Published in Proceedings of SPIE International Symposium on Intelligent Systems and Advanced Manufacturing Vol. 4191,
Boston, MA, November 6-7, 2000

Analysis of Behavioral Requirements for Component-Based Machine Controllers

Frederick Proctor[a], John Michaloski[a], Sushil Birla[b], and George Weinert[c]

[a]National Institute of Standards and Technology, MS 8230, Gaithersburg, MD
[b] General Motors Corporation, North American Operations, Warren, MI
[c]Lawrence Livermore National Laboratories, P.O Box 808, Livermore, CA

ABSTRACT

Machine controllers built from standardized software components have the greatest potential to reap open architecture benefits – including plug-and-play, reusability and extensibility. A challenge to component-based controllers relates to standardizing behavior in a non-restrictive manner to accommodate component packaging and component integration. Control component packaging requires behavior to be dependable, well-defined, and well-understood among a variety of users to help ensure the reusability of the component, the reliability of the component, and the correctness of the system built using the component. Integration of control components requires that the behavior model is consistent not just within a single component, but across all components in a system so that the components interoperate correctly. At the same time, the component behavioral model must be reasonably flexible to accommodate all behavioral situations and not be restrictive to a single programming methodology. Further, not all the behavior in the system may be pre-packaged as part of a component. Thus, another issue is the suitability of the standard behavior model for programming and integration of new control logic. Ideally, we need a vendor-neutral, tool-neutral, controller-neutral behavior model to allow the export/import of any and all types of control logic programs. This paper will analyze the requirements of component-based, machine controller behavior, then offer a refinement of a Finite State Machine as the basis of a behavior model to satisfy these requirements. Examples will be presented based on the behavioral model the efforts of the Open, Modular, Architecture Controller (OMAC) User's Group Application Programming Interface (API) for standardized, interchangeable machine controller components.

Keywords: Finite State Machine, component, module, control, standard, modularity, machine

1. BACKGROUND

Historically, computing in manufacturing operations has lagged behind mainstream hardware and software technology. The physical constraints of factory floor operation, including harsh environment, continual operation, and strict safety requirements, has forced manufacturers to rely on controls vendors and systems integrators to provide control systems based on special-purpose hardware and software. Unfortunately, such factory automation is costly to support and cannot easily adapt to meet new manufacturing demands or technology innovations. Manufacturers realize the shortcomings of proprietary solutions and are moving toward open solutions using commodity, standards-based computer products in machine control. Machine controllers used in factory automation built with this open, standards-based premise have been termed open architecture controllers.[1,2] The open architecture strategy is part of an effort to improve the agility of manufacturing by providing the highest level of flexibility for integration of manufacturing control elements as well as for integrating the shop floor directly into the enterprise business systems. Organizations that can adapt to open architecture practices could experience potential benefits that include improved time to market, increased production yield, increased production capacity, reduced inventory, and optimization of production workforce.

Open architecture systems differ widely in the degree to which they are "open" ranging from systems built from commodity hardware and open-source software to systems consisting primarily of proprietary components with exposed interfaces. However, in order to maximize benefits, open-architecture systems should be component-based with interfaces and interaction standardized; otherwise, interoperability and plug-and-play are unfeasible. Component-based technology stresses

the desire for independent pieces of software that can be reused and combined in different ways to implement complex software systems. Component-based technology offers to make it easier and more cost-effective to design, develop, integrate, deploy and reconfigure open-architecture controllers and applications. In this regard, the seminal foundation for the open controller effort in the discrete-parts manufacturing industry is based on the "Open Modular Architecture Controller" (OMAC) requirements document.[3] This document lead to the formation of the OMAC Users Group, which is an industry forum to advance the state of modular open architecture controller technology. As part of the OMAC forum, an effort was undertaken to define an open 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 the "OMAC API" specification for machine controllers.[4]

The objective of the OMAC API is to "enable control vendors to supply standard components that machine suppliers configure into machine control systems. The integrated control system and machine are then delivered to the end-user." The OMAC API application domain ranges from multi-axis, coordinated motion control typical of Computer Numerical Control (CNC) to single axis controllers one would find in the packaging industry or as part of a transfer line. The OMAC API adopted an object-oriented approach for the specification where interface classes define the API. This approach contrasts to other industry standards efforts[5,6] which are message-based. The OMAC API uses a paradigm in which systems are built primarily of components, which offers the opportunity to lower costs by sharing the components across a broader market, thus amortizing them over a larger population, while taking advantage of the investments that industry is putting into the development of new technologies.

The OMAC API defines a component as a reusable piece of software that serves as a building block within an application. To be useful, component interfaces must be well-defined so that they are flexible enough to solve a general class of problem, and at the same time robust. Example components defined in the OMAC API specification include: IO Point, Control Plan, Kinematics, and Control Law. Interfaces define component functionality. A component may contain multiple interfaces, through either aggregation or inheritance. New components may extend functionality by means of aggregation or specialization. A module is also a type of component, but 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. A summary of the OMAC API components and modules with a brief description is given in Table 1.

Table 1. OMAC Module and Component Interfaces

Axis	Modules responsible for servo control of axis motion, transforming incoming motion setpoints into setpoints for the corresponding actuators.
Axis Group	Modules 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.
Omac	Serves as the base class that provides a uniform API base class for an OMAC module. The OMAC base class defines a state model and methods for start-up and shutdown. The OMAC Base Class defines a uniform name and type declaration and provides an error-logging interface. The OMAC Base Class maintains a global directory service for name lookup and reference binding.
Control Law	Components responsible for servo control loop calculations to reach specified setpoints.
Connection	Interface used to resolve component dependencies to other component, including interface references, events-in and events-out.
Discrete Logic	Modules responsible for implementing discrete control logic or rules that can be characterized by a Boolean function from input and internal state variables to output an internal state variables.
Finite State Machine	Finite State Machine (FSM) is a component interface to handle state machine event processing logic
Human Machine Interface (or HMI)	Modules responsible for human interaction with a controller including presenting data, handling commands, and monitoring events. Defining a presentation style (e.g., GUI look and feel, or pendant keyboard) is not part of OMAC API effort.
I/O Device	Modules responsible for managing communication between the physical hardware device and IO software.
I/O Points	Components responsible for the reading of input devices and writing of output devices through a generic read/write interface. Logically related IO are be clustered within an IODevice module or under a Discrete Logic module.
Kinematics Model	Component responsible for geometrical properties of motion.
Program	Components that manage a series of linked Tasks, with ability to restart and navigate.
Task Coordinator	Modules are responsible for sequencing operations and coordinating the various motion, sensing, and event-driven control processes. The task coordinator can be considered the highest level Finite State Machine in the controller.

Task	Component interface that derives from a FSM and adds functionality in support of the Task life cycle, including methods for starting, stopping, restarting, halting, and resuming a Task.
Task Generator	Modules responsible for translating application programs into Tasks.

Since an open-architecture standard must allow system integrators to mix and match components from different vendors, one of the greatest challenges is standardizing behavior in a non-restrictive manner to accommodate component packaging and component integration. Control component packaging requires behavior to be dependable, well-defined, and well-understood among a variety of users to help ensure the reusability of the component, the reliability of the component, and the correctness of the system built using the component. Integration of control components requires that the behavior model is consistent not just within a single component, but across all components in a system so that the components interoperate correctly. At the same time, the component behavioral model must be reasonably flexible to accommodate all behavioral situations and not be restrictive to a single programming methodology. Further, not all the behavior in the system may be pre-packaged as part of a component, so that the components must be extensible.

This paper will review the behavior and programming requirements of machine controller components and review how the OMAC API effort addressed these requirements. First, the paper will examine the control logic requirements needed in machine control. The paper will describe the Finite State Machine as the basis of a behavior model of machine controller and then show how it was generalized to handle a wide range of control logic needs. Next, the paper will review the use of OMAC API Task abstraction and how it is used for programming controller components as well as handling process plans. Throughout the paper examples will illustrate the application of the behavioral model as it relates to the efforts of the Open, Modular, Architecture Controller (OMAC) User's Group Application Programming Interface (API) workgroup's effort to specify API for standardized, interchangeable machine controller components.

2. PROGRAMMABLE COMPONENT MODEL

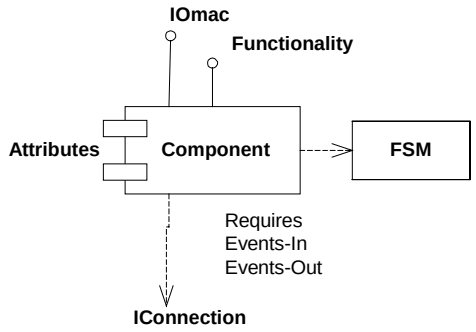
In machine automation, the control logic specifies the actions a machine controller can perform, that in turn define the controller behavior. In specifying the potential actions, the control logic provides the means by which certain actions are taken in response to different situations. Control logic is usually encoded as conditional instructions that specify alternative routes through a program, depending on conditions as evaluated at execution time. It is the control logic that gives controllers the semblance of intelligence. The more encompassing the control logic, the more "intelligent" the controller appears. The problem domain of which we are interested includes sensor-based controllers, typical of CNC machines or robots, as applied to manufacturing applications. In this realm, sensor feedback control systems fall under the domain of hybrid systems, in which digital and analogue devices and sensors interact over time. Issues of purely control system modeling concern include, but are not limited to, representation of states, uncertainty of state transitions, and the relationship to controllability in general.

Modeling the control logic of machine controllers is a well-studied problem. A survey of the literature finds numerous modeling approaches used in machine control logic programming, including State Machines, Hierarchical State Machines, Discrete Event Dynamic Systems (DEDS), Hybrid Systems modeling, Probabilistic Automata, and Petri nets and derivatives.[8,9,10] From a component-based perspective, these control logic models can be categorized as state-based and can be generalized as state machine models. By state-based we mean there is a dependence on their history. In contrast, stateless means there is no state history (or memory) as typified by a cosine function, which for example, always returns the same cosine independent of previous invocations. From a programming standpoint, 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. For components to properly use other components, a standard model of control logic behavior must exist. For the OMAC API, behavior is event-driven and the model abstraction is based on finite state machines (FSM). The OMAC API defines interface event methods to specify the service available to component clients, while hiding the internal event-handling and control logic implementation. Thus, the OMAC API FSM behavioral model is general enough to allow any state based control logic model to be used in the component implementation, which includes most control logic models.

Formal modeling of control logic is only a part of the software life cycle. The OMAC API has defined a formal integration process to establish the relationships between components for multiple aspects of the system life cycle. Historically, the difficulty in component integration has been a significant barrier to realizing code reuse on a large scale. With the difficulty encountered in integration, each new system often becomes as much a new invention as if it had been created from scratch. The OMAC API has tried to address this problem by defining a controller integration framework in order for components to understand how to collaborate, how to advertise functionality, where and under what framework they operate. The OMAC

API workgroup has spent considerable effort in defining an API to facilitate component integration in a controller environment.

As the basis, the OMAC module API describes services for component integration and deployment in a consistent manner. Using the Unified Modeling Language (UML) notation[7], Figure 1 illustrates the high-level OMAC API component model, that implements the **IOmac** and **IConnection** interfaces.



Functionality interface - defines the control capability of a component. This interface includes methods for behavior, state, and parameter manipulation.

IOmac interface - provides infrastructure 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.

IConnection 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 - parameterizes the features of a component that can be customized.

Figure 1. OMAC Deployable Component Model

An OMAC API component has a set of features, which applications can exploit in building control applications. These features include composition (Functionality and Connections); loose coupling using events (for publishing events-out, for emitting event via connection points, and consuming events-in); and configuration (through parametric attributes). These interfaces provides a uniform API for dealing with the common controller behavior, 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, and naming.

Implementing behavior as packaged components introduces programmability constraints, including reusabilty, extensibility, and flexibility. The OMAC API allows programmable control logic through the concept of a Task. Tasks like all OMAC components and modules adhere to a Finite State Machine (FSM) behavior model. In this case, programmed behavior is defined by the **ITask** interface corresponding to start-running-completed-restart program behavior. Tasks can also be used as components in order to customize the system for specific application needs. However, Task are components that implement the **ITask** interface, as opposed to the **IOmac** interface which defines a more complete component lifecycle API.

Axis Homing is an example of a component where control logic functionality should be encapsulated into a Task component since homing can vary greatly depending on the underlying hardware. Figure 2 uses Unified Modeling Language (UML) notation to illustrate how the OMAC task model is used to handle multiple Axis Homing configurations. To handle these different hardware limit switch configurations and position capture, the Axis Homing is a component that aggregates an **IConnection** 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 component is either hard-coded or configured. Then the Axis Homing component 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 component is connected to these IOPoints and to the controlling axis supervisor.

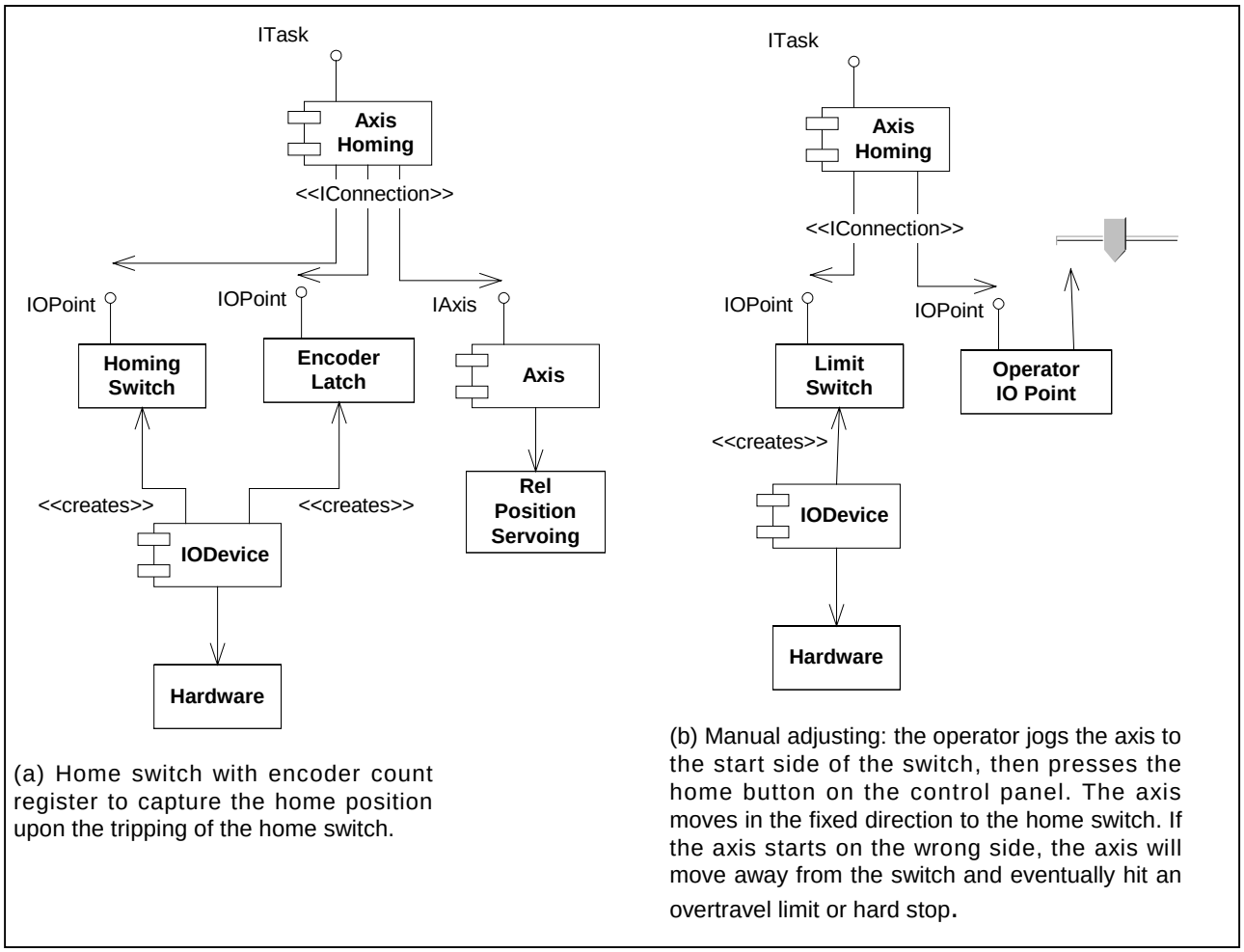


Figure 2. Axis Homing Task Examples

Tasks that support both the **IConnection** and the **ITask** interfaces are **Resident Tasks** that operate as a permanent part of the controller. Resident tasks execute periodically while active or on an event-driven basis. Resident Tasks 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 from an IO Push Button 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 can be abstracted into a Resident Task and programmed as either a hard or abnormal stop to achieve a safe emergency stop.

3. PROGRAMMABLE PROCESS BEHAVIOR

We have established that the behavior of an OMAC modules and components are based upon FSM, but the relationship of a module and FSM can vary. Modules can categorized as either Event-driven or Task-driven modules. Event-driven modules exclusively accept events as the means to determine and alter behavior, but can offer component plugs to customize behavior. Task-driven modules accept new and different Tasks, while still using event-service to sequence these Tasks.

3.1 Pluggable Event Handlers

Event-driven modules are restricted to only accepting events to alter behavior. Event-driven module examples include modules consisting of one Resident Task (e.g., IEC61131-3 Program to handle Tool Changes) or the Axis module, which only uses events to change control strategies. In the case of the Axis module, a client, such as an Axis Group, can switch between motion control strategies (e.g., follow position, stop, follow velocity, home, etc.) by sending events that adhere to the Axis module FSM. The Axis module FSM implementation itself is hidden, and the Axis interface only provides methods

for FSM event propagation and state query. There is no changing or modifying of the internal FSM for the module, but customizing by using different event-handler plugs is possible.

The OMAC API has defined modules that accept event-handler "plugs" or predefined **Resident Tasks** that implement the **IConnection** and the **ITask** interfaces as described previously. Plugs allow module customization in the OMAC API through the selection of different component Resident Task implementations. Figure 3 labeled "Axis Module and Component Plugs" shows the plugs defined for the Axis module. Plugs are customized to meet specific controller requirements. For different applications, designers select different plugs to be used at system integration. Earlier we showed how different Axis Homing Resident Task could be implemented, and now these can be used as alternative plugs in the Axis module at system integration. To perform homing, the Axis Module would receive a **startHoming** event that the Axis module FSM would process, which would then trigger the execution of the Homing task plug. As another example, consider the Command Output plug. This plug could be developed to communicate with a SERCOS network or to communicate with CAN bus depending on the application needs. This pluggable deployment does not place any extra programming burden on the end-users. Control vendors would provide a set of default plugs, so that users would customize only if the need arises. Reusability is provided by deploying the same plug in different controllers.

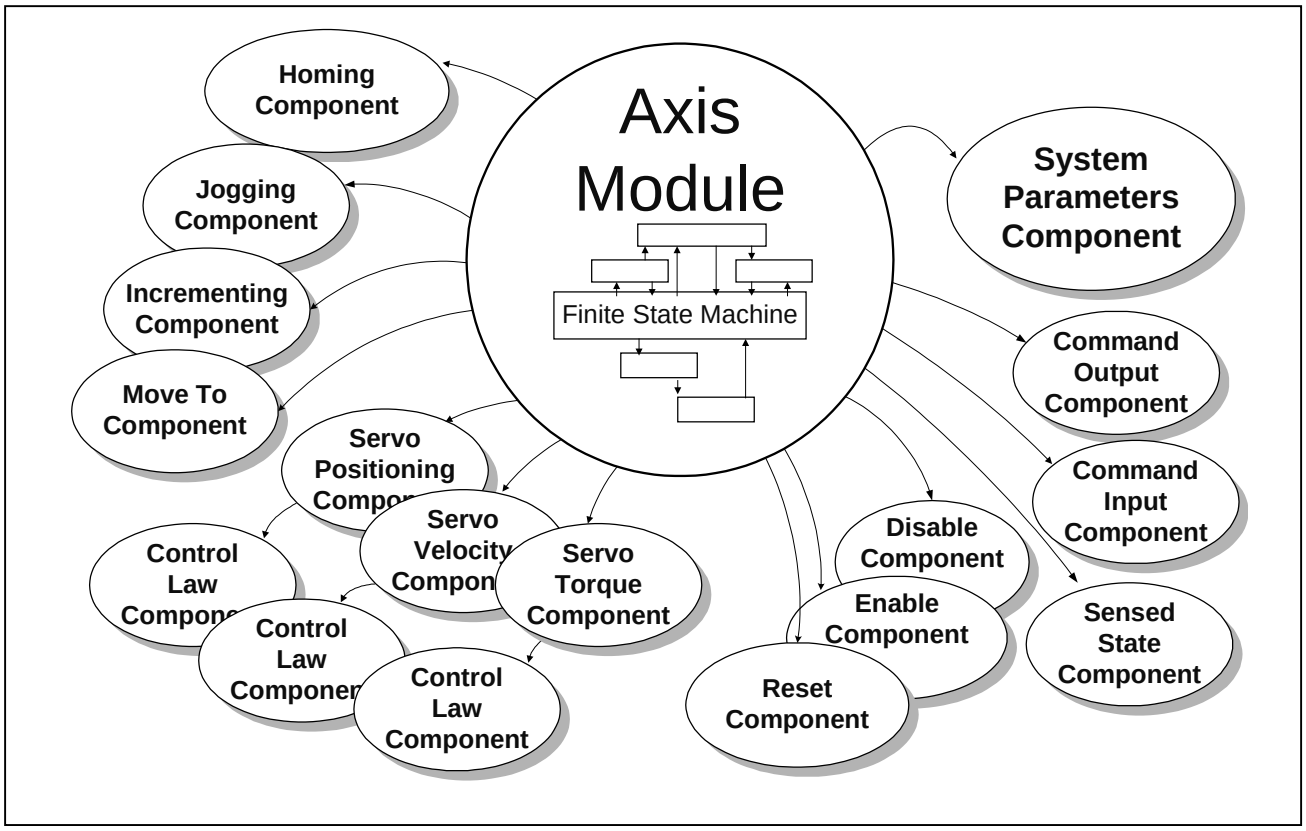


Figure 3. Example use of Component Plugs in the Axis Module

3.2 Task-Driven Modules

Task-Driven modules are also event-driven, but change behavior by accepting new Tasks to be used within the module as a behavioral FSM at runtime. For example, an Axis Group module can have a separate "Transient" Task for StraightLine motion, and another Task for Arc motion. The Axis Group module need not be preprogrammed to understand that it supports both Straight Line and Arc Tasks. This contrasts to the Event-driven module, which restricts Tasks to be "pre-installed" as plugs at system integration time. Further, OMAC Task-Driven modules allow nesting of FSM, so that a full range of behavior is possible. In the case of nested Task FSM, the OMAC Task-Driven modules are responsible for activating the master Task FSM, which then coordinates its nested or subordinate FSM. The master Task can activate and coordinate its nested tasks within a single program, on a single platform or across a distributed multi-platform controller. Thus, Task-Driven modules

provide a uniform model for the extensibility and programmability of control logic in a vendor-neutral, tool-neutral, controller-neutral manner.

Part of the role Task-driven module is to provide a programming generalization satisfying most models. In machine control, the process plan customarily serves as the programming model for manufacturing part production. A process plan coordinates actions in a production operation to control one or more devices (i.e., machines, tools, fixtures, etc.) and allows "re-programming" a controller. A 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. Figure 4 gives the inheritance hierarchy of the Task model in UML notation.

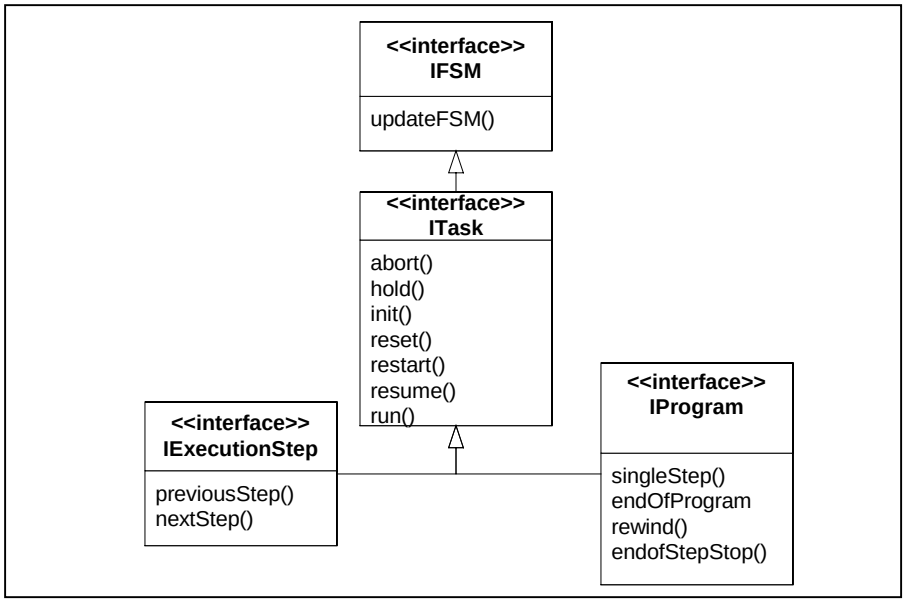


Figure 4. Task Inheritance Hierarchy

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. The Task Generator module uses a Transient Task "factory" that is responsible for generating Execution Steps. A system integrator rewrites the Transient Task factory depending on the equipment and facilities of the machine controller. Figure 5 illustrates the variety of elements that are involved in the system integration reprogramming of a Transient Task for a Tool Change. The figure offers two perspectives for handling a tool change cycle in a CNC Machining Center. The implementation of the Tool Change Transient Task depends on the underlying hardware and devices associated with the Tool Changing operation. 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., StraightLine motion segments). For case (a) in the figure, a Machining Center has a Tool Changer (with a Tool Magazine, Collet/Chuck, etc.) that requires a separate Tool Changer module to coordinate the activity. For case (b) in the figure, a Machining Center is not equipped with a moving (e.g., chain-driven) tool magazine or specialized tool-changing device so the Tool Change functionality would have to be realized as a Resident Task rather than a component/module, because it is specialized to a particular machine configuration.

A Part Program Generator may produce a program with one or more Tool Change Transient Tasks, each with appropriate parameterization (i.e., selected tool). During machining, the Transient Tool Change Task(s) run in the Task Coordinator and use the programmed logic to sequence the motion (i.e., Axis Group Transient Motion Tasks) and discrete IO (i.e., Tool Changer Module or Tool Change Discrete Logic Task) necessary to coordinate the tool change operation.

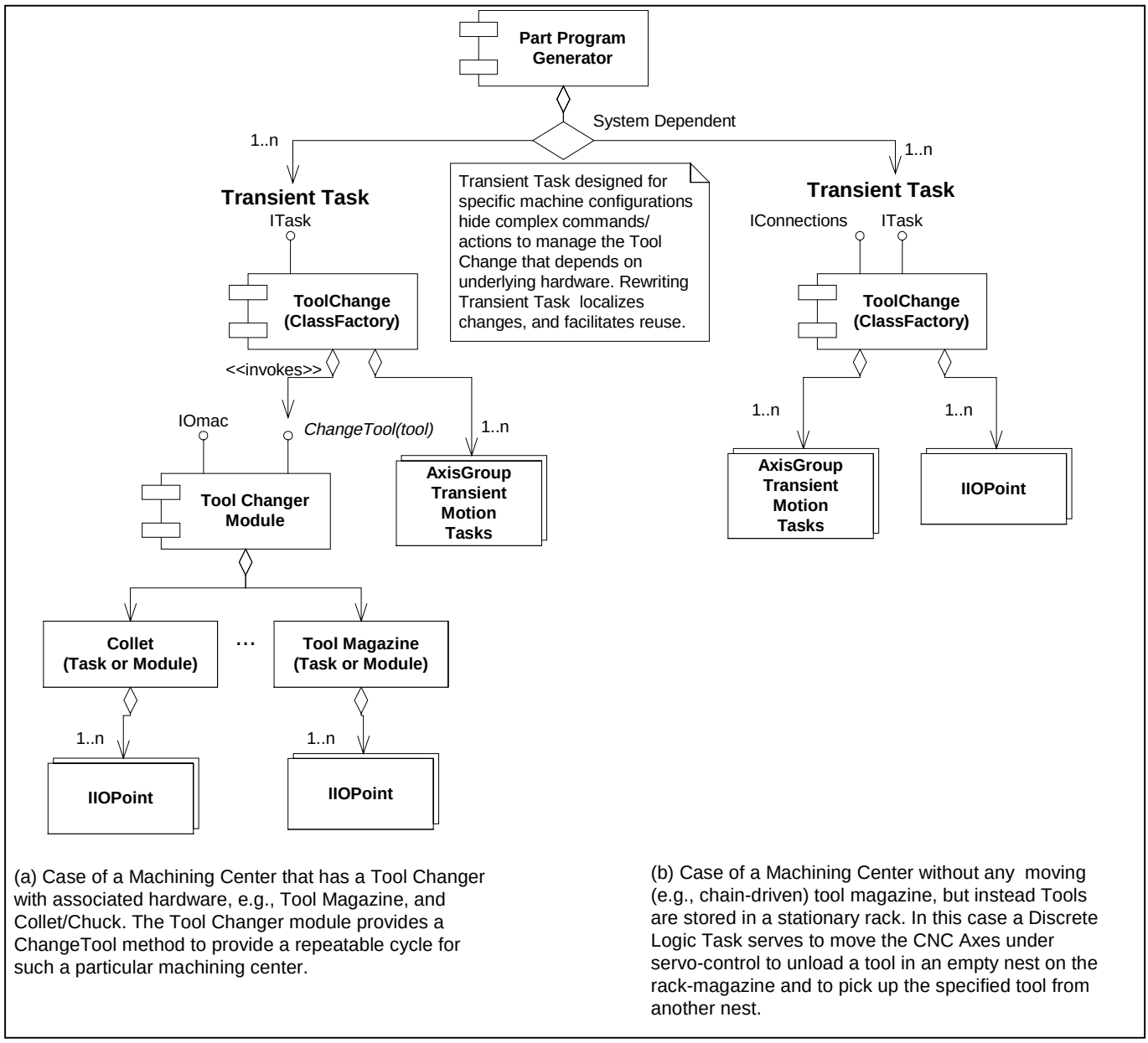


Figure 5. Relationship of Tasks, Programs, and Modules

4. DISCUSSION

We have shown numerous aspects to standardizing runtime behavior for components, but the OMAC API also contains provisions for embedding smarts in a component to be used at design time. Support for the OMAC **IConnection** interface defines a framework for resolving object dependencies to be used at design time in a drag-and-drop Integrated Developer Environment (IDE), such as Visual Basic, or other visual programming tool. Embedding design smarts also enables component introspection that allows users to determine the attributes of the component and to then customize the component. With introspection, users could graphically drag and drop components onto a component container (possibly a form or an Axis module container), and customize the properties of the components. The OMAC API workgroup foresees the ability of IDE builder tools 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 ends, the controller integration process can be greatly simplified

through IDE configuration tools so that complete systems can be integrated graphically without any source code programming required.

In the machine controller industry, there already exist controller IDE products[11,12] that are make use of component-based technology. Unfortunately, these products are not open, but proprietary and the resulting components are not interoperable outside the vendor's IDE. Without high-volume standardized component technology, it is economically unfeasible to create and maintain components in IDE particular to the manufacturing automation industry[13]. The challenge for the OMAC API is to be adaptable to these existing products in establishing controller component API standards, of which we are currently investigating. In the long term, the OMAC API component connection and dependency model may help with the automation of the integration process entirely so that instead of an IDE, systems are built from a formal requirements process.

Overall, 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." To that ends, the OMAC API specification provides many provisions to standardize component behavior in a plug and play environment in a non-restrictive approach. These provisions include:

- Integration framework with a behavior model that is consistent not just within a single component, but across all components in a system so that the components interoperate
- Behavioral model in support of task programming allowing extensible and flexible components
- Vendor-neutral, tool-neutral, controller-neutral behavior model to allow the export/import of all types of control logic
- Component smarts to support multiple life cycle phases to customize the behavior of the component in IDE.

5. REFERENCES

1. F. Proctor, and J. Albus, "Open Architecture Controllers," *IEEE Spectrum*, **34**(6), pp. 60-64, 1997.
2. P. K. Wright, and I. Greenfeld, "Open Architecture Manufacturing: the Impact of Open-System Computers on Self-Sustaining Machinery and the Machine Tool Industry," *Proc. Manufacturing International 90*, ASME Vol. 2, pp. 41-47, March 1990.
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. Open, Modular, Architecture Controls (OMAC) Users Group API Working Group, http://www.isd.mel.nist.gov/info/omacapi/
5. OSACA, "Open System Architecture for Controls within Automation Systems," http://www.osaca.org, 2000.
6. OSEC, "Open System Environment Consortium," http://www.sml.co.jp/OSEC, 2000.
7. Rational, "Unified Modeling Language," http://www.rational.com/uml, Rational Software Corporation, 2000.
8. Y. Brave and M. Heymann, "Control of Discrete Event Systems Modeled as Hierarchical State Machines," *Proc. of 30th IEEE Conf. Decision and Control*, pp. 1499-1504, December 1991.
9. P. J. Ramadge and W. M. Wonham, "Modular Feedback Logic for Discrete Event Systems," *SIAM Journal of Control and Optimization*, September 1987.
10. T. M. Sobh and R. Bajcsy, "Visual Observation of a Moving Agent, " in *Engineering Systems with Intelligence; Concepts, Tools, and Applications*, Kluwer Academic Publishers, 1991.
11. ControlShell, Real-Time Innovations, Inc., http://www.rti.com
12. LabVIEW, National Instruments, Inc., http://www.ni.com
13. S. Birla, J. Yen, J. Skeries, and D.Berger, "Control Systems Requirements for Global Commonization," Control Engineering Online Extra, http://www.manufacturing.net, January 1999.