) Application Programming Interface (API) specification for reconfigurable, plug-and-play open-architecture controllers [10]. Within the OMAC API specification, the finite state machine model was determined to be sufficiently robust at modeling control logic, while at the same time conducive to best programming practices. FSM representation is a rigorous approach to modeling the machine control logic and can incorporate measures and tests of correctness beyond testing as well as provide for flexible reuse and reconfiguration [2], [12], [13]. For this reason, FSM control logic is pervasive throughout the OMAC API. FSM are especially appealing to the manufacturing industry as their use can contribute to better error traceability, faster diagnostic troubleshooting and easier maintenance [1]. The better diagnostic capability cannot be overstated as this has been considered the number one challenge for manufacturing systems [9].

In this paper, we look at the potential to expand the software utility and functionality FSM within machine control. First, we explore the normal requirements and special considerations for modeling machine control in a FSM framework. Then we look at FSM design and implementation reuse as a means to provide a consistent control methodology as well as offer a flexible way to handle extensions, modifications and reconfiguration. Next, we look at the policy mechanism to reuse state logic under different FSM methodologies. Finally, we demonstrate the expanded role of FSM in dynamic timing and report generation to provide improved testing, diagnostics and comprehension. The paper includes commentary on the importance of various FSM modeling aspects within the GMC testbed.

2 FSM CONTROL MODEL

A FSM models a machine controller as being in one of a finite set of the possible states, known as the state-space, at any given time. In order to handle the FSM logic within

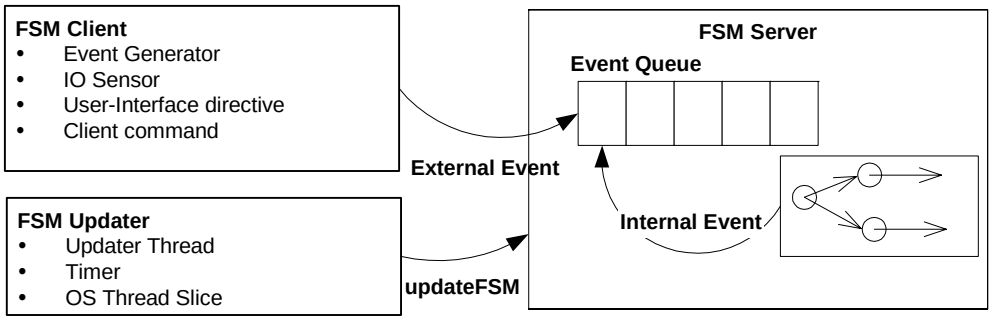

Figure 1. FSM Architecture

our GMC testbed, we felt that programming of control logic that relies on embedding control logic through if/then/else statements fails to leverage software techniques such as inheritance and aggregation. We considered using commercial Computer Aided Software Engineering visual programming environments that feature diagrams and state charts, but found them to be an all-or-none type of proposition and not conducive to our validation effort. We experimented with some open-source FSM code generators [4], [7], [11] but programming changes were hard to tie back into the initial FSM model. Instead, we developed a FSM framework following an "abstract factory" pattern integrated in C++ and Java.

We incorporated numerous programming capabilities you would expect of a FSM framework. The framework contains related class definitions for states, events, conditions, state transitions and actions necessary in the building of an abstract FSM. A FSM is programmed by factoring gross levels of behavior into states, identifying events, and then mapping the events into state transitions. Within the abstract FSM, the ability to nest FSMs into a hierarchical state machine is available. An abstract FSM can refine logic based on specific conditions, timeouts, and event triggers while in a given state. An abstract FSM can provide a mechanism for outside observers to monitor the FSM behavior in accordance with the "Observer Pattern" [3].

Overall, the FSM framework provides features found in the Unified Modeling Language (UML) statechart notation [5] with some extra features and refinements added to account for machine control. Machine control depends on periodic-updating behavior that can be handled differently based on the underlying state machine models and requires special consideration. In general, Mealy machine associate behavior with state transitions and Moore machine associates behavior with states. The FSM framework provides a model that is a combination of the Moore and Mealy models – actions occur at state transitions and periodically when in a given state for an extended amount of time. To satisfy control applications needs, a system must respond to events, such as externally generated commands, and then execute some action, following the Mealy model. Yet a control system must periodically update itself, and by inference its FSM, to monitor its state. Adopting the Unified Modeling Language terminology, our framework provides a class to activity model behavior associated while in a state, and a class to model action behavior associated with a state transition.

The handling of within-state updates, as opposed to state transition updates, is another matter affecting FSM modeling. In the Mealy model, periodic updates of the FSM will require a separate event to trigger a self-transition and associated action within a state. In the Moore model, each state executes an activity associated with the state to service the periodic update. As is evident, either model can handle the FSM update, but there is a requirement to support the UML State Change diagrams, which allow optional entry action to be executed upon entering a state and an optional exit action to be executed upon leaving a state. Thus, in the Mealy model, the self-state transition is actually leaving and reentering the state; consequently executing exit and entry actions. By comparison, the Moore model executes only the activity associated within the state. Within the GMC, the FSM framework uses the Moore activity modeling for within-state updates, which corresponds to the "do" internal state event of the UML state chart notation.

3 REUSE

The object-oriented FSM framework encourages reuse because of the use of abstraction and encapsulation of control logic. The reuse of FSM within the GMC helped to provide a consistent design methodology to reduce system development time and follow-on training time. Inheritance, composition or delegation provide alternative strategies for reusing abstract FSM.

Class reuse through inheritance defines new subclasses derived from one or more super-classes. The subclass inherits all methods and instance variables of its super-classes. One can overload inherited methods in the subclass and add new methods and instances variables. Within the FSM framework, class reuse allows sharing as well as modification or reconfiguration to meet particular control requirements. Since class inheritance defines a relationship at compile-time, a derived class cannot change its base class dynamically during program execution. Moreover, modifications to the base class automatically propagate to the subclass, thus complicating software maintenance. Within the GMC, we did not find static class inheritance to be a problem as the OMAC API control logic was relatively stable.

By comparison, reuse through the delegation hands logic over to another object. The delegation pattern establishes the relationship between objects where one object forwards certain method calls to another object, called its delegate. The idea in the delegation pattern is to create the FSM object that will handle the all required control

logic thus limiting any logic changes to one object. Delegation is useful for the dynamic change of object relationships. Delegation enables the encapsulation of certain functionality in another object and allows code to be loosely coupled, a positive in software maintenance. One advantage of delegation is run-time flexibility – changing the delegate is easier at run-time. Delegation can be a powerful reuse technique, but unlike inheritance does not facilitate dynamic polymorphism so it places an onus on the programmer to code all the delegation methods.

In building the GMC testbed, the reuse of a inherited FSM classes reduced component testing and was quite simple to do in the FSM framework. The delegation of FSM logic was attempted but required extensive recoding of source code in C++ so that it was rarely used. Instead, the use of different setup methods to describe state logic was found to be more useful than delegation. Dynamic reconfiguration of FSM logic was still possible using inheritance by clearing and redefining state logic. We found that the inherited base FSM class offered a clean mechanism to establish sequencing of control logic by defining abstract or limited functionality event, state transition, and action/activity methods. The overriding of action/activity methods customized behavior for the component while the base FSM handled the sequencing of control logic in calling these methods.

4 RECONFIGURATION

Component-based development assembles systems from existing building blocks of code. Different programmers prefer different methodologies so the ability to integrate components in a flexible manner is crucial to component-based development. For flexibility, a FSM should support numerous methodologies and allow reconfiguration to handle differing event, error, and execution procedures. In order to use FSM under different methodologies, the FSM framework contains a policy class to customize FSM behavior in a flexible manner. Of course, a default policy is loaded when an instance derived from abstract FSM class is constructed. Users can customize this policy or develop a separate policy for repeated use across many different abstract FSM classes.

Interaction between events and the state logic is a major issue in control logic methodology. Figure 1 illustrates this interaction as a connection between an event producing FSM client and an event consuming FSM Server as modeled within the client/server paradigm. It is assumed that more than one client or more than one server could be connected to each other. The relationship between events and the FSM can vary greatly depending on the system architecture and can differ regarding event generation, event transmission, event validity, and event handling. The fundamental architecture issue is in regard to event handling: events can be serviced immediately (i.e., synchronously), or can be queued and handled later (i.e., asynchronously). For an asynchronous architecture, a FSM Updater running in a separate thread is necessary to service events. Another consideration is that in a synchronous event-driven FSM, we only perform actions when an event has occurred so there are no state-based periodic activities. Further architecture consideration must be given to error handling. Events without a corresponding state transition from the current state could be ignored or cause the FSM to fail. To achieve a reliable component framework, the FSM policy allows a variety of configuration in a standard way so that interoperability and plug-and-play of components is possible.

The FSM policy provides control over how to handle FSM execution, whether the FSM is synchronous or asynchronous. The FSM policy provides for options for either priority or standard FIFO queuing. Asynchronous options exist for automatic FSM update by a separate background thread on or by a timer for tighter update regulation. The FSM policy determines whether to maintain timing statistics during the FSM operation and whether any error-conditions monitoring (such as timeouts) should be done if timing is performed. Under asynchronous execution, the FSM policy controls the thread execution period for timed or threaded updates. The FSM policy determines error handling for such situations as an invalid event for the FSM, state duplicate event matches, and invalid events in current state.

In the GMC testbed, the FSM flexible error policy mechanism was helpful in that errors could be either strictly handled or more loosely handled, depending on the circumstances. This is especially true for dealing with the user-interface, as event errors, such as invalid events in a given state, which are not catastrophic, could be handled in a graceful manner. Because invalid events are so common when dealing with the user-interface, the FSM framework provides a **`canHandleEvent`** method so that the user-interface can determine if an event is valid in the current state, (i.e., has a state transition for the event). This event handling method assumes synchronous event handling or an empty event queue, but this was not found to be a limiting constraint.

5 REPORTING FACILITIES

The FSM framework expands the role of FSM from control logic to include reporting facilities that can assist in system analysis, diagnostics and maintenance. The FSM framework can generate HTML or XML describing the state transitions of a given FSM. Further, the FSM can also generate HTML or XML describing the execution processing as summary timing statistics. This information is available for hard-copy documentation or for on-line during design- or run-time to improve comprehension as well as for tracking state transitions.

The FSM framework generates state table documentation describing the FSM as an alphabetized list by state of each internal and external state transition for the given state. Each table row contains the state name, the event triggering the state transition, any guard conditions for the event, the ending state, and the associated action/activity. The documentation will generate tables for all substate FSM as well. The generated documentation corresponds directly to the actual control logic and is not generated from embedded comments or other programming artifacts that may not reflect the actual control logic.

Software understanding is a prerequisite for system support tasks such as testing, maintaining, modifying and renovating. Typically, system support consumes over half the software engineering resources. Clearly the self-documenting FSM feature can help in providing analysis for understanding system operation. The availability of automated, source-code-generated, documentation is

FSM Processing (in seconds)						
State	**Average Duration**	**Worst Case Duration**	**Total Elapsed Duration**	**Average Processing**	**Worst Case Processing**	**Total Processing**
RUN	11.151000	33.453000	63.703000	0.000000	0.204000	4.378000
STOPPED	8.265667	24.797000	37.391000	0.000000	0.000000	0.000000

Table 1. Timing Report Example

especially useful, as experience in industry shows that even if documentation is available it may not be up-to-date [8]. The on-the-spot report generation based on internal code is also more accurate than if generated by comment extraction tools. Self-documenting components is an especially critical aspect for reuse, as understanding how a component works is critical for proper systems integration.

The FSM framework can generate an HTML or XML diagnostic documentation string that describes the FSM performance by state in table format. Table 1 shows the statistics for two FSM states RUN and STOPPED, that includes performance statistics for average duration, worst case duration, total duration, average processing time, worst case processing time, and total processing time.

In the FSM timing report table, the duration statistic means how long was spent in a given state according to the wall clock. By wall clock, we mean that if we enter a state at 1:00PM and leave the state at 1:01PM, then the duration was 1 minute, irrespective of the fact that only 1 millisecond of processor time was required to enter, update, and leave the state. The average duration represents the average for the total wall clock time in a given state. The worst case duration gives the longest wall clock time spent in the state. The total elapsed duration gives the total amount of wall clock time spent in the state. The average processing represents the average for the CPU processing time in a given state. The worst case processing gives the CPU longest the processor required in the state. The total elapsed processing gives the total amount of CPU processing time spent in the state.

Because timing is an automatic part of the FSM framework and available while running, timing statistics can allow easy analysis and evaluation of condition-based hardware maintenance. A standardized FSM within the packaging industry (PackML) has been adopted as a way access line performance metrics [6]. The PackML state model was easily modeled with the FSM framework, and the reporting facilities provide an easy way to provide performance metrics.

6 DIAGNOSTICS AND MAINTENANCE

The FSM framework integrates diagnostics capability into every abstract FSM class. The FSM incorporates a parameter table class to handle state variables in a textual "ini" format like string. The parameter table class creates a table of entries containing parameter name, parameter type, and an address for storing and retrieving data values. The FSM reporting feature saves snapshots of data stored in the parameter table at each major state transition that can be dumped later if necessary. The FSM also provides event and state breakpoints within the parameter table. These breakpoints can trigger an execution breakpoint whenever an event is received or a state is entered greatly augmenting the existing debugging environment. Policy flags are also stored as parameters to catch program logic errors by "breakpointing" at any deviation from the accepted FSM logic, such as unexpected events or invalid state

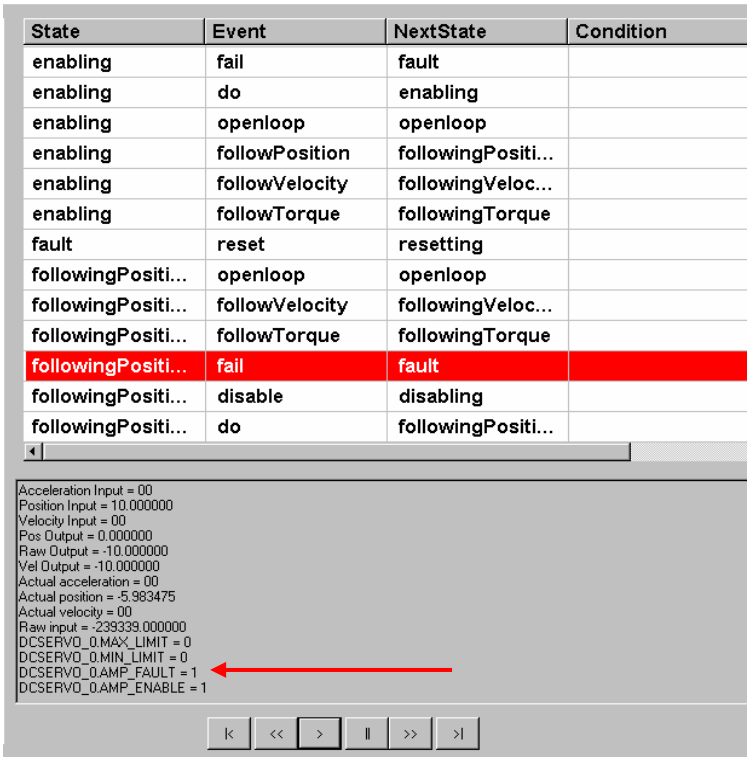

Figure 2. FSM Diagnostic Playback

transition.

One benefit of using FSM framework within the NIST GMC testbed was the eased traceability of program logic. The GMC testbed integrated these features with sequencing buttons under a FSM display to allow forward and reverse playback to “reenact” errors. Thus, hardware faults or other system errors can be easily pinpointed and the sequence of events leading to the error can be traced within the FSM. Figure 2 illustrates the NIST GMC testbed controller after an amp fault causes the **`followingPosition`** state to transition to a **`fault`** state. The arrow (added for this paper) shows the Amp Fault IO point as 1, indicating a fault.

7 SUMMARY

This paper described the public-domain, object-oriented, source-code FSM framework developed at NIST to validate the OMAC API. While the standard practice for coding a finite state machine is to use long conditional branches or while loops, the FSM framework uses a class factory pattern to encapsulate states, events, actions, transitions, and conditions necessary in the programming of machine control behavior. Unlike while loops or conditional branches, the FSM framework imposes structure on the code and makes its intent clearer. Encapsulating each state transition and action in a class elevates the idea of an execute state to full object status. Within the GMC testbed, the use of a FSM framework to develop common FSM provided a high degree reuse and reconfigurability when developing OMAC API components, which simplified coding and testing. In addition, the framework adds classes for policy, logical correctness, and parameter definitions to extend the framework. These additional facilities added to the life cycle control support by providing differing FSM methodology support especially for user-interface development; automated documentation generation; diagnostic playback for troubleshooting; reconfiguration of logic; and improved program correctness.

REFERENCES

[1] Birla, B., Yen, J., Skeries, F., and Berger, D., 1999, “Controls Software Requirements for Global Commonization at General Motors Corp.”, Control Engineering.

[2] Brave, Y. and Heymann, M., 1991, “Control of Discrete Event Systems Modelled as Hierarchical State Machines,” Proc. of 30th IEEE Conf. Decision and Control, 1499-1504

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1994, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison Wesley, Reading, MA.

[4] Grail++, http://www.csd.uwo.ca/research/grail/

[5] Harel, D., 1987, “State charts: A Visual Formalism for Complex Systems,” Science of Computer Programming, 8, 231 - 274.

[6] Kowal, J., 2003 “Best practices: implementing the PackML™ state model in Sequential Function Chart”, Packaging World.

[7] Libero, http://www.imatix.com/html/libero/

[8] Liu K., Alderson A. and Qureshi Z., 1999 , “Requirements Recovery from Legacy Systems by Analyzing and Modeling Behavior,” ICSM, 3-12.

[9] NSF Workshop on Logic Control for Manufacturing Systems, 2000, University of Michigan, Ann Arbor, MI.

[10] Open, Modular, Architecture Group (OMAC) User’s Group, http://www.omac.org

[11] Ragel State Machine Compiler, http://www.essemage.com/ragel

[12] Ramadge, P. J. and Wonham, W. M., 1987, “Modular Feedback Logic for Discrete Event Systems”', SIAM Journal of Control and Optimization.

[13] Wang, S. and Shin, K.G., 2000, "Generic programming paradigm for open architecture controllers," WAC 2000/Seventh International Symposium on Manufacturing with Applications.