SANDIA REPORT

SAND96–2499 • UC–705
Unlimited Release
Printed October 1996

An Intelligent CNC Machine Control System Architecture

David J. Miller, Clifford S. Loucks

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

DISCLAIMER

SAND96-2499
Unlimited Release
Printed October 1996

Distribution
Category UC-705

An Intelligent CNC Machine Control System Architecture

David J. Miller, Clifford S. Loucks
Intelligent Systems and Robotics Center
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

Intelligent, agile manufacturing relies on automated programming of digitally controlled processes. Currently, processes such as Computer Numerically Controlled (CNC) machining are difficult to automate because of highly restrictive controllers and poor software environments. It is also difficult to utilize sensors and process models for adaptive control, or to integrate machining processes with other tasks within a factory floor setting. As part of a Laboratory Directed Research and Development (LDRD) program, a CNC machine control system architecture based on object-oriented design and graphical programming has been developed to address some of these problems and to demonstrate automated agile machining applications using platform-independent software.

ACKNOWLEDGEMENTS

This project resulted from the efforts of several people. The design, implementation, and testing of the VME-based subsystem software was done by Mike Rogers. The design, implementation, and testing of the graphical programming environment was done by Jim Pinkerton. Machining expertise was provided by Cliff Loucks. Pro/Engineer and Pro/Manufacture CAD files and tool paths were generated by Terry Smith.

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

1.1 Executive Summary

Intelligent, agile manufacturing relies on automated programming of digitally controlled processes. Currently, processes such as Computer Numerically Controlled (CNC) machining are difficult to automate because of highly restrictive controllers and poor software environments. It is also difficult to utilize sensors and process models for adaptive control, or to integrate machining processes with other tasks within a factory floor setting. As part of a Laboratory Directed Research and Development (LDRD) program, a CNC machine control system architecture based on object-oriented design and graphical programming has been developed to address these problems and to demonstrate automated agile machining applications using platform-independent software.

This architecture includes a VME subsystem to control any CNC machine using an object-oriented generic programming language that is EIA-274D compliant. Serial and quadrature encoder interfaces from the VME subsystem to a Fadal Inc. 5-axis milling machine are used to command this testbed commercial controller and receive axes position feedback. A workstation-based graphical programming environment incorporates a menuing system, IGRIP simulation of kinematically correct 3D models of the CNC machine, and a supervisory control program for communication with the subsystem.

A user designs parts and generates initial tool cutting paths using standard CAD/CAM packages such as Pro/Engineer and Pro/Manufacture. These models and paths are imported into the graphical programming environment to set up a virtual IGRIP-based CNC work cell for interactive tool path editing and simulated machining. When satisfied with the resulting verified machining script, the user then calibrates the virtual work cell to the actual setup, downloads the generic script and calibrated tool paths to the subsystem, and executes the machining operation. This script can be archived for future production runs.

1.2 Background

Previous Sandia research has shown a very high payoff in productivity and software reliability resulting from the use of a generic language for the programming of intelligent robot applications. This is the Robot Independent Programming Environment (RIPE) and Language (RIPL). Use of RIPE permits application programmers to use a uniform methodology for structuring the software systems for intelligent robot systems. RIPL is the common language for expressing commands to intelligent robot systems. Use of RIPL enables wide reuse of code across different applications. As part of the Laboratory Directed Research and Development (LDRD) program, this concept of a generic language has been extended for manufacturing processes involving devices other than robots, including Computer Numerically Controlled (CNC) machining centers.

There are several major features in RIPE which contribute to the success of this technology. RIPE models an intelligent system as a set of software classes. A class is a complex data structure which defines all of the attributes and behaviors of whatever entity it logically represents. Classes can therefore be defined for devices such as robots, machining workstations, and sensors, as well as

for "virtual"concepts such as communication handlers and world models. By defining libraries of classes for all components of an intelligent system, it is possible to hide low-level device integration and communication details from end users by encapsulating those details inside each class, and then providing a uniform interface to the class through a higher-level language. Within this context, we have defined the Robot Independent Programming Language (RIPL).

RIPE currently divides a manufacturing work cell into three generic classes, *Workpiece*, *Station*, and *Device*. This is derived from the concept that devices carry out actions on work pieces, and stations are locations in the work space for storing these devices or work pieces. When a *Station* or *Workpiece* object is created in an executing application program (an object is a particular instance of the class), database information is used to "fill in the slots" with the attributes of the particular station or work piece being modeled and used by the program. The *Device* class, on the other hand, is expanded into a hierarchy of subclasses for the different kinds of devices normally found in an intelligent manufacturing system. Active devices which have the property of being able to move or transport a work piece or tool are derived from the *Transport* subclass. Transport devices include robots, computer numerically controlled machining centers, conveyors, translation tables, or autonomous vehicles. Passive devices which are manipulated by the active devices are derived from the *Tool* subclass, which is further divided into particular subclasses of tools, such as a *Sensor* or *Grabber*. A generic set of messages or commands are defined for each of these generic classes, and these messages constitute the Robot Independent Programming Language (RIPL). Below these generic levels, subclasses are created for the specific devices used by a particular system. These devices are programmed using the same RIPL commands defined at the generic level.

Because RIPE is object-oriented, it also shares all of the advantages of object-oriented technology as applied to intelligent machine systems. Since RIPE is organized around class representations of the objects in a manufacturing work cell, its structure reflects the physical structure of the system. This aids system integration because it is possible to build a software application in parallel with the hardware, therefore allowing software engineers to better communicate with the hardware system integrators during development. This also controls complexity because each object is defined and tested independently of the application and is known to be reliable before it is used.The application simply creates, combines, and manipulates these well-behaved objects through RIPL commands to perform the specific tasks of the system.

In addition, object-oriented design concepts such as inheritance and polymorphism allow software reusability, extensibility, reliability, and portability. Inheritance is the mechanism used to define the hierarchies of objects through subclassing, where a subclass inherits the attributes and behaviors of its parent class while extending its definition with specialized characteristics. RIPE defines a generic abstract base class such as *Robot*, and then extrapolates this by defining subclasses for specific types of robots. Through this mechanism, the software is extensible and reusable because RIPE extends the concept of a generic robot to a specific robot while at the same time reusing all of the software already written for the generic robot. Polymorphism is the mechanism used for implementing RIPL because it allows different objects derived from the same parent class (i.e. different types of robots) to respond to the same messages in an appropriate manner. This implies that a standard set of RIPL commands for a generic robot is defined, and

then this same set of commands is used to talk to any specific type of robot for which a subclass has been defined. RIPL commands have also been implemented for other types of classes as well, such as sensors, programmable tools, and communication handlers.

RIPL provides consistent interfaces and device independence for application code. Therefore, a device in the work space can be replaced without having to rewrite the application software. Also, an entirely different application can be implemented which will have a similar design structure and will use the same objects and same RIPL commands, but will perform very different tasks. All of these factors contribute to the speed, reliability, and cost of developing complex intelligent systems.

Current RIPE implementations have focused primarily on robots, force sensors, and various communication facilities and protocols. The work perfomed in this project exploits the RIPE methodology to develop new software classes and RIPL interfaces for another major component of a manufacturing work cell, namely the CNC machine. In addition, there is a need to integrate machining processes with other activities in a manufacturing facility. This integration is accomplished through the development of a Generic Intelligent System Controller (GISC) architecture.

The GISC concept was originally developed as part of the U.S. Department of Energy's Robotic Technology Development Program to design and implement prototype intelligent systems for performing hazardous operations. It is now being used for a variety of applications, including laboratory automation, painting of large structures, and agile machining. GISC is communication oriented and is based on the premise that sophisticated intelligent system performance is achieved by coordinating a collection of semi-autonomous subsystems, each with complementary capabilities. Each subsystem has a well-defined command-and-control interface, and a supervisory control program coordinates the overall activities of the system through these subsystem interfaces. Individual subsystems may also possess real-time low-level control functions which can be performed autonomously and asynchronously. With the right combination of supervisor and subsystem capabilities, such an approach supports the implementation of model-based control and sensor integration within reusable software structures. This approach also promotes the use of modularity, distributed multi-processing environments, and standard commercial interfaces. In order to build a GISC-based system, tools are needed for developing and integrating the supervisor and subsystems into a complete operational control system. Four such tool kits have been developed to provide a range of capabilities required at all levels of an intelligent system.

1.3 Application of the GISC Architecture to a CNC Machine

An application of the GISC architecture in the area of information-driven manufacturing involves the development of an intelligent CNC machine control system architecture which enables one to more fully automate the process from CAD design to finished part. The software implementation consists of a graphical programming environment coupled with a generic transport subsystem which controls a Fadal Inc. vertical machining center through a RIPL translator. The Fadal machine encoders are interfaced to the subsystem for real-time position tracking. In addition, a

touch probe and structured lighting system are also interfaced to the subsystem for part and fixture location.

A typical scenario for using the system begins with the operator opening a window onto his favorite CAD system and designing a part containing features which require machining. When the design is completed, CAD models for the finished part, raw stock, and fixtures are imported into a simulation environment such as Deneb's IGRIP. A kinematically correct model of the milling machine is available within this environment, and the operator performs the necessary setup of the virtual machine by interactively arranging the CAD models of the parts and fixtures in an optimal way for machining operations. The operator then imports a tool path from a package such as Pro Manufacture or interactively generates a tool path by using a space ball to maneuver the machine tool around the part. The system automatically records the motions which can be played back in a simulation mode to verify that there are no collisions and that an acceptable material removal sequence is being performed. When the operator has completed the generation of the program, he can then mount the actual parts and fixtures onto the selected machine bed and use a sensor such as the touch probe to locate the parts and fixtures with respect to the machine coordinate system. This information can be uploaded to the graphical programming environment which uses it to perform its own calibration process to accurately register the model with the real physical world. Then the tool paths derived from the previous simulation are automatically adjusted based on this calibration. Finally, the graphically generated program is downloaded to the generic transport subsystem and executed as a sequence of generic commands to machine the part.

2.0 CNC HARDWARE/SOFTWARE TECHNOLOGIES

2.1 General Requirements for Agile Machining

There are numerous requirements to be met in order to provide an agile machining capability. Although the architecture researched by this project only provides a starting point for meeting these requirements, one of the project goals was to make industry aware of what is needed for agile machining in order to encourage evolution toward such a capability. The following sections briefly enumerate some of these requirements.

2.1.1 Powerful Controller Architectures

- much faster processing of materials with higher speeds and lower forces (milling rates over 10 inches/sec and drilling rates over 6 inches/sec in aluminum)
- much faster non-machining motions (60 inches/sec and up to 4g acceleration)
- highly automated, high-volume production modes
- many axes operating at the same time in the same workspace (two 6-axis machine tools processing a workpiece mounted on a 3-axis table with access to two 3-axis tool changers, where 9 axes could be controlling a tool-to-workpiece motion within .0002 inch at a relative speed of 6 inches/sec)
- very accurate (in general, .0004 inch on a 20 cubic inch work space, with variations, depending upon types of possible errors and requirements)
- very reliable, running unattended overnight and on weekends (5 years mean time between failures that interrupt production)
- look-ahead processing to generate servo commands based on expected behavior, expected processing loads, and feedback of deviations from expected behavior and expected loads
- a .3 millisecond servo update rate
- ability to program and select from a range of algorithms for motion control
- fast, reliable communication interfaces to access external networks of computers for obtaining engineering information, work schedules, task completion status, etc.
- open, distributed, multiprocessing architecture with modularity and upgradeability
- selection of a system language and programming environment which is efficient yet allows one to develop complex software in a structured way
- standardization of automation interfaces:
  - position feedback devices and other sensory inputs
  - signals exchanged by servo amplifiers
  - communication
  - human interfaces
- intelligent software
- based on U.S. leadership in computer hardware, software, and analytical abilities

2.1.2 Ability to Perform Complex Machining Operations without Years of Training

- versatile or flexible (able to perform a variety of operations within a known range on a family of workpieces, requiring little programming effort specific to each workpiece)
- packaged sets of generic task-level machining operations which can be selected and

sequenced easily for a variety of applications

2.1.3 Utilization of Sensor Technologies for Real-Time Tool-Path Generation, Modification, and Monitoring

- built-in quality assurance to estimate and correct repeatable dynamic errors and slowly changing operational errors
- health monitoring of controller, tools, and current operations, with interruptibility:
  - propagate alarm signals in 0.1 millisecond when failures in the position feedback subsystem or other critical sections of the control system are detected
  - propagate alarm signals in 1 millisecond when other out-of-control conditions are detected
  - interrupt power to servo control amplifiers or initiate a feed-hold-mode within 1 millisecond of such alarms
  - initiate dynamic braking of all motors immediately after such alarms
- sophisticated position feedback capabilities:
  - ability to interface a controller to a variety of position feedback devices (rotary encoders,optical scales, laser interferometers, incremental as well as absolute position)
  - ability to interface multiple position feedback devices per axis (a feedback device for control and a device for monitoring)
  - much higher ratio of peak velocity to measuring resolution (up to 20 million counts per second)
  - real-time tool-path modification based on position feedback signals (extraction of mean position at a given instant with vibrations filtered out; applying smoothing algorithms and feeding results back into the servo loop; maintaining time histories of position measurements and synchronizing them with other signals)
- fast acquisition and processing of sensors that measure forces, vibrations, acoustic emissions, and angular displacements for monitoring and altering equipment, tool, and process behaviors

2.1.4 Utilization of Solid Modelers, Databases, and Graphical Programming to Realize the Art-to-Part Concept

- framework in which the kinematic and dynamic models of the system can be created, stored, and updated
- models of motion error and those induced by temperature distribution, tooling, and loads
- knowledge of how to automatically calibrate and characterize the behavior of the system
- knowledge of equipment capacity, degradation models, life expectancy models
- libraries of form features to be machined
- libraries of cutting operations (cycles or macros) and ranges of feeds and speeds to machine these form features
- models of process loads for machining these form features which can be used to plan precompensated motion commands
- models of tool degradation and life expectancy
- libraries of geometric, structural, and parametric models of cutting tools, fixtures, and families of parts, including tolerances, and the sequence of operations to produce them (relating them to the form features and cutting operations above)
- having a common, efficient, easy-to-use, easy-to-learn programming environment for defining the sequence control logic, continuous motion control, other servo controls, monitoring functions, generating visual indicators of status, generating alarms, etc.,
- software toolkits for:
  - program generation which combines knowledge and data of the machine, its controller, processes,

and products
- emulation of various parts of the control system
- animated simulation of operations
- diagnosis/troubleshooting

- a user interface which provides access to any level of the control system, access to external computer systems, color graphic displays of system state and flow of of operations

2.1.5 Integrated environment incorporating all of the above technologies

2.2 Agile Machining Advantages

The following is a list of some of the advantages of agile machining:

1) Automated programming (CAD design to finished part)
2) Short cycles (speed development, minimize programming errors, cut costs)
3) Small learning curve (reduce skilled shop floor labor)
4) Process control (increased accuracy)
5) On-line verification (fewer reworked or scrapped parts)
6) Concurrent design (reduce design change time and eliminate dry runs)
7) Responsiveness, scalability (increased part complexity, variety, variable lot sizes)
8) Standards (tooling, fixtures, software interfaces)
9) Rapid reconfiguration (cost-effective prototyping, smooth transition to production)

2.3 Future Integrated Technologies

While significant advances have been realized in recent years in areas such as metal part fabrication and inspection, integration of these recent technologies into a single processing cell has been limited. The use of separate forging, measuring, and machining centers remains the norm. This situation is not lost on industrial suppliers of machining and inspection systems, yet they face a conservative market which is painfully slow to accept technological advances, thus limiting their resources directed toward advanced system integration. A single machining/inspection cell incorporating recent, proven technological advances would provide an efficient, productive means to realize waste minimization. Such a system would also provide the capability of performing machining operations on cast parts which, due to variances associated with casting and forging processes, are not compatible with today's highly structured machine tool programming and operating environments.

The development of such an integrated system requires the marriage of sensing, data interpretation, and real-time control into a package understandable and acceptable to an "average machine tool operator." Currently available on-machine inspection techniques are limited to collecting a single data point at a time. These processes are based on contact probe or simple laser triagulation techniques which can only resolve a single data point per machine axis motion. The use of current structured light techniques enables data collection rates much more suitable to the continually advancing processing power of today's computers.

Process control, a technique employing feedback control of the input versus the desired output of

a system, has yet to find significant inroads into the machine tool industry. As advanced as today's CNC machining centers appear, only a few offer anything in the way of process control. Referred to as adaptive machining, these controllers provide automatic modulation of feedrates based on sensed spindle torque. This technique provides some benefit in compensating for process variables associated with tool wear and casting uniqueness, but is of limited use since it provides a scalar measurement of what is truly a 3-dimensional process. The application of real-time sensing/control of tool contact force into a machining center would allow the accepted use of non-traditional tooling (shaped cutters designed to create radii, chamfers, and simple edge breaks) as well as providing a means of process control and early (salvagable) error detection (and waste minimization).

Another area of development necessary for the acceptance of productive process control and inspection techniques into a single machining center lies in the system's operator interface. Current CNC controllers rely on "G Code" programming which is adequate for 2D and 2.5D features, resulting in a small percentage of in-production machines producing parts with complex 3D contours, which are programmed only by those few capable of generating "true 5-axis" tool trajectories. Providing the NC programmer with the utility of specifying real-time parameters like desired tool contact force would require an enhanced machine tool programming environment.

3.0 A GISC-BASED CNC CONTROL SYSTEM ARCHITECTURE

3.1 Toolkits For Building a GISC-Based System

As mentioned in the introduction, in order to build a GISC-based system, tools are needed for developing and integrating the supervisor and subsystems into a complete operational control system. Four such tool kits have been developed to provide a range of capabilities required at all levels of an intelligent system. These include:

1) the GENISAS tool kit which provides the communication facilities needed for the distributed supervisor/subsystem paradigm;
2) the RIPE/RIPL tool kit which enables development of generic subsystems by providing object-oriented interfaces to intelligent system devices;
3) the SMART tool kit which enables development of underlying control systems that provide the performance and flexibility for sensor-based control and teleoperation;
4) the Sancho tool kit which provides for easily reconfigurable menu-based operator interfaces and a dynamic simulation environment.

Figure 1. conceptually illustrates how a GISC-based system is organized with respect to these tool kits.

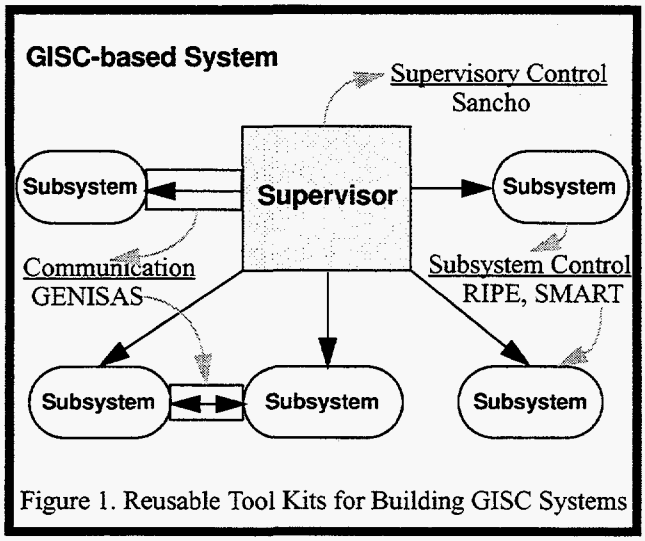

Figure 1. Reusable Tool Kits for Building GISC Systems

3.1.1 GENISAS

One of the key elements of any distributed intelligent system architecture is a powerful communication mechanism. The General Interface for Supervisor and Subsystems (GENISAS) is a client/server-based tool kit which provides general communication software interfaces between a supervisory control program and semi-autonomous subsystems, such as those which would be defined in a GISC-based system. There are four main components comprising the tool kit. The first component consists of low-level communication and utilities libraries which are provided to support

reliable transmission of atomic messages and virtual multi-channels for commands, data, status, and exceptions. The next two components include supervisor (client-based) and subsystem (server-based) command and event processing libraries. Finally, there are facilities for message construction, parsing, and conversion. All of these libraries provide capabilities which allow the user to define command sets for table-driven command processing between supervisor and subsystem, data transfer requirements based on single point of control, events for asynchronous processing, and symbol manipulation.

The tool kit uses an object-oriented approach to define standard client and server base classes implemented in the C++ programming language. Through inheritance, application-specific subclasses can be derived. The base classes supply all of the supervisor-to-subsystem communication facilities. The subclasses, which are normally defined by the user, provide the specific command sets and command implementations for control of a particular subsystem, such as for a manipulator or sensor subsystem.

3.1.2 RIPE/RIPL

Another tool kit, the Robot Independent Programming Environment and Robot Independent Programming Language (RIPE/RIPL) , is the culmination of one of the earliest efforts to apply object-oriented technologies to building robotic software architectures. RIPE models the major components of a system as a set of C++ software classes. It consists of two main class inheritance hierarchies, *Device* and *CommunicationHandler*. The *Device* hierarchy contains subclasses for different kinds of devices normally found in an intelligent system. Active devices which have the property of being able to move or transport a tool or work piece are derived from the *Transport* subclass. Transport devices include robots, CNC machines, conveyors, translation tables, or autonomous vehicles. Passive devices, which are manipulated by the active devices, are derived from the *Tool* subclass, and *Tool* is further partitioned into particular types of tools such as *Sensor* or *Grabber*. The *CommunicationHandler* class hierarchy defines different ways of communicating with these devices, including serial, parallel, or network-based message passing. A clear separation is maintained between device class implementations and communication interfaces. Figure 2. illustrates the inheritance hierarchy for *Device*.

A generic set of object messages or "commands" are defined for each of the abstract base classes, and these messages constitute RIPL. For example, a generic set of RIPL calls is defined for the *Robot* class, and these commands are used for all robots. RIPL object messages are implemented as methods of the robot subclasses defined for each robot type. These subclass implementations serve as "translators" from the generic language to the robot-specific control environment. Implementations are obviously different for different vendors, but the interface is the same. Inheritance and polymorphism are used to associate these generic messages with each subclass defined for a particular robot type, thereby providing a mechanism for generically programming any robot for which a RIPE subclass has been implemented. The entire RIPE/RIPL tool kit is packaged as a set of class libraries.

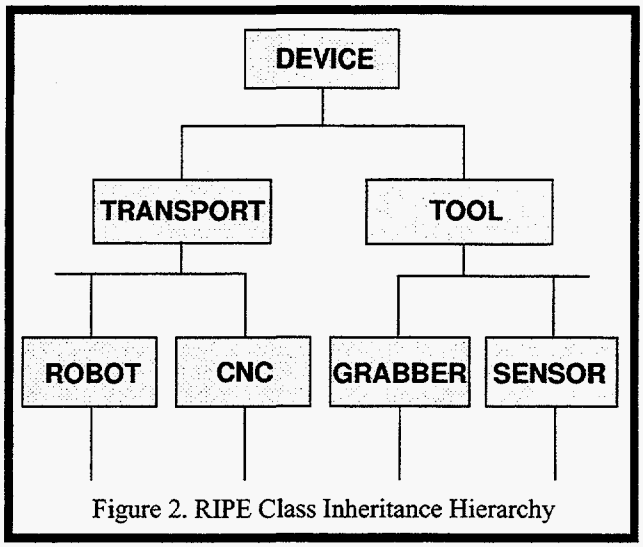

Figure 2. RIPE Class Inheritance Hierarchy

3.1.3 SMART

For low-level control of actuators and sensors, a third tool kit called SMART (Sequential Modular Architecture for Robotics and Teleoperation) provides the capabilities required for stable autonomous and teleoperated closed loop feedback control. This tool kit can be used with any robot that is capable of accepting external position set points, and it can be used with any sensor that has a VME-based interface. The tool kit consists of a collection of C language libraries, each of which defines an interface to a distinct system "module" such as a sensor, actuator, input device, or kinematic/dynamic element. These "modules" can be asynchronously distributed across multiple CPUs and can execute in parallel with individual fixed-rate servo loops ranging from 100Hz to 1KHz.

SMART is based on 2-port network theory in which each module has a network equivalent. For example, inductors represent inertia, resistors represent damping, capacitors represent stiffness, and transformers represent Jacobians. Modules are connected to create a complete circuit which represents a control system. Typical modules include trajectory modules, kinematic modules, robot joint modules, sensor modules for force control and compliance, and input modules for space ball teleoperation or force reflection. To use the tool kit, an application must define description files which indicate the number and types of modules to be used, how they are distributed, how information is passed between them, their period of operation, and appropriate filter constants.

3.1.4 Sancho

Sancho, a workstation-based tool kit, provides a GISC supervisory control program coupled with interface libraries which connect this supervisor to a graphical programming environment. This environment includes a menuing system based on X-Windows. Through these menus, an operator can command tasks and control the state of the system. Multiple active menu palettes allow for operations to be initiated in parallel. Communication objects from the GENISAS tool kit are used

internally by the supervisory control program to connect it with an appropriate GISC subsystem such as a manipulator subsystem.

The functions performed by the menus are reconfigurable through ASCII file definitions, thereby allowing the supervisory control program to be reused for controlling different subsystems. A simulation interface library also provides facilities for the operator to execute a commercial simulation package such as Deneb's IGRIP. The operator can then interact with the work cell models that are loaded into this environment in conjunction with the menuing system and supervisor. The simulation environment is also linked through GENISAS to the real-time control system, providing for dynamic model updating and position tracking.

3.2 Generic Tool Kit Interfaces

Each of these tool kits, aside from the supervisory control program, can be used completely independently of each other. This implies that they can be used and reused to implement robotic systems based on paradigms which are different from the GISC concept. On the other hand, by integrating them, a very powerful environment can be created for building intelligent system applications which are based on GISC.

This requires the development of interfaces between the tool kits which allow them to maintain their autonomy and, at the same time, allow them to interact with each other according to the GISC philosophy. Such interfaces have been developed, and complete intelligent control systems have been implemented. These systems utilize the tool kits to perform tasks related to problems in such diverse areas as waste remediation and information-driven manufacturing.

3.2.1 Sancho to GENISAS Interface

Beginning at the operator interface level, the supervisory control program provided with Sancho automatically supplies an interface to the GENISAS tool kit because its function is to control the subsystems required for a particular application. This interface includes menu callback routines which use GENISAS client objects and their associated messages to communicate appropriate commands to the available subsystems. The set of commands, as reflected by the menuing system, may be application-specific. However, as mentioned previously, the command set can be easily changed through ASCII configuration data files. Similarly, the menuing system can be interactively redesigned in order to meet customer specifications. Both of these tailoring operations can be performed with minimal programming effort.

The other tools in Sancho provide an interface to Deneb's IGRIP simulation package which is simply treated as another GISC subsystem. If a different simulation package is selected for an application, then a new interface library must be implemented. However, the application programmer's interface between the supervisory control program, menuing system, and the simulation environment should remain the same. Only the underlying simulation interface library implementation must reflect the requirements of the particular simulation package used.

3.2.2 GENISAS to RIPE/RIPL Interface

The next required interface is between GENISAS and RIPE/RIPL. This interface occurs at the subsystem level and is relatively straightforward since both tool kits are object-oriented and implemented as C++ class libraries. A GISC subsystem is normally controlled by a server process which is defined as a subclass of the GENISAS *StdServerProcess* base class. It therefore inherits all of the communication facilities required by any server. This subclass also defines the methods which implement the command set associated with the subsystem it services. These methods, in turn, are implemented by using RIPL methods defined for the device or devices controlled by the subsystem. The integrated use of RIPE with GENISAS allows for distribution of RIPE objects across multiple CPUs and environments, and provides an ASCII-based script file interface which translates into C++-based RIPL methods.

3.2.3 RIPE/RIPL to SMART Interface

The interface between RIPE/RIPL and SMART is somewhat complex due to the asynchronous, distributed nature of the underlying SMART modules. This interface has two primary components, one associated with the server subclass and one associated with the RIPL methods used by the server. Normally when a subsystem is booted which uses SMART, the desired SMART modules are automatically downloaded as part of a startup script, and numerous tasks associated with them are spawned. The number, type, and distribution of modules are determined by configuration files which are currently compiled with the subsystem initialization code. If multiple CPUs are utilized by SMART, an exact copy of the server code is downloaded to each CPU. These servers are started after SMART module initialization is completed. They also use configuration files to build a "roadmap" which indicates where the SMART modules are located. Through data-driven logic, the server on the first CPU behaves as a "traffic cop" by directing commands received from the supervisor to either itself or to the other servers according to where the SMART modules are located and according to which modules are required to carry out each command. Note that the server code does not have to be modified for different SMART configurations. Only the ASCII configuration files need to be changed. This essentially comprises the first interface to SMART.

The second interface is simpler. The RIPL methods used by the server to carry out commands call routines from the SMART tool kit. These routines, in turn, cause the asynchronous control tasks to change state and thereby affect the state of the devices being controlled by the subsystem. However, a problem with this approach is that RIPL methods now appear to be directly tied to the SMART tool kit rather than remaining autonomous. This can be prevented by defining a *SMARTRobot* class in RIPE which isolates the RIPL methods that must be implemented in terms of the SMART tool kit. Then subclasses can be derived from *SMARTRobot* for particular robot types. These subclasses can inherit either a standard robot interface or the SMART robot interface. Therefore, only the *SMARTRobot* class is dependent upon the SMART tool kit.

3.3 Generic Subsystem for Transport Devices

Using the interface templates just described, a generic server subsystem has been implemented which can be reused with minor modifications to control any transport device that has a RIPL translator. A generic command set has been defined for this transport subsystem, thereby eliminat-

ing the need to reconfigure the Sancho interface whenever a different manipulator is required for a new intelligent system application. Brief descriptions of the generic commands are given in Figure 3.

During a graphical programming session using Sancho, these commands are sent to the generic server subsystem by a GENISAS client which is contained within the supervisory control program. They are sent as ASCII strings with variable numbers of arguments and argument formats. GENISAS internally handles the parsing of the commands and their arguments to determine which method in the server subsystem should be invoked to carry out the command.

The generic transport subsystem is defined as a *RobotServer* subclass of the GENISAS *StdServerProcess* base class. It therefore inherits all of the communication facilities required by any server. The *RobotServer* subclass itself contains the methods which implement the generic command set. These methods, in turn, are implemented by using RIPL methods defined for the appropriate RIPE device driver subclass. This is accomplished by defining a generic pointer (*ptr_robot*) to the RIPE subclass inside *RobotServer* and establishing a containment relationship between them. Whenever a *RobotServer* object is created during subsystem initialization, the *RobotServer* constructor will create the appropriate RIPE object or objects for the transport device in use. This, in turn, provides the initialization for the device so that it is ready to be controlled through the generic commands.

Lock:	give supervisor exclusive REMOTE control
Release:	give subsystem exclusive LOCAL control
Activate:	place transport device in an active state
Deactivate:	place transport device in an inactive state
Configure:	configure subsystem for subsequent cmds
SetUnits:	set the linear and/or angular units
SetSpeed:	set the absolute speed
SetAcceleration:	set the absolute acceleration
SetToolLength:	set the tool length for the current tool
ReportState:	return the current device state
MoveTo:	perform a motion in world space
MovebyJoint:	perform a motion in joint space
MoveReact:	move until a sensor threshhold is exceeded
MoveComply:	move while complying to a surface
ManualControl:	move under control of a teleoperated device
LoadPath:	download a path segment to a motion queue
MoveAlongPath:	perform a path move using current queue
ClearPath:	clear path motion queue
StopMotion:	stop current motion gracefully
GetTool:	get specified tool
PutTool:	put specified tool
OpenGripper:	enact motion for current tool (open jaws)
CloseGripper:	enact motion for current tool (close jaws)
InitRecordFile:	record a log of subsequent trajectories
CloseRecordFile:	stop recording trajectories

Figure 3. Generic Transport Subsystem Commands

The *RobotServer* generic command implementations are identical for any transport device because all RIPE transport device subclasses use the same RIPL calls to program their associated hardware. An example of a simple template for the *RobotServer* method which implements the

Activate command is shown in Figure 4. In this code, the server first determines which CPU the command should be executed on if the control system is distributed across multiple CPUs. If this particular copy of the server resides on CPU 0, which is by convention the CPU that the supervisor communicates with, then message routing must be handled correctly. *RobotServer* on CPU 0 uses an internal GENISAS client to ship the command to another copy of *RobotServer* on a different CPU if the command must be executed somewhere other than CPU 0.

The command is actually executed by calling RIPL method *change_state*. This method will somehow interact with the device to place it in an active state. For a SMART-based controller, this involves calling SMART library routines for activating the SMART control system. As long as each RIPE subclass required by the server has the standard RIPL calls, such as *change_state* for activating the transport device, the same implementation can be used by any server for any transport device. Note in Figure 6. how the *change_state* method is called using the generic *ptr_robot*. Therefore, for each different transport server implementation, the only code modifications required are redefinition of this pointer for the desired RIPE device object contained in *RobotServer* and substitution of the correct RIPE constructor call used to initialize that device. In other words, for a subsystem that controls a Puma robot, *RobotServer* will define a containment relationship with the RIPE class *PRobot*, and the generic *ptr_robot* will be initialized to point to a *PRobot* object. Likewise, for a subsystem that controls a CNC machine, *RobotServer* will define a containment relationship with RIPE class *CNCMachine*, and the generic *ptr_robot* will be initialized to point to a *CNCMachine* object. All of the *RobotServer* command methods will remain unchanged from subsystem to subsystem, producing a high degree of software reuse.

Application-specific information is maintained in ASCII configuration files which are accessed by the *RobotServer* constructor. Such information includes network configuration information, tool and sensor tables, and SMART configuration information if the SMART tool kit is being used for low-level control. The SMART configuration includes which SMART modules are required, which CPUs they are resident on, and which modules are accessed for each generic command implementation.

```
int RobotServer::Activate(int argc, void ** argv, char *e_msg) {
        int ret = OK ;
        static char fname[] = "Activate";
        int location ;
        char cntlCmdMsgCopy[100] ;

        entering(fname);

        // Determine where the command should be executed
        location = WhichCPU(fname) ;

        // If this is the main server and the command is to be executed
        // somewhere else, send the command to the appropriate cpu.
        // If the transmission is successful, also execute the command
        // on the main server to update state variables
        if ((location > my_cpu_number) && (my_cpu_number == 0))
        {
           sprintf(cntlCmdMsgCopy, "%s", fname) ;
           ret = clientP[location]->SendCommand(cntlCmdMsgCopy, e_msg) ;
           if (ret == OK)
              ret = ptr_robot->change_state(ACTIVATE) ;
        }

        // If this is the correct cpu, execute the command
        else if (location == my_cpu_number)
           ret = ptr_robot->change_state(ACTIVATE) ;
```

```
        // This server is not suppposed to execute the command
        else
            ret = ERROR ;

        return(ret);
}
```

Figure 4. Sample Code for a Generic Command Method

Currently this generic server is used to control several different manipulators and a CNC milling machine. Extension of the generic tool kits to support other devices is a straightforward, methodical process because existing detailed designs can be reused. For example, to support a new manipulator, a RIPE subclass must be implemented which provides the translation from RIPL commands to corresponding hardware signals that produce motion. Because the RIPL interface design is already well-defined, the process basically involves implementing each of the methods associated with the RIPL command interface. Then a new version of the generic transport subsystem can be cloned which utilizes this new RIPE object to control the new manipulator. A similar scenario can be followed for extending the SMART tool kit. Development effort may still be significant since different devices have different interfaces with varying degrees of complexity. However, the amount of reuse and resultant savings in time and cost are also significant.

3.4 Applications

Complete intelligent control systems have been implemented which utilize all four tool kits and their interfaces to perform several prototype applications for environmental remediation and information-driven manufacturing. The resulting systems are based on the interactive menuing interface and simulation environment from the Sancho tool kit for automated planning and programming. The supervisory control programs use the set of generic commands described previously to control a transport device required by a given subsystem. This command set is easily extended or modified through Sancho ASCII configuration files and new *RobotServer* methods to reflect changing requirements. The generic transport server subsystem defined by subclass *RobotServer* is used to control either a manipulator or CNC machine. This subsystem connects to the supervisor through GENISAS and executes the generic commands for any manipulator or CNC machine that is supported by the RIPE/RIPL and/or SMART tool kits. Currently this includes a Schilling Titan2 manipulator, a Schilling ESM long reach manipulator, various models of the Puma robot, and a Fadal vertical machining center. By starting out with this base system, task-level programming can be accomplished by generating scripts containing sequences of generic commands that perform useful operations.

4.0 SYSTEM OPERATION

4.1 Desired Scenario: CAD Design to Finished Part

The following is a 5-step scenario for machining a part with the graphical-programming-based CNC architecture:

4.1.1 Part Design - CAD System

a) design desired part using standard CAD package (Pro Engineer)
b) design any required fixtures and raw stock models which do not already exist in a library
c) maintain a library of modular standard fixture and raw stock models to reuse

4.1.2 Interactive Setup of Simulated Machine Work Cell

a) import and position models of fixtures, raw stock, and finished part into the simulation environment
b) other information to include in the modeled environment:
tool selection (based on material, cut length, # flutes, diameter, cost, sharpness)
material selection (based on hardness, strength)
part features (geometry, topology, tolerances, surface finish, top of stock, final cut depth, dimensions)
machine tool selection (based on suitability, limitations, power)
fixture selection (based on rigidity, orientation, type)
part quantities
speeds, feeds, incremental cut values, coolant
roughing and finishing operations
ordering of tasks: setup, approach and entry, cutting methods, intermediate motions, exit/withdrawal
economics, safety

4.1.3 Tool Path Planning

a) let the CAD system (Pro Manufacture) do the initial planning and import it into the simulation environment
b) simulate the tool path for verification, and interactively edit it as needed
c) automatically generate the "program" or "script" which consist of generic commands that will actually implement the machining task on the subsystem
d) playback the script in simulation mode for verification with full-body collision detection and material removal monitoring

4.1.4 Model Registration and Calibration

a) mount actual fixtures and raw stock onto machine
b) use sensors (probe or structured lighting) to locate them with respect to the machine coordinate system and upload this information to the simulation system

c) perform calibration to register the graphical work cell with the actual one
d) automatically adjust the program tool path to reflect the calibration

4.1.5 Execution of the Program to Machine the Part

4.2 Using the Architecture to Machine a Turbine Blade

In the first step of the turbine blade machining process, the fixture which holds the blade has to be aligned with the machine axes. The fixture consists of an adaptive plate attached to the CNC table and another fixture attached to the plate which actually holds the blade. When mounted, the fixture is automatically centered with respect to the default coordinate frame (fixture or part frame) whose origin is located at the top center of the CNC table. However, there is slop in the rotation, so a touch probe is used to determine the rotary angle (A) and the tilt angle (B) for the fixture which are used as part of the new Home location (machine coordinate system made to coincide with the new fixture frame). This is done by touching the fixture with the probe on corner cubes which have been precisely machined onto the fixture. Eight points are generated by touching each end of the 4 sides of the fixture. These 8 points are stored in a file on the workstation as X, Y data which are tool positions with respect to the fixture or part coordinate system. This file is used as input to the next step for calibrating the sensor with respect to the located fixture.

In the second step, a structured lighting sensor has to be calibrated. The sensor is mounted manually in the tool holder. There will be slop in its orientation with respect to the tool holder. Therefore, the sensor coordinate system needs to be computed relative to the current machine coordinate frame which is the fixture or part coordinate system. The touch probe points determined in step one are used to compute locations around the fixture to which the sensor is moved, and readings are taken which are used to calibrate the sensor's position with respect to the fixture frame. It generates 24 locations (3 at each of the 8 fixture points from step 1 but with different orientations). The sensor readings consist of Y and Z values in the sensor's coordinate system. These values are used to compute the calibration parameters for the sensor.

Now the turbine blade has to be scanned to determine how much material needs to be removed from its tip. The blade is mounted in the fixture. The sensor is moved around with a fixed tool path to scan the blade. This path is based on what an ideal blade should look like (engineering drawings stored in a Unigraphics CAD file). Normally 120 readings are taken around the circumference of the blade to generate Y and Z values. The results of the blade scan are used to generate a CAD representation of the actual blade. It consists of 120 points along cubic spline representations of the blade tip. This CAD representation is then used to compute a tool path which represents the path which should be followed by a cutting tool to remove excess material from the blade tip in order to get it to match the ideal blade configuration as closely as possible. When generating this data, it is possible to select the number of points to be generated between sensor scan nodes.

Now the blade can be machined. The sensor is removed from the tool holder, the necessary setup functions are performed, and the cutting tool is inserted in preparation for machining the blade tip. The generated tool path is then used to allow the cutting tool to remove the correct amount of material from the blade tip.

5.0 CONCLUSIONS

Developing software for complex systems continues to be a difficult problem in many areas of industry and government. One of the reasons why progress in the automation of manufacturing processes has been slow is because of the difficulty in programming and integrating diverse devices to interact intelligently together and with their environment. The advanced CNC machine architecture provides a uniform, integrated environment for all steps in the process from CAD design to finished part. This includes interactive tool path generation based on the entire work cell, automatic post-processing, and program verification using full-body collision detection. This results in reduced time and cost for programming, reduction in design change time, and smoother transition from prototype to production.

The GISC-based architecture also provides for virtual collaborative environments which enable design and machining to occur at different locations using the same integrated environment. The real-time subsystem can also be transparently interfaced to low-level open architecture controllers such as NIST's Enhanced Machine Controller for sensor-based adaptive machining and in-process monitoring. These benefits can be applied to the modernization of the Defense Production Complex and to increasing economic competitiveness of U.S. manufacturers.

6.0 REFERENCES

Achi, P. B. U., "The Software Aspects of Computer Control of a Semi-Autonomous Interface to a Milling Machine," *Modelling, Simulation and Control*, VOl. 27, No. 2, 1990, pp. 31-45.

Allcock, A., "A New 'Strata' of Off-Line Programming," *Machinery and Production Engineering*, Vol. 147, No. 3769, October 1989, pp. 86-88.

Aramanda, Gregg, "CAM Software Gets Expert Advice," *Machine Design*, June 20, 1991, pp. 42-45.

Bahns, C. H., Barash, D. D., Cescato, D. J., Schneider, M. L., "Intelligent Machining Workstation Initiative," *WRDC-TR-90-8031*, January 1991.

Bakanau, F., "Modeling the Manufacturing Process," *The British Library Document Supply Center*, March 1990.

Beckert, Beverly A., "NC Programming Moves to Micros," *Computer-Aided Engineering*, June 1990, pp. 40-49.

Ben-Arieh, D. and Miron, I., "Modeling Advanced Manufacturing Systems Using Concurrent Logic Programming," *Artificial Intelligence in Engineering*, Vol. 5, No. 1, January 1990, pp. 43-49.

Buckley, C. P., "DNC: The First Step Towards Factory Floor Data Communications," *AUTOFACT'89*, pp. 27-7 to 27-16.

Bullen, George N., "Selective Programming," *Production and Inventory Management Journal*, Vol. 30, No. 4, 1989, pp. 49-51.

Callihan, H. D. and Loyacona, P. J., "From Form-Featured Solids Modeling to Production," *AUTOFACT' 89*, pp. 5-1 to 5-11.

"Consider Controls and Software," *Machinery and Production Engineering*, Vol. 147, No. 3764, July 1989, pp. 48-52.

Cook, R., "CAD Animation Makes Big Strides in Motion Control," *Managing Automation*, Vol. 6, No. 9, September 1991, pp. 58-59.

"FANUC Series 15-TA/TF/TTA/TTF, 150-TA Operator's Manual," *B-61214E/02*, FANUC LTD, July 1990.

Finster, Douglas P. and Carrier, T., "On the Direct Programming of CNC Milling Equipment," *Computers and Industrial Engineering*, Vol. 17, Nos. 1-4, 1989, pp. 252-257.

Fuller, J. E., "Vertical EDM Using Modular Programming," *CONF-8910214*, September 1989.

Haicheng, Y. and Zhenya, Z., "The GNCES System for Turbine Parts," *CAD and CG'89, Proc. International Conference on Computer-Aided Design and Computer Graphics*, Beijing, China, August 10-12, 1989, pp. 456-8.

Haynes, T. L., "The Infrastructure of Third Party VARs: A New Era of Manufacturing Controls," *IPC 92*, Detroit, MI, April 6-9, 1992.

Hudson, C. A., "Low End Controller (LEC) Project," *IPC 92*, Detroit, MI, April 6-9, 1992.

Kapustin, N. M. and Korotaev, M. Y., "Preparation of Control Programs for a Group of NC Machine Tools Under the Conditions of Combined Automation of Component Design and Manufacture," *Vestnik Mashinostroeniya*, Vol. 69, No. 4, 1989, pp. 37-41.

Karjalainen, J. A. and Ollila, A., "Towards Automated Thermal Cutting," *Computer Applications in Production and Engineering, Proc. 3rd International IFIP Conference CAPE'89*, North-Holland: Amsterdam, 1989, pp. 427-434.

Knutton, P., "Total Control in Any Language," *Machinery and Production Engineering*, Vol. 147, No. 3766, August 1989, pp. 53-54.

Koelsch, James R., "NC Programming: Obstacle or Asset," *Manufacturing Engineering*, July 1991, pp. 49-52.

Koncewicz, D., "Numerical Control Programming System - NCS," *Prace Naukowe Instytutu Cybernetyki Tech. Pol. Wroclawskiej, Seria:Konferencje*, Vol. 39, 1991, pp. 13-19.

Mahieddine, F. and Webb, D. C., "An Evaluation of Postprocessors in Generalized CAM Systems," *Advances in Manufacturing Technology, Proc. 5th National Conference on Production Research*, Kogan Page: London, UK, September 1989, pp. 312-316.

Martin, J. M., "Picking a CNC," *Manufacturing Engineering*, May 1989, pp. 59-62.

Mayer, R. J., SU, C., and Keen, A. K., "An Integrated Manufacturing Planning Assistant - IMPA," *Journal of Intelligent Manufacturing*, Vol.3, 1992, pp. 109-22.

Mayr, H. and Stifter, S., "Off-line Generation of Error-Free Robot/NC Code Using Simulation and Automatic Programming Techniques," *Robotic Systems and AMT. Proc. IFIP TC5/WG 5.3 International Conference*, Jerusalem, December 1989, pp. 126-136.

Miller, David J. and Lennox, R.C., "An Object-Oriented Environment for Robot System Architectures," Proceedings of 1990 IEEE International Conference on Robotics and Automation, Vol. 1, pp. 352-361.

Neelamkavil, J. and Graefe, U., "Automatic Generation of Manufacturing Control Instructions -

An Expert Systems Approach," *Information Control Problems in Manufacturing Technology,* selected papers from the *6th IFAC/IFIP/IFORS/IMACS Symposium*, Madrid, Spain, 1989, pp. 431-436.

Norrie, D., Roy, G., Fauvel, R., and Guo, D., "Microcomputer Simulation of a CNC Machining Center and its Application," *Computer Modeling and Simulation of Manufacturing Processes*, 1990, pp. 217-223.

Prun, J., "Maneuvering in the NC Minefields: Here are the questions to ask yourself before buying NC software," *Machine Design*, January 11, 1990, pp. 107- 13.

Quinlan, J. C., "Getting into DNC," *Tooling and Production*, February 1989, pp. 43-46.

Richardson, C., Copeland, L., and Wheeler, B., "Obstacles to Shop-Floor CNC Programming in the United States," *11th Triennial World Congress of Int. Fed. of Automatic Control*, Tallinn, USSR, August 13-17, 1990, pp. 487-91.

Roth, S. G., "Early Applications of NGC: Evolving an Open Architecture Control Solution," *IPC 92*, Detroit, MI, April 6-9, 1992.

Saito, T. and Takahashi, T., "NC Machining with G-buffer Method," *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 207-16.

Sprow, E., "Maybe you CAM, maybe you CANC," *Tooling and Production*, March 1990, pp. 117-122.

Stevens, L., "In Small Packages: PC-based NC Programming Gets Serious," *Manufacturing Systems*, Vol. 8, No. 7, July 1990, pp. 20-23.

"Tapping Advanced Technology: Today's leading-edge hardware and software become tomorrow's everyday tools," *Machine Design*, Vol. 61, July 1989, pp. 30-40.

"Unsnarl your shop - with DNC," *Tooling and Production*, Vol. 56, No. 2, May 1990, pp. 73-75.

Vukobratovic, M. and Stokich, D., "Software Support of the Dynamic Approach to the Control of Flexible Manufacturing Modules," *Soviet Journal of Computer and Systems Science*, Vol. 23, 1990, pp. 18-25.

Wakabayashi, N., Honda, N., and Mimaki, T., "A Supporting System for Making NC Programs by way of Knowledge Engineering Approach," *Skill Based Automated Production*, selected papers from *IFAC/IFIP/IMACS Symposium*, Vienna, Austria, November 15-17, 1989, pp. 213-218.

APPENDIX A - EXAMPLE SPECIFICATION OF A GENERIC CNC COMMAND

CNCMachine Class/Subclass Method *point_to_point_positioning* (G00) Software Requirements Specification and Design

1. General Description

Point-to-Point Positioning method

1.1 Overall Perspective

The following describes a method which belongs to the generic *CNCMachine* class and its derived subclasses. The *CNCMachine* class is a C++ class which is part of the RIPE/RIPL *Device* class hierarchy. It is a subclass of the *Transport* class and resides at the same level in the hierarchy (third tier) as the generic *Robot* class. They both model generic transport devices. The *CNCMachine* class logically models a Computer Numerically Controlled machining workstation. Applications which create objects from the *CNCMachine* class or its derived subclasses use them to control the operations of a CNC machine through a generic programming language as defined by the class methods. The following is a specification for one of those methods.

1.2 Functions

Method *point_to_point_positioning* is a motion command which, when executed, moves a tool attached to the CNC machine from one location to another. How the motion is carried out is determined by parameters passed into the method when it is called. Any combination of the axes available to the machine can be controlled by this positioning command. The final destination of the tool can be specified in absolute or relative (incremental) coordinates. The coordinate system used to specify the destination can be selected according to the capabilities of the machine. Most machines provide for at least 3 coordinate systems: a machine or tooling coordinate system (MCS, TCS, G28, G53), a work or program coordinate system (WCS, PCS, G92, G54-59), and a local coordinate system (G52). The units for the coordinate system are also machine-dependent and are usually specified in inches or millimeters.

The motion is always carried out at the maximum traverse rate of the machine which is usually set for each axis independently by the machine tool builder. Therefore, the traverse rate cannot be controlled by this method. During motion, the tool is accelerated to the predetermined speed by the controller and decelerated as the tool approaches its final destination. On some machines, "in-position" checking is performed at the completion of the move to verify that the feed motor is within a specified range of the destination location.

This type of motion is normally used when moving in free space between operations performed on a workpiece because it is the fastest and most efficient type of machine tool motion.

1.3 User Characteristics

Users of this method include software developers of CNC machine subclasses and possibly applications programmers. It can be used by any developer of CNC machine applications.

1.4 General Constraints

Current implementations of this method require that the CNC machine have a communication interface which allows a host computer executing this method to transmit the appropriate positioning command to the CNC controller. The controller then activates the machine axes to perform the motion. Hopefully sometime in the future, CNC machine controllers will be directly programmable in the "high-level language" defined by these methods.

The number and types of axes controllable by this command are machine-dependent.

This command cannot control the speed of the motion.

1.5 Assumptions and Dependencies

This method assumes that point-to-point positioning for a generic CNC machine is implemented using the standard G00 code.

2. Specific Requirements

The following sections describe all of the details of the *point_to_point_positioning* method which are needed to generate an actual implementation. This includes the method name, its parameters and their interpretation, and the processing sequence required to accomplish the specified function of this method.

2.1 Functional Requirements

Project Name: Generic Programming Languages for Manufacturing Processes
Class Name: *CNCMachine*
Method Name: *point_to_point_positioning*
Form of Call: return_code = point_to_point_positioning(destination, attributes) ;
Return Type: Integer status code

2.1.1 Introduction

Method *point_to_point_positioning* is a motion command which, when executed, causes the axes of a CNC machine to move from one point to another at the maximum traverse rate of the machine tool. How the motion is carried out is determined by the *destination* and *attributes* parameters.

2.1.2 Inputs

Parameter *destination* - This is a vector object which specifies the point or location to which the machine moves. It contains the coordinate values of each machine tool axis. The number and type of axes are machine-dependent. For example, a 5-axis machine would require that the vector contain 5 coordinates for X, Y, and Z (translation axes), and A and B (rotational axes). The units and ranges of these coordinate values are also machine-dependent.
Parameter *attributes* - This is a bit mask whose fields specify what coordinate system to use and whether the destination is in absolute or relative coordinates.

2.1.3 Processing

The input parameters must first be validated. If an invalid attribute is passed in (unrecognizable bit pattern), the method will simply return an error code without moving the machine tool. If the *attributes* parameter makes sense, it is parsed to determine the coordinate system and mode (absolute vs relative). Based on this information, the *destination* vector can be examined to determine if the values make sense (within range, correct dimension). If the destination is unreachable, the method will return an error code without moving the machine tool. If the parameters are valid, the appropriate command(s) will be encoded in an ASCII message and sent to the controller for execution.
A group of commands at each step in the sequence of a typical CNC machine controller program is called a block. The encoded ASCII message sent to the controller will comprise one block. A block typically has the following generic configuration:

NOOOOOGOOXOO.OYOO.OZOO.OAOO.OBOO.OCOO.OMOOSOOTOO<CR>

The N field contains the sequence number. This is optional when the block is sent from a remote host. The G field contains the G code associated with the particular action to be performed. For this type of

move, the code is G00. The X, Y, Z, A, B, and C fields contain the destination coordinate values for each machine axis. The M field contains a code for miscellaneous functions. It is not used in this method. The S field contains a value for spindle speed. It is not used in this method. The T field contains a tool function code. It is not used in this method.

Multiple G and M fields may be placed within a single block to specify several actions which must occur to complete the desired function. For this method, depending upon the attributes specified, it may be necessary to encode several G commands within the block. If the CNC machine is not currently in absolute mode and an absolute move is specified, G90 is used. If the CNC machine is not currently in incremental mode and a relative or incremental move is specified, G91 is used. If the desired coordinate system for the move is not the current coordinate system setting, then one of the following codes will have to be specified, assuming that origin locations for these coordinate systems have already been defined and saved internally by the *CNCMachine* class: G52, G53, G54-G59, or G92 (see the specifications for the commands which set up coordinate systems in order to learn how to build the appropriate forms of the message for these particular G codes). A typical message format for an absolute point-to-point positioning move of the X and Z axes in the current coordinate system would be:

G90G00X25.0Z10.0<CR>

This message is sent over a serial port using a communication object defined internally by the *CNCMachine* class or its derived subclass. If the CNC controller has the capability of returning a status message after the command has been executed, the *point_to_point_positioning* method will wait until it receives that message before returning control to the caller of the method.

2.1.4 Outputs

An ASCII message as described above is output to the CNC machine being controlled. The resulting side effect is the motion of the machine tool axes as specified by the commands embedded within this message. The status message (if possible) returned by the machine controller is converted into a status code which is the return type of the method call. This will be 0 if the command executed correctly. Otherwise it will be a value which is normally less than zero.

2.2 External Interface Requirement

2.2.1 User Interface Requirements

This method must be called within a C++ program which has created a *CNCMachine* object or a derived subclass object. A graphical user interface built on top of the C++ environment allows the operator to graphically program a machining application using this command.

2.2.2 Hardware Interface Requirements

As stated above, a communication handler object serves as the logical interface between this method which is executing on a host computer and the CNC machine controller which performs the actual motion. This communication object has generic *send_msg* and *receive_msg* methods which can be used to send the message block defined above to the controller and receive back status. These methods hide the actual communication mechanism from the *point_to_point_positioning* method, thereby providing device independence and a measure of portability and reusability. Currently, most CNC machines are interfaced to an external computer through a serial port. One possible protocol used is DNC (Distributed or Direct Numerical Control). Currently there are serial classes available in RIPE/RIPL which are used to provide this interface.

2.2.3 Software Interface Requirements

The methods of the *CNCMachine* class and its derived subclasses interface to RIPE/RIPL and the CNC machine controller software environment.

2.2.4 Communication Interface Requirements

The communication interface to the CNC machine controller has already been discussed in the hardware interface requirements. More specific details will depend upon the particular CNC machine controller used to implement the software. This will therefore fall under the purview of the subclass derived from the generic *CNCMachine* class for a particular CNC machine.

2.3 Performance Requirements

There are none for this method. However, if the controller is capable of sending back status after executing the motion, and if this status fails to arrive within a certain time frame, then the communication handler will timeout and inform this method that something has gone wrong. This method will then give up and return a timeout status to the caller. The length of the time frame is dependent upon the CNC machine.

2.4 Design Constraints

The number and types of axes controllable by this method are machine-dependent.
This method cannot control the speed of the motion.

2.4.1 Standards Compliance

EIA, "Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines: ANSI/EIA Standard RS-274D", Electronic Industries Association.
EIA, "32 Bit Binary Exchange (BCL) Input Format for Numerically Controlled Machines: EIA RS-494 standard", Electronic Industries Association.

2.4.2 Hardware Limitations

This method should be capable of running on any host computer which can compile and execute C++ code and which can be physically interfaced to a CNC machine controller, using the necessary interface software (communication handler) to send and receive messages.

3. Class Elements Dependencies

The following describes those elements of the *CNCMachine* class and its subclasses which the *point_to_point_positioning* method depends upon for its implementation.

3.1 Data Structures

Data structures are defined to model the following components required by this method:
- number of axes available (degrees of freedom),
- symbolic constants to reference or index the individual axes in a vector object (X, Y, Z, A, B, C),
- coordinate systems (transformation matrices or origin references),
- symbolic constants to define the various bit mask fields for the *attributes* parameter,
- templates for encoding CNC program blocks,
- hooks to communication objects,
- status messages and status codes.

3.2 Other Information

The following information is maintained by an object created as an instance of the *CNCMachine* class:

- the current mode of the CNC controller (absolute or relative),
- the current coordinate system in use,
- the current units used by the CNC controller (inches or millimeters),
- the minimum and maximum travel ranges for each axis,
- the types of status messages returned and their corresponding integer codes,
- the type of communication protocol used between the host and CNC controller (DNC),
- timeout parameters.

Distribution:

2	MS	1436	Donna L. Chavez, LDRD Office, 4523
2		1006	Clifford S. Loucks, 9671
2		1138	David J. Miller, 6532
1		9018	Central Technical Files, 8523-2
5		0899	Technical Library, 4414
2		0619	Review & Approval Desk, 12630 For DOE/OSTI