The TEAM API Open Architecture Methodology

John L. Michaloski
National Institute of Standards and Technology

Sushil Birla
University of Michigan

Richard E. Igou
Y12 and Oak Ridge National Laboratory

Harry Egdorf
Los Alamos National Laboratory

C. Jerry Yen
General Motors

Douglas J. Sweeney **George Weinert**
Lawrence Livermore National Laboratory

Abstract

Open architecture technology offers great potential for improving tomorrow's controllers. With an open architecture, controllers can be built from multi-vendor, plug-compatible components. The desire for open-architecture controller components is high, but vendors are slow to respond. One reason for the delay in industry action is that no clear open-architecture solution has evolved. In an effort to promote open architecture control solutions, the Technologies Enabling Agile Manufacturing (TEAM) program for Intelligent Closed Loop Processing (ICLP) sponsors an Application Programming Interface (API) workgroup. This paper reviews the TEAM API workgroup's open architecture efforts. The goal of the TEAM API workgroup is to specify standard APIs for a set of open architecture controller components. The review includes the TEAM API definition of open architecture, as well as the advantages and impediments to open architectures. An overview of the TEAM API reference model including the TEAM API core modules, application framework and specification methodology is given. Overview of the TEAM API specification methodology includes discussion of the API definition and language strategy, definition of the client/server behavior model, motivation for the use of proxy agent for distributed communication, and benefits of foundation classes and reusable software.

1 BACKGROUND

Most Computer Numerical Control (CNC) motion and discrete control applications incorporate proprietary control technologies that have associated problems: non-common interfaces, higher-integration costs, and specialized training. On the other hand, a modular, standard-based, open-architecture controller is built from multi-vendor, plug-compatible modules and component parts. With openness, modules can be added, replaced, reconfigured, or extended based on the functionality required. Modifications to a module should provide equivalent or better functionality as well as offer different performance levels. Ideally, the modules should be platform and execution environment independent.

However, it is important to note that openness alone does not achieve plug-and-play. One vendor's idea of openness need not be the same as another vendor's. Openness is but one step towards plug-and-play. In reality, plug-and-play openness is dependent on a standard. This leads to the following definition of an open architecture controller:

No approval or endorsement of any commercial product by the authors or their employers is intended or implied. Certain commercial equipment, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the authors or their employers, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

An open architecture control system is defined and qualified by its ability to satisfy the following requirements:

Open provides access to module internals, ability to piece together systems module by module, ability to modify the way a controller performs certain actions, and ability to start small and upgrade as a system grows.

Modular refers to the ability of controls users and system integrators to purchase and replace controller modules without unduly affecting the rest of the controller, or requiring extended integration engineering effort.

Extendible refers to the ability of sophisticated users and third parties to incrementally add functionality to a module without completely replacing it.

Portable refers to the ease with which a module can run across platforms.

Scalable like portability, refers to the ease with which a module can be made to run in a controller based on another platform. But, unlike portability, scalability allows different performance based on the platform selection. Scalability means that a controller may be implemented as easily and efficiently by systems integrators on a high-speed processor, as a distributed multi-processor system, or on a stand-alone PC to meet specific application needs.

Maintainable supports robust plant floor operation (maximum uptime), expeditious repair (minimal downtime), and easy maintenance (extensive support from controller suppliers, small spare part inventory, integrated self-diagnostic and help functions.)

Economical allows the controller of manufacturing equipment and systems to achieve low life cycle cost.

Standard allows the integration of off-the-shelf hardware and software components into a controller infrastructure that supports standard interfaces and a standard computing environment. A standard is vital to plug-and-play.

Satisfying more, if not all, of the open requirements leads to better openness. Thus, quantifying and measuring openness can be done by comparing a claim of openness against the above requirements. Herein, the concept of an open-architecture control system that supports openness, and the auxiliary requirements will be identified as *"open, openness or open architecture."*

1.1 Advantages of Open Architecture Technology

The advantages of open architecture system can best be illustrated by the types of problems it could solve. The TEAM API solicited specific instances of problems encountered by users of proprietary controllers. Complaints about the work-arounds to overcome proprietary systems were common. From this solicitation, the following list containing explicit requirements of an open-architecture was generated. An open architecture should be able to do the following:

- provide a migration path from existing practices;
- allow an integrator/end user to add, replace, and reconfigure modules;
- provide the ability to modify spindle speed and feed rate according to some user-defined process control strategy;
- allow access to the real-time data at a predictable rate up to the servo loop rate;
- allow full 3-D spatial error correction using a user-defined correction strategy;
- decouple user interface software and control software and make control data available for presentation;
- provide communication functions to integrate the controller with other intelligent devices;

- increase the ability for 3rd party software enhancements. Examples of 3rd party enhancements include:
  - replace a PID control law with a more sophisticated Fuzzy Logic control law
  - collect servo response data with a 3rd party tool, and set tuning parameters in the appropriate control law
  - add a force sensor, and modify the feed rate according to a user defined process model
  - perform high resolution straightness correction on any axis
  - replace the user interface with a 3rd party user interface that emulates a user interface familiar to your machine operators.

The initial validation strategy for the TEAM API would be to insure that this list of capabilities can be addressed.

1.2 Impediments to Open Architecture Technology

On paper, defining an open architecture specification sounds simple. Yet, it is not because of the difficulty defining a specification that is safe, cost-effective, and supports real-time performance.

A specification must factor in current practices, as well as anticipate evolving technologies. To be successful, the open architecture definition must be cost-effective and implementable given current market practices. Further, an open architecture specification cannot be so rigidly defined as to later impede future technologies. An open architecture specification must be able to grow.

Of great importance within the controls domain is the requirement for guaranteed, hard-real-time performance. Without this, safety is at risk. Safety is a major concern voiced within the controller industry which is especially concerned with the issues of liability and delegation of responsibility within an open architecture paradigm. It is assumed that new industry practices would have to be adopted for open architecture controllers. A greater responsibility would be placed on the integrator. Conformance testing would now play a larger role. Conformance could require regression and boot-up testing and verification procedures to guarantee proper operation.

Another problem is that defining a specification that can map into a broad range of platforms and implementations is not easy. Here the problem of using a specification language that can map to any number of implementations is encountered. The TEAM API resolution to this problem was to use a technology neutral language - Interface Definition Language (IDL) - which will be discussed later.

A further hindrance is the fact that modules are not "self-contained." Defining an infrastructure within which the modules can operate is necessary and quite difficult. We consider the *infrastructure* to be defined as the services that tie the modules together and allow modules to use platform services. The infrastructure is intended to hide specific hardware and platform dependence; however, this is often difficult to achieve.

Containing the scope of the specification is also difficult. Openness goes beyond run-time APIs. There can be "other" APIs, including configuration, integration, and initialization. As an example, consider the simple use of a math library API. Even there, specification of the math library implementation must be done to select either a floating point processor or software emulation.

Finally, group and industry dynamics can be a problem. From a workgroup perspective, getting people to agree can be a challenge. From an industry perspective, many companies do not perceive any direct benefit from an open architecture. Overcoming the workgroup inertia and industry skepticism by promoting and articulating the benefits of open architecture remains a fundamental key to open architecture acceptance.

2 REFERENCE MODEL

The TEAM API requirements were derived from the OMAC or "Open Modular Architecture Controller" requirements document [OMA94]. The OMAC document describes the problem with the current state of controller technology and prescribes open modular architectures as a solution to these problems. OMAC defines an open architecture environment to include Platform, Infrastructure, and Core Modules. TEAM

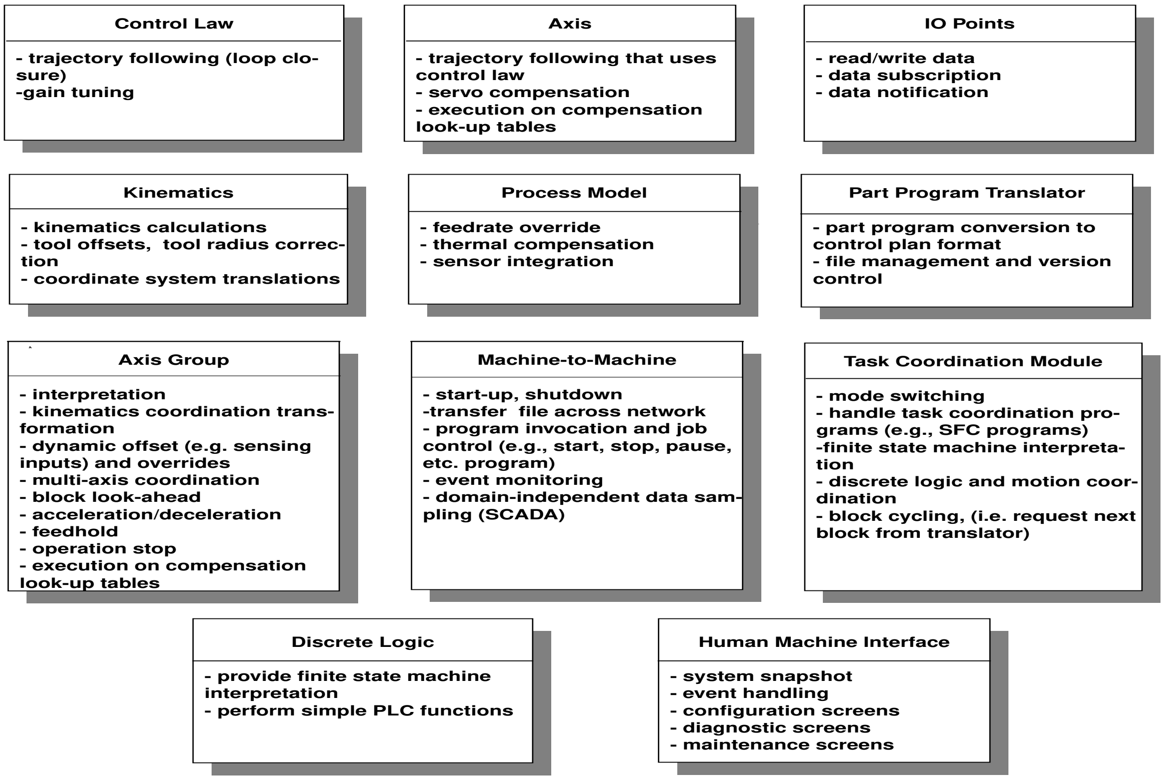

Figure 1: TEAM API Core Modules

API defines a *module* to be a piece of the system that is sufficiently defined such that it can be replaced by another piece from a third party. This, at a minimum, will maintain the same service through the same interfaces, have the same potential states, and have the same state transition conditions.

Using the OMAC specification model as a baseline, Figure 1 diagrams the TEAM API Core Modules including a brief description of a module's general functional requirements. The Core Modules have the following general responsibilities:

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 a finite state machine (FSM) interpreter accepting directives from an operator or as Sequential Function Charts (SFC) programmed in IEC 1131-3[IEC93].

Part Program Translator modules are responsible for translating the part program into control sequences.

Axis Group modules are responsible for coordinating the motions of individual axes.

Axis modules are responsible for motion control.

Kinematics Models modules are responsible for kinematics configuration and calculation, geometric correction, tool offsets, radius corrections, and effects of tool wear. Computing forward and inverse kinematics, mapping and translating between different coordinate systems, and resolving redundant kinematic solutions are examples of kinematic model functionality.

Control Law modules are responsible for loop closure calculations to close the motion loop.

Human Machine Interface or HMI modules are responsible for remotely handling data, command, and event service of an internal controller module. Defining a presentation style (e.g., GUI look and feel, or pendant keyboard) is not part of TEAM API effort.

Process Model is a module that contains dynamic data models to be integrated with the control system. Based on external stimuli or static modeling techniques, the process model produces adjustments or corrections to nominal rates and path geometry. Feedrate override and thermal compensation are examples of process model functionality. The process model is important to the concept of extensible open systems.

Discrete Logic modules are responsible for implementing system control laws that can be characterized by a Boolean function from input and internal state variables to output and internal state variables. More than one discrete logic module is permitted, but not necessary. Multiple discrete logic modules is similar to having many PLC's networked together within the same computing platform.

I/O Points form an IO system that is responsible for the reading of input devices and writing of output devices through a generic read/write interface.

Machine-to-Machine modules are responsible for connecting and coordinating controllers from different domains or address spaces. An example of this functionality is the communication from a Shop Floor controller to an individual machine controller on the floor.

2.1 Reference Architecture

The TEAM API does not have an explicit reference architecture. Instead, TEAM API endorses component based technology in order to support the OMAC core modules. At a higher level, the assembly of the OMAC modules into a system requires an integration or reference architecture. TEAM API does not prescribe a reference architecture. Rather, TEAM API provides API for each module and offers an assembly strategy described below for connecting modules. Suggestions are offered, but are not mandated.

TEAM API assumes a module assembly described by this abstraction hierarchy:

- Foundation Classes
- Framework Components
- Core Modules
- Integration or Reference Architecture
- Application Architecture

The foundation classes are the building blocks that may be found in multiple modules. For example, the class definition of a point would be found in most modules. Framework components are instances of foundation classes that can be integrated into the core modules. For example, line, helix, and NURB are framework components for the motion plan class. The core modules have the functionality as previously outlined. An integration or reference architecture describes a configuration methodology for component topology, timing, and inter-component communication protocols. The output of the integration architecture is the application architecture. With the application architecture, users can develop and run programs. Some candidate distributed reference architectures include the following: agent-based, DCOM [DCO], CORBA [COR91], RCS [Alb91], OSACA [OSA96], or EMC [PM93].

2.2 Application Framework

As stated, the TEAM API provides core modules, but does not mandate an integration architecture. The TEAM API strategy is to build application control systems as a set of connected modules that communicate through the published API. The emphasis of TEAM API module assembly is low-level that is aimed for the

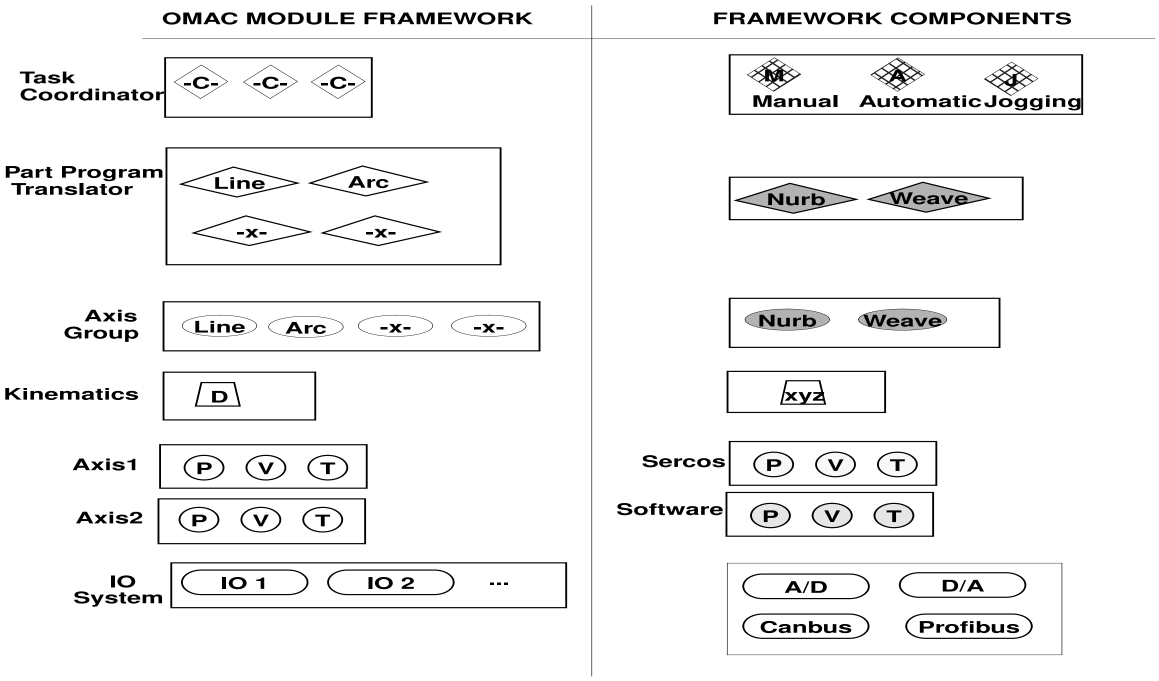

Figure 2: Control Framework

system integrator. At this level, the system integrator links ".o" object files (or linked libraries) to assemble a controller. The .o's correspond to procured modules bought as commercial off-the-shelf technology (COTS). The assembly of TEAM API modules in such a manner is referred to as the *framework paradigm*.

Object-oriented *frameworks* are sets of prefabricated software and building blocks that are extensible and can be integrated to execute well-defined sets of computing behavior. Frameworks are not simply collections of classes. Rather, frameworks come with rich functionality and strong "pre-wired" interconnections between the object classes.

This contrasts with the procedural approach where there is difficulty extending and specializing functionality; difficulty in factoring out common functionality; difficulty in reusing functionality that results in duplication of effort; and difficulty in maintaining the non-encapsulated functionality. With frameworks, application developers do not have to start over each time. Instead, frameworks are built from a collection of objects, so both the design and the code of a framework may be reused.

In the TEAM API framework the prefabricated building blocks are the COTS implementations of the OMAC modules and framework components. As a simple example, Figure 2 illustrates a framework for a typical controller application. An application developer buys the modules, and then the application developer "puts the pieces together."

Within the example, there is a task coordinator module which has containers for inserting capabilities (in the figure represented by a -C- framed by a diamond). The capabilities include Manual, Automatic or Jogging. The application developer is free to put one or more of these capabilities into the task coordinator or develop a unique capability. For Part Program Translation and Axis Group, the application developer is already provided line and arc path descriptions but can plug in Nurb (Non-Uniform Rational B-Spline) or Weave path descriptions. Once again, application developers could uniquely develop a path description. For the Axis modules, the application developer has the possibility to do position (P), velocity (V) or torque (T) control in software, hardware or some combination of hardware and software. For software P control, the application developer would select a control law object from the Software set. For hardware P control, the application developer would select a control law object from the Sercos set.

Using the TEAM API framework paradigm, application development involves three groups:

Users define the behavior requirements and the available resources. Resources include such items as hardware, control and manufacturing devices, and computing platforms. For behavior, the user defines the performance and functionality expected of the controller. Performance includes such characteristics as how fast or how accurate the application must be. Functionality defines the controller capability such as the ability to handle planar part features versus complex part features.

System Integrators select modules and framework components to match the application performance and functional requirements. The system integrator configures the modules to match the application specification. The system integrator uses an integration architecture to connect the selected modules and verifies the system operation. The system integrator also checks compliance of modules to validate the user-specification of performance and timing requirements.

Control Component Vendors provide module and framework component products and support. For control vendors to conform to an open architecture specification, they would be required to conform to several specifications including the following:

- customer specifications
- module class specification
- system service specification

The system service describes the platform and infrastructure support (such as communication mechanisms) and the resources (disks, extra memory, among others) available. Computer boards have a device profile that includes CPU type, CPU characteristics and the CPU performance characteristics. Included within the profile is the operating system support for the CPU. A spec sheet or computing profile [SOS94] is required to describe the system service specification that would include such areas as platform capability, control devices, and support software.

2.3 Application Example

Figure 3 illustrates the major systems of a controller as they might be configured for a typical numerical control application. The application example describes programmed numerical control for a two-axis lathe. Axis components are assumed be the same for each axis and consist of a PWM motor drive, an amplifier enable control, an amplifier fault status signal, an A-QUAD-B encoder with marker pulse and switches for home and axis limits. Spindle drive components are assumed to provide a facility for setting spindle speed and direction and to start and stop spindle rotation. A machine sensor system is assumed to consist of a set of analog and digital sensors monitoring coolant temperature and oil pressure. A machine safety system is assumed to consist of a set of input switches monitoring E-Stop, Power-Up and Reset. A control pendant is assumed to provide an operator with a simple set of control functions including part program selection, Cycle Start/Stop, Feedhold, Feedrate Override, Manual Data Input and Manual Jogging. Machine part programs are assumed to be in EIA RS274D format. Control pendant is assumed to display machine status to an operator including indication, machine modes, program status and machine diagnostics.

Figure 3 shows the implementation as two sets of components, the larger box of components is the real-time controller and the smaller box contains the HMI mirror. More on the mirrored HMI system will be presented later. The example controller is made of seven major systems. Each system is made up of one or more replaceable modules. Modules are tied together through exposed interfaces. A key concept in modular open architectures is that the system may be incrementally adapted to changing requirements. Three mechanisms for adapting the system are adding modules, replacing modules and reconfiguring modules by reconnecting the interfaces. The seven major systems that make up the example controller include:

IO System consists of one or more IO modules. Each IO module corresponds to a sensor or actuator. The IO module interfaces are used by axis modules, axis group modules, logic control modules and human interface modules. Sample timing for IO modules is controlled by the task coordination module.

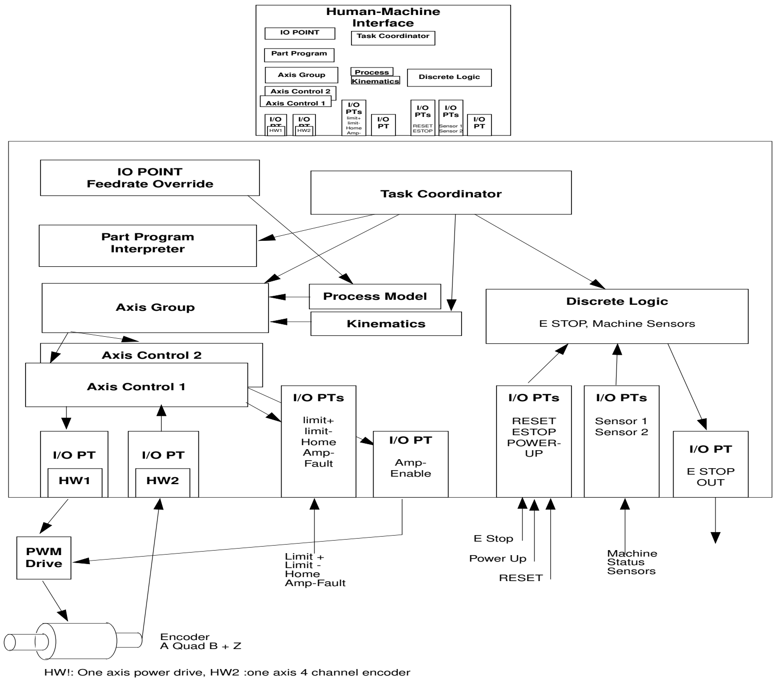

Figure 3: TEAM API Example System

Axis System consists of one or more axis modules which, in turn, contain control law modules. Each axis module requires two or more IO module interfaces. These represent sensor input and actuator output. Each axis module provides a command interface that is normally connected to an axis group module. Control law modules may provide additional interfaces that allow features such as status information for the human interface, monitoring/tuning of internal parameters, real time data collection and real time algorithm modification. Each axis module references one or more control law modules which apply a servo algorithm to generate a new actuator command based on current sensor readings, commanded set points and machine state.

Trajectory Generation System consists of one or more axis group modules, a process module and a kinematics module. An axis group module requires at least one control loop interface for each coordinated degree of freedom in the computed trajectory. It may also require additional control interfaces if it supports algorithmically related motions (electronic gearing). An axis group module may also require one or more IO module interfaces to provide sensor modified generation such as impedance control. An axis group requires a process model module and a kinematic model module to handle application and device modeling. An axis group module provides at an interfaces which is normally connected to a task coordinator module.

Process Model provides support to each axis group to receive dynamic input such as feedrate and spindle override.

Kinematics Model provides support to an axis group for coordinate transformation information, such as relative offset.

Task Coordinator System normally consists of one task coordinator module. A task coordinator is the central point for coordination of actions. A task coordinator understands the controller configuration to say what modules are in the system and how to start up the modules. A task coordinator is the controller's main sequencing engine, "what happens when," or the highest level Finite State Machine within the controller. A task coordinator may provide an interface normally used by the human interface module for machine mode and program sequencing.

Discrete Logic System consists of one or more discrete logic modules. Each of these discrete logic modules is a finite state machine, similar in functionality to the task coordinator module. The system normally contains a large number of discrete logic modules with a variety of requirements for IO module interfaces. Logic control modules provide an interface to the task coordinator module that allow status and event transition information to be conveyed. Logic control modules may also provide an interface to be used by part program interpreter modules. Discrete logic module operation is distinguished from control law module operation by the fact that logic control modules execute Boolean equations to generate new IO output values and detect event transitions based on IO inputs and machine state.

Part Program Translator System consists of one or more part program translator modules. Part Program Translator is responsible for reading and translating programs which represent machine operation and tooling. Part Program Translator output is a list of control plans to be interpreted by the task coordinator, motion subsystem or discrete logic subsystem. A part program interpreter uses several system infrastructure services - primarily file system services. A part program interpreter provides an interface that is normally connected to a human interface module.

Human Machine Interface system is composed of a set of HMI objects which mirrors the state of the controller objects. The main assumption is that HMI objects are a snapshot of control system objects and use proxy agents for communication.

3 SPECIFICATION METHODOLOGY

To satisfy the OMAC open architecture specification, a standard API for each of the Core Modules would be defined. Consequently, the primary goal of the TEAM API workgroup is to define standard API for the Core Modules. This section will refine the concept of "API" and describe the TEAM API specification methodology. The API specification methodology applies the following principles:

- Stay at API level of specification. Use IDL to define interfaces.
- Do not specify an infrastructure.
- Use Object Oriented technology.
- Use general Client Server model, but use state-graph to model state behavior.
- Use Proxy Agents to hide distributed communication.
- Finite State Machine (FSM) is model for data and control.
- Define Foundation Classes to foster the concept of reusable assets.
- Mirror system objects in human machine interface.

The following sections will discuss these principles.

3.1 API Specification

API stands for Application Programming Interface, and refers to the programming front-end to a conceptual black box. The math function `‘‘double cos(x)’’` specifies the function name, calling sequence, and return parameter, not how the cosine is implemented, be it table lookup or Taylor series. Of importance to the API specification is the function "signature" and its calling and return sequence, assuming of course, that cosine doesn't take too long. Behavior is an explicit element within the API definition and relies on a defined state transition model. A (standard) API is helpful because programming complexity is reduced when one alternative exists as opposed to several. For example, the cosine signature is generally accepted as `cos(x)`, not `cosine(x)`. This is a small but significant standardization.

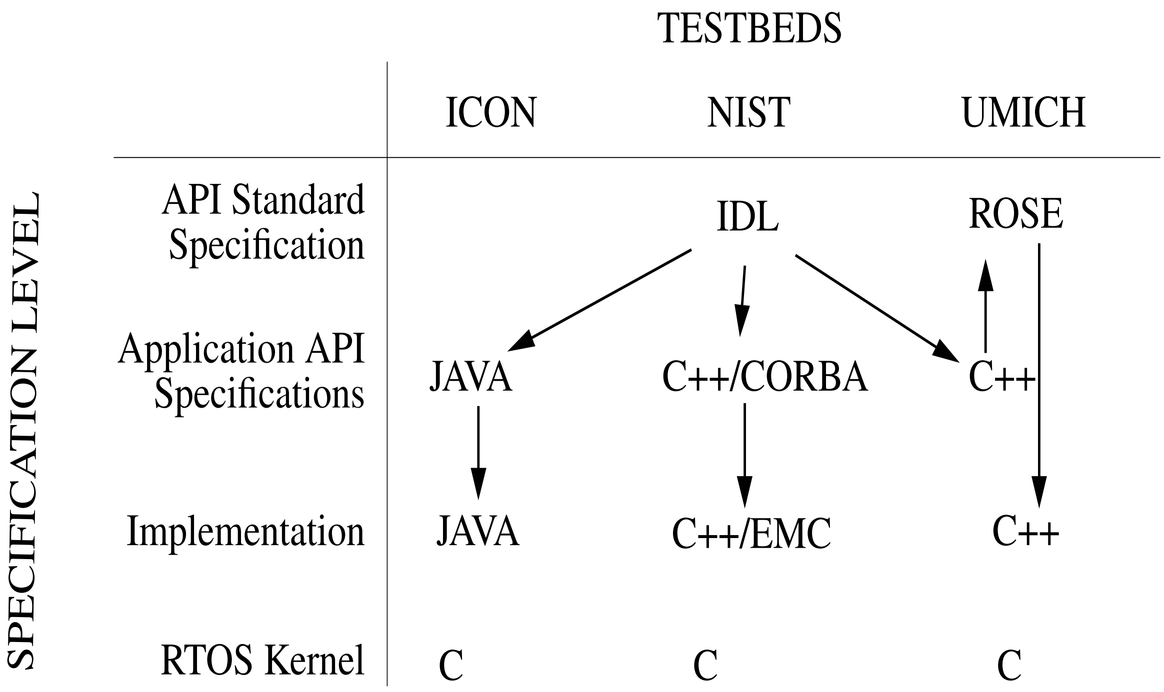

Figure 4: Specification Language Mapping

At a programmatic level, the importance of a standard API can be seen within the Next Generation Inspection Project (NGIS) at NIST[NGI]. The NGIS project has integrated three commercial sensors and one generic sensor into the Coordinate Measuring Machine controller. Taming diversity was a problem. Each sensor had a different "front-end" - one had a Dynamically Linked Library (.DLL) interface, one had a memory mapped interface, one had a combination port and memory mapping. None of the sensors had the same API. Yet, all of the sensors were "open."

There exists a problem selecting the API specification language. The specification language must be flexible enough to support a variety of implementation languages and platforms. TEAM API chose IDL, or the Interface Definition Language, for its specification language [COR91]. IDL is a technology-independent syntax for describing interfaces. In IDL, interfaces have attributes (data) and operation signatures (methods). IDL supports most object-oriented concepts including inheritance. IDL translates to object-oriented (such as C++ and JAVA) as well as non-object-oriented languages (such as C). IDL specifications are compiled into header files and stub programs for direct use by application developers. The mapping from IDL to any programming language could potentially be supported, with mappings to C, C++ and JAVA available.

To clarify the problem of unifying the specification, consider the mapping of the TEAM API IDL onto three different validation testbeds. Figure 4 illustrates mapping IDL to the different implementation strategies. For ICON, the standard API in IDL has to be mapped into JAVA. At the University of Michigan, they are using the ROSE CASE tool to design their controller. ROSE accepts C++ header through a reverse engineering process. At the NIST testbed, the IDL will be translated into C++ headers and use the Enhanced Machine Controller and its infrastructure[PM93]. For these three implementations, only the IDL specification can be mapped into all the languages needed to support the applications.

3.2 Object Oriented Technology

TEAM API uses an object-oriented (OO) approach to specify the modules' API with class definitions. The following terms will define key object-oriented concepts. A *class* is defined as an abstract description of the data and behavior of a collection of similar objects. Class definitions aggregate both data and methods to offer *encapsulation*. An *object* is defined as an instantiation of a class. For example, the class `SERCOS-Driven`

`Axis` describes objects in the running machine controller. A 3-axis mill would have three instantiations of that class – the three objects described by that class. An *object-oriented program* is considered a collection of objects interacting through a set of published APIs. A by-product of an object-oriented approach is *data abstraction* which is an effective technique for extending base types to meet the programmer needs. A "complex number" data abstraction, for example, is certainly more convenient than manipulating two doubles.

3.2.1 Inheritance

Inheritance is useful for augmenting data abstraction. OO classes can inherit the data and methods of another class through class derivation. The original class is known as the *base or supertype class* and the class derivation is known as a *derived or subtype class*. The derived class can add to or customize the features of the class to produce either a specialization or an augmentation of the base class type, or simply to reuse the implementation of the base class. To achieve a framework strategy, all TEAM API class signatures (methods) are considered "virtual functions." Virtual functions allow derived classes to provide alternative versions for a base class method.

Using an Axis module as a server, assume that all the axis does is set a variable x.

```
class Axis
{
  virtual void set_x(float x);
private:
  double myx;
}

application()
{
  Axis ax1;
  ax1.set_x(10.0);
}
```

To extend the server, a base class to add an offset to its value before each set is derived. This could also be achieved on the server side if so desired.

```
class myAxis : public Axis
{
    virtual void set_x(float x){ x= x + offset; Axis::set_x(x); }
  private:
    double myx;
    double offset; // set elsewhere for offset calculation
}

application()
{
    Axis ax1;
    myAxis ax2;
    double val;
    double offset;

    val=10.0;
```

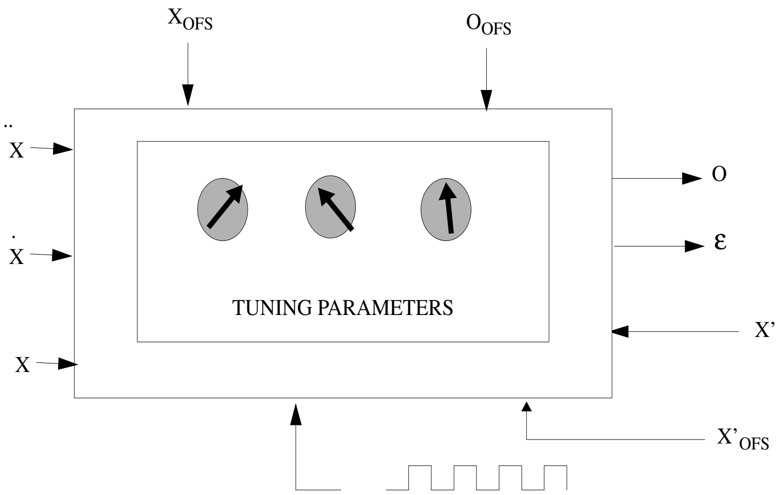

Figure 5: General Control Law

```
    ax1.set_x(val+offset); // explicit offset in application code
    ax2.set_x(val);        // offset hidden by configuration
}
```

3.2.2 Specialization

TEAM API leverages the OO concept of inheritance to use base and derived classes to add specialization. When defining a control law, one has many options including PID, then Fuzzy, Neural Nets, and Nonlinear. This plethora of options implies a need to contain the realm of possibilities. The TEAM API approach is to define a base type (generally corresponding to one of the OMAC Core Modules) and then add specialized classes.

The control law module illustrates the base and derived class specialization. The responsibility of the Control law module is conceptually simple – use closed loop control to cause a measured feedback variable to track a commanded setpoint value using an actuator.

Figure 5 illustrates the definition of a base control law. The concept of tuning is encapsulated within the black box and is conceptually controlled via "knob turning." The concept of accepting third party signal injection is handled by the inclusion of pre-and post-offsets (or injection points). These offsets allow sensors or other process-related functionality to "tap" and dynamically modify behavior by applying some coordinate space transformation. The IDL definition of the illustrated control law module follows. Each IDL `attribute` maps into a get and a set accessor methods.

```
interface CONTROL_LAW{
    // Attributes
    attribute double X;
    attribute double Xdot;
    attribute double Xdotdot;
    attribute double output;
    attribute double actual;
    attribute double following_error;
    attribute double XOffset, OutputOffset, XprimeOffset;
    // Operations
    API::Status calculate_control_cmd();
    API::Status init();   // clear time history
```

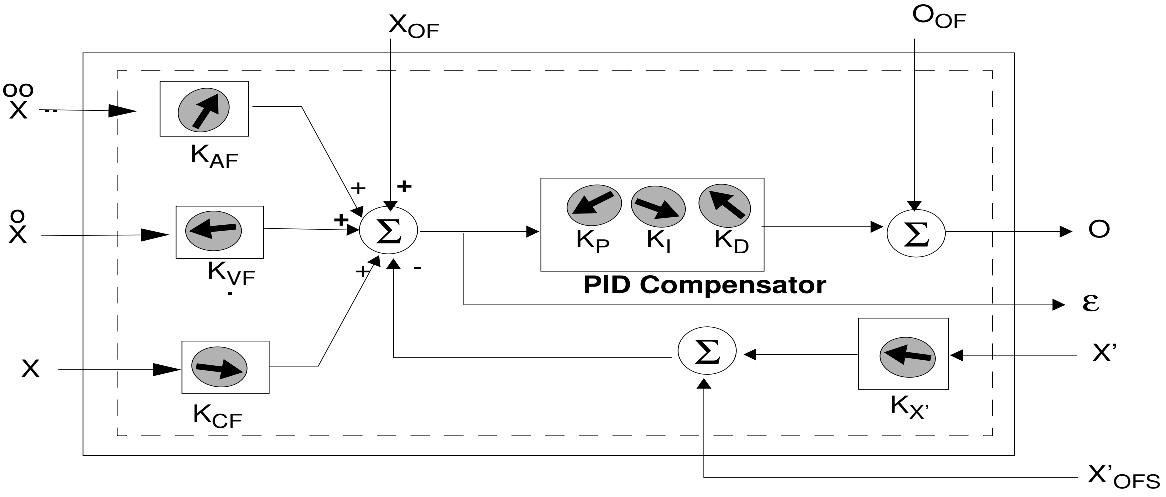

Figure 6: PID Control Law

```
};
```

Each `CONTROL_LAW` specialization is a subtype whereby each subtype inherits the definition of the supertype. By applying this concept, an evolutionary process evolves to adapt to changes in the technology. At first, only highly-demanded subtypes, such as PID, were handled. Figure 6 conceptually illustrates the PID specialization of the control law. The IDL definition of the PID control law follows.

```
interface PID_TUNING: CONTROL_LAW{
    // Attributes
    attribute double Kp, Ki, Kd;
    attribute double Kaf, Kvf, Kcf, Kxprime;
};
```

TEAM API also uses inheritance to maintain levels of complexity. Level 0 would constitute base functionality seen in current practice. Level 2 would constitute functionality expected of advanced practices. Level 3, 4,..., n would constitute advanced capability seen in emerging technology, but unnecessary for simple applications.

3.3 Client Server Behavior Model

TEAM API adopts a client server model of inter-module communication. In the client/server model, a module is a *server* and a user of a module is called a *client*. Modules can act as both a client and a server and cooperate by having clients issue requests to the servers. The server responds to client requests. A client invokes *class methods* to achieve behavior. A client uses *accessor methods* to manipulate data. Accessor methods hide the data physical implementation from the abstract data representation. The server reacts to the method invocation and performs the corresponding method implementation and sends a reply (either an answer or a status) back to the client.

As a server, a module services requests from clients that can be immediately satisfied or that may require multiple cycles. Multiple cycle service requests require state space logic to coordinate the interaction. TEAM API define three types of service requests: (1) parametric requests, (2) commanded requests, and (3) administrative methods.

Parametric service requests are generally the get/set methods and are, in theory, immediately satisfied. They do not require state space logic.

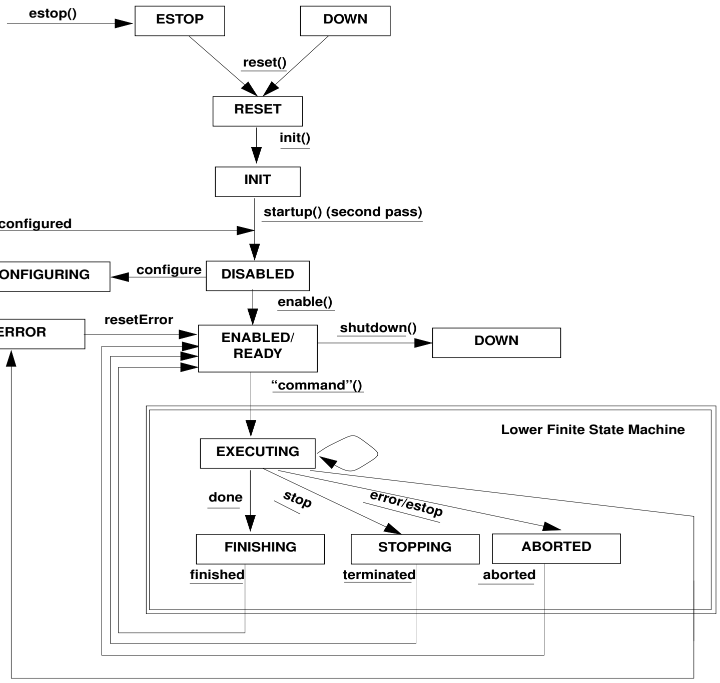

Figure 7: Generalized State Diagram

Command service requests are command methods which may run one or many axis cycles - such as move_to absolute position. Repeated cycles of the same command methods require a state transition mechanism for coordination between the client and the server. Service requests require a state space to coordinate the client server interaction.

Administrative service requests coordinate the execution of a module, for example, `processServoLoop, enable, disable` for Axis module. The `processServoLoop` function provides cyclic execution - e.g., axis module is executed once per servo loop period. In this mode, the axis software would be running as a data flow machine: at every period, it accesses the data (e.g., commanded position, actual feedback) and derives a new setpoint. Administrative methods can require a state space, such as enabled/disabled/running, but will be considered as part of the service request state space.

Service requests that run multiple cycles require a state space mechanism for coordination between the client and the server. Without such state coordination, the client could not monitor the server's progress toward satisfying the client's request. TEAM API adopts a generalized state model as illustrated in Figure 7. A state model describes the behavior of a module and consists of states and state transitions. For clients to understand a server module's control logic and react accordingly, the client and server must agree on the same state graph representation of valid states and state transitions. Figure 7 shows the typical states found in any control module – start, initialized, configured, enabled/ready, executing, and aborted.

For purposes of representing a module's state space, the concept of administrative states and process commanded states are combined within the state graph. Most of the enumerated states are administrative in order to coordinate the module computational engine. To service a command request, the module enters into the "executing" state. In the "executing" state, client/server coordination uses a lower finite state machine for coordination. This lower finite state machine for command services is module dependent.

3.3.1 Threads of Control

Parametric and Command service methods may be separated from Administrative methods and executed in separate threads of control. Figure 8 illustrates a server with multiple clients and two processes: an Axis Group process for issuing setpoints and an integration architecture process to coordinate execution.

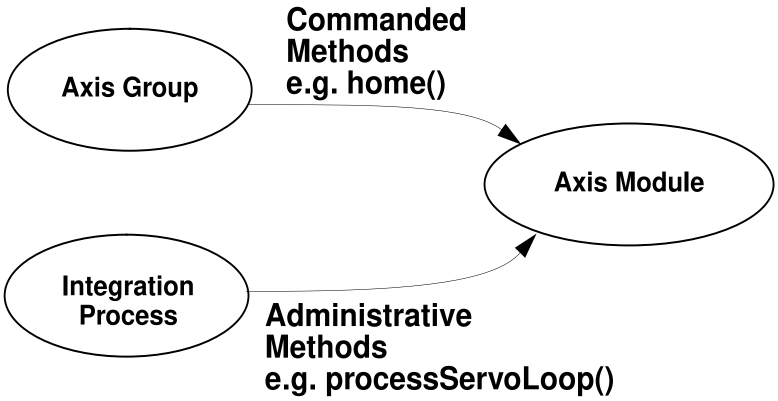

Figure 8: Multiple Threads of Control

Generally, the commanded service requests would come from an Axis Group module that is issuing setpoints to multiple axes. Another thread of execution will handle module integration by sequencing execution of the axis module. This integration module may be tied to some hardware device (such as a timer) to guarantee periodic execution behavior.

The following code snippet illustrates the "Small Picture" in assembling the modules. An integration architecture reads and writes between external interfaces between modules, and then in a separate thread of execution calls each modules administrative "execute function."

The example will develop a connection between an Axis Group module, an Axis module and an actuator and encoder IO points. First, the object naming and registry will be sketched. The integration creates object references (i.e., `io1`, `io2`, `ax1`, `trajgen1`) and then binds addresses to the objects through some name registration.

```
integration_process_init(){
    // initialize parameters
    IOpoint  io1= new IOpoint("encoder1");
    IOpoint  io2= new IOpoint("actuator1");
    Axis ax1= Axis("Axis1");
}
```

In this case, IO points were created and then an Axis and AxisGroup constructor was called. This information would typically be furnished by the Task Coordinator module because it is defined to have the knowledge of the application configuration.

Next, an integration process to synchronize the execution of the modules follows.

```
// Integration architecture puts this together
integration_process(){

    // Use state to cause module to execute - probably at different rates
    if(ax1.state() == running) ax1.processServoLoop();
    if(axgrp1.state() == running) axgrp.execute();
    ...
}
```

The Axis module ax1 has a method `processServoLoop` which is cyclical process that inputs, computes and outputs. This process could also be a finite state machine that depend on the state of the `ax1` object.

```
Axis::processServoLoop(){
    Measure value;

    // Read sensor - i.e, the current encoder value with IO system
    value=io1.get();

    // Set the next actuator value
    ax1.set(value);

    // Get the next value set by the trajectory generator
    value=get_command();

    // Put out value to DAC, (scaling done by io system)
    value=ax1.get_output();
    io2.put(&value);
}
```

3.4 Proxy Agent Technology

Client/server interaction can be local or distributed. In **local** interaction, the client uses a class definition to declare an object. When a client accesses data or invokes object methods, interaction is via a direct function call to the corresponding server class member. At its simplest, local interaction can be achieved with the server implemented as a class object file or library. Interaction is connected by binding the client object to a newly created server object implementation. Such a binding could be done by static linking, or with a dynamically linked library (DLL) or through a register and bind process that does not use the linker symbol table.

When **distributed** service is needed a *proxy agent* is used which is a set of objects that are used to allow the crossing of address-space or communication domain boundaries[M.S86]. The class describing a proxy agent uses the API of some other class (for which it is a proxy) but provides a transparent mechanism that implements that API while crossing a domain boundary. The proxy agent could use any number of lower level communication mechanisms including a network, shared memory, message queues, or serial lines.

Below is a code example to illustrate the concept of proxy agents. We will assume that we have defined an axis module by the class Axis that has but one method `set_x();`. The following code would be found in the axis module header file (or API specification):

```
class Axis : Environment
{
public:
  void set_x();
private:
  double myX;
}
```

As a user, one would develop code to connect or bind to the axis module server, which in this case has the name "Axis1." The `_bind` service is similar to a constructor method, but returns a server reference pointer and keeps track of the number of client pointer references to the server. The bind establishes a client/server relationship with the axis module. The application code is the client, and when Axis methods are invoked, a message is sent to the server. In the following code, the application sets the x variable to 10.0:

```
application(){
    Axis * a1;
```

```
    a1 = Axis::_bind("Axis1");
    a1→set_x(10.0);
}
```

If the server is colocated with the application, it is trivial to implement the object server. The `Axis::set_x` implements the value store.

```
Axis::set_x(double _x){ myX = _x; }
```

However, for distributed communication, `Axis::set_x` is defined twice - once on the client side and once on the server side. On the client side we set up the remote communication, which in this case, is a sketch of a remote procedure call.

```
Axis::set_x(double _x){
    callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
}
```

On the server side, a server waits for service events (such as the `bind`, and the `set_x` method). A corresponding `Axis::set_x` is defined to handle the x variable store. The server technology could handle events in the background or use explicit event handling. In either case, the server actions are transparent to the client.

```
Axis::set_x(double _x){ myX = _x; }

server(){
    /* register rpc server name */
    while(1) { /* service events */ }
}
```

Within TEAM API, in order to achieve transparency within source code, all methods contain a parameter field to allow customization of the infrastructure by defining an environment variable at the end of the parameter list. This is an implicit augmentation performed by an IDL compiler. For any TEAM API calling parameter list, the `ENVIRONMENT` parameter appears at the end of the calling sequence, as in:

```
void move(double x, double y, double z, ENVIRONMENT env = default);
```

The `ENVIRONMENT` can be used in several ways to tailor the infrastructure, such as to specify the remote communication protocol and the necessary parameters during transmission. The `ENVIRONMENT` can also be used to set an invocation time-out value or to pass security information. The `ENVIRONMENT` can be a stubbed dummy and ignored by the called method.

The goal of the `ENVIRONMENT` parameter is to provide transparency between invoking function calls locally or invoking function calls remotely. To provide for transparency between local and remote calls, the `ENVIRONMENT` parameter field has a default argument initializer so that local (or remote) calls need not supply this parameter.

The actual infrastructure supported by the `ENVIRONMENT` parameter will not be specified within this TEAM API document. Systems with a proprietary remote communication technology may use the `ENVIRONMENT` parameter field to enable distributed processing. The `ENVIRONMENT` can also be used as a trap door to hide other nonstandard operations. To enable compatibility with known remote processing requirements, TEAM API uses accessor functions to manipulate object data members. The data format creates one or two accessor functions – one to set and one to get – as defined by the cases for read only, write only, or read-write combinations.

```
void set_x(double inx, ENVIRONMENT env=default);
double get_x(ENVIRONMENT env=default);
```

Note that the `ENVIRONMENT` parameter at the end of the parameter list is necessary.

3.5 Infrastructure

The infrastructure deals primarily with the computing environment including platform services, operating system, and programming tools. Platform services include such items as timers, interrupt handlers, and inter-process communications. The operating system (OS) includes the collection of software and hardware services that control the execution of computer programs and provide such services as resource allocation, job control, device input/output, and file management. Real Time Operating System Extensions can be considered platform services since these extensions are required for semaphoring, and pre-emptive priority scheduling, as well as local, distributed, and networked interprocess communication. Programming tools include compilers, linkers, and debuggers.

The TEAM API does not specify an infrastructure because many of the infrastructure issues are outside the controller domain and would be better handled by the domain experts. Further, it is more cost-effective to leverage industry efforts rather than to reinvent these technologies. For example, commercial implementations of proxy agent technology are available. Microsoft has developed and released DCOM (Distributed Common Object Model) for Windows 95 and Windows NT. Many implementations of CORBA (Common Object Request Broker Architecture) are available and Netscape incorporates an Internet Interoperable ORB Protocol (IIOP) inside its browser. The question concerning the hard-real-time capability of such products remains. But, industry is acting to solve this problem. In the interim, control standards that could provide a real-time infrastructure are available [OSA96].

Because there are so many competing infrastructure technologies, TEAM API has chosen to allow the market to decide the course of the infrastructure definition. As such, to achieve plug-and-play module interchangeability, a commitment to a *Platform + Operating System + Compiler + Loader + Infrastructure suite* is necessary for it to be possible to swap object modules.

3.6 Behavior Model

For the TEAM API, *behavior* in the controller is embodied by finite state machines (FSM). TEAM API uses state terminology from IEC1131[IEC93]. An FSM *step* represents a situation in which the behavior, with respect to inputs and outputs, follows a set of rules defined by the associated *actions* of the step. A step is either *active* or *inactive*. *Action* is a step a user takes to complete a task which may invoke one or more functions, but need not invoke any. A *transition* represents the *condition* whereby control passes from one or more steps preceding the transition to one or more successor steps. Zero or more actions shall be associated with each step.

For TEAM API, the finite state machine (FSM) is the principal element of both the data flow and control flow. As outlined previously, the client/server model has command requests. The FSM data are passed within command methods from the sending TEAM API module to the receiving TEAM API module to effect behavior. FSM are then used within a module to handle the control flow. A module executes an FSM until it ends or is superceded by another FSM. How the FSM are implemented is not important to the TEAM API. Rather, a general model of FSM behavior is defined in the API so that a TEAM API module calls FSM state transition methods to initiate state changes.

Passing FSM between modules is fundamental to a flexible, extensible controller. Without an FSM, there is no explicit mechanism for sequencing an RS274 block containing simultaneous instructions. Without the FSM, intelligence must be hard-coded into the Task Coordinator to “understand” in what order to sequence and synchronize the block operations. Internally hard-coding the block decoding within the Task Coordinator prevents easy controller modification or extension.

Figure 9 illustrates the different computational elements of a TEAM API module. The TEAM API module supports module system *administration* (`reset, init, startup, shutdown`) that is handled by an administrative FSM. One or more lower level *operation* FSM are possible depending on system complexity.

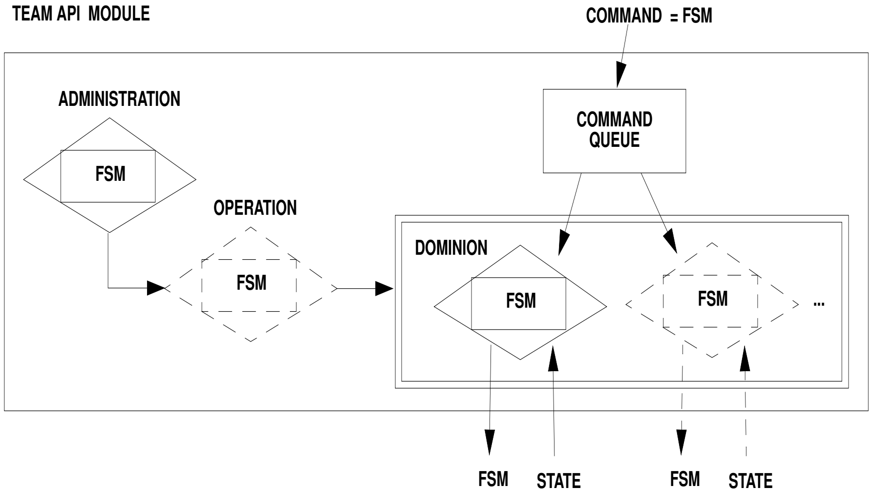

Figure 9: Computation Paradigm

The TEAM API module contains a queue, possibly of length 1, for queuing commands. Commands are in the form of FSM. The TEAM API module may have one or more FSM executing on a dominion FSM list. The *dominion* FSM list contains FSM that "rule" over other objects. In the diagram, the FSM are represented by a rectangle within a diamond. The dotted line indicates an optional FSM.

For a TEAM API module, there can be several levels of FSM applicability. TEAM API does not dictate the levels of FSM. In general, an outer FSM exists to handle module system administration. Module system administration activities can include initialization, startup, shutdown, and, if relevant, power enabling. The system administrative FSM must follow established safety standards. When the administrative FSM is in the `READY` state, it is possible to descend into a lower level operation FSM. The operation FSM is optional (as indicated by the dashes in the figure), but is necessary in the task coordinator module. The task coordinator operation FSM is called a `Capability.` Different `Capability` are used to handle the different machine modes (manual, auto). When the operation FSM is in the `READY` state, it too can descend into a lower programming or *dominion* level FSM. The dominion FSM "rules" over other objects by invoking administrative and command methods. Since the module could "rule" over several objects, the potential for multiple dominion FSM exists.

Within any of the three nested levels of FSM mentioned above, there may be more nested levels. For example, at the operation level for part programming, there may be another level of FSM to handle a family of parts. When a particular part is specified, it may invoke a nested FSM that specifies processing to be performed specific to that part. The designer of a particular control system determines the number of nested FSM levels, depending upon the complexity and organization of the controlled system.

A module comes up executing the administrative FSM and after several steps of initialization and insuring safety of operation, a module is `READY`. At this point, a module is capable of "stepping down" the FSM hierarchy to the dominion FSM. Clients can still invoke administrative methods but can now also invoke command methods to queue an FSM. When enabled, a module will transfer FSM from the queue onto the dominion list. Executing the FSM will generate output to subordinate servers. Clients can invoke parametric methods to query server status. At any point during the processing of commands, a module administrative state can change which will be reflected in the lower level FSM. For example, instructing the module administration to `stop()` will result in the administrative FSM and all dominion FSM stopping.

Command methods are defined as list management methods that put FSM objects onto the queue. Below, a Task Coordinator would call the Axis Group `ag` to append the motion segment `ms_homing` onto the axis group queue.

```
MotionSegment ms_homing;  // parameters set by the part program translator(PPT)
ag→set_next_motion_segment(ms_homing);
```

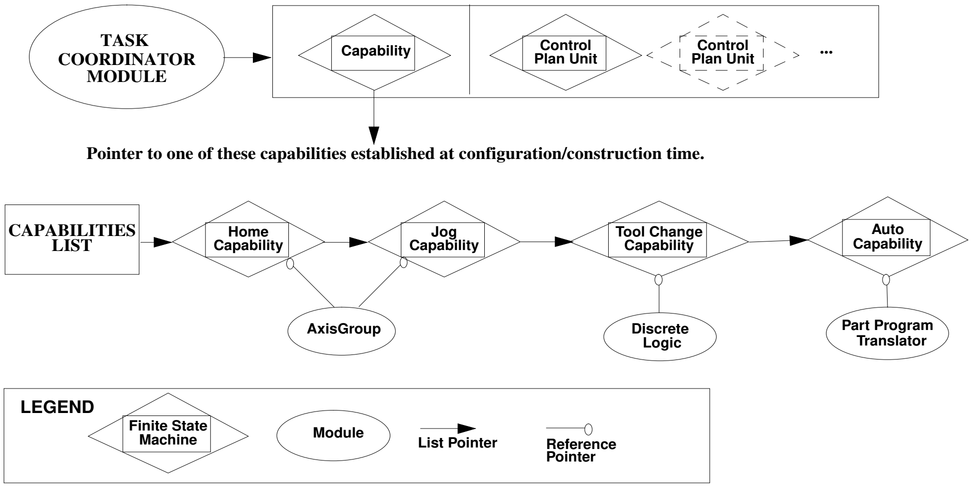

Figure 10: Controller Task Coordinator Capabilities

To sequence an FSM list, at a minimum, calling the `execute()` method until the `isDone()` boolean condition is true, can sequence a FSM from start to finish.

```
interface axis_homing : ControlPlanUnit
{
    attribute MotionSegment ms_homing;// setup by PPT
    attribute AxisGroup ag;           // set by PPT - discussed later

    execute()                         // called by Task Coordinator
    {
      if(firsttime) ag→set_next_motion_segment(ms_homing); // message passing!
      else if(!ag→isOK());            // do error checking each cycle
    }

  isDone(){ return(!ag→isHomed()); } // called by Task Coordinator
}
```

With this computation paradigm, different TEAM API modules have different command queue and FSM dominion list sizes. The Task Coordinator has a one-element queue as well as a one-element dominion FSM. The Discrete Logic module may have a one element queue, but generally has a multi-item dominion FSM list, some active, some not active, to coordinate the IO points. The Axis Group has a minimum two-element command queue, and generally a one-element dominion FSM list unless some blending of operations or synchronization with a spindle FSM is required. The Axis module only has an extensive administrative FSM. These differences will be further explored.

3.6.1 Task Coordinator

The Task Coordinator has a one-element FSM dominion list. The dominion FSM list is defined by the `Capability` class definition. Associated with the `Capability` FSM is a `ControlPlan` list.

The `Capability` FSM supports `stop, start, execute, isDone` methods. For an application controller, there is list of capabilities that a Task Coordinator can use. Figure 10 illustrates a typical milling CNC application with `Capability` instances. Each `Capability` has reference pointers to TEAM API modules that it uses. Thus, the `Home Capability` and the `Jog Capability` each have reference pointers to the

Axis Group. When a `Capability` is executing, it coordinates the servicing of requests from the HMI. When the `Auto Capability` FSM is executing, it interacts with the Part Program Translator.

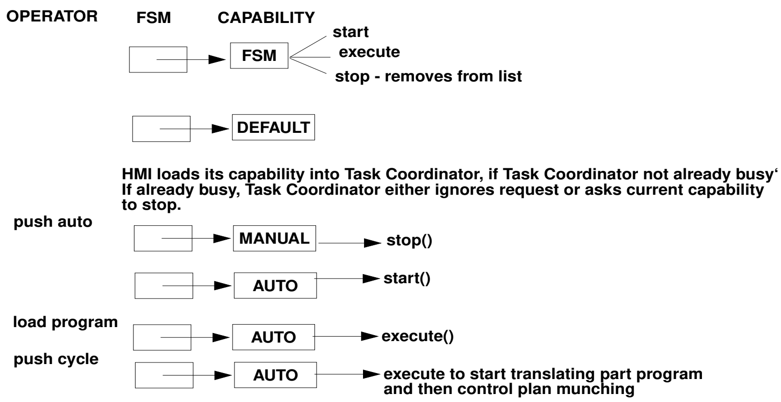

Figure 11: Step Through of a Task Coordinator Capability Sequence

Figure 11 illustrates a sequence of operations that takes a milling CNC from manual mode to automatic mode. The diagram illustrates that a `Capability` FSM has `start, stop, execute` methods. There is the assumption that there is a default `Capability`, probably an `Idle Capability`. In the scenario, the operator pushes the `auto` button that causes the HMI to execute the `Manual Capability stop` method, and load the `Auto Capability` onto the Task Coordinator queue. That cycle, the Task Coordinator will see that the `Manual Capability` boolean `isDone` is True and will swap the `Auto Capability` FSM into the dominion FSM list. The operator action to load a program will result in a program name loaded into the Part Program Translator. When the operator pushes the cycle button, it will cause the Auto `Capability` FSM to start sequencing Part Program Translator generated information. Part Program Translator information is called `ControlPlan` and will covered in the next section.

3.6.2 Control Plan Units

When the Task Coordinator dominion is the `Auto Capability`, it coordinates with the Part Program Translator to generate control information. For different applications, the Part Program Translator generates different `ControlPlanUnit` FSM. For the TEAM API, the base type control information is an FSM and is called a `ControlPlanUnit` that may embed other `ControlPlanUnit` FSM. For different control behavior, an FSM has a unique class definition derived from the `ControlPlanUnit`. A series of `ControlPlanUnit(s)` is a `ControlPlan`. A `ControlPlan` can be a simple list to represent sequential behavior or a complex tree to represent parallel controller behavior.

A `ControlPlanUnit` FSM understands how to coordinate and sequence the logic and motion submodules. The `ControlPlanUnit` FSM could put `MotionSegments` on the AxesGroup motion queue. The `ControlPlanUnit` FSM can either put `LogicUnits` on the DiscreteLogic queue or activate `LogicUnits` on the DiscreteLogic dominion list similar to a PLC scanning list. There are also `ControlPlanUnits` for decision making. (e.g., loops, end program and if/then/else). Figure 12 illustrates a `ControlPlan` with one of the `ControlPlanUnits` expanding the hierarchy of possible `ControlPlanUnit` options.

`ControlPlanUnit` has a method `execute_control_plan()` which does not need to be entirely self-contained. It may make use of services of other objects. In addition, the `ControlPlanUnit` acts as a container for embedded `ControlPlanUnits`. These embedded `ControlPlanUnits` are passed to the appropriate server, such as, a `MotionSegment` is passed to the Axes Group module. To use such a sequence of control, the Part Program Translator builds a `ControlPlanUnit` for the Task Coordinator FSM that causes a `MotionSegment` FSM to be pushed onto AxesGroup Queue. It is important to understand that this rippling effect is a fundamental mechanism for passing data through a TEAM API controller. The following section provides a simple code example to illustrate this rippling effect.

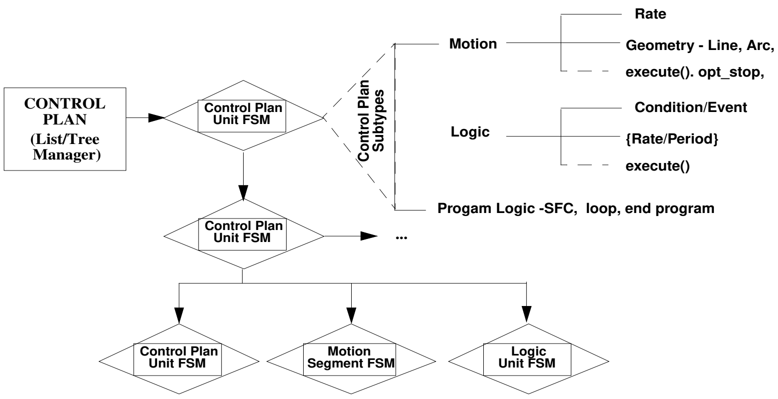

Figure 12: Control Plan Hierarchy

3.6.3 Forward Reference

The TEAM API specifies that `ControlPlanUnit` objects allow embedding of dynamic module references and direct method calls. On the surface this approach appears implausible. However, because of proxy agent technology, creating a "forward reference" by dynamically binding to an object is not hard to do. This dynamic binding is beneficial since it eliminates the need for static encoding of methods with id numbers so that methods can execute across domains (address spaces). To enable forward references, the requirement does exist for the infrastructure to support some "`lookup()`" method to map object names to addresses.

As an example, the application of proxy agent technology will be used by Part Program Translator to generate a `ControlPlanUnit` for an `axis_homing` FSM. The `axis_homing` is an FSM with a transition method `execute` and a query method `isDone` to determine FSM completion.

```
interface axis_homing : ControlPlanUnit
{
  attribute MotionSegment ms_homing;  // parameters set by the PPT
    execute()                          // called by Task Coordinator
    {
      if(firsttime)
    ag->set_next_motion_segment(ms_homing); // message passing!
      else if(!ag->isOK());            // do error checking each cycle
    }

  isDone(){ return(!ag->isHomed()); } // called by Task Coordinator
  set_axgrp(char * axgroupname ) { ag=lookup(axgroupname); }
private:
  Axis Group *ag;                      // ag set by the PPT
}
```

The `execute` and `isDone` methods use explicit calls to an Axis Group object. A "forward reference" to the Axis Group object is required. Suppose the Part Program Translator (`PPT`) receives at constructor time the name "*axisgroup1*" for the Axis Group object. Lookup of the "*axisgroup1*" must be available through the underlying proxy agent technology. Without the proxy agent technology, one has to encode the object `ag` and the methods `ag->home` and `ag->isDone`. This extra programming overhead is hidden by the proxy agent technology.

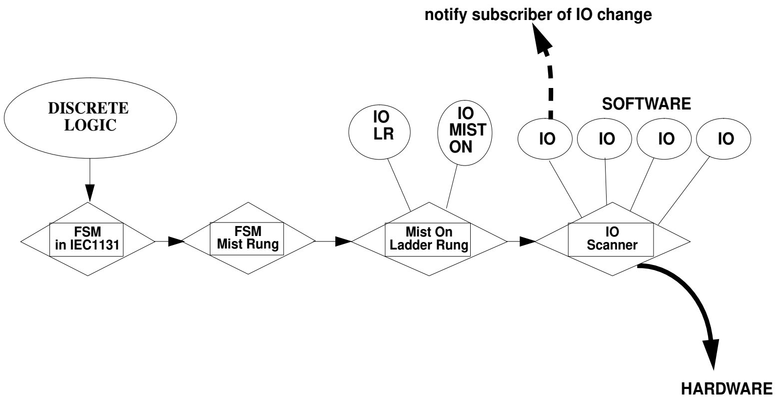

Figure 13: Discrete Logic FSM List

3.6.4 Discrete Logic

The Discrete Logic module is similar to the Task Coordinator module in that it sequences and coordinates actions through dominion FSM. However, for clarity, instead of a monolithic one-element dominion FSM, the Discrete Logic module has a multi-item dominion FSM list. In general, a Discrete Logic dominion FSM could be coded in any of IEC-1131 languages. Figure 13 illustrates the types of FSM that may be found on the Discrete Logic dominion list for a typical milling CNC application. An FSM to handle IO scanning would be expected. An FSM implemented as a Ladder Rung could be expected to handle a relay for turning a Mist pump on. Below one finds a sketch of the activity for turning the IO mist pump on.

```
mist_pump_on_rung()
execute()
{ logic:  trigger relay to turn pump on
          wait till IO/pt says pump is on
          IOmist← on;
}
```

At a higher level, a hardware-independent Mist FSM would be required to coordinate turning Mist on and off. Below is a sketch of pseudo code to sequence the Mist on operation. For coordination between FSM logic, polling or event-drive alternatives exist to wait for the IO Mist on activity to complete.

```
mist_on_fsm()
{ "MistOn LR IO <- on"  to turn LR=ladder rung on
  "subscribe to event that IO Mist On ==on"
  "wait for event or poll for IO point for Mist On == on "
  "done - deactivate FSM for scanning"
}
```

3.7 Foundation Classes and Data Representation

Exchange of information between modules relies on standard information representation. Such control domain information includes units, measures, data structures, geometry, kinematics, as well as the framework component technology. Figure 14 portrays the conceptual organization of framework component software as defined by foundation classes.

Consider the analogy of building materials. The primitive data types, shown at the bottom of Figure 14, are similar to such raw materials as sand, gravel, and clay. Using foundation classes and aggregating

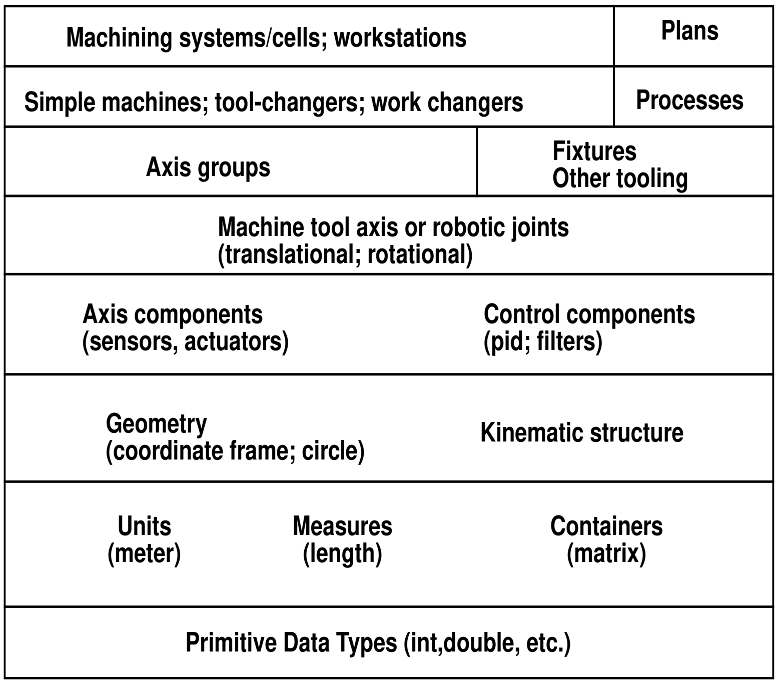

Figure 14: Software Reusable Assets

structural components, a control hierarchy of reusable software components can be built. Based upon the reusable foundation classes, these assets can be used to build class libraries for such motion components as sensors, actuators, and pid control laws.

Not all software objects have physical equivalents. Objects such as axis groups are only logical entities. Axis groups hold the knowledge about the axes whose motion is to be coordinated and how that coordination is to be performed. Services of the appropriate axis group are invoked by user-supplied plans (process programs).

TEAM API has chosen two levels of compliance for data definitions. The first level defines named data types to allow type-checking. The TEAM API uses the IDL primitive data types and builds on these data types to develop the foundation classes and framework components. For control domain data modeling, the TEAM API used data representations found in STEP Part Models for geometry and kinematics [Inta, Intb]. Internally, one could, of course, use any desired representation. The STEP data representations were translated from Express into IDL. Representation units are assumed to be in International System of Units, universally abbreviated SI. Below is the basic set of data types which use STEP terminology for data names but reference other terms for clarification.

Primitive Data

- IDL data types include *constants*, *basic data types* (float, double, unsigned long, short, char, boolean, octet, any), *constructed types* (struct, union and enum), *arrays* and *template types* bounded or unbounded sequence and string.
- IEC 1131 types - 64 bit numbers
- bounded string

Time

Length

- Plane angle

- Translation commonly referred to as position
- Roll Pitch Yaw (RPY) commonly referred to as orientation
- STEP notion of a Transform which is composed of a translation + rpy, also commonly referred to as a "pose."
- Coordinate Frame which is defined as a Homogeneous Matrix

Dynamics

- Linear Velocity, Acceleration, Jerk
- Angular Velocity, Acceleration, Jerk
- Force
- Mass
- Moment
- Moment of Inertia
- Voltage, Current, Resistance

The second level provides for more data semantics. The TEAM API adopted the following strategy to handle data typing, measurement units, and permissible value ranges. Distinct data representations were defined for specific data types. For example, the following types were defined in IDL to handle linear velocity.

```
// Information Model - for illustrative purposes
typedef Magnitude double;

// Declaration
interface LinearVelocity : Units  {

    Magnitude  value; // should this value be used?
    // Upperbound and Lowerbound, both zero ignore
    Magnitude ub, lb; // which may be ignored

    disabled();
    enabled();
};

// Application
LinearVelocity vel;
```

In this case, linear velocity is a special class. Unit representation is inherited from a general units model. Permissible values are defined as a range from lowerbound to upperbound. The units and range information are optional and may not be used by the application.

Another data typing problem that must be resolved concerns the use of a parameter. Not all parameters are required or need be set by every algorithm. For example, setting the jerk limit may not be necessary for many control algorithms. To resolve the parametric dependency issue it was decided to use a special value to flag a parameter as "not-in-use". This approach seems simpler than having a `use_xxx` type method for each parameter. For now, TEAM API has decided that setting a parameter to a unrealistic "Not a Number" value (such as `MAXDOUBLE` or 1.79769313486231570e+308) renders a parameter to be ignored or not-in-use. This works for level 1 and level 2. Within level 2, the methods `enable` and `disable` were added for convenience.

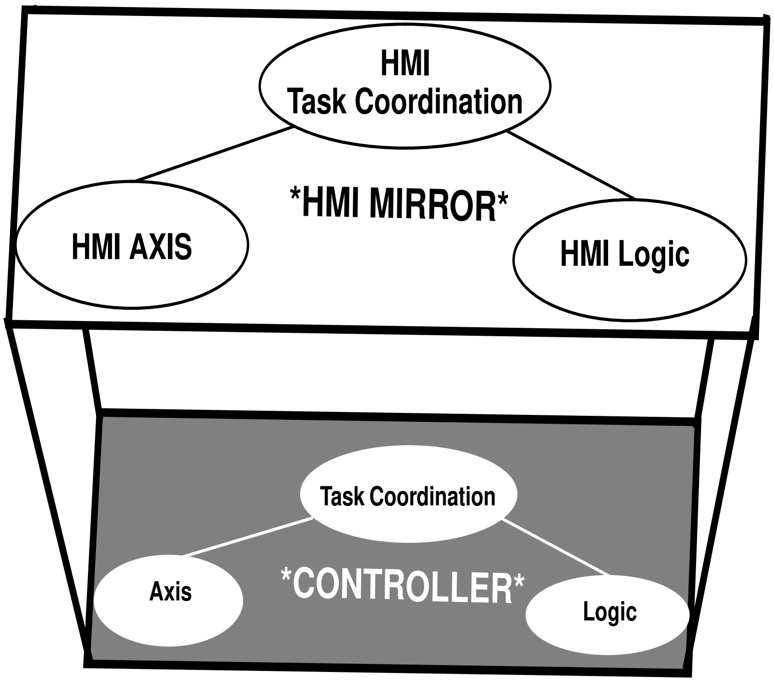

Figure 15: Human Machine Interface Mirrors Controller

3.8 Human Machine Interface

The primary HMI objective of the TEAM API is to provide the ability to "bolt-on" a Human Machine Interface to the controller. The HMI is intended to be independent of the choice of presentation medium, the dialogue mechanism, the operating system, or the programming language.

TEAM API specifies that every controller object has a corresponding HMI object "mirror". A simplifying assumption is that HMI objects communicate to control objects via proxy agents. Figure 15 illustrates the mirroring of a one axis controller that uses a task coordination module for coordination and sequencing in conjunction with a discrete logic module.

The desired HMI functionality is best understood in the context of simple problems. Three "canonical" problems exist that an HMI module must be able to handle regardless of the interface device. First, the user must be able to receive *solicited information reports* about the state of the controller, such as a current axes position. Second, the user must have *command capabilities* such as set manual mode, select axis, and then jog an axis. Third, the user must be alerted when an exception arises, in other words, handle *unsolicited information reports.* Following is an analysis of how the HMI mirror handles these cases.

To handle the information report functionality, an HMI mirror acts as a remote data base that replicates the state and functionality of the controller object and then adds different presentation views of the object. These HMI mirrors are not exact mirrors of the controller state, but rather contain a "snapshot" of the controller state. Figure 16 illustrates the interaction of the HMI mirror and the control object. In the basic scenario of interaction, the control object is the server and the HMI mirror object is the client. Each HMI mirror uses the accessor functions of get and set to interact with the control object. You will notice that each host controller object and corresponding HMI mirror have a proxy agent to mediate communication.

To handle command functionality, the HMI mirror contains the same methods as the controller object so that a command is issued by invoking a method remotely.

To handle abnormal events when polled monitoring may not be possible, an HMI mirror must serve as a client to the control object so that it can post alert events. For such unsolicited information reports, the control object uses an event notification function, `update_current_view`, in which to notify the HMI mirror that an event has occurred. This notification in turn may be propagated to a higher-authority object.

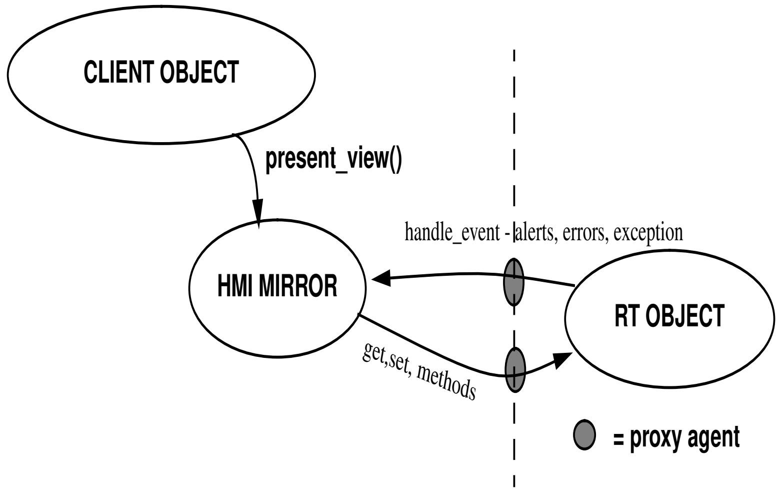

Figure 16: Human Machine Interface

The following HMI definition gives the method extensions that a control object must support to become a mirrored object.

```
interface HMI
{
  // Presentation Methods
  void present_error_view();
  void present_operational_view();
  void present_setup_view();
  void present_maintenance_view();

  // Events - to alert HMI that something has happened
  void update_current_view();
};
```

A benefit to using the HMI mirrors is the potential for vendors to supply a control object, as well as a presentation HMI object that can be incorporated into their Operator Interface. As an example of this technology, a tuning package can provide a Windows-based GUI to do some knob turning. Another example, is a tuning package that offers this capability to be plugged inside a Web browser. With this development, unlimited component-based opportunities are available.

4 DISCUSSION

TEAM API has developed an API specification that is scalable for the system design, integration and programming for systems ranging from a single-axis device to a multi-arm robot. The TEAM API working group's initial focus was to establish programming requirements for precision machining. Applicability to other control environments may be possible but is not be guaranteed. The TEAM API primary focus has been to define Application Programming Interfaces for certain modules that the ICLP community routinely wants to upgrade. In addition, the workgroup has defined an assembly framework with which to connect these modules.

The TEAM API effort is not all-inclusive. The focus of effort has been to develop module APIs and to create a methodology for assembling and reconfiguring modules. The desire is that the the methodology is general enough to handle most architectures, but specific enough to offer a path to standardization. At this time, the TEAM API effort discusses, but does not attempt to specify procedures, for such issues as the following:

- configuring modules,
- performance evaluation,
- validation, verification,
- resource profiling and environment.

TEAM API has posted on the Web numerous documents describing the module API. Other papers describe related TEAM API information on life cycle, general computation models, and control models. For more information, see the Wide World Web at the Universal Resource Locator address:
`http://isd.cme.nist.gov/info/teamapi`.

Acknowledgements

The authors would like to thank Harry Scott for his detailed technical review and Janet Land for her significant editorial assistance.

References

[Alb91] J.S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3), may/june 1991.

[COR91] Object Management Group, Framingham, MA. *Object Management Architecture Guide, Document 92.11.1*, 1991.

[DCO] Distributed Common Object Model.
See Web URL: http://www.microsoft.com/oledev/olemkt/oledcom/dcom95.htm.

[IEC93] International Electrical Commission, IEC, Geneva. *Programmable controllers Part 3 Programming Languages, IEC 1131-3*, 1993.

[Inta] International Organization for Standardization. *ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 42: Integrated Resources: Geometric and Topological Representation.*

[Intb] International Organization for Standardization. *ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 105: Integrated Application Resources: Kinematics.*

[M.S86] M.Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society Press, May 1986.

[NGI] Next Generation Inspection System (NGIS).
See Web URL: http://isd.cme.nist.gov/brochure/NGIS.html.

[OMA94] Chrysler, Ford Motor Co. , and General Motors. *Requirements of Open, Modular, Architecture Controllers for Applications in the Automotive Industry*, December 1994. White Paper – Version 1.1.

[OSA96] OSACA. European Open Architecture Effort.
See Web URL: http://www.isw.uni-stuttgart.de/projekte/osaca/english/osaca.htm, 1996.

[PM93] F. M. Proctor and J.L. Michaloski. Enhanced Machine Controller Architecture Overview. Technical Report 5331, National Institute of Standards and Technology, December 1993.

[SOS94] National Center for Manufacturing Sciences. *Next Generation Controller (NGC) Specification for an Open System Architecture Standard (SOSAS)*, August 1994. Revision 2.5.