TECHNOLOGIES ENABLING AGILE MANUFACTURING (TEAM) INTELLIGENT CLOSED LOOP PROCESSING

Open Architecture Specification

Part 1: General Information

January 11, 1996

CONTENTS

Foreword

This draft document has been prepared by the Technologies Enabling Agile Manufacturing (TEAM) Intelligent Closed Loop Processing (ICLP) Application Programming Interface (API) working group. The TEAM API working group developed a specification that will consist of the following series of documents:

Part 1: General Information

Part 2: Module Capabilities

Part 3: Module Specification Class Descriptions

Part 4: Module Specification C++ Header Files

Part 1 includes a Life Cycle model that describes the steps and actions required to build a controller. The breadth of satisfying Plug-and-Play compatibility is extensive. Not all elements in the Life Cycle have been addressed. The focus of the effort has been on defining Application Programming Interfaces for certain modules routinely that the ICLP community wants to upgrade. The TEAM API effort discusses but does NOT attempt to specify procedures for such issues as:

- performance evaluation
- validation, verification
- resource profiling and environment

This specification is scaleable 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 focus was programming requirements for precision machining. Applicability to other control environments may be possible but cannot be guaranteed.

Background

Most 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, an open-architecture controller is built from multi-vendor, plug-compatible modules and component parts. The operation to build a controller from module components is multi-faceted and includes the following major elements:

- User defines “initial conditions” such as hardware, control devices, and computing platforms that in general constitute the application resources.
- Platform supplies system low-level services (e.g., file-management, etc.)
- Integrator connects selected modules together via standard integration and configuration techniques.
- Analysis checks compliance of modules to support user-specification of performance and timing requirements.
- Flexibility of modules provides default or minimal functionality where user has not selected any.
- Scalability of modules allows convenient methods of experimentation - to reconfigure modules quickly; and optionally, capture results in order to analyze the experimentation.

These operations form the basis for the open architecture requirements which are:

Open

Allows the integration of off-the-shelf hardware and software components into a controller infrastructure that supports a “de facto” standard environment.

Modular

Refers to the ability of controls users and system integrators to purchase and replace components of the controller without adversely affecting the rest of the controller nor requiring extended integration engineering effort, permitting “plug and play” of a limited number of components for selected controller functions.

Extendible

Refers to the ability of intelligent users and third parties to incrementally add functionality to a module without replacing it completely.

Portable

Refers to the ease with which a module can run across platforms. Standards such as ANSI C and POSIX serve as a reference to which programmers adhere.

Scaleable

Like portability, refers to the ease with which a module can 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 standalone 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 parts inventory, integrated self-diagnostic and help functions, etc.)

Economical

Allows the manufacturing equipment and systems that the controller controls achieve low life cycle cost.

TEAM API System Environment

The Open, Modular Architecture Controller (OMAC)[1] defines the system requirements for an Open-architecture, Application Environment that includes Platform and Infrastructure, Core Modules, and a standard Application Programming Interface to the Core Modules. Using the OMAC model as a baseline, Figure 1 specifies the modules in a standard environment.

Adhering to the Open Architecture requirements, modules can be added, removed, and extended based on the functionality required (extensibility), be replaced by other modules with equivalent functionality but at different performance level (scalability & modularity), and are operating environment independent (portability).

The modules have the following responsibilities:

Part Program Interpreter

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

---

[1]. Initial requirements defined in White Paper jointly co-written by GM, FORD, and Chrysler

Task Coordinator

Task Coordinators are responsible for sequencing operations and coordinating motion, sensing, and event-driven control processes.

Axis Group

Axis Groups are responsible for coordinating the motions of individual axes.

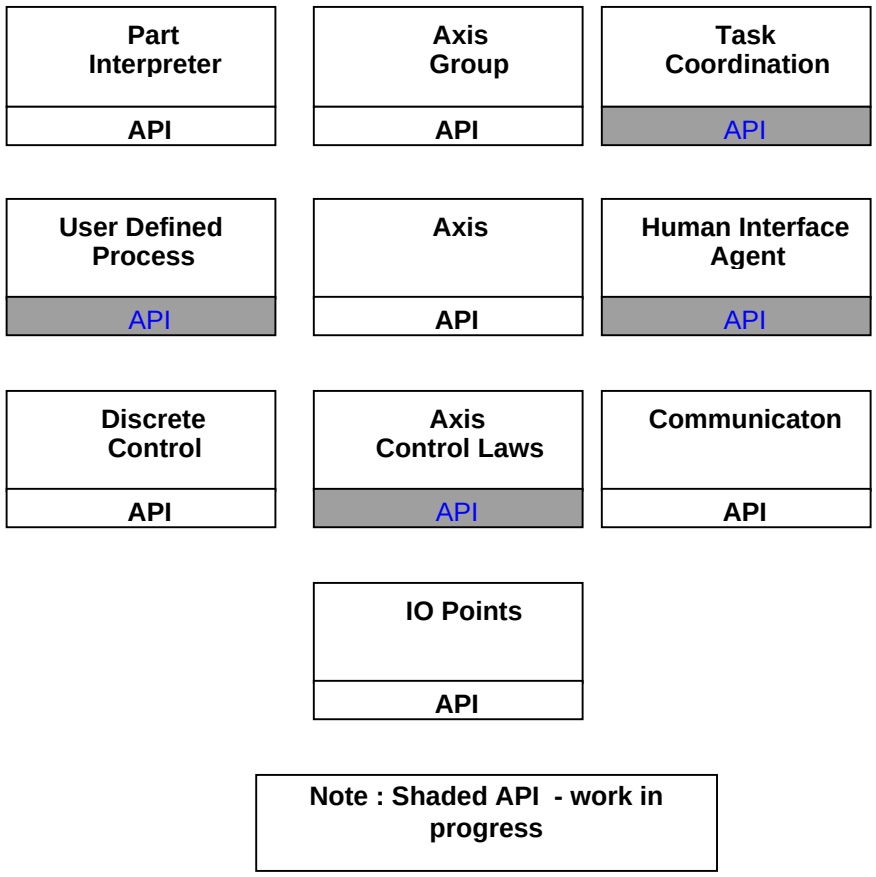


Figure 1. Major Module Types

Axis

Axes are responsible for motion control.

Kinematics Model

Kinematics Models are responsible for kinematics configuration, geometric correction, thermal compensation, tool offsets, and the effects of tool wear.

Control Law

Control Laws are responsible loop closure calculations to close the motion loop.

Human Interface Agent

Human Interface Agents are responsible for the data access between controller internal modules and appropriate mirrored graphical user interface objects in an Operator Interface environment outside of the controller.

I/O Points

I/O Points are responsible for the reading of input devices and writing of the output devices through a generic read/write interface.

Communication

Communication modules are responsible for both local area communications and sensor effector bus communications.

TEAM ICLP - Open Modular Architecture Controller: Example

Figure 2 illustrates the major systems of a controller as they might be configured for a typical numerical control application.

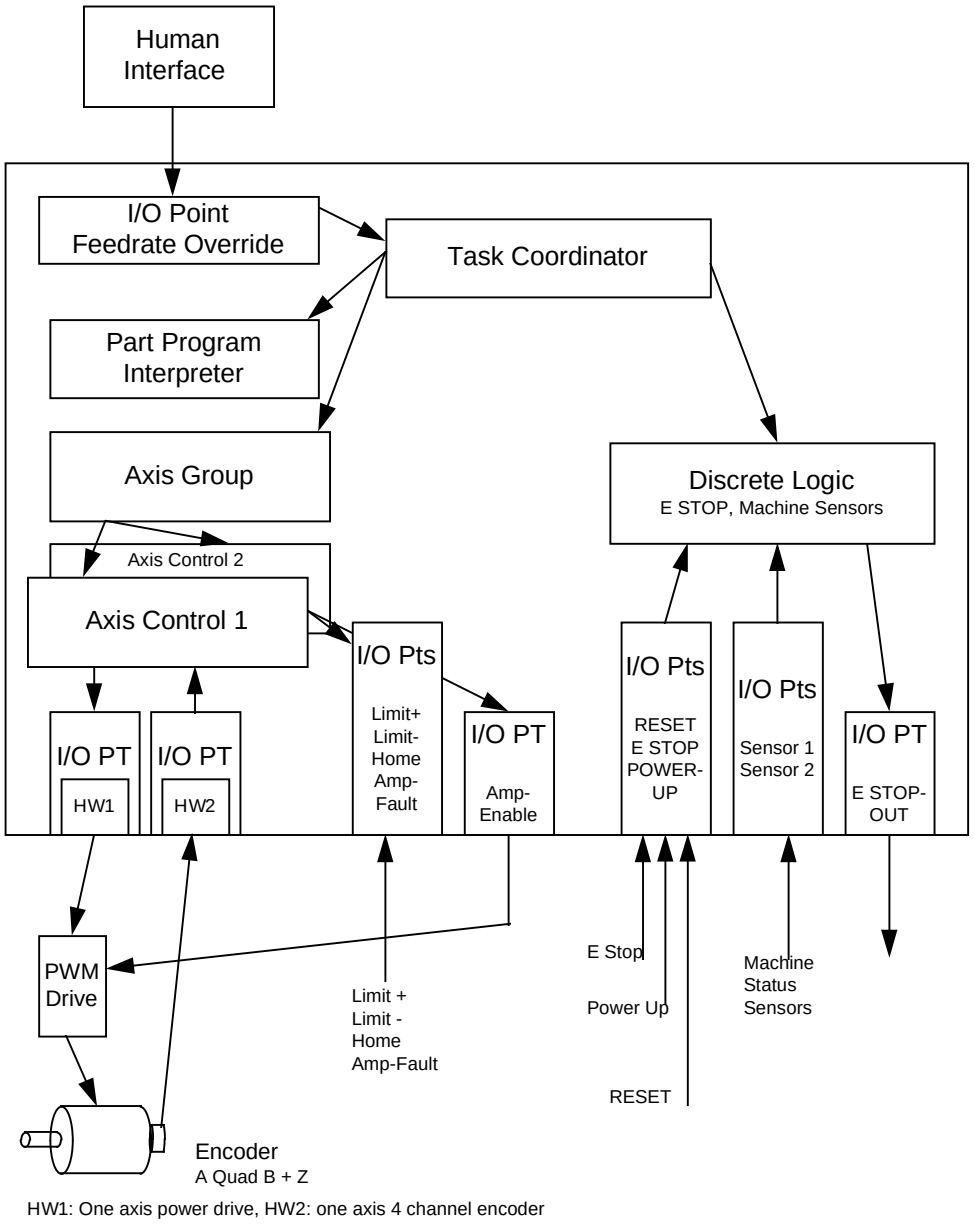


Figure 2. Example of an Controller

The application used for this purpose is programmed numerical control for a two-axis lathe. The 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. The spindle drive components are assumed to provide a facility for setting spindle speed and direction and to start and stop spindle rotation. The machine sensor system is assumed to consist of a set of analog and digital sensors monitoring coolant temperature and oil pressure. The machine safety system is assumed to consist of a set of input switches monitoring E-Stop, Power-Up and Reset. The 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. The machine part programs are assumed to be in EIA RS274D format. The control pendant is assumed to display machine status to an operator including indication, machine modes, program status and machine diagnostics.

This application is used as a reference example for open architecture modular controllers. Some possible modifications to the reference application that would be instructive are:

- add tachometer feedback to the axes/add IO modules/control loop module;
- use SERCOS drive instead of PWM/replace IO modules/replace control loop module;
- add secondary linear slide/add IO modules/add control loop module/ replace axis group module;
- add spindle position control/add IO modules/add control module/replace axis group module;
- add tool changer/add IO modules/add logic control module;
- add error compensation/replace IO modules;
- add probing capability/add IO modules/add logic control module;
- add closed loop scanning probe/add IO modules/replace module;
- add acoustic emission sensor for tool breakage/add IO module/add logic control module;
- add real time image recognition of surface finish/add IO add logic control module;
- add LAN communications functions to support part program upload part program download, remote status monitoring and remote data acquisition;
- replace human interface face module with industry de facto standard operator interface;
- replace part program interpreter to support CL data generated from AutoCad or ProEngineer.

The figure shows seven major systems that make up the components of open architecture modular controller. These are the IO system, control loop system, logic control system, axis group, part program interpreter system, human interface system and the task coordinator system. Each system is made up of one or more replaceable modules. The 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. The three mechanisms for adapting the system are to add modules, to replace modules and to reconfigure modules by reconnecting the interfaces

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

The control loop system consists of one or more axis control loop modules. Each axis control loop module requires two or more IO module interfaces. These represent sensor input and actuator output. Each axis control loop module provides a command interface that is normally connected to an axis group module. Control loop 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. Control loop module operation is to compute an algorithm to generate a new actuator command based on current sensor readings, commanded set points and machine state.

The logic control system consists of one or more logic control modules. The system normally contains a large number of logic control 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 modules may provide an interface that would normally provide status information to the human interface module. Logic control modules may also provide an interface to be used by part program interpreter modules. Logic control module operation is distinguished from loop 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.

The trajectory generation system consists of one or more axis group modules. 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 module provides at least two interfaces one of which is normally connected to either a task coordinator module or part program interpreter, and the other of which is normally connected to the human interface to provide manual jog capabilities.

The part program interpreter system consists of one or more part program interpreter modules. A part program interpreter module requires one or more trajectory modules interfaces. A part program interpreter may also require one or more logic control 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.

The human interface system consists of one or more human modules. The collection of modules in the human interface system will normally require at least three interfaces - an interface to a part program interpreter, an interface to an axis group for manual jog capabilities and one or more IO module interfaces for low level interactions such as button activation, feedrate override, etc. The human interface system may also use any interfaces available for system modules to supply status information.

The task coordinator system normally consists of one task coordinator module. A task coordinator module may require a variety of logic control module interfaces to detect event transitions and evaluate system state. The task coordinator module operation is to system state and to schedule execution of system modules. A task coordinator may provide an interface normally used by the human interface module for machine status information.

TECHNOLOGIES ENABLING AGILE MANUFACTURING — INTELLIGENT CLOSED LOOP PROCESSING — Part 1: General Information

1 General

1.1 Scope

This part of the TEAM API specification constitutes part 1 of a series of specifications for intelligent closed loop processing and should be read in conjunction with the other parts of the series. This TEAM API specification applies to closed loop processing - including module interface programming; command, control and communication; infrastructure and system services; and the scaling of functionality based on selected equipment and desired application. The purposes of this specification are:

- to establish the definitions and identify the principal components relevant to closed loop processing;
- to specify the requirements for scaleable service, infrastructure, and resources;
- to define the behavior, communication and input/output as specified by an Application Programming Interface (API) for the closed loop processing modules;
- to give examples and application guidelines to the user.

1.2 Object of the Specification

The key objective of defining an API specification is to enable the design and implementation of an open, modular control architecture which:

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

- increases the ability of 3rd party software to interoperate with the long-term goal of true plug-and-play. For example:
  - 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 Module
  - 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 user-interface with third party package and configure identically to existing machine user-interface

1.3 Object of Part I

This part of the TEAM API specification gives the following information:

- the rationale and need for an Open Architecture specification;
- the basic requirements of an Open Architecture controller;
- the definitions and the principal characteristics relevant to a common set of open architecture module definitions. The term module will be considered equivalent to the notions of primitive and aggregated components and will be defined with class definitions.
- the steps in general Life Cycle to build a Open-Architecture controller;
- the roles of the major participants;
- the responsibility of the TEAM API working group;
- some examples to clarify the general API specification model.

1.4 Normative References

ISO/IEC 9506-1 1990, Industrial automation systems - Manufacturing Message Specification (MMS) - Part 1**:** Service Definition.

ISO/IEC 9506-2 1990, Industrial automation systems - Manufacturing Message Specification (MMS) - Part 2: Protocol Definition.

ISO 10303-41 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 41: Integrated Resources: Fundamentals of Product Description and Support.

ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 42: Integrated Resources: Geometric and Topological Representation.

IEC 1131-3 Programmable controllers - Part 1: General Information, Oct. 1992.

IEC 1131-3 Programmable controllers - Part 3: Programming Languages, March 1993.

NCMS (National Center for Manufacturing Sciences), “Next Generation (NGC) Specification for an Open System Architecture Standard (SOSAS), Revision 2.5”, August 1994.

EIA Standard 441, “Operator Interface Functions of Numerical Controls”, Electronics Industries Association, Washington, D.C., January 1979 (Reaffirmed July 14, 1992)

EIA Standard - EIA-274-D, Interchangeable Variable, Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines,” Engineering Industries Association, Washington, D.C., February, 1979

2 Definitions

The TEAM API Workgroup use the following definitions for the following terms. Terms that are more specific to closed loop processing are defined in the corresponding parts in order to make them self-readable.

Application Programming Interface

The protocol by which one interacts with a module.

Class

An abstract data type and inheritances. 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 3 objects described by that class.

Configuration

The specification of a module mapping it from a general solution set into a specific solution. Compare: integration.

Design

A description of a system model in terms of classes and APIs (consisting of member function descriptions associated with classes).

Integration

The capability to allow the connection and cooperation of two or more modules within a system. Compare: configuration.

Module

A piece of the system that is sufficiently defined such that it can be replaced by another piece, by a third party, with same service through the same interfaces, possible states, and conditions of state transitions;

Object

An instantiation of a class.

Object-Oriented Program

A collection of objects interacting through a set of published APIs (Application Programming Interfaces).

Plug and Play

The ability to replace a TEAM API specified module in a control system with a complying version from any supplier and achieve the same or better results.

Proxy Agent

A set of objects that allow the crossing of address-space or communication domain boundaries. 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.

3 Glossary

Terms already defined in other standards that are frequently used in this specification are listed in this clause for convenience and comprehension.

Component

Adopts the SOSAS concept of a reference architecture consisting of primitive and aggregate components. Components are defined as abstract building block elements that describe functionality and communication. The application architecture is built from these components. Components have the following attributes:

- responsibility;
- peer-to-peer or collaborative relationships;
- behavior (specific functionality encapsulated by the component);
- messages, that is, the complete set of specific instructions necessary for invoking all of the behaviors encapsulated by the component;
- Application Program Interface(s) or the interfaces a component uses specifically to access services provided by the SOSAS notion of an Open Systems Environment.

TEAM will use the term module to refer to both a primitive and an aggregate component.

4 Development and Integration Life Cycle Model

Openness provides benefits and savings through flexibility and extensibility - but openness alone does not achieve interoperability. Application programming interfaces for one vendor's open system will generally not run under another vendor's system. Openness is but the first step towards "plug-and-play" interoperability which in turn is dependent on some form of a standard. Requirements for a standard "open solution" include the ability to allow the development of controllers by users or system integrators who want to piece together their own systems component by component, modify the way their controller perform certain actions, apply their modifications to another controller, or start small and upgrade as they grow.

Control systems are built as a set of connected components that require assumptions as to the scope and operation of interdependent components. The development of a general set of control system components assumes that each module can span a broad range of applicability. To build an application system from the general component set, configuration of individual modules and the integration of modules must be specified, tested and evaluated in the development of "plug-and-play" systems. Describing the general API for a module is extensive. Additionally, describing the services and levels of efforts of dependent resources requires a Life Cycle in order to understand the roles and relationships of elemental and aggregate components in the developmental process. The TEAM API Workgroup has developed a Life Cycle that divides the TEAM API Development and Integration Process into five phases:

1. Module Definition Phase,
2. Module Creation Phase,
3. Module Configuration,
4. Module Integration, and
5. System Initialization.

More detailed steps of the TEAM API Development and Integration Process is shown in the series of figures 3-7. The TEAM API specification realizes that in addition to the standards developers - such as the TEAM API workgroup - there are other perspectives within the model:

- control component development
- system integrator
- users

Each perspective will be further reviewed within the Life Cycle.

4.1 Control Component Developers' Tasks

Control vendors provide component products and support for hardware or software modules. For control vendors to conform to an open architecture specification, they would be required to conform to several specifications including:

- module class specification;
- system service specification;
- customer specifications.

A mechanism similar to the NGC profile is required to describe the system service specification that would include such areas as platform capability, control devices, and support software. The system services describe the platform and infrastructure support (such as communication mechanisms) and the available resources (disks, extra memory, etc.) Computer boards in turn 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. Sensor-effector devices such as controller cards or drives, would subscribe to a general electro-mechanical classification and then provide a more detailed capability feature profile.

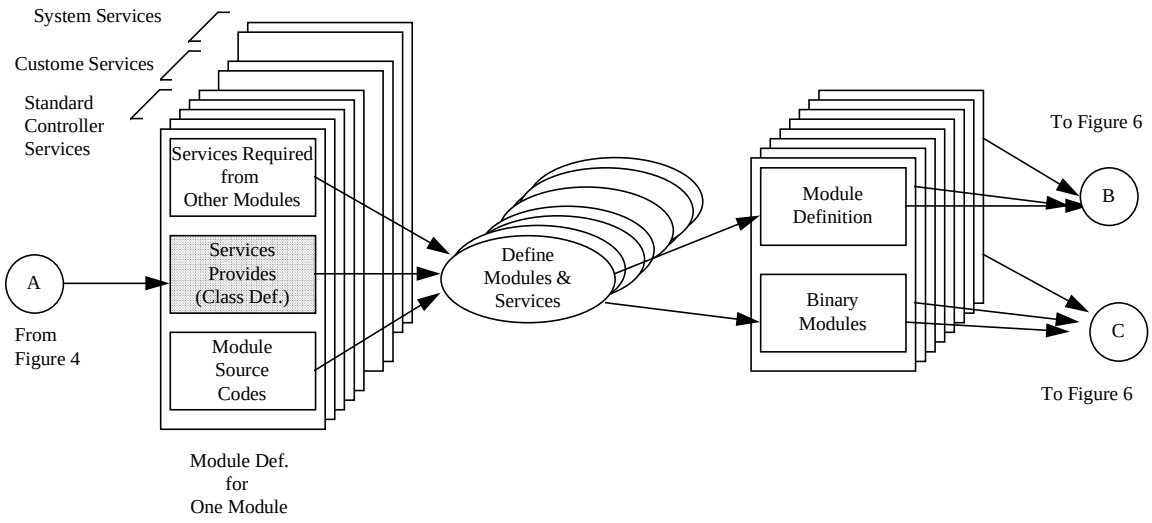


Figure 3. Life Cycle - Control Component Suppliers Tasks

4.1.1 Services Required from Other Modules

Created by:	(Control Component Providers)
Definition:	Parameterization and definition of services that are required from other modules
Fed into:	Define Modules & Services

4.1.2 Services Provided

Created by:	(TEAM API Specification)
Definition:	Specifications of a class
Fed into:	Define Modules & Services

4.1.3 Module Source Codes

Created by:	(Control Component Providers)
Definition:	Implementation of functions for a particular class
Fed into:	Define Modules & Services

4.1.4 Define Modules & Services

Input:	1. Services Required from Other Modules 2. Services Provided 3. Module Source Codes
Action:	**Generate definitions of the module and create binary codes from source codes**
Output:	1. Module Definition 2. Binary Modules

4.1.5 Module Definition

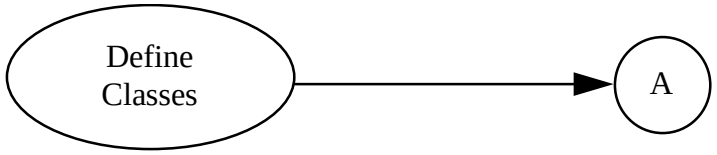
Created by:	Define Modules & Services
Definition:	**Includes the following information for a module:** • class definitions of the module • services provided by the module • persistent data in the module that needs to be initialized • creation information • services used by other modules • resources consumed and services required by the module (including operating system, etc.) class definitions of the module • system services supported by the module • other system services the module uses • response time benchmarks (e.g., published latency) • items requiring configuration (interrupt lines, buffers, etc.)
Fed into:	Build Module Database

4.1.6 Binary Modules

Created by:	Define Modules & Services
Definition:	**Binaries of the module (e.g., libraries or object files) ready to be integrated into a control system**
Fed into:	Configure and Integrate System

4.2 TEAM API Workgroup Tasks

The Workgroup recognizes that specifications, guidelines, and implementation examples must be developed before other parties can develop components of an open architecture system. The specification developed so far will concentrate its efforts in defining the necessary information and documents for **the Module Definition Phase**, and the resulting documents will be available for general review and comments. The effort necessary to develop documents for the other phases of the Life Cycle will be initiated if the feedback for the Module Definition Phase indicates a need to continue the specification efforts.



to Figure 3

Figure 4. TEAM API Life Cycle - TEAM API Workgroup Tasks

The tasks of the TEAM API Workgroup in the “Module Definition Phase” include:

1. defining a set of required classes and their services in a controller;
2. defining items within a module that need to be configured;
3. defining services of other modules that will be required by a module (integration information);
4. defining information that is needed to create other modules;
5. defining a standard description of configuration and conformance of a module.

The combination of each individual module’s configuration and conformance descriptions will be a full description of the control system. The infrastructure of a controller, that includes platform services such as timers, interrupt handlers, and inter-process communications, will be treated as a module. Specific interfaces to system services will be explicitly specified by the Workgroup.

4.2.1 Define Classes

Input:	(TEAM API Workgroup)

Action:	Create Class Definitions for Standard Controller Modules
Output:	Class Definition

4.3 Control System Integrators' Tasks

The control system is built from component parts as specified by the user (as based on market demand) by the system integrator. Many times the system integrator and the component builder are one in the same. However, to derive the major long-term benefits of standard open-architectures, it is necessary that the component builder and the system integrator have a clear separation within the Life Cycle. The control component builder provides binaries (as some form of an object library) from which the system integrator selects based upon the design criteria (controller performance: cost, accuracy, speed, reliability, tolerances, available parts, etc.). Figure 6 illustrates the concept of deriving two implementations based on different design and implementation specifications. The actual Commercial Off-the-Shelf (COTS) component set is a subset of the total available component sets.

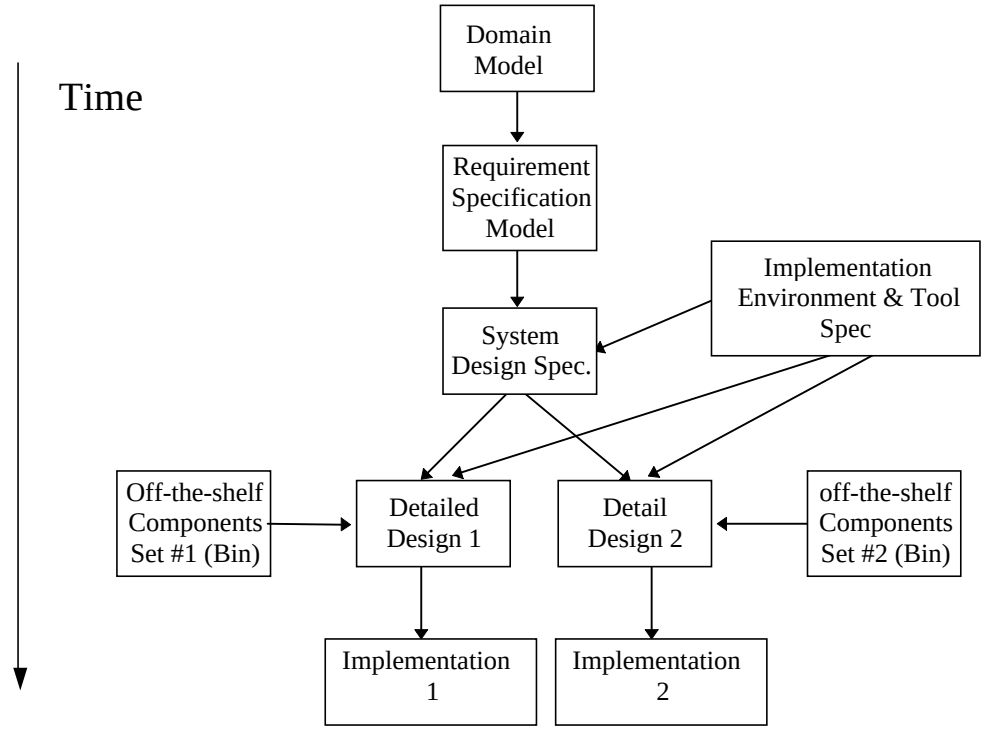


Figure 5. Controller Design Life Cycle vis a vis Module

The main responsibilities of the control system integrator include:

- Build Module Database
- Create Modules
- Initialize and Configure Modules
- Create System Resource Model
- Design System
- Configure & Integrate System

as illustrated in Figure 6.

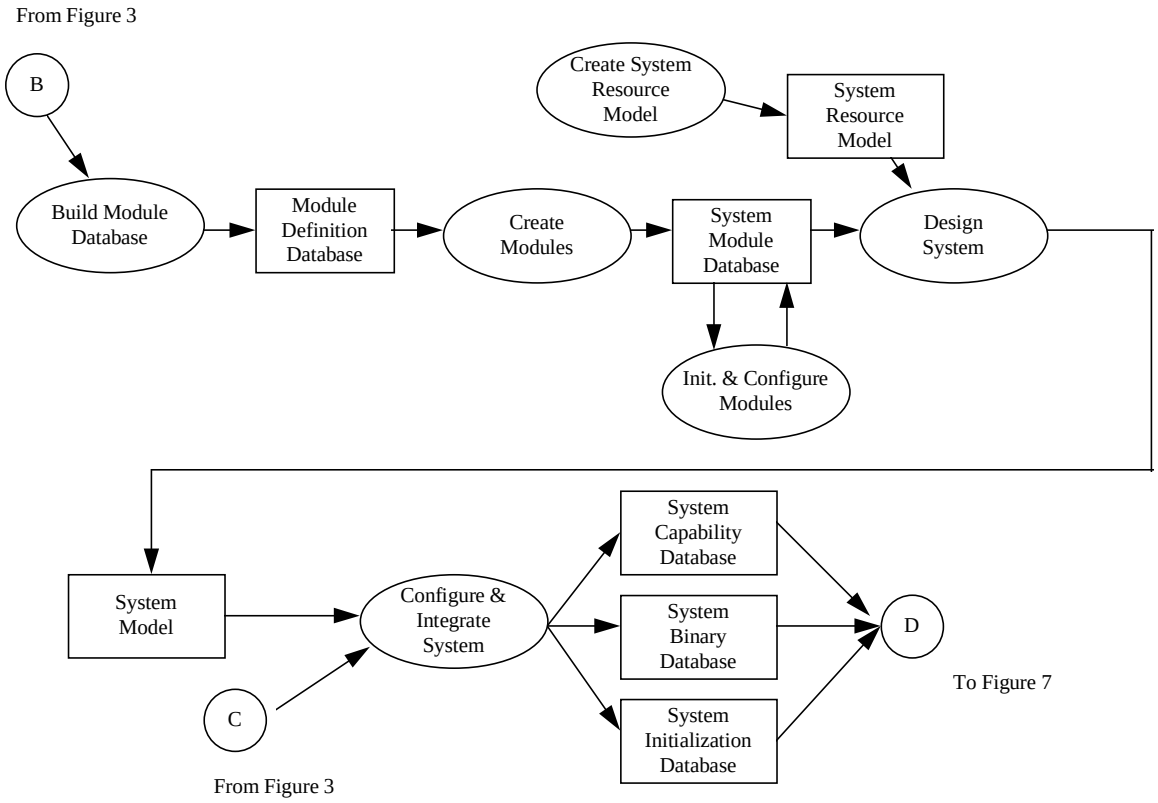


Figure 6. Life Cycle - System Integrator's Tasks

4.3.1 Build Module Database

Input:	Module definitions of all modules to be included in the system
Action:	Put information of all modules in a system into a database
Output:	Module Definition Database

4.3.2 Module Definition Database

Created by:	Build Module Database
Definition:	Includes module definitions of all modules in the system
Fed into:	Create Modules

4.3.3 Create Modules

Input:	Module Definition Database
Action:	**Select the subset and the number of instances of all modules participating in this controller** **Assign a name to each instance**
Output:	System Module Database

4.3.4 System Module Database

Created by:	Create Module
Definition:	**Contains information of modules contained in the system. Includes resource consumption information, services provided, services required, a reference to the binaries and information needed to configure the module.**
Fed into:	Create System Model

4.3.5 Initialize and Configure Modules

Input:	System Module Database
Action:	**Give initial values to persistent data for each module selected in the system**
Output:	System Module Database

4.3.6 Create System Resource Model

Input:	(Control System Integrator)
Action:	**Based on the hardware and software platforms selected, the control system integrator creates the information to be included in the system resource model**
Output:	System Resource Model

4.3.7 System Resource Model

Created by:	Create System Resource Model
Definition:	**Includes information on specified computing resources, storage resources (e.g., memory), communication channels, etc. for a particular implementation**
Fed into:	Create System Model

4.3.8 Design System

Input:	System Resource Model System Module Database
Action:	**Verify module resource consumption against system resource definition** **Allocate modules to resources** **Define interaction of modules to IPCs** **Assign timing information** **Create prototype implementation and test; iterate if necessary**

Output:	System Model

4.3.9 System Model

Created by:	Design System
Definition:	**Includes information on:** description of resource and performance assignments
Fed into:	Configure & Integrate System

4.3.10 Configure & Integrate System

Input:	System Model
Action:	**Execute a "make file"-like sequence** **Execute "compile" and "link"-like tools**
Output:	System Capability Database System Binary Database System Initialization Database

4.3.11 System Capability Database

Created by:	Configure & Integrate System
Definition:	**Contains processor and memory capabilities for load balancing; node, network and module connection information.**
Fed into:	Start System

4.3.12 System Binary Database

Created by:	Configure & Integrate System
Definition:	**Maintains information on load images, executables, libraries or system binaries**
Fed into:	Start System

4.3.13 System Initialization Database

Created by:	Configure & Integrate System
Definition:	**Contains startup information for each module in the system for initialization, includes initial values of init files of each module and persistent data**
Fed into:	Start System

4.4 End Users' Tasks

The user is responsible for creating application programs for the control system. The user can be expected to handle startup and shutdown operations of the controller. The user can be expected to test and debug application programs on the controller. Different classes of users can be expected. Some can be tasked with program generation, some with maintenance and others with operation. A run-time system configuration registry would be expected for handling the general

startup and shutdown sequencing and specific system customization. The end user's tasks, as illustrated in Figure 7, can be summarized as: start system, create application programs, and run system. Other user tasks are beyond the scope of this discussion.

.

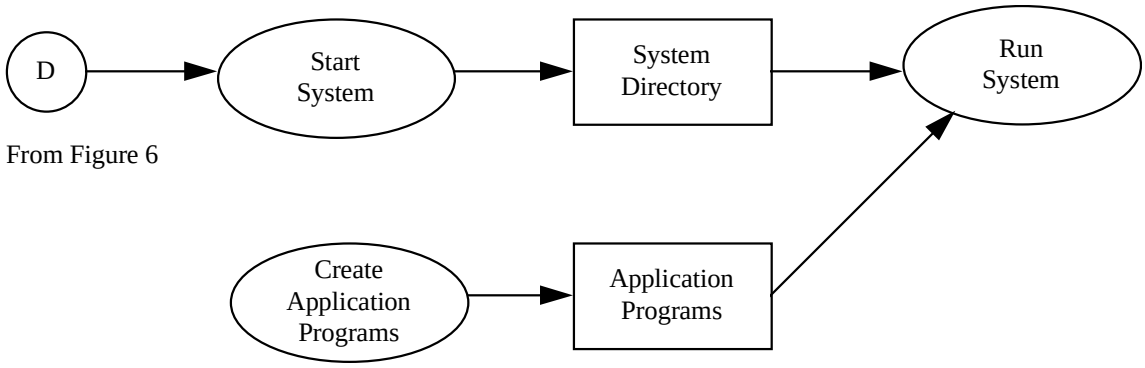


Figure 7. Life Cycle - End Users Tasks

4.4.1 Start System

At this time, an integrated controller that meets the requirements is available. The controller is integrated on a machine or equipment it will control. Desired operation sequences are specified in these application programs. The control system is also at a stage that it can be modified, purchased, and tested by users, and integration of sensors, video, Autocad, or other commercial packages and equipment is doable.

Input:	• System Capability Database • System Binary Database • System Initialization Database
Action:	Test and debug control system
Output:	System Directory

4.4.2 System Directory

Created by:	Start System
Definition:	Directory of all names in the integrated controller
Fed into:	Run System

4.4.3 Run System

At this stage, controller has been tested and debugged, and user programs for both discrete and motion applications can be executed.

Input:	• System Directory • Application Programs

Action:	Execute Application Programs
Output:	(Applications)

4.4.4 Create Application Programs

Input:	(End Users)
Action:	Write application programs
Output:	Application Programs

4.4.5 Application Programs

Created by:	Create Application Programs
Definition:	Application programs
Fed into:	Run System

Annex A

(informative)

Specification Modeling Overview

The modeling strategy is to use component based technology for integration of off-the-shelf components. This strategy implies the need for strongly defined Life Cycle considerations - design, configuration, integration and extensibility. Interface Definitions use the ROSE class definition format. ROSE is a CASE tool from Rational Software Corporation, that generates C++, ADA or Smalltalk code. The Object Management Group Interface Definition Language may be used in follow-up modeling work. Validation of the initial API specification models will be done on several testbeds (e.g., NIST Enhanced Machine Controller testbed, and the University of Michigan EECS).

A.1 CONFIGURATION - Base and Derived Class API Strategy - Subclassing

A major obstacle to defining a suitable specification is limiting the scope to a reasonable level of effort. To achieve reasonably general models, subclassing will be used to scale the interfaces. The specification model will start with a minimum API specification and then extend the interface to meet specific needs, e.g., PID, FF, or Neural Net.

For example, suppose we have a loop closure module as represented below in Figure 8:

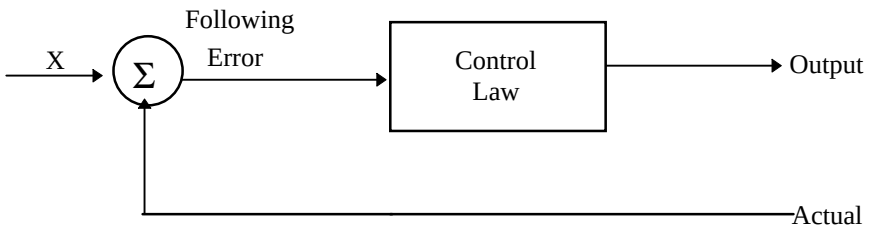


Figure 8. Minimum or Base Class for Loop Closure

Within this example, the interface has the following io points: *Output*, *Actual*, and *X* which have the associated functions to manipulate the contents:

- *X* has *set_command, get_command*
- *OUTPUT* has *set_output, get_output*
- *ACTUAL* has *set_actual, get_actual*

A class function definition such as *close_loop*(), is required to trigger the control law to update the current set point. One specializes (or derives in C++ terminology) a application module from this general loop closure with a specific control law such as PID or FUZZY logic. A PID module would then inherit the functionality from the loop closure base class and derive its more specialized control law functionality, so that one sees:

$$PID_{module} = LoopClosure_{Inherited\ functions} + PID_{Subclass\ functions}$$

where the PID adds the following io points to the loop closure base class: *X dot, X dot dot, Command Offset, Output Offset, Scaled Output, Following Error, etc.* as illustated in Figure 9. The subclass functions to manipulate the contents of these IO points are: *set/get_dt_command, set/get_dt2_command, set/get_command_offset*, etc. which matches wires in/out of the PID.

To set the gains and scaling parameters that effect the control law computation (e.g., P, I, D, and K_{VF}), one would use the *set/get_PID_prop_gain, set/get_PID_int_gain, set/get_PID_der_gain,* and *set/get_dt_command_gain.* Once again the class function definition *close_loop*(), is used to trigger the control law to update the current set point, and a series of class functions such as *scale_command_offset()* would now be required to trigger the associated control law scaling functions.

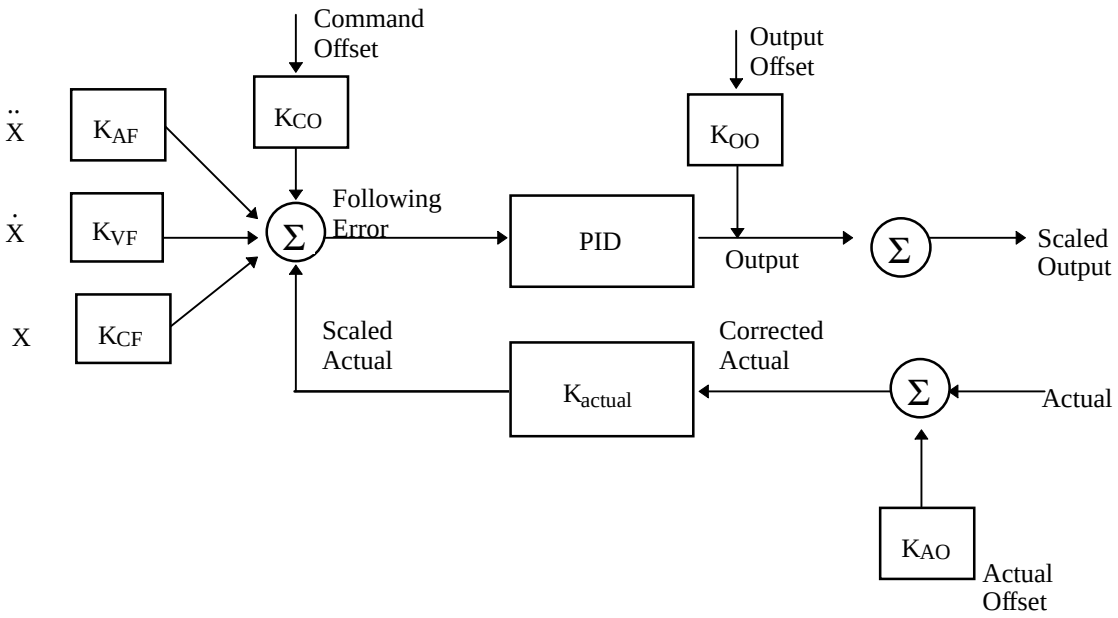


Figure 9. PID Derived Class Line Loop Closure

A.2 INTEGRATION - Module Connections

Effective integration of modules is fundamental to the open architecture. The ability to build a controller of integrated components is similar to piecing a puzzle together. [2] We will assume a controller consists of modules that are either standalone processes with a communication channel for a connection; or are constructed as a cyclically executing task (ibid, agent) using component modules. In general, one must assume that a general framework is available in which to build a controller. Figure 10 illustrates one example framework in which component pieces are used. Within the example framework, there are several useful capabilities. First, substitution of a different component piece within the framework puzzle - such as a closed loop module - is key to plug and play solutions. Second, the ability to rewire an IO connection, such as that between the segment and set-point generator, to use either a direct message connection or insert an extra component piece to allow indirection through some distributed communication mechanism, such as shared memory mailbox or socket. A potential rewiring solution could use different Dynamically Linked Libraries DLL) so that one library does a direct class method call while the other library uses PROXY AGENTS so that when a class method is invoked it understands to access shared memory.

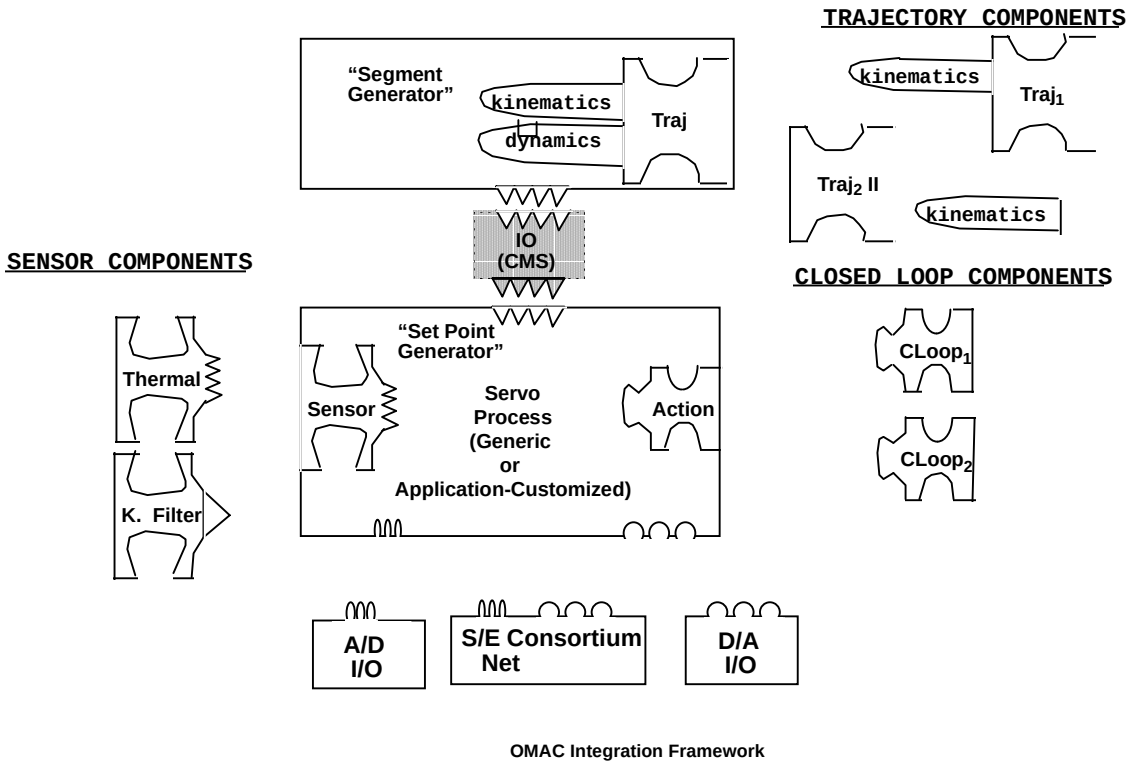


Figure 10. Integration Framework (Draft)

[2]. This analogy is drawn from the Taligent OS concepts. See: ``Building Object-Oriented Frameworks", White Paper, Taligent, Inc. 1994

A major objective of the specification is to allow the addition or change of control laws and control loops. As an attempt to meet this objective, we will illustrate how to build a one-axis "closed loop process" that cyclically executes. We will suppose that loop_closure modules are commercially available that meet our controller environment profile. We will further assume that API definitions will resolve the naming and instantiation conventions (either direct or with proxy agents) for object classes **io1**, **loop_closure1**, **axesgrp1**. Then, we can write the following code to produce our "servo" process. Once we attach a thread of execution (such as timing, priority, stack size, etc.) we have built an agent. The servo process reads and writes to all the relevant io connection points, and at the same time calls the loop_closure module to generate the next set point. We will assume that all the io connection points have been established by another routine.

```
// Most basic cyclic process - could be C++ template
closed_loop_process(){
// Read the current encoder value with IO system
io1.get_encoder(&value)

// Set the next actuator value
loop_closure1.set_actuator(value)

// Get the next value from the axis group
axesgrp1.get_axis1(&value)

// Use traj generator value to set next  closed loop command
loop_closure1.set_command(value)
...

 // Use state to cause module to compute next value
if(state == running) loop_closure1.close_loop();
 ...

// Read CLC commanded output after it finishes cycle
 loop_closure.get_output(&value);

 // Put out value to DAC, (scaling done by io system)
 io1.put_DAC1(&value);
}

// initialize parameters
closed_loop_init(){ }

// read commanded modes, change if necessary
closed_loop_mode(){ }
```