Analysis of Module Interaction in an OMAC Controller

John Michaloski
National Institute of Standards and Technology

ABSTRACT

Machine controllers built from standardized software parts, commonly referred to as components or modules, have the greatest potential to reap open architecture benefits – including plug-and-play, reusability and extensibility. Modularity is the key to enabling component technology. Naturally, module interaction is a by-product of modularity, and must be explicitly modeled to allow plug-and-play technology. This paper will present a high-level model of component interactions in order to allow component-based machine controllers. Discussion will focus on Functionality, Infrastructure and Connection interfaces for dealing with the common software functionality such as handling normal operation, creation and destruction, parameter manipulation, connection, wiring, licensing, security, registration, binding, discovery, naming, and introspection. The concept of introspection will be explored as it relates to designing a machine controller architecture using an Integrated Development Environment.

KEYWORDS: Finite State Machine, component, module, client/server, control, standard, modularity, proxy agent, machine

BACKGROUND

Open architecture controllers [11] have the potential to transform manufacturing by exploiting the power of standardized software parts, much like standardized mechanical parts contributed to the industrial revolution. The key to standardized software parts is modularity, which refers to encapsulating common functionality into a separate entity. In current software parlance, a modular entity is referred to as a "component," while a grouping of components will be called a "module" (akin to a hardware assembly). Modularity makes "plug-and-play" possible so that you can replace a component (module) without affecting the rest of the system. If properly designed, components and modules can be reused in different applications. Reusability, when combined with feature customization, allows engineers to the tailor components to meet the specific needs of individual applications. All in all, modularity can make it easier and more cost-effective to design, develop, and integrate applications.

Assuming object-oriented technology, the fundamental aspect to attaining modularity is encapsulating functionality by specifying interfaces. However, there are other factors besides explicit functionality that play an important role in making plug-and-play possible. A module must understand the framework in which it will operate and how the module interacts with other modules; a module must advertise what it does, where and how it operates, as well as what the module requires in order to operate. This paper will attempt to address these module interaction questions as they apply to open architecture controllers. Much of the discussion will be drawn from the efforts of the Open Modular Architecture Controller (OMAC) Application Programming Interface (API) workgroup [10] to standardize modular technology in open architecture controllers. The paper will start with an overview of the OMAC API specification methodology. The paper will continue by examining the different OMAC API constraints required to allow component interaction in a machine controller. Finally, the paper concludes with discussion on the issue of development using mainstream, high-volume, component technology.

SPECIFICATION METHODOLOGY

The goal of the OMAC API workgroup is to develop a specification for machine controllers yielding the benefits of open architecture technology – e.g., plug-and-play, extensibility, and reusability. The OMAC API workgroup advocates component-based technology as the best alternative to attaining open-architecture benefits. As a consequence, the specification defines components that cover the gamut of machine control functionality and a framework in which the components cooperate. Figure 1 illustrates the OMAC API relationship between different elements in a component-based abstraction hierarchy. An *application* is a program designed for machine control. The OMAC API application domain is multi-axis, coordinated motion control typical of Computer Numerical Control (CNC) machines or robots. Representative controller applications include cutting, manipulation, and grinding. The targeted range of controller complexity is quite broad – from multiple robotic arms to single axis controllers. A *framework* is the infrastructure for integrating software. Examples of frameworks include Microsoft's Component Object Model (COM) [8] or the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) [2].

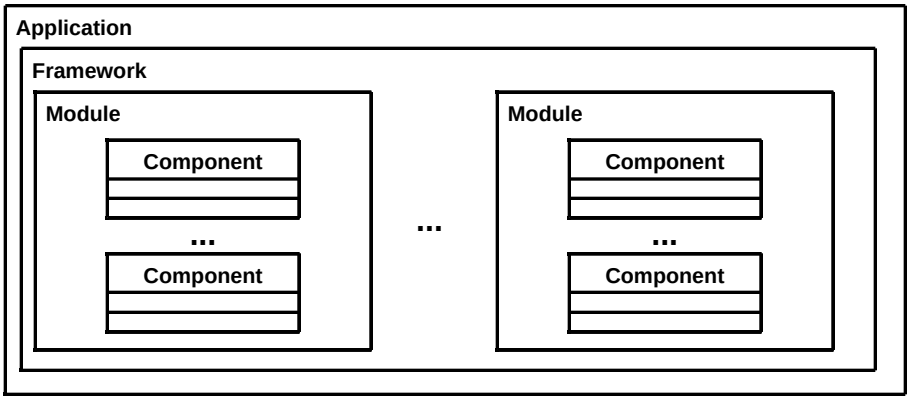

Figure 1. OMAC API Component-Based Abstraction Hierarchy

A *component* is a reusable piece of software that serves as a building block within an application. Example components defined in the OMAC API specification include: IO Point, Control Plan, Kinematics, Process Model, and Control Law. *Interfaces* define component functionality. A component may contain multiple interfaces, either through aggregation or inheritance. New components may extend functionality by means of aggregation or specialization. A *module* is a type of component, one that acts as a container of components of some other type. A module provides a means of storing component references and then providing access to the component references. The OMAC API specification defines a series of modules, including Axis, Axis Group, Task Coordination, and Discrete Logic.

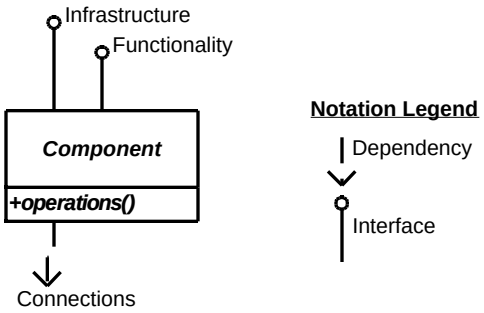

Figure 2. OMAC Component High-Level Interface Model

Using the Unified Modeling Language (UML) notation [12], Figure 2 illustrates a high-level model common to all components that includes Functionality, Infrastructure and Connection interfaces. These interfaces provides a uniform API for dealing with the common software functionality such as handling normal operation, installation, creation and destruction, parameter configuration, initialization, startup and shutdown, licensing, security and registration, persistent data saving and restoring, enabling and disabling, binding and discovery, naming, and introspection.

The Functionality interface exposes the control capability of a component. This interface includes methods for behavior, state, and parameter manipulation. The Infrastructure interface is vital in support of a broader application framework. The Infrastructure interface handles component identification and registration, which is used for the component to advertise what it does, where it is, and how it operates. The Connection interface advertises the module dependencies such as the services it requires and enumerating the events in and events out it supports, thereby answering the question of what a module needs.

FUNCTIONALITY

Functionality refers to the range of application-specific behavior a component is capable of performing. In the case of machine controllers, functionality involves different aspects of motion control, discrete logic and program coordination depending on the component. For components to properly use other components, a standard model of functional behavior must exist. For the OMAC API, behavior is event-driven and modeled with finite state machines (FSM). Event-driven systems correlate actions to discrete changes in the world. Change is modeled as an *event*, which triggers a FSM state

transition. Within a FSM, actions can be associated with states and/or state transitions. Associating an action with a state is categorized as a Moore machine. Associating an action with a state transition is categorized as a Mealy machine. The OMAC API adopts the Mealy machine model since it is well-suited for discrete event-driven systems. Given this background, component interaction in the event process will be studied as it applies to various ways in which components communicate.

The discussion will start with the prevalent model of interacting components, the client/server architecture, in which clients send messages requesting service to the server, which generates a response. Component-based technology uses an object-oriented approach where the client invokes server methods to achieve message passing. For distributed communication, component-based technology prescribes a *proxy agent* to handle method invocations that cross process boundaries [14]. Since responses can take time, clients have a number of options in awaiting a response. *Synchronous communication* involves the client issuing a request to the server and then blocking, waiting for a server response. Upon receipt of the server response, the client resumes processing. *Asynchronous communication* differs from synchronous communication in that the client returns immediately after issuing the request and does not wait for a response. At some future point in time, the server generates an event signaling that it has finished servicing the request.

In the context of a machine controller, client/server communication can be of a query, command, or event service type.

Query requests simply manipulate parameter values through set and get accessor methods. Query requests are typically synchronous due to the minimal response time required. Query requests via a set method can cause a side effect resulting in an internal event. As a query example, consider the interaction between a Human Machine Interface (HMI) client and an Axis server in control of a single axis of motion. A query-based interaction would have the HMI request from the Axis its current position.

Command requests are events that cause a state-change and a resulting behavioral action to be taken by the server. The client/server command behavior is widely known as the supervisor/subordinate software pattern [5]. Since having the client wait until commands complete before a response is returned is inefficient and can lead to deadlock, OMAC API prescribes asynchronous communication for command-related requests so that server components can buffer command requests to minimize command request/response turnaround. Returning to the HMI-Axis example, the HMI client may issue a command request to an Axis server component to “jog” to an absolute position, continue processing and either poll or use event service to determine when the Axis completed jogging.

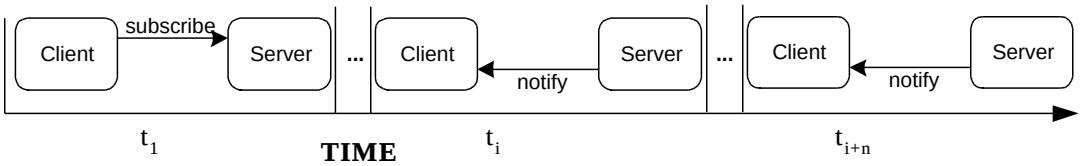

Figure 3. Event Service over Time

Event Service is a variation of asynchronous communication, illustrated in Figure 3, in which the client issues one request and then receives one or more asynchronous response(s) from the server. The initial client request is known as event subscribing.

After subscribing to an event, the client is notified when the event occurs, which is called *notification.* Event service with a single server and multiple clients is modeled as the Subject/Observer or Publish/Subscribe software pattern [6]. A more general model of event service is the Producer-Consumer software pattern where one or more producers generate events to be forwarded to one or more consumers. Events generated by the producers result in a call to each consumer's *event handler*. Event handlers are methods supported by the consumer, which execute other actions in response to the event reception. In this way the consumers can handle actions like I/O changes, state transitions, and elapsed timers. Event service can take on Push and Pull variations. In the Push event model, events occur asynchronously. In the Pull model, a consumer pulls events from a producer corresponding to the idea of polling.

There are several approaches to implementing event service including a two-tier or peer-to-peer, as well as a three-tier or brokered approach managing multiple producers/consumers. In a two-tier (or peer-to-peer) architecture, a producer interacts directly with consumers, with no intervention. This protocol defines a one-to-many relationship so that when the producer generates an event, all its dependent consumers are notified. In Microsoft COM, peer-to-peer event service is implemented as "Connection Points" [9]. Javaä calls delegation through the peer-to-peer event service the Source-Listener software pattern. A three-tier architecture introduces a *broker* between event producers and event consumers, which allows for many-to-many communication. Brokers act as intermediaries similar to queues in a queuing model, but can provide extra features such as event filtering, event correlation, and event quality of service. Different push/pull combinations are possible between the producers, the broker and the consumers leading to different computational paradigms: Notifier, Agent, Queued, and Procurer [13]. OMAC API endorses the Notifier paradigm, the case where the producer pushes an event to the broker, which pushes the event to all the subscribed consumers.

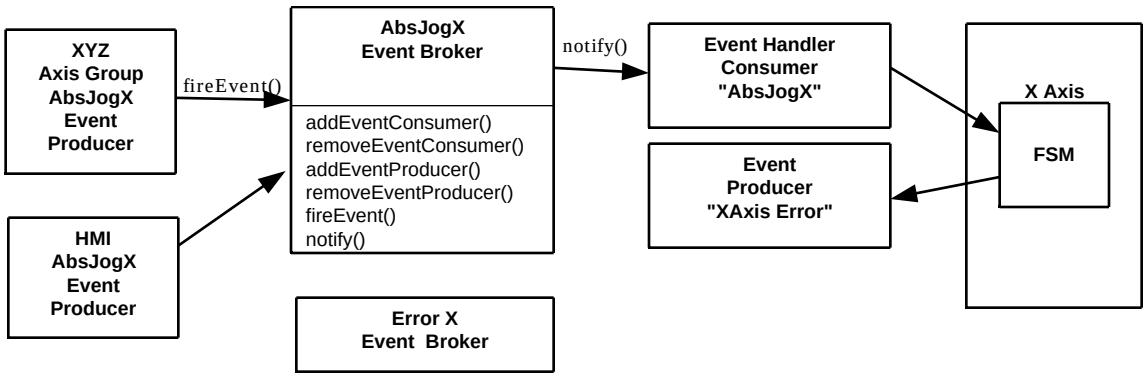

Figure 4. Example Brokered Event Service

Illustrated in Figure 4 is an example of brokered event notification for the Axis absolute jogging example. A separate X Axis Absolute Jog (AbsJogX) broker and X Axis Error broker are defined. The X axis uses the `addEventConsumer()` broker method to subscribe as a consumer of AbsJogX events, while the HMI and XYZ Axis Group use the broker method `addEventProducer()` to subscribe as producers of AbsJogX events. When the XYZ Axis Group uses the `fireEvent()` method to generate an AbsJogX event, the X Axis consumer event handler will receive notification of the AbsJogX event.

For the X Axis, the event is propagated to its FSM, which initiates the appropriate action.

The OMAC API prescribes the following event service protocols. Peer-to-peer event service allows client event pushes are advertised through the Functional interface. Peer-to-peer server event pushes are special Functional interfaces, known as sources, to which clients connect. Three-tier event service is achieved by components advertising Events-In and Events-Out with a separate broker required for each unique Event In/Out.

INFRASTRUCTURE

Component interaction for the Infrastructure interface will be examined using one of many potential frameworks, Microsoft's COM. In COM, a server is a class factory that creates components. COM servers can be inprocess, acting as a Dynamic Link Library (DLL), and/or local or remote Executables (EXE). In COM, each component is identified by a class id (CLSID), which is a unique 128-bit globally unique id (GUID). COM uses this CLSID to associate a specific DLL or EXE server to a component. For each component, an association is made to the server in the Windows registry. In COM, c*ategory* ids (CATID) identify those interfaces that a component implements and those interfaces it requires. Components are expected to enter its Implemented/Required interfaces in the Windows Registry. OMAC API adds the concept of CATIDs per OMAC module in the Windows registry to reflect the addition of multiple servers per component/module. For example, vendor A and vendor B might both install Axis module servers. In this case, both servers are registered under the Axis Module CATID. Browsers could then use this category information to determine available Axis Module servers.

Discovery is a way for components to dynamically find component servers. In COM, each component CLSID is associated to a server, which may be automatically activated. *Activation* uses a component identifier and server type (inprocess, local, remote, any) to locate a server, which creates the desired component, and then launches the server. Activation is contingent upon satisfying licensing and security requirements. *Licensing* allows only the authorized use of a component. *Security* determines who is granted access to the component services and the type of access that is granted. Assuming activation, the server can now create new components.

Finding existing components, i.e., those already created by a server, will be called connecting. COM does not support this feature and limits finding to component servers. The next section will review the OMAC API recommendation for this framework need.

CONNECTIONS

Connecting refers to resolving component dependencies and results in a system architecture. The OMAC API has defined three types of dependencies: Instance References, Events-In and Events-Out. Making connection information dynamically accessible as part of the component will be called *introspection*. Introspection allows components to be manipulated in Integrated Developer Environment (IDE) builder tools. OMAC components are naturally expected to operate in a running application, but with introspection, components can be used in support of the design and run-time lifecycle phases. This concept is common elsewhere in the software industry (JavaBeans [15] and

Active/X [3]), but is rare in the machine controller domain. Using introspection at design time, an IDE builder tool can query the component for the references it publishes, the types of OMAC interfaces it requires as references, the events-in it requires, and the events-out it generates. The designer can then connect the "wires" among the various OMAC components.

To distinguish the use of introspection at different lifecycle phases, the act of resolving component dependencies at design time will be called *wiring*, and at runtime *binding*. *Connecting* is the process for discovering, activating and wiring/binding component instances. An *instance* is an existing component and *naming* assigns a global identification to each instance. OMAC API adds component instance discovery through naming so that components can find other components by global name, or if no matching component is found, to create a new component with the given name.

OMAC component instances publish their name and properties in an instance directory. *Publishing* associates names with instances that implement a given interface and makes the instance reference address globally available to other components through an instance directory. Components find each other through the instance directory by name (Directory Service) or property (Trader Service). If a matching component instance is not found, a new instance is created that may require server activation.

Connecting matches a component dependency name to a global instance name. Since dependency names are vendor-specific, they are effectively hard-coded so that it would be difficult to change them at design time. To match dependencies to instances, OMAC API specifies that design-time wiring map a local dependency name to a global instance name. Local naming is responsible for the names associated with a particular component. A vendor would be responsible for distributing a local naming table associated with each component. For example, the following table sketches a local naming table for an Axis module.

Local Name	**Type**	**Connected**
"ENCODER"	IIOPointFloat	Y
"ACTUATOR"	IIOPointFloat	Y
"POSITION_CONTROL_LAW"	IControlLaw	Y
…	**…**	**…**

Global naming is responsible for mapping local names to global names. Global naming serves two purposes. First, global naming allows system access to local address references. Second, global naming enables familiar naming conventions. For example, a three-axis mill would have three instances of the parameter ENCODER that could be resolved into corresponding global names of X-ENCODER, Y-ENCODER, and Z-ENCODER. The table below sketches the global-local name mapping for the X-Axis.

Global Name	**Component**	**Local Name**
"X-AXIS-ENCODER"	"X-AXIS"	"X-ENCODER"
"X-POSITION-CONTROL_LAW"	"X-AXIS"	"POSITION_CONTROL_LAW"
…	**…**	**…**

DISCUSSION

The OMAC API vision of plug-and-play involves reduced cost and higher fidelity through leveraging pervasive, off-the-shelf, high-volume, component technology, such as

Microsoft COM for component framework, or UML for component design. This paper presented an overview of some additional component constraints, as specified by the OMAC API, to make such plug-and-play machine controller development possible. To that end, a high-level component model was presented with Functional, Infrastructure and Connection interfaces. The component interaction based on these interfaces was examined.

In the machine controller industry, there exist controller IDE products, e.g., [4, 7], that achieve many of the desired goals of the OMAC API prescribed plug-and-play component-based technology. Unfortunately, these products are highly proprietary and the resulting components are not interoperable. Further, these IDE products do not directly support mainstream, high-volume, component technology. Without high-volume component technology, it is economically unfeasible to create and maintain components in a manner specific to the manufacturing automation industry [1]. At NIST, we have used high-volume component-based technology to develop control components. We have developed OMAC API control components in COM that support the additional OMAC API component constraints. We have found the initial overhead in developing control components in COM is significant, but the long-term prospect for component reusability and customization is encouraging.

REFERENCES

1. Birla, S., Yen, J., Skeries, J., and Berger, D., "*Control Systems Requirements for Global Commonization*," Control Engineering Online Extra, http://www.manufacturing.net, January 1999
2. Common Object Request Broker Architecture, Object Management Group, Framingham, MA, 1995
3. Chappell, D., "*Understanding ActiveX and OLE*," Microsoft Press, Redmond, WA, 1996
4. ControlShell, Real-Time Innovations, Inc. http://www.rti.com/
5. Flater, D., Barkmeyer, E., and Wallace, E., "*Towards Unambiguous Specifications: Five Alternative Job Control Models for An Object-Oriented, Hierarchical Shop Control System*," In Proceedings of the 1999 ASME Design Engineering Technical Conferences, Las Vegas, Nevada, ASME Technical Publishing, September 1999
6. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "*Design Patterns: Elements of Reusable Object-Oriented Software*," Addison Wesley, Reading, MA, 1994
7. LabVIEW, National Instruments, Inc., http://www.ni.com
8. Microsoft Corporation, "*Component Object Mode*l," http://www.microsoft.com/com
9. Microsoft MSDN, "*Connection Points*," In MSDN, see mk:@ivt:vccore/F26/D2B/S4CC4A.HTM
10. Open, Modular, Architecture Controls (OMAC) Users Group API Working Group, http://www.isd.mel.nist.gov/projects/omacapi/
11. Proctor, F., and Albus, J., "*Open architecture controllers*," IEEE Spectrum, 34(6), pp. 60-64, June 1997
12. Rational Software Corporation, UML 1.3 Documentation, http://www.rational.com/uml/resources/index.jtmpl
13. Schmidt, D., "*An Overview of OMG CORBA Event Services*," http://www.cs.wustl.edu/~schmidt/
14. Shapiro, M., "*Structure and Encapsulation in Distributed Systems: The Proxy Principle*," In 6TH International Conference On Distributed Computing Systems, IEEE Computer Society Press, pp. 198-204, May 1986
15. Voss, G., "*What is a Java Bean*," http://developer.java.sun.com/developer/onlineTraining/Beans/Beans1/simple-definition.html