Software Models for Standardizing the Human-Machine Interface Connection to a Machine Controller

John Michaloski
National Institute of Standards and Technology

Sushil Birla and Jerry Yen
General Motors Corporation

ABSTRACT

In manufacturing, the Human Machine Interface (HMI) handles the human interaction with a machine controller. Currently, in computer numerically controlled (CNC) machines, most HMIs are tied to a CNC using a proprietary connection that leads to non-reusable, vendor-specific, application software resulting in costlier integration. With a standard HMI-Controller Application Programming Interface (API), users should be able to reduce costs by shortening the development cycle associated with each vendor's software/hardware interface. A worldwide effort among a variety of participants in the CNC standards community is underway to develop a single HMI-Controller API. This paper will review a series of potential API approaches for the HMI-Controller connection as a progression from a flat name space library to a component-based technology. Included is a discussion on the approach selected by the worldwide HMI-Controller API effort.

KEYWORDS: Human-Machine Interface, MVC pattern, Computer Numerically Controlled (CNC) machine, object-oriented, control, standard, user-interface, data model, proxy agent

BACKGROUND

In manufacturing, the Human Machine Interface (HMI) handles the connection between the human and a machine controller. The HMI is responsible for supervisory command and control as well as status monitoring. The extent of HMI functionality can be summarized by three canonical features a HMI must support. First, the HMI must have the capability for soliciting and displaying information reports about the state of the controller, such as current axis position. Second, the HMI must have command capabilities, such as the ability for the user to set manual mode, select an axis, and then

---

jog an axis. Third, the HMI must alert the user when events occur, in other words, handle unsolicited information reports.

These different aspects of an HMI are best understood when viewed as a generalization of the traditional model-view-controller (MVC) architecture, a well-known object-oriented design pattern for Graphical User Interfaces (GUI) [5,7]. The MVC architecture contains three parts: a MODEL, a VIEW, and a CONTROL. Figure 1 shows the relationship between the human, the MVC components and the machine controller.

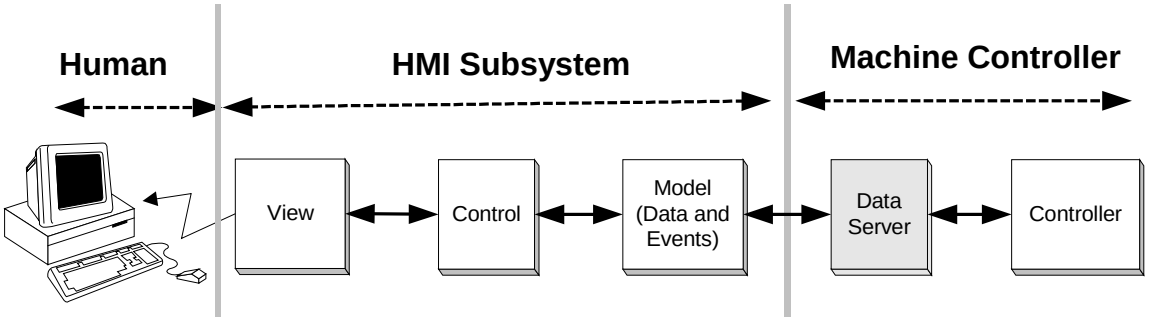

Figure 1. Relationship of MVC Design Pattern to HMI Machine Control

In the MVC architecture, the VIEW corresponds to the front-end or visual presentation with which the user interacts, for example, either a GUI or a teach pendant. The MVC CONTROL is not the same as the machine controller, but instead refers to an object that determines what happens when the user interacts with an HMI VIEW component – for example, what occurs when the user clicks a button control. Different MVC CONTROL modes correspond to different presentation views. For example, there can be a VIEW for configuration, calibration, error handling – as well as normal operation. The MVC MODEL corresponds to the back-end or database, and manages whatever data or values the HMI uses – such as the feedrates, velocities, and current axes values. The MODEL also provides translation services in order to resolve HMI and Controller data representation differences. An additional element between a Controller and the MODEL may exist (as indicated by the shaded box), which we call a DATA SERVER. A DATA SERVER is an optional piece of software and corresponds to the broker design pattern [1]. The DATA SERVER filters data communication by selectively extracting and injecting data into the controller. The DATA SERVER usually operates at a slower frequency than the Controller.

There is a worldwide unification effort underway to standardize the HMI-Controller connection. The primary participants include: (1) Japan FA Open Systems Promotion Group (JOP) [6], (2) Open System Architecture for Controls within Automation Systems (OSACA) [7], and (3) Open Modular Architecture Controller (OMAC) HMI workgroup [10]. Each group has proposed its own solution to standardizing the HMI-Controller API. The effort to unify these contrasting approaches is called the “Global HMI API Project.”

The Global HMI API Project is not attempting to standardize all elements of the MVC architecture. The Global HMI API Project is not concerned with the VIEW as pertaining to “look or feel” of an HMI. The MVC VIEW should be driven by the needs of the customer. The Global HMI API Project is not concerned with defining the CONTROL of views presented to the user and the CONTROL for a given view. Once again, the MVC CONTROL should be established by the needs of the customer.

The primary emphasis of the Global HMI API Project is to define a MVC MODEL API that allows the exchange of data and events between the HMI and the machine controller.

Currently, the MODEL connection between most HMI and Controllers is proprietary. There are difficulties associated with a proprietary connection including increased training costs for maintenance and operation, non-reusable, vendor-specific, application software, and higher integration costs. Preferably, one common HMI-Controller API should exist for all CNC devices, so that users should be able to shorten the development cycle associated with each vendor's software/hardware interface and have wider range of software/hardware choices available.

The goal of the Global HMI API Project is to unify the work of existing HMI standardization efforts. This means agreeing to a common approach, common modularization, common names, common data types, common event models and a common data exchange mechanism. Some requirements have been established in determining the final approach. The API should be independent of the data transport hardware (e.g. communication link) between an HMI and a Controller. The HMI API's should be platform independent. Further, an object-oriented approach is preferred as it directly promotes extensibility, modularity and reusability.

Numerous solutions exist to unify the different HMI-Controller approaches. This paper will review a series of potential API methodologies including:

- Legacy Solutions
- "Flat" C API Name Space
- C++ "Wrapper" Classes
- Centralized Dictionary Component "Wrapper"
- Distributed Components
- Distributed Components with Presentation Views

The JOP, OSACA, and OMAC approaches will be used as illustrations of the different HMI API methodologies. The paper concludes with a discussion on the Global HMI API Project selection of the Centralized Dictionary Component "Wrapper" as its HMI API methodology.

LEGACY SOLUTIONS

In general, legacy HMI-Controller products implement the MVC MODEL as a "dictionary" containing data and events for communication. The overall legacy HMI-Controller interaction is shown in Figure 2. The MVC MODEL is a DICTIONARY that acts as a data server and brokers data communication. The primary purpose of the DICTIONARY is to provide a simple mechanism for binding and naming. We will assume the HMI understands that the DICTIONARY is a linked library (e.g., Microsoft Dynamic Linked Library or DLL, Unix shared library). To bind, the HMI loads the library by name. For data naming, the library exports a list of predefined data and event names. The DICTIONARY resides on the HMI platform and can be static or active.

Figure 2. Legacy HMI

A purely data exchange model would have the MVC MODEL act as a static repository or dictionary of information, with no active role in the shipping or receiving of data. In the static model, events and data are identical and each side would poll for new events. This static approach could be implemented as a DLL without threads that maps variables into some predefined buffer in Interprocess Communication (IPC) shared memory.

To enable dynamic event handling, the DICTIONARY must become an active process in order to forward events. Asynchronous event notification could be in the form of callbacks or interrupts. In this approach, the DICTIONARY is a hybrid because it allows both pushes and pulls on its data and events. One implementation of dynamic event handling is with a threaded DLL that communicates to the Controller via some proprietary transport mechanism.

Flat C API Name Space

The flat name space is a legacy approach that defines the API with C function calls and is illustrated in Figure 3. An advantage to a C API is its universal applicability. In general, linking C libraries generated by different C compilers is possible because of the existing ANSI C standard. Event notification in this approach would be through the use of C callback functions.

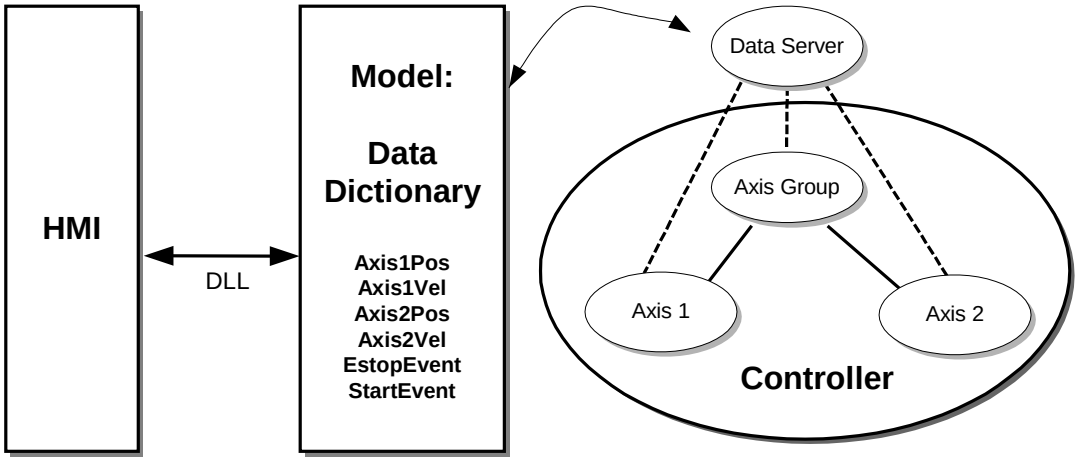

Figure 3. Flat Name Space HMI-Controller Architecture

JOP adopts this approach as it provides a simple, yet flexible, approach to HMI-Controller interaction. A drawback to defining C API is that it does not support object-oriented capabilities and hence offers no inherent encapsulation or modularity.

C++ Wrapper Classes

A C++ object-oriented wrapper design wraps each data element in the DICTIONARY as a separate object. Instead of one or more C API functions per data element, a C++ wrapper groups the wrapper methods into one class definition. The DICTIONARY then "presents" objects to the HMI, as represented in Figure 4 by bubbles originating from the DICTIONARY. OSACA uses the wrapped object approach because it allows every object to seamlessly embed OSACA communication support.

A major problem with C++ wrappers is that they restrict data access to C++ enabled HMIs. Furthermore, C++ implementations are vendor specific since exporting C++ class

and method names (known as demangling) is not standardized. Without a universally supported C++ demangling convention, this C++ wrapper approach degrades to "exporting" a flat C name space.

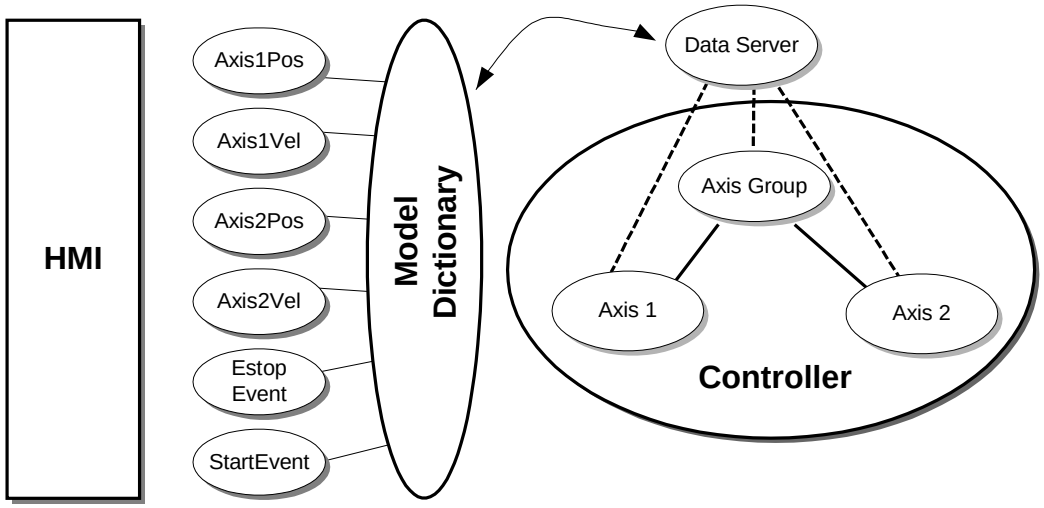

Figure 4. C++ Wrapper Model

COMPONENT SOLUTIONS

The next logical step would be to use a "pure" C++ object-oriented approach, which would tie Controller objects to DICTIONARY objects. Unfortunately, a C++ object-oriented approach would suffer from the same demangling drawback as the C++ wrapper approach. A more flexible solution is to use a component-based approach. A component-based approach differs from object-oriented programming, which is a way to build object-based software components. By contrast, component-based software can be created using many different programming languages, all of which can then work with each other. Component based technology, such as the Component Object Model (COM) [4] or the Common Object Request Broker Architecture (CORBA) [3], would allow an HMI to communicate with the Controller in a programming language-independent, object-oriented manner. The API are defined in a neutral programming language, such as the Interface Definition Language (IDL) and translated into the different programming languages.

For our component-based discussion, we will focus on COM technology, although CORBA offers similar functionality. COM defines globally unique interfaces to solve the naming problem and for handling version control. For binding, COM offers a systematic way for discovery, activation and connecting components. COM defines a peer-to-peer event model, known as connection points, that can be used for event subscription and notification. Implementing the HMI to Controller connection in a component-based approach can be done in several alternative ways, which will be explored in detail.

Centralized Dictionary Component Wrapper

The simplest migration to a component-based approach would be to would wrap the centralized DICTIONARY in COM. This assumes the desire for a centralized dictionary,

which is reasonable, since specifying a complete interface for a CNC machine as a dictionary would increase across-the-board standardization. Figure 5 illustrates wrapping a legacy dictionary as a COM component.

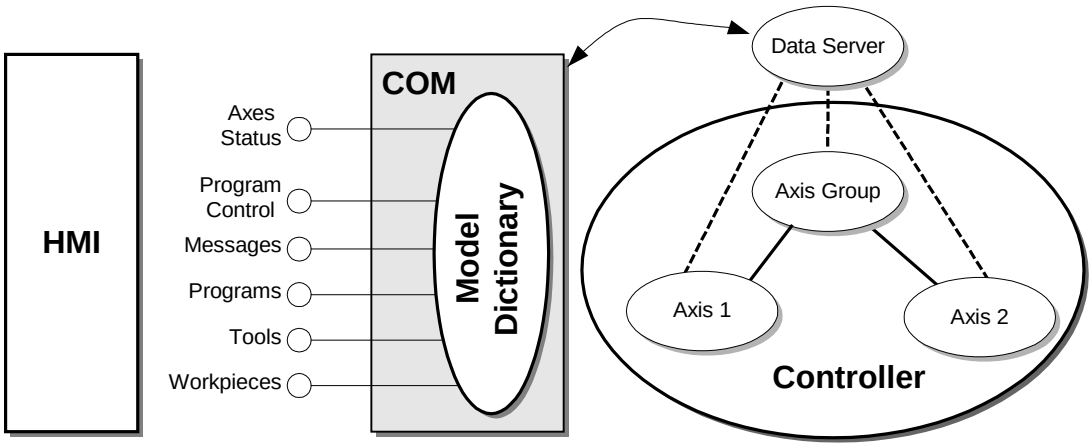

Figure 5. Dictionary Wrapped as COM Component

It would be expected that vendors still employ a data server, which may use an underlying COM model, but would not be exposed to the HMI. In this model, the DATA SERVER is integrated into the Controller and performance issues are handled by the vendor. The major drawback to this approach is that it is a limited form of component-based technology. Because so much of the information is merged into one component, this diminishes the scope of plug-and-play capability and restricts the use of some of emerging component-based features covered later in the paper.

Distributed Components

A transition to a component-based approach raises doubts about the need for a centralized dictionary. In a distributed component-based approach, the HMI could directly access any Controller component. Instead of a single data server, each Controller component provides its own data services through a proxy server [12]. A component proxy server is an essentially "free" by-product of component-based technology that comes in support of location transparency. Figure 6 illustrates the distributed component-based approach.

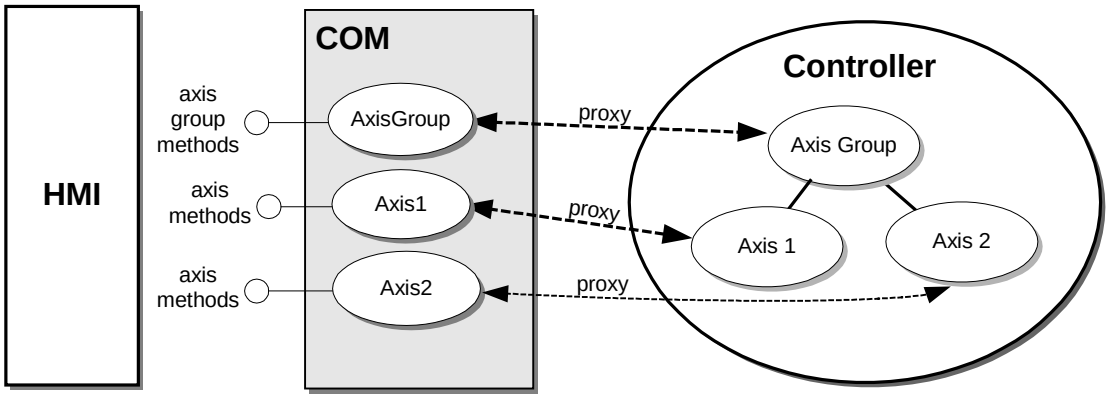

Figure 6. Distributed HMI-Controller COM model

A distributed approach assumes that each Controller component exposes COM interfaces. Since COM components are location transparent, the HMI can bind to a COM component anywhere, be it in-process, local-process or remote process. The most likely case is remote binding because it would be assumed that the HMI and the Controller would reside on different platforms.

The primary benefit to the decentralized COM components would be the increase in reusability of components. An exciting consequence of component-based technology occurs when a component provides "introspection" so that it can be visually manipulated in Integrated Developer Environment (IDE) builder tools. ActiveX [2] and Java Beans [13] are two technologies that enable reusable component deployment from within IDE builder tools. Introspection allows for easier component reuse corresponding to less costly HMI development as well as greater HMI extensibility and customization.

The primary drawback to decentralized components is the uncertainty of real-time controller performance generally resulting from poorly designed proxy agent use, for example, if the HMI samples Controller data at too high a frequency.

Distributed Components with Presentation Views

A natural consequence of components supporting introspection is allowing components to provide their own MVC VIEW as can be done with ActiveX controls. An ActiveX control can draw itself in its own window, respond to events (such as mouse clicks), and be managed through an interface that includes (data) properties and (event) methods. ActiveX components are network independent and can be used in Windows applications or over the Internet in Web Pages. ActiveX controls have properties that allow the user to change the appearance of the control, change certain values of the control, or make requests of the control, such as accessing a specific piece of data that the control maintains. Now, the HMI becomes an ActiveX control container, which manages a set of ActiveX controls. Figure 7 illustrates the use of ActiveX in the HMI-Controller architecture. The open-architecture standards workgroup, the OMAC API [11], champions the notion of introspection-capable components that can provide different presentation views based on their state, such as normal, maintenance, or error.

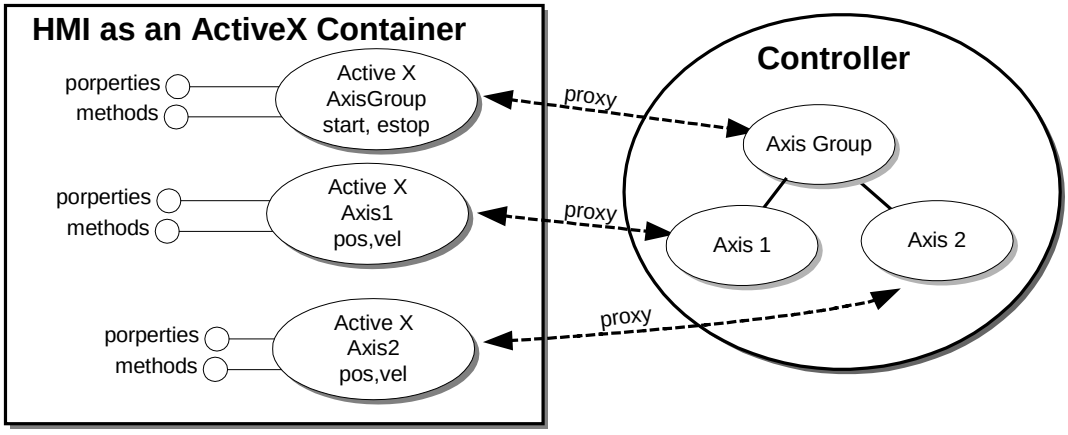

Figure 7. HMI-Controller Architecture using ActiveX

The major advantage of the ActiveX approach is the cost-savings due to the ease of HMI generation, customization and scripting. Very little code needs to be written to get the initial HMI component interaction working properly, at least in comparison to traditional approaches. An integrator would buy and then integrate a set of Controller components with ActiveX support that could then be used during HMI design as well as HMI run-time. The level of programming skill required is reduced because of the simplicity of using an IDE, such as Visual Basic. The major drawback to this approach is the lack of a standard look and feel for HMI components.

DISCUSSION

We have shown there are numerous approaches to standardizing the HMI-Controller API. Each approach has its advantages and disadvantages, so there are tradeoffs in selecting a single approach. The Global HMI-Controller Project has selected a COM component-wrapper of the centralized dictionary as the approach for its standardization effort. Arguably, the HMI would only be realizing some of the benefits of COM software technology. On the plus side, the advances in technology are readily achievable for vendors of legacy HMI-Controller products. Further, the component dictionary wrapper approach reaps many of the component-based lifecycle benefits, such as discovery, activation and binding, without raising numerous performance and compliance issues.

REFERENCES

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., "Pattern-Oriented Software Architecture: A System of Patterns," John Wiley and Sons Ltd, Chichester, UK, 1996
2. Chappell D., "Understanding ActiveX and OLE A Guide for Developers and Managers," Microsoft Press, Redmond, WA, 1996
3. Common Object Request Broker Architecture, Object Management Group, Framingham, MA, 1995.
4. COM Specification, Microsoft Corporation, http://www.microsoft.com/com
5. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley, Reading, MA, 1994
6. Japan FA Open Systems Promotion Group, http://www.mstc.or.jp/jop
7. Krasner, G., and Pope, S., "A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80," Journal of Object-Orientated Programming, 1(3):26-49, August/September 1988
8. "Microsoft Interface Definition Language (MIDL)," Microsoft Corporation, http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/psdk/midl/mi-laref_1r1h.htm.
9. Open System Architecture for Controls within Automation Systems (OSACA) Association, http://www.osaca.org/
10. Open, Modular, Architecture Controls (OMAC) Users Group HMI Working Group, http://www.arcweb.com/omac/
11. Open, Modular, Architecture Controls (OMAC) Users Group API Working Group, http://www.isd.mel.nist.gov/projects/omacapi/
12. 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
13. Voss, G., "What is a Java Bean," http://developer.java.sun.com/developer/onlineTraining/Beans/Beans1/simple-definition.html