539-63
207662
P- 9

AIAA-94-1214-CP

USING GENERIC TOOL KITS TO BUILD INTELLIGENT SYSTEMS*

6060

N94- 30565

David J. Miller
Sandia National Laboratories
Albuquerque, New Mexico

Abstract

The Intelligent Systems and Robotics Center at Sandia National Laboratories is developing technologies for the automation of processes associated with environmental remediation and information-driven manufacturing. These technologies, which focus on automated planning and programming and sensor-based and model-based control, are used to build intelligent systems which are able to generate plans of action, program the necessary devices, and use sensors to react to changes in the environment. By automating tasks through the use of programmable devices tied to computer models which are augmented by sensing, requirements for faster, safer, and cheaper systems are being satisfied. However, because of the need for rapid cost-effective prototyping and multi-laboratory teaming, it is also necessary to define a consistent approach to the construction of controllers for such systems. As a result, the Generic Intelligent System Controller (GISC) concept has been developed.[1] This concept promotes the philosophy of producing generic tool kits which can be used and reused to build intelligent control systems.

Introduction

There have been many approaches taken in developing robotic control systems.[2,3,4,5,6,7,8,9,10,11,12,13,14,15] In examining these efforts, a common set of requirements can be derived. This set minimally includes such elements as fast servo-level response based on sensory inputs, trajectory planning based on world models of the tasks to be performed, and an extensible computing environment that supports asynchronous control with multi-tasking and multiprocessing. Therefore, any approach used for designing and implementing intelligent systems should support these requirements.

There also continues to be discussions within the user community about the need for guidelines or standards for robotic architectures. Because the primary purposes of standards are to save time and money when developing new systems and to facilitate integration of multi-supplier components, any standards adopted should reflect these goals. Also, because software is becoming the most critical component of complex intelligent systems, any potential standards should address the issues of how to make it easier and more cost-effective to develop software for new intelligent system applications.

The primary solution to this problem is software reusability. Although this may seem too simplistic, and many would argue that more encompassing standardization should be pursued, designing software for reuse is technically a very difficult task.[16] Also, because most software is developed within the context of a specific project, budgets and deadlines normally preclude developers from doing anything beyond the scope of the immediate task at hand. To overcome this dilemma, long-range thinking and planning need to be performed in order to encourage a philosophy of producing generic tool kits. Such tool kits, although developed within the context of a specific application, should transcend the application to provide reusable capabilities which reflect the common set of requirements for intelligent systems. Reuse makes subsequent applications easier to develop, thereby saving time and money. As a result, relatively complex systems with "standard" components can be developed as cost-effective solutions to difficult problems. Sandia is pursuing this philosophy in the development of the Generic Intelligent System Controller.

In this paper, we first describe the GISC approach to developing control systems. Then we discuss four different generic tool kits which have been developed in support of this approach. Next we illustrate how these tool kits can be integrated to build an intelligent robotic system, with particular emphasis on the development of a reusable generic subsystem to control any transport device such as a manipulator or CNC machine. Finally, we show how this system is utilized in two prototype applications.

An Approach to Building Intelligent Systems

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

*This work performed at Sandia National Laboratories supported by the U.S. Department of Energy under contract DE-AC04-94AL85000.

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.

Generic Tool Kits

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; [17]

2) the RIPE/RIPL tool kit which enables development of generic subsystems by providing object-oriented interfaces to intelligent system devices; [18]

3) the SMART tool kit which enables development of underlying control systems that provide the performance and flexibility for sensor-based control and teleoperation; [19]

4) the Sancho tool kit which provides for easily reconfigurable menu-based operator interfaces and a dynamic simulation environment.[20]

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

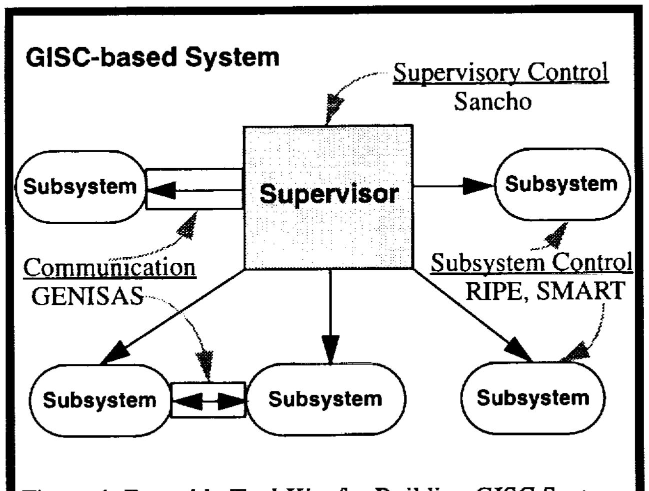

Figure 1. Reusable Tool Kits for Building GISC Systems

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.

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.

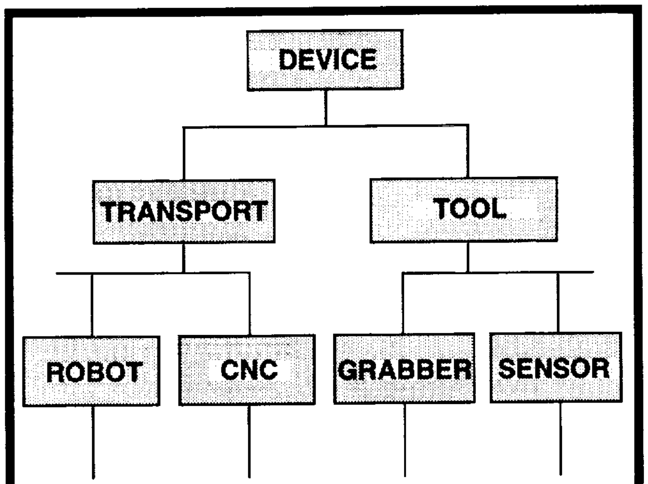

Figure 2. RIPE Class Inheritance Hierarchy

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. Figure 3. illustrates a simple control system using three SMART modules. 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.

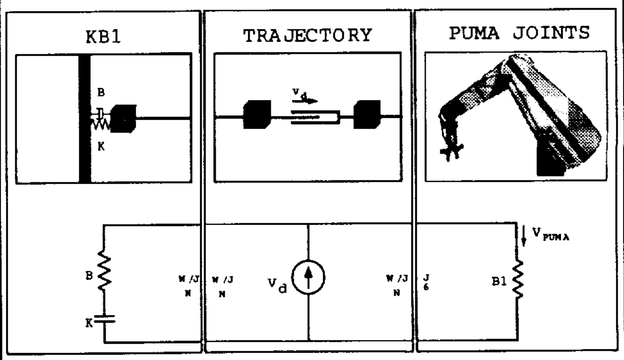

Figure 3. SMART Joint Controller for a PUMA 560

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. Figure 4. shows an example of the graphical user interface for CNC machines as it appears to the operator on a Silicon Graphics workstation.

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.

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.

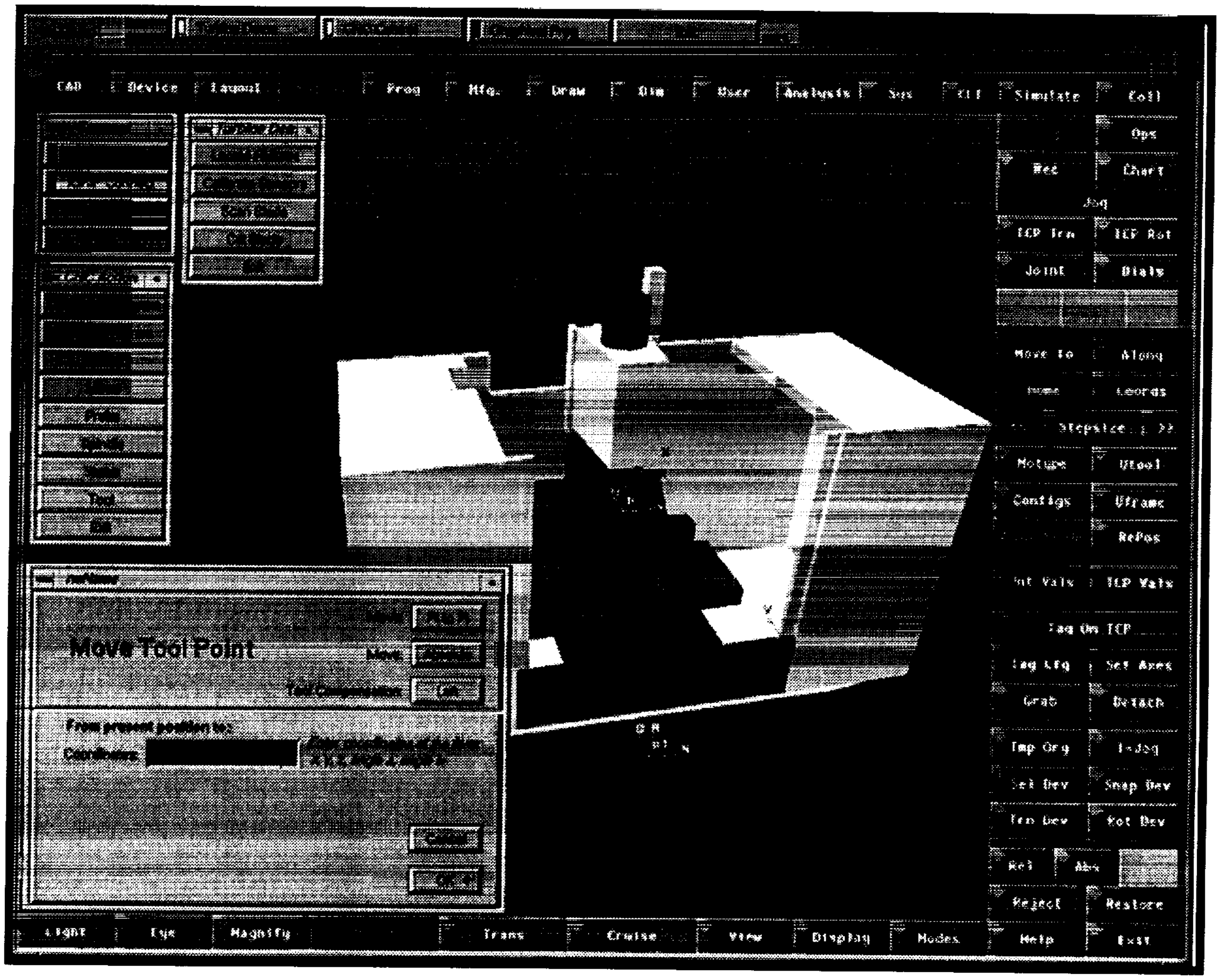

Figure 4. Sancho Graphical Programming Interface

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.

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.

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.

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.

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 eliminating 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 5.

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 5. 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 6. 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 6. 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.

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 *Robot-Server* 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.

Underground Storage Tank Remediation

One application for environmental remediation involves the clean up of waste sites in which human exposure to radiation or other hazardous elements is unacceptable. Traditional manual master-slave methods for performing such remote operations have very low productivity and consequently a very high cost. Therefore, systems which use automated planning and programming and sensor-based and model-based control to perform these operations are faster, safer, and cheaper.

One of the tasks which has been implemented using this system is the cutting and removal of structures such as pipes from underground storage tanks. A Schilling Titan2 manipulator is used to perform the task. The operator first commands the manipulator to pick up a hydraulic cutter end effector and approach a pipe under graphical control, based on a model of the tank environment. The operator uses a mouse to select any point along the pipe where he wishes to perform the cut. Using knowledge of the location and orientation of the pipe in the graphical model as well as knowledge of approved pipe shearing practices, the control system automatically computes the correct motions to position the cutter approximately one foot from the pipe surface. This approach can be simulated first and previewed by the operator to verify that it can be executed safely. Once the manipulator is near the pipe, the operator can then command the system to perform a docking operation using ultrasonic sensors to center the pipe within the jaws of the cutter. Once docked, the operator commands the cutter to shear off the pipe, followed by an undocking operation.

All of the manipulator motions are executed through the generic robot server using the generic command set. Additional subsystems are used in GISC-like fashion to control the sensors and the cutter. The docking operation is therefore actually a "macro" command which consists of a sequence of generic commands to perform compliant motion. This macro is an example of how application-specific software can be developed within the context of the generic control system to perform specific tasks.

Intelligent CNC Architecture for Agile Machining

Another application 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 once again consists of the graphical programming environment coupled with the 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 would begin 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 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.

Summary

In summary, rapid, cost-effective deployment of intelligent systems to perform useful operations requires a software infrastructure which allows a system builder to immediately focus on the application-specific requirements of the task. Such an infrastructure is best provided through a set of complementary, integrated generic tool kits which serve as the building blocks for new application development. Such tool kits should provide the necessary communication, device, and operator interfaces within reusable software structures. As standalone products, they are independent of any particular application, but in the hands of the system integrator, they can be used to build very powerful intelligent systems for a variety of automated tasks.

As generic tool kits proliferate and are made more robust and easier to utilize, then de facto standards may evolve for intelligent systems which are based on common interfaces established within these tool kits. Obviously, there are many barriers to overcome in terms of defining these interfaces and learning how to develop truly reusable code. Technology transfer and commercialization of these packages is also essential in order to establish a market-driven standardization climate. Companies such as Adept, Schilling, PAR Systems, and Trellis are already developing and marketing more open, modular approaches to control systems due to repeated requests from the robotics R&D community. With continued efforts within this community to define the necessary interfaces and then transfer them to the commercial sector, we may gradually see an evolution toward the availability of standard tool kits which can be used to construct whatever kind of intelligent robotic system is needed for future applications.

References

1. Griesmeyer, J. M., McDonald, M. J., Harrigan, R. W., Butler, P. L. , and Rigdon, B., "Generic Intelligent System Controller (GISC)," *Sandia Internal Report, SAND92-2159*, Sandia National Laboratories-New Mexico, October, 1992.

2. Albus, J. S., McCain, H. G., Lumia, R., "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)," *NIST Technical Note 1235*, April 1989.

3. Backes, P., Hayati, S., Hayward, V., and Tso, K., "The KALI Multi-arm Robot Programming and Control Environment," *Proc. of NASA Conf. on Space Telerobotics*, January 31-February 2, 1989.

4. Brooks, Rodney A., "Elephants Don't Play Chess," *Robotics and Autonomous Systems*, No. 6, pp. 3-15, 1990.

5. Dilts, D. M., Boyd, N. P., Whorms, H. H., "The Evolution of Control Architectures for Automated Manufacturing Systems," *Journal of Manufacturing Systems*, Vol. 10, No. 1, pp. 79-93, 1991.

6. Elfving, A., Kirchhoff, U., "Design Methodology for Space Automation and Robotics Systems," *ESA Journal*, Vol. 15, 1991.

7. Hayati, S., Venkataraman, S. T., "Design and Implementation of a Robot Control System with Traded and Shared Control Capability," *Proceedings 1989 IEEE International Conference on Robotics and Automation*, Vol. 3, pp. 1310-15, Scottsdale, AZ, May 14-19, 1989.

8. Hayes-Roth, F., Erman, L. D., Terry, A., Hayes-Roth, B., "Domain-Specific Software Architectures: Distributed Intelligent Control and Communication," *IEEE Symposium on Computer-Aided Control System Design*, pp. 117-128, Napa, CA, March 1992.

9. Hormann, A., "A Petri Net Based Control Architecture for a Multi-Robot System," *Proceedings. IEEE International Symposium on Intelligent Control 1989*, pp. 493-8, Albany, NY, Sept. 25-26, 1989.

10. Martin Marietta, "Draft Volume I of Next Generation Workstation/Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS)," *Document No. NGC-0001-13-000-SYS*, March 1992.

11. Mitchell, T.M., "Becoming Increasingly Reactive," *AAAI-90 Proceedings: Eighth National Conference on Artificial Intelligence*, Vol. 2, pp. 1051-8, Boston, MA, July 29-Aug. 3, 1990.

12. Rossol, Lothar, "Nomad Open Architecture Motion Control Software," *Proceedings of the International Robots & Vision Automation Conference*, Detroit, Michigan, April 5-8, 1993.

13. Saridis, G. N., "Architectures for Intelligent Controls," *Symposium on Implicit and Nonlinear Systems*, Ft. Worth, TX, December 14-15, 1992.

14. Sorensen, Steve, "Overview of a Modular, Industry Stan-

dards Based, Open Architecture Machine Controller," *Proceedings of the International Robots & Vision Automation Conference*, Detroit, Michigan, April 5-8, 1993.

15. Stewart, D. B., Volpe, R. A., Khosla, P. K., "Integration of Real-Time Software Modules for Reconfigurable Sensor-Based Control Systems," *Proceedings 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Raleigh, NC, July 7-10, 1992.

16. Freeman, P., "Tutorial: Software Reusability," *IEEE Computer Society Press*, ISBN 0-8186-0750-5, 1989.

17. Griesmeyer, J. M., "General Interface for Supervisor and Subsystems (GENISAS)," *Sandia Internal Report*, Sandia National Laboratories-New Mexico, October, 1992..

18. Miller, D. J. and Lennox, R. C., "An Object-Oriented Environment for Robot System Architectures," *IEEE Control Systems*, Vol. 11, No. 2, February 1991.

19. Anderson, R. J., "SMART: A Modular Architecture for Robotics and Teleoperation," *International Symposium on Robotics and Manufacturing (ISRAM '93)*, Santa Fe, NM, April, 1993.

20. McDonald, M. J. and Palmquist, R. D., "Graphical Programming: On-Line Robot Simulation for Telerobotic Control," *Proceedings of the International Robots & Vision Automation Conference*, Detroit, Michigan, April 5-8, 1993.