Reconfigurable Machine Controllers using the OMAC API

Sushil Birla[a], David Faulkner[b], John Michaloski[c],
Steve Sorenson[b], George Weinert[d], and Jerry Yen[e]

[a]General Motors Corporation, North American Operations, Warren, MI
[b]Cimetrix, Inc. Midvale, UT
[c]National Institute of Standards and Technology, Gaithersburg, MD
[d]Lawrence Livermore National Laboratories, Livermore, CA
[e]General Motors, Powertrain Headquarters, Pontiac, MI

ABSTRACT

With enterprises facing tremendous time-to-market pressures, manufacturing systems must be implemented quickly and modified easily. The ability of the open architecture approach to reconfigure or extend existing machine controllers to meet new needs is one of its advantages in meeting these challenges. Another open architecture advantage is the ability to reuse existing designs or components. However, without open architecture standards, design and component reuse and reconfiguration is economically unfeasible in creating and maintaining machine controllers. This paper will review the Open Modular Architecture Controller (OMAC) Application Programming Interface (API) Workgroup specification that allows systems integrators and end users to extend, modify or upgrade controller components or modules used in developing, assembling, and reconfiguring machine controllers. The paper will examine the OMAC API component-based approach and the various categories of reuse and reconfiguration that the OMAC API supports.

Keywords: open architecture, component, module, reconfigurable, control, standard, machine

1. BACKGROUND

To be agile, control systems need to offer the necessary flexibility in order to handle reoccurring changes in product design and fluctuations in product mixes and volumes without major replacement of equipment. The idea of building control systems through open architecture technology offers advantages in developing, adapting, and reconfiguring factory automation systems for meeting these agile manufacturing challenges. The reconfiguration of open architecture factory automation allows control modules to be added or removed from the control system and provides appropriate control capability to match application needs. This paper looks at the OMAC API component-based open architecture approach to building and reconfiguring control systems.

The foundation for the open controller effort within the discrete-parts manufacturing industry originated with the Open Modular Architecture Controller (OMAC) white paper, entitled Requirements of Open, Modular Architecture Controllers for Applications in the Automotive Industry, Version 1.0 dated August 15, 1994, that was published by the U.S. automotive companies [3]. A great deal of interest from industry in the open, modular

Commercial equipment and materials are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

architecture technologies led to the formation of the Open Modular Architecture Controllers (OMAC) Users Group as a forum to advance the state of open controller technology. As part of the OMAC forum, an effort was undertaken to define an open, modular architecture controller specification based on application programming interfaces (API). This work was done under the auspices of the OMAC API Workgroup, of which the authors are members, and has lead to the development of a component-based specification for machine controllers.

The goal of the OMAC API specification is to enable control vendors to supply standard components that machine suppliers can easily configure and integrate to build machine control systems, and that end-users can then easily reconfigure based on evolving manufacturing requirements. The OMAC API technology vision provides for easily customized plug-and-play control components to reduce cost and provide higher fidelity while leveraging pervasive, off-the-shelf, high-volume, software technology. This vision means that building a control system by writing code would be replaced with building a control system by assembling and integrating existing software modules or components. Likewise, the notion of modifying a controller by rewriting code will be replaced with reconfiguring or replacing modules within the control system. In this component-based vision, a system integrator selects an appropriate set of components, configures the components through a combination of plugging in components into pre-wired slots and/or connecting interface-compatible components together [1]. To make such a component-based arrangement work, the OMAC API was required to define a formal integration API to establish the relationships and collaborations between components. The OMAC API also embraced the trend towards introspective components, or components that contain information about themselves found previously on bookshelves. This introspective ability allows components to be used at design time in an Integrated Development Environment (IDE) as well as runtime.

The OMAC API differs from other industry open-architecture standardization efforts in its component-based approach. Open System Architecture for Controls within Automation Systems (OSACA) is a message-based open-architecture effort that has developed a reference architecture and communication scheme directed at equipment controllers [11]. The Open System Environment Consortium (OSEC) has a message-based, open-architecture specification also, which is based on the Factory Automation Equipment Description Language (FADL) for exchanging messages in a distributed, agent-based, networked environment [12]. These efforts use a message-based approach, which is suitable for system-to-system asynchronous communication, but is not as conducive to smaller component reuse or reconfiguration [6].

This paper will review the OMAC API specification as it relates to developing and reconfiguring component-based systems using different reuse and reconfiguration strategies. The paper will start with an overview of the OMAC API specification. The paper will continue by examining component extensibility and reusability, system integration, and then reprogrammable tasks. Finally, the paper summarizes the approaches with discussion on the issue of controller reconfiguration using mainstream, high-volume, reusable component technology.

2. OMAC API OVERVIEW

To accommodate changes within a manufacturing process, it may be necessary to increase or decrease the functionality of the control systems associated with the equipment in the process. At such times, a module of a control system needs to be replaced to provide additional capability and/or lower cost to the user of the control system. Modularity, which refers to encapsulating common functionality into a separate entity, allows a user to replace a component or a module in a control system easily without having to devote a great amount of engineering effort to re-integrate the control system before it is functional. In this scenario, end users plug in the new module and start to play with minimal effort such as simply loading a new software driver. Plug and Play in this sense means to integrate and validate in a standardized environment.

The OMAC API adopted an object-oriented approach to plug-and-play modularization, using interface classes to define the Application Programming Interface (API). However, plug-and-play on a per-class basis is not practical. Instead, a coarser granularity is necessary that resulted in the OMAC API defining different sizes and types of reusable plug-and-play entities. To differentiate, the OMAC API uses different terms to distinguish them — component, module, and task — with each based on a Finite State Machine (FSM) so that collaboration is performed in a known manner. The term component applies to a reusable piece of software that serves as a building block within an application (analogous to a hardware part), while the term module refers to a grouping of components (analogous to a hardware assembly). Modules can themselves be then used as components, so that both terms are used interchangeably within this paper. A component may contain multiple interfaces, either through aggregation or through inheritance. A module acts as a container of components by providing a means of storing component references and then providing access to the component references. In general, new OMAC API component interfaces may extend functionality of existing components by means of aggregation or specialization. Further, new components can aggregate or inherit from one or more interfaces.

Figure 1 sketches the relationship between an application control system, modules and components as defined within the OMAC API specification. An application is a computer control system built by assembling and integrating software components. 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 one would find in the packaging industry or on a transfer line. The application can be distributed with parts of it running on one or more platforms, so that one master, the System Coordinator, is responsible for bringing up all the distributed modules on each controller platform, the Domain Coordinator, for each platform.

The OMAC API specification defines a series of modules as shown in Figure 1, including Axis, Axis Group, Task Coordinator, and Discrete Logic. The Axis module is responsible for servo control of axis motion, transforming incoming motion setpoints into setpoints for the corresponding actuators. The Axis Group module is responsible for coordinating the

motions of individual axes, transforming an incoming motion segment specification into a sequence of equi-time-spaced setpoints for the coordinated axes. The Task Coordinator module is responsible for sequencing operations and coordinating the various modules in the system based on programmable Tasks. The Task Coordinator can be considered the highest level Finite State Machine in the controller. The Input-Output (IO) Device module is responsible for managing communication between the physical hardware device and IO software, and is responsible for managing a group of IO points. The Human Machine Interface (or HMI) module is responsible for human interaction with a controller including presenting data, handling commands, and monitoring events. The Discrete Logic module serves as a general-purpose mechanism to organize and present IO control software and allows discrete logic and IO programming using Tasks.

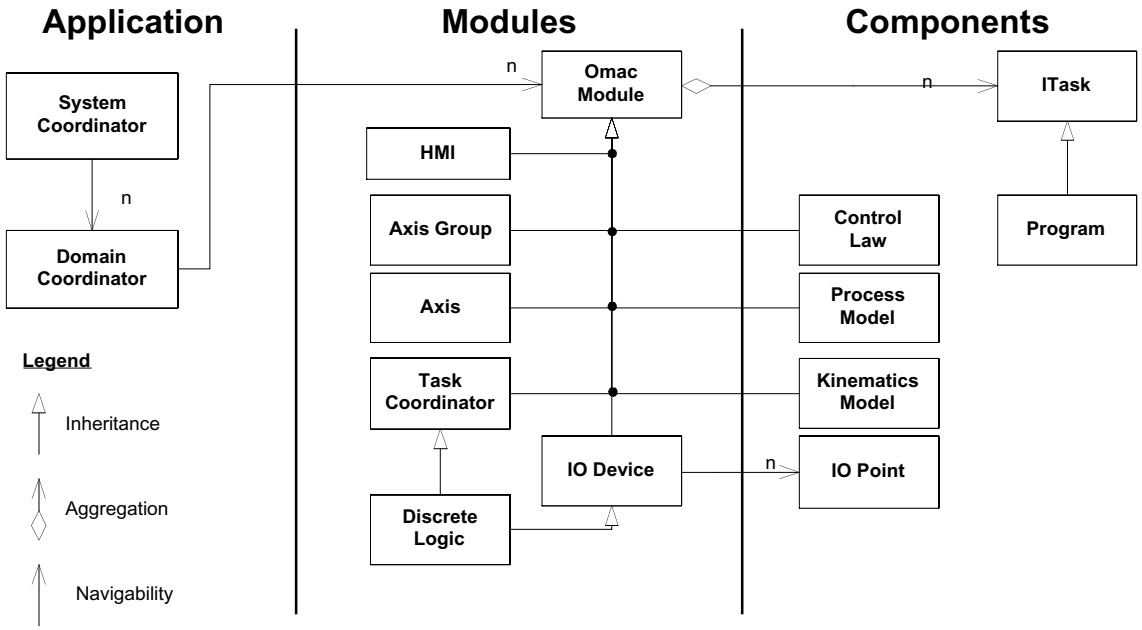

Figure 1. Relationship of OMAC Application, Modules and Components

The OMAC API specification defines components as shown in Figure 1 that include Control Law, IO Point, Kinematics, Process Model, and Task. The Control Law component is responsible for servo control loop calculations to reach specified setpoints. The IO Point component is responsible for the reading of input devices and writing of output devices through a type-based read/write interface. Logically related IO points are clustered within an IO Device module or within a Discrete Logic module. The Kinematics Model component is responsible for geometrical properties of motion. A Process Model component contains a dynamic data model that is integrated with control functionality.

Tasks are a type of component that encapsulates functional behavior characterized as lighter-weight than an OMAC module, contains a series of steps, runs to completion and may be run multiple times while a controller is running. The Task interface defines a FSM model that has functionality in support of the Task life cycle, including methods for starting, stopping, restarting, halting, and resuming a Task. The Program component manages a

series of Tasks, with ability to restart and navigate. Tasks serve a variety of roles in the realm of a controller and a later section will cover the differences in functionality.

3. SYSTEM AND COMPONENT RECONFIGURABILITY

To integrate components, a framework is necessary. Frameworks formalize the life cycle, collaborations and other aspects of the manner in which components operate. Example frameworks include Microsoft s Component Object Model (COM) [7] or the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) [10]. The potential productivity gains for end-users and Original Equipment Manufacturers (OEM) from component-based technology using a common framework is obvious, but can also work to the advantage of control vendors. With a common framework in which to develop components, control vendors can concentrate on application-specific improvements that define their strategic market-share — as opposed to spending valuable programming resources reinventing and maintaining software plumbing. With the shortage of skilled software professionals continuing to rise, the urgency to leverage commodity framework technology is even more compelling. It was the OMAC API workgroup's desire to produce an API specification that is platform and framework independent using a specification language such as the Unified Modeling Language (UML) [13]. However, to hasten the development and validation of the API, Microsoft COM was selected as an initial development framework. The OMAC API has produced a set of reference documentation to define the API based the Microsoft Interface Definition Language (MIDL) [9].

Historically, the difficulty in components integration has been a barrier to realizing code reuse on a large scale. With the difficulty encountered in integration, each new system often becomes as large an effort as if it had been created from scratch. To alleviate this problem, the OMAC API has defined a formal integration process — beyond that found in the Microsoft COM framework — in order to establish the relationships and collaborations necessary for components in control system applications. The OMAC API has defined a component-level integration so that components understand how to collaborate, how to advertise functionality, where and under what framework they operate. The OMAC API integration process defines an IOmac interface, which describes base services for component integration and deployment in a consistent manner [9]. Using UML notation, Figure 2 illustrates the high-level OMAC API component model common to all components and includes:

- **Functionality** interface — defines the control capability of a component. This interface includes methods for behavior, state, and parameter manipulation.
- **Infrastructure** interface — provides support for the 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.
- **Connection** interface — advertises the module dependencies such as the interfaces it requires and enumerating the Events-in and Events-out it supports, thereby answering the question of what a module needs.
- **Attributes** — parameterize the features of a component that can be customized.

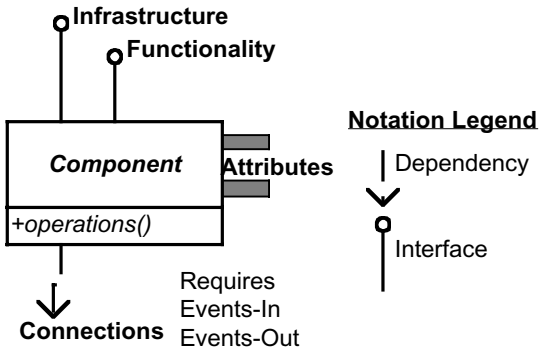

Figure 2. Omac Component Overview

The potential to reuse software is indisputable. A study by Tracz found 60-to-70% commonality has been found when comparing similar software, which includes code, design, functional and architectural similarities [13]. However, component-based modularity alone does not guarantee reusability. Modularity must be combined with feature customization to make it possible for engineers to tailor the modules to meet the specific needs of individual applications. Thus, to attain maximal potential for reconfiguration, the OMAC API designers realized that modules and components must allow as much customization as possible in as flexible a manner as possible to provide a broader basis for the reusable components to exploit cross-industry purchasing power.

The OMAC component set of interfaces can be exploited in numerous ways in building and reconfiguring control applications. Potential approaches include the ability for composition (Functionality and Connections), loose coupling using events —publishes (events-out), emitters (connection points), consumers (events-in), and configuration (attributes). These interfaces also provide 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. As part of the OMAC module base services, a connection API is defined that allows components to advertise what other components they require and assist in resolving component dependencies.

To further streamline the ability for system reconfiguration, the OMAC API also provided support for the growing trend in software towards embedding information in a component about itself. With embedded information, a component has the ability to be used at design time in a drag-and-drop Integrated Developer Environment (IDE), such as Visual Basic, or other visual programming tool. For example, with embedded information, a component can be queried locally on the shop floor about its last maintenance. Embedding information also enables component introspection that allows users to determine what the component can do and how to customize the component. With introspection, components can be easily customized in IDE builder tools so that users can graphically drag and drop components onto a component container (possibly a Visual Basic form or an Axis module container), and customize the properties of the components. The OMAC API workgroup foresees the

ability of IDE builder tool to query an OMAC component for the functional properties, the types of components it requires, and the events-in it requires and the events-out it generates. To that end, the controller construction process can be greatly simplified through IDE configuration tools so that complete systems can be integrated graphically without any source code programming required.

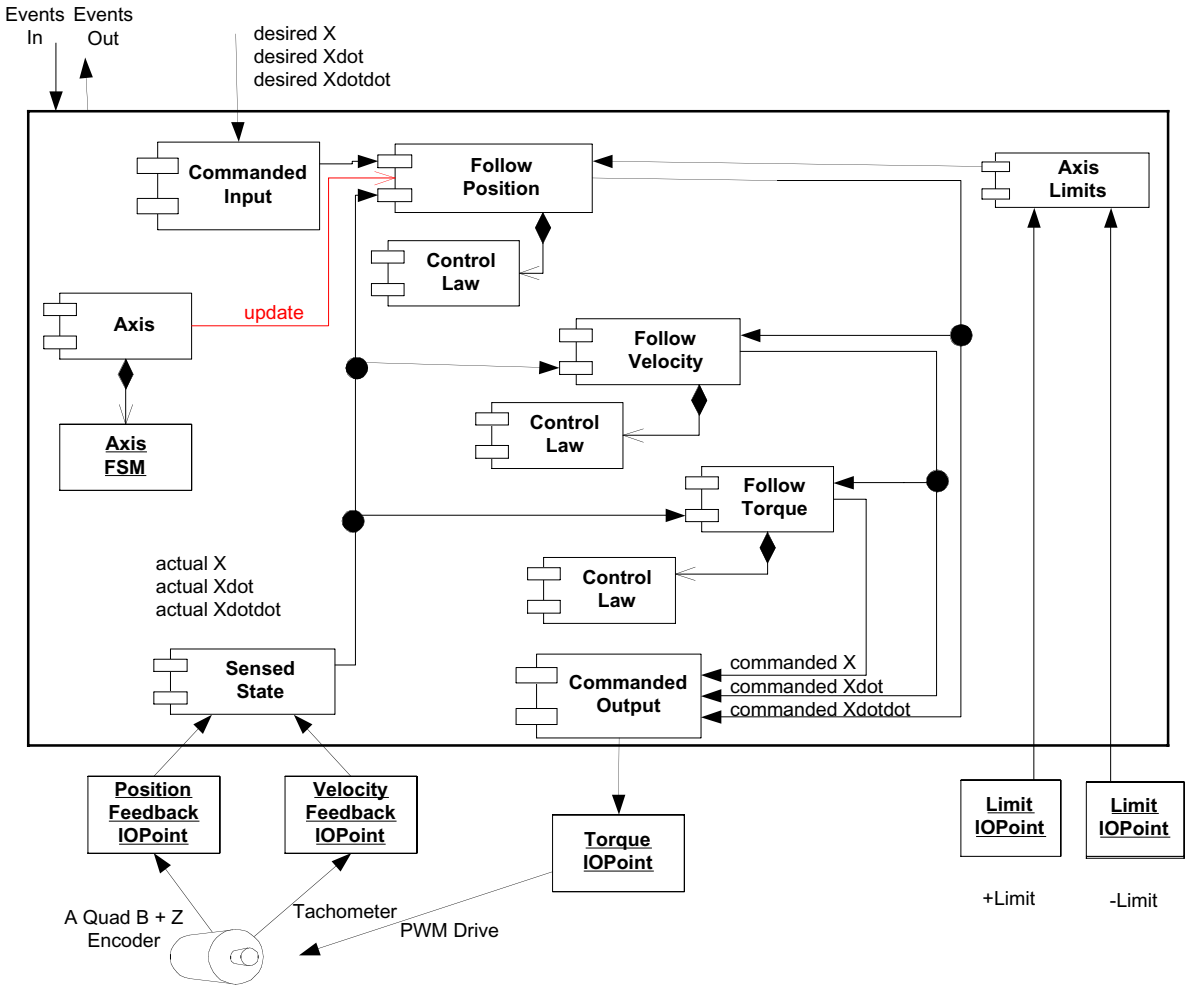

Figure 3. Example Axis Module with Component Plugs

Another step towards maximizing reconfigurability in the OMAC API is through component plugs . Figure 3 shows the relationship of an example Axis module, the related plugs including coordination, (such as CommandedInput, CommandedOutput, SensedState), servo plugs (such as FollowPosition), and the external components such as IO points (such as Axis Limit switches). In Figure 3, the axis is wired to IO components including a motor drive, an encoder with marker pulse and IO switches for home and axis limits. Axis modules can be customized to accommodate need, for example, handling a spindle through a single Follow Velocity plug, but the concept of plugs remains consistent.

Plugs can be customized to meet specific controller requirements. When functional add-ons or changes to a control system are required, such as sensors, communications, diagnostics,

etc., the most appropriate technologies can be selected and integrated without relying on specific control vendors to develop custom solutions. In the case of the Axis module as illustrated in Figure 3, the module contains pre-wired component plugs for Commanded Input, Commanded Output, Sensed State, etc. The appropriate communication plugs can be inserted via set methods accessible through the Axis module, which acts as a container for these plugs. Consider the Command Output plug: this plug could be replaced by one that communicates over a Serial Realtime Communication System (SERCOS) network or a plug that communicates over Controller Area Network (CAN) bus.

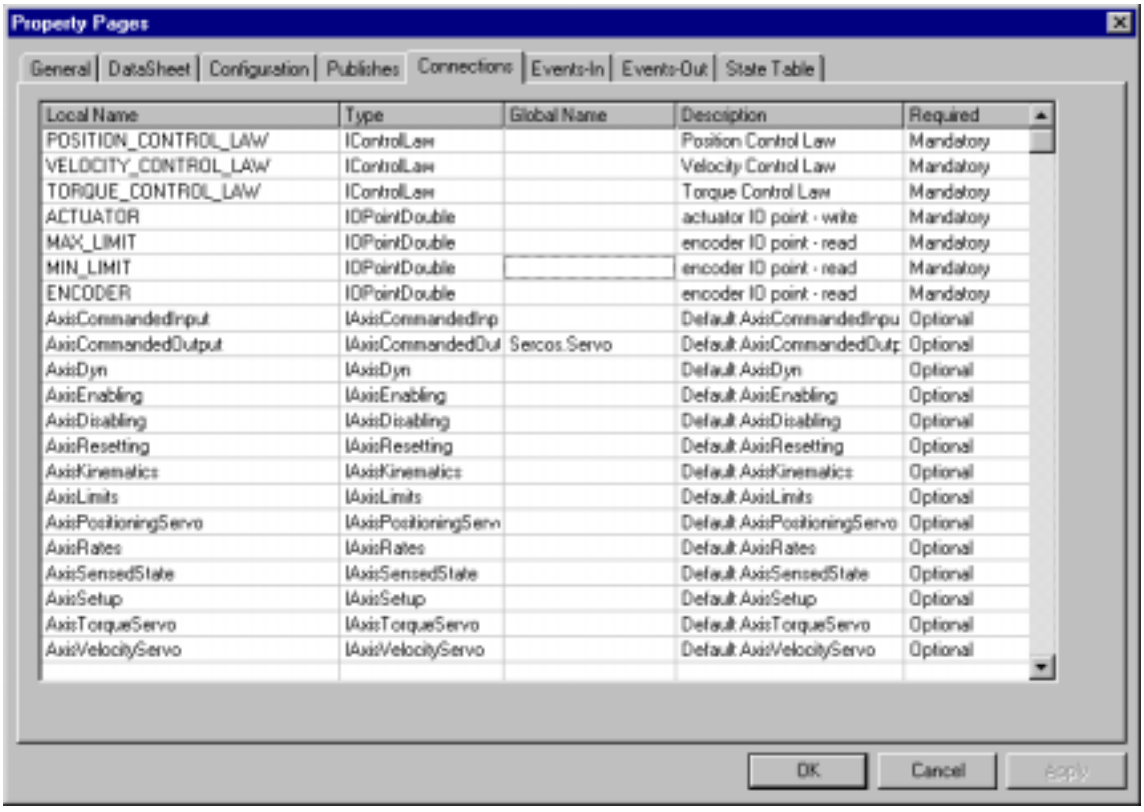

Figure 4 Axis Module Connection Property Page.

As an alternative to rewriting source code to change a set plug method parameter, component plugs can be changed by asking the component what component plugs or connections it needs, and assigning a component to the connection. In this case, components and modules advertise what other component connections they need and a system integrator picks and chooses an appropriate fitting plug and connect the components together. Figure 4 illustrates a property page for an Axis module under a Visual Basic IDE. Within the Connections property page, a connection is made using a Sercos.Servo plug so that the Axis module Commanded output will communicate over a SERCOS network to the axis servo drive. Pluggable deployment does not place any extra programming burden on the end-users. Control vendors would be expected to provide a default set of plugs, so that users would customize only if the need arises. This is seen by the Connections Property Page Required column that indicates whether the connection is mandatory or optional, so that the axis module uses an internal plug.

4. REPROGRAMMABLE TASKS

The OMAC API defines the Task API to allow programmable control components. To allow component collaboration in a known manner, Tasks follow a Finite State Machine (FSM) logic defined by the ITask interface corresponding to start-running-completed-restart program behavior. Tasks are also component plugs because they offer a mechanism to customize the system for specific application needs. However, Task are component plugs that implement the ITask interface, while heavier-weight OMAC modules follow and extend the FSM state logic defined by the IOmac interface, which defines a more complete component lifecycle API. IOmac compliant components would operate continuously from the moment the controller is brought up until the controller is shutdown. ITask compliant components would run from start to completion any number of times while the controller is running.

An example of a Task is the homing of an Axis. Since homing can vary greatly based on the underlying hardware, this control logic functionality is best encapsulated into a Homing Task component plug. Figure 5 uses UML notation to illustrate how the OMAC task model is used to handle Axis Homing configurations. Case (a) illustrates an automatic detection of crossing the homing switch and capturing the position. Case (b) illustrates a very low-cost machine, where the axis switch is wired into a single digital input bit on the controller and any overtravel limit causes an abort. The operator manually cranks the axes and is responsible for monitoring when the switch trips and then backing the machine off.

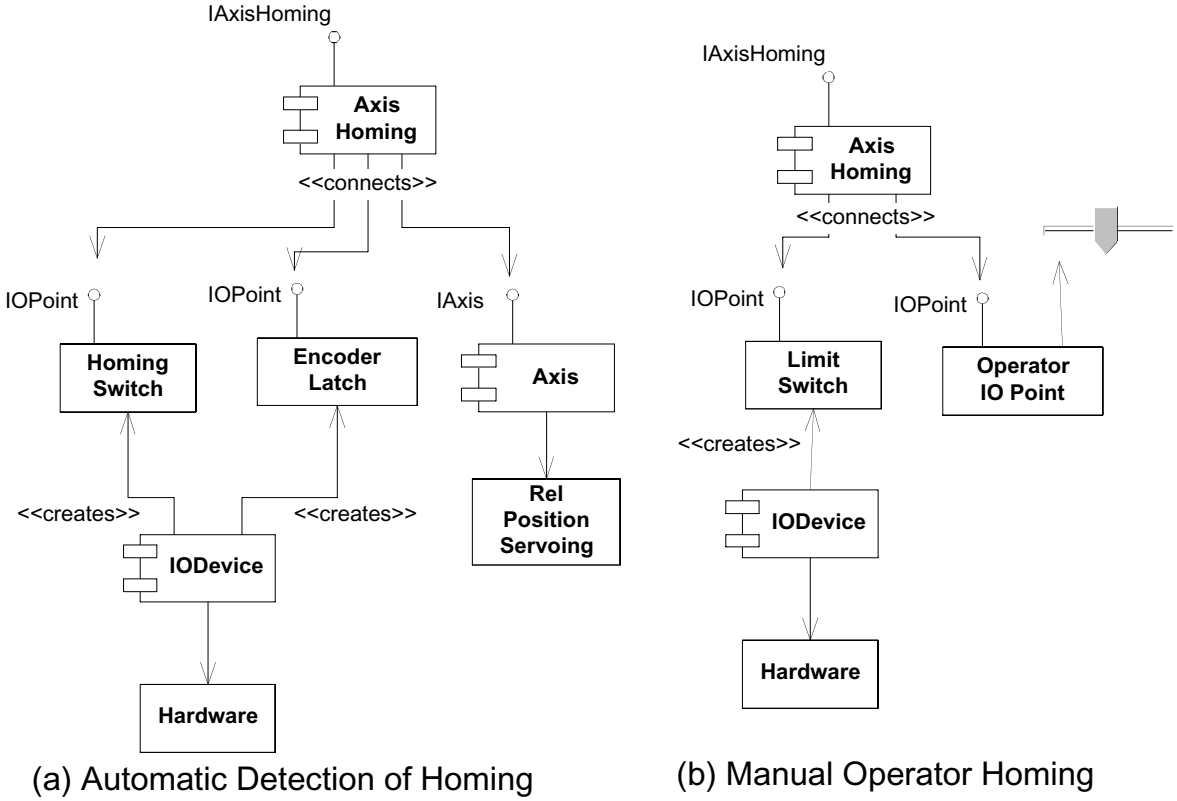

Figure 5 Axis Homing Task Examples

To handle these different hardware limit switch configurations and position capture, the Axis Homing is a plug that aggregates an IConnectionTable interface in support of˚design and runtime connection to IO Points it needs in performing the homing process. As part of this procedure, the hardware setup for the Axis Homing plug could be either hard-coded through source code or configured with an ini file.˚Then the Axis Homing plug advertises the names and˚types of IO it requires, including homing or limit switches, latches, encoder pulse count, and possibly even an operator IO point. Then, the Axis Homing plug is˚connected to these IO Points and to the controlling object - either an Axis Group or Axis Supervisor depending on the type of machine. In some cases where hardware latching is not available and software sampling must be done, the Axis Homing plug could contain direct access to the hardware for better performance.

Tasks that support both the IOmac and the ITask interfaces are Resident Tasks that operate as a permanent part of the controller. Resident tasks execute periodically "forever" or on an event-driven basis. Resident Tasks are independent of any OMAC module, and depend on a scheduler or an arrival of an event to carry out execution. Estop is an example of Resident Task since it maps an Estop event into several events that can be tailored to accommodate domain-specific safety codes. For example, Estop could be mapped into stop as fast on each limit using the software limits as configured by OEM, put brakes on, cut power , or cut power after certain time limit depending on pertinent safety codes. To allow for the different domain-specific forms of Estop, this functionality has been abstracted into a Resident Task and would use either a hard or abnormal stop to a achieve safe emergency stop as dictated by the domain-specific safety code.

In manufacturing, a series of Tasks corresponds to a process plan, which coordinates actions in a production operation to control one or more devices (i.e., machines, tools, fixtures, etc.) and allows "re-programming" a controller. An RS274 part program is an example of a process plan for milling a part. Process plans are handled in the OMAC API by translating them into a series of process-oriented Transient Tasks . In the OMAC API model, each process plan step is translated into a sequence of Execution Steps as defined by the IExecutionStep interface, which is a specialization of the ITask interface. The relationship between Transient Tasks and Execution Steps corresponds to distinction between a class factory and an instance of the class. In this case, each Execution Step is a parameterized clone of a Transient Task that has been pre-wired to resolve connection dependencies to components, modules and other Tasks. IProgram is an interface that has been defined to provide programming control (e.g., start, stop, single step, rewind) that is made up a sequence of Execution Steps.

The OMAC API provides a model for translating Process Plans into Execution Steps in a device independent manner. All potential process plan steps have a corresponding Execution Step. A Transient Task factory is responsible for generating an Execution Step. A system integrator rewrites a Transient Task depending on the equipment and facilities of the machine controller. For example, consider the case of handling˚a tool change cycle in a CNC Machining Center.˚ A Tool Change may include orientating motion of the spindle. It may also include motion of the axes to a fixed tool-change position, as performed by the Axis Group Transient Motion Tasks (e.g., Straight Line motion segments). The implementation

of the Tool Change Transient Task depends on the˚underlying hardware and devices associated with the Tool Changing operation. For the case of a Machining Center that has a Tool Changer with a Tool Magazine, Collet/Chuck, etc., a more permanent Tool Changer Discrete Logic module would be used to coordinate the activity. For the case of a Machining Center˚that is equipped with no moving (e.g., chain-driven) tool magazine or specialized tool changing device, the Tool Change functionality would have to be realized as a Resident Task rather than a Component/Module, since there is no corresponding actual device.

5. SUMMARY

With the pressure on manufacturing systems to adjust to the fast changing demands from the marketplace, the time required to design and integrate a control system must greatly improve in the near future. The open architecture ability to reconfigure or extend existing equipment to meet new needs is particularly powerful in meeting these challenges. However, realization of the open architecture concept demands more than the selection of control modules and the definition of standard interfaces. It also requires components conforming to the standard interfaces, software tools to assist the users to integrate commercial components, and support from control component suppliers and system integrators for implementations. There exist several controller IDE products on the market today, (e.g., [4, 5]), that achieve many of the desired goals of the OMAC API prescribed plug-and-play component-based technology. Unfortunately, these products are proprietary and the resulting components are not operable in other IDE environments. Without high-volume interchangeable component technology, it is economically unfeasible for a manufacturing industry to create and maintain components in an IDE [1].

The OMAC API specification is a draft proposal for a standard component-based, plug-and-play, controller technology that leverages pervasive, off-the-shelf, high-volume, component strategy. Currently, validations and commercial product development based on the proposal are underway. This paper presented an overview of the OMAC API plug-and-play capabilities for machine controller development and reconfiguration. The OMAC API component-based specification encourages reuse and reconfiguration by providing numerous mechanisms for plugging, rewiring, and tweaking components. Of note is the fact that OMAC application systems need not be monolithic, but rather a system could be pieced together that merely consisted of a HMI and an Axis. This scaleable nature of the OMAC API allows systems to be built and tested incrementally. More importantly, OMAC API based systems can be reconfigured to fit memory-constrained applications or other deployment constraints. Solution scalability is essential for achieving the commodity-valuations necessary if component-based controller technology is to succeed.

6. REFERENCES

1. S. Birla, H. Egdorf, R. Igou, J. Michaloski, D. Sweeney, D. Uchida, and G. Weinert, J. Yen, An Open Architecture Model of System Development, ASME International Mechanical Engineering Conference and Exposition, Atlanta, GA, Nov.17-22, 1996.

2. S. Birla, J. Yen, J. Skerie and D. Berger, Requirements for Global Commonization, Control Engineering Online Extra, http://www.manufacturing.net, January 1999.
3. Chrysler, Ford Motor Co., and General Motors, Requirements of Open, Modular, Architecture Controllers for Applications in the Automotive Industry, White Paper, Version 1.1, Dec. 1997.
4. ControlShell, Real-Time Innovations, Inc. http://www.rti.com/
5. LabVIEW, National Instruments, Inc., http://www.ni.com
6. J. Michaloski, S. Birla, R. Igou, G. Weinert, and J. Yen,˚˚ An Open System Framework for Component-Based CNC Machines, ACM Computing Survey Symposium on Object-Oriented Application Frameworks, March 2000.
7. Microsoft Corporation, Microsoft Interface Definition Language (MIDL), http://msdn.microsoft.com
8. Microsoft Corporation, COM Specification, http://www.microsoft.com/com
9. OMAC API Reference Documentation, http://www.isd.mel.nist.gov/info/omacapi/ReferneceDocumentation.
10. OMG, CORBA Basics, http://www.omg.org/gettingstarted/corbafaq.htm.
11. OSACA, Open System Architecture for Controls within Automation Systems. See Web URL: http://www.osaca.org.
12. OSEC, Open System Environment Consortium. See Web URL: http://www.sml.co.jp/OSEC.
13. Rational, Unified Modeling Language, http://www.rational.com/uml Rational Software Corporation, 2000.
14. W. Tracz, ed., Software Reuse: Emerging technology, IEEE Press, New York, 1988.