# An Object-Oriented Environment for Robot System Architectures*

*David J. Miller*
*and*
*R. Charleene Lennox*

Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185

## ABSTRACT

An object-oriented Robot Independent Programming Environment (RIPE) developed at Sandia National Laboratories is being used for rapid design and implementation of a variety of applications. A system architecture based on hierarchies of distributed multiprocessors provides the computing platform for a layered programming structure that models the application as a set of software objects. These objects are designed to support model-based automated planning and programming, real-time sensor-based activity, error handling, and robust communication. The object-oriented paradigm provides mechanisms such as inheritance and polymorphism which allow the implementation of the system to satisfy the goals of software reusability, extensibility, reliability, and portability. By designing a hierarchy of generic parent classes and device-specific subclasses which inherit the same interface, a Robot Independent Programming Language (RIPL) is realized. Work cell tasks demonstrating robotic cask handling operations for nuclear waste facilities are successfully implemented using this object-oriented software environment.

## 1. INTRODUCTION

This paper discusses the Robot Independent Programming Environment (RIPE) developed at Sandia National Laboratories. RIPE is an object-oriented approach to robot system software architectures. The primary accomplishment of this effort is a software environment which facilitates the rapid design and implementation of complex robot systems to support diverse research efforts and applications. RIPE allows robot system developers to concentrate on algorithm design and optimization, as well as testing and evaluation of new control, sensing, computing, and communications technologies without having to focus on overall system software development and integration. This is achieved by modeling the robot system as a set of software classes. As a result, RIPE hides device integration details and provides uniform interfaces to all objects in the system. A separation of concept from implementation characterizes RIPE's software classes and provides software reusability, extensibility, reliability, and portability.

In the following sections, the problems associated with complex robot software systems (which we have experienced first hand) are reviewed, together with past approaches dealing with these problems. We then discuss the underlying object-oriented concepts and distributed computing architecture upon which RIPE is based. RIPE currently supports automatic motion planning and programming of robotic and machining devices based on models of the environment, sensor-based control, error handling, and robust communication. The RIPE architecture also supports development of advanced software concepts such as graphical interfaces for robot system control.

The detailed design of RIPE is then defined, followed by a discussion of two implementations involving robotic cask handling operations for nuclear waste facilities. These systems show how the class hierarchy, consisting of generic parent classes and device-specific subclasses sharing the same interface, results in a Robot Independent Programming Language (RIPL). Finally, a brief discussion of error handling and additional future work is presented.

## 2. CURRENT APPROACHES TO ROBOT SOFTWARE

Sandia is currently developing robot systems for applications including hazardous material handling, automated assembly, and robotic edge finishing. Supporting this work are research laboratories investigating controls and optimization, telerobotics, grasping and dexterous manipulation, vision, tactile and proximity sensing, path planning and collision avoidance, oscillation damped movement, autonomous vehicles, flexible arms, and simulation. The diversity of Sandia's robotics effort implies that the software environment must support a wide variety of requirements and devices. It also must serve users with different levels of expertise in robot system programming.

### 2.1 The Problems

As our robotics effort has developed we have experienced many of the problems common to robot programming:

- the inability of robot languages to handle integration of sensors into the motion control system
- the difficulty of extending application code to include new tasks or new devices
- the high expense of application programming
- the small amount of code that is reusable for new systems
- the focus on manipulator-level rather than on task-level programming
- the management of system complexity
- the time consuming process of writing and debugging low-level (e.g. communication) software before beginning to test new algorithms
- the need for real-time control of mechanical devices
- the necessity of implementing robust error handling and recovery code

*This work was supported by the U. S. Department of Energy under Contract DE-AC04-76DP00789

---

It has been our experience that software development costs are a significant part of the overall intelligent robot system development costs. Finding solutions to robot software problems reduces these costs and thereby increases the viability of using intelligent robot systems in applications where it was once considered too costly.

## 2.2 Other Approaches to Solving the Problems

Various approaches have been used to try to solve these problems. Vendor supplied robot programming languages have been enhanced to provide extended sensor capabilities, real-time path modification, and user-friendly interfaces. VAL-II [18] and AML/X [14] are examples of such languages which have evolved until they resemble general-purpose, high-level computer languages. However, applications written in these languages are specific to the vendor's robot and, therefore, are not portable.

One approach is to replace the robot control system with a new control system embedded within an existing operating system, written in a general purpose language, and using primitive functions included in a library. RCCL [6] and KALI [1], for example, provide such intermediate level robot control primitives. These primitives can be called by off-line programming systems and can provide an environment for research into control and optimization algorithms. Frequently neither RCCL nor KALI is a viable option for industry work cells or research labs since they require the replacement of the robot control system hardware.

In attempts to move beyond manipulator-level programming of basic robot motions in a three-dimensional work space, task-level languages such as AUTOPASS [9] and LAMA [11] were defined. They were somewhat premature and were never fully implemented due to the unsolved subproblems of grasp planning, path planning, and fine motion planning [8]. However, it is generally agreed that task-level languages are needed to solve the problems relating to the expense and complexity of robot system programming.

A number of the more recent approaches [7, 19, 20] use conventional programming languages to help solve the problems of system complexity, real-time constraints, sensor integration, and modeling. These approaches have the advantages gained by using a language which is reliable, well-supported, portable, and familiar. In addition, these approaches focus on the total work cell system rather than only on the robot. The drawback is that, to date, the software has often been single purpose and not easily extended to other applications.

## 2.3 Approach for RIPE

RIPE was developed to support model-based automated planning and programming of robotic and machining devices, integration of sensor technologies, development of next generation robot system programming languages with graphical interfaces, error handling, and robust communication. It is built upon well-established software operating systems and programming languages. RIPE is an environment for complex system integration which stresses use of off-the-shelf hardware (e.g. commercial robots) where appropriate as well as providing support for advanced system development.

### 2.3.1 Computing Architecture

The RIPE computing architecture consists of a hierarchical multiprocessor approach which employs distributed general and special purpose processors. This architecture provides the computing power required by the RIPE software to control complex diverse subsystems in real-time while coordinating reliable communications between them. Advances in microprocessor technology allow general purpose processors to achieve the computing performance required by complex robot control algorithms while remaining compatible with a large base of existing software.

### 2.3.2 Object-Oriented Design

Central to the design of RIPE is object-oriented programming. Object-oriented design results in software architectures based on the objects that comprise a system and its environment rather than on the functions it performs [12]. Robot systems perform actions on certain objects within a defined work space. The software controlling the robot system can be viewed as an operational model of the world in which the robot exists. Therefore, RIPE is organized around representations of the objects in the work space so that its structure reflects the physical structure of the system. Controlled complexity is achieved by creating, combining, and manipulating software objects instantiated from previously defined software classes to perform the specific tasks of the system.

Object-oriented programming is based on the concepts of abstract data types, classes, objects, generic operations, message passing, type hierarchies/inheritance, and polymorphism [12]. Applying these concepts to RIPE results in modularity, encapsulation, abstraction, and information hiding. Abstract data types, classes, and objects allow the designer to model the physical robotic work cell entities in RIPE by defining only their attributes, behavior, and interfaces. Examples of such entities include work pieces or parts with geometric attributes, devices (e.g. NC machines, fixtures, machine tools, tool changers, robots, grippers, other end effectors), and sensors for contact switches, force control, and vision.

Since the software classes in RIPE are defined to represent the physical objects that are commonly found in a work cell, the communication interfaces to these generic software classes in RIPE become the general device independent language used to program the cell. This is achieved by applying the concept of polymorphism. Polymorphism (inherent in the object-oriented programming environment of RIPE) enables objects of a generic parent class and objects of its device-specific subclasses to receive the same messages and respond to them appropriately. The device independent programming language in RIPE resulting from object-oriented design of robotic work cells is RIPL.

Separating robot system concepts from the actual RIPE implementation results in robot system software which is reusable, extensible, reliable, and portable. RIPE's reusability is the basis for the design process. Extensibility is provided in RIPE by defining new software classes which in turn become part of the general work cell programming language. Use of inheritance to define subclasses which are extensions or restrictions of RIPE parent classes greatly lowers the cost and complexity of software development. System reliability is enhanced by reusing well-

defined RIPE objects, and portability is realized because RIPE classes are tightly encapsulated and relatively independent of their environment.

### 2.3.3 The Development Environment

Our development environment for RIPE has four primary layers: task-level programming, supervisory control, real-time control and device drivers. The choice of software at each layer is influenced by the primary requirements for modeling, sensing, and motion specification, as well as the widely acknowledged levels of robot software (task, manipulator, servo) [2, 21]. In addition, there is a strong relationship between the architecture employed at each particular layer and robot performance requirements.

The first layer is synonymous to what is generally referred to as task-level programming. At this level, world modeling, planning, and simulation are performed. Currently, this layer is in the initial stages of definition in our architecture.

The second layer is the supervisory control layer implemented on a UNIX-based workstation. This layer contains the primary control programs which coordinate all devices and activities of the system. UNIX is used by this layer because it provides a rich set of software development tools, is a mature operating system with emerging standards, and is available on almost all computers. The C++ language [16] is used to implement the object-oriented work cell class hierarchies and the supervisory code which manipulates these classes. C++ is a standard high level language which offers all the necessary features for object-oriented programming. Because C++ is a superset of C, a large existing base of C code is used, and all of the advantages of C programming are retained (portability, versatility, and systems programming facilities).

The third layer in the programming environment handles real-time control of devices for tasks such as force control. This layer consists of multiple VME-based 68000 family processors on a backplane network running the VxWorks operating system [22]. VxWorks was selected because of its real-time kernel, full-featured development and run-time environments, and its compatibility with UNIX. C++ runs effectively in this environment, and therefore, the same software can be used both at the workstation level and the real-time control layer. An Ethernet-based local area network ties together the workstations and VME systems.

The bottom layer contains the device drivers for each subsystem in a work cell. Some device drivers are relatively simple and consist of interfacing commands for tasks such as controlling a bar code reader. Others are sophisticated programming environments, such as the CIMCORP XR100 gantry robot software system. It is often more practical and efficient to utilize the vendor-supplied packaged software rather than attempt to create a new language and external interface. However, RIPE objects can use KALI or RCCL based commands to communicate with specialized controllers as well.

In the case of intelligent devices such as robot controllers, a monitor program located at the controller for the device is written in the robot programming language. This monitor establishes communication with an external host CPU, waits for a command from that host, carries out the command when one is received, and then waits for the next command. The host may send the command in a format which is directly executable by the robot controller (such as a VAL-II statement), or it may send a command code which triggers a subroutine call in the monitor. The monitor is treated as the part of a distributed robot object which resides on the robot controller. The messages that it understands and interprets are defined in the robot class and frequently have a one-to-one correspondence with the user interface routines defined for the robot class.

## 3. DEFINITION OF ROBOT SYSTEMS IN RIPE

The design of RIPE is based not only on our goals of ease of use,

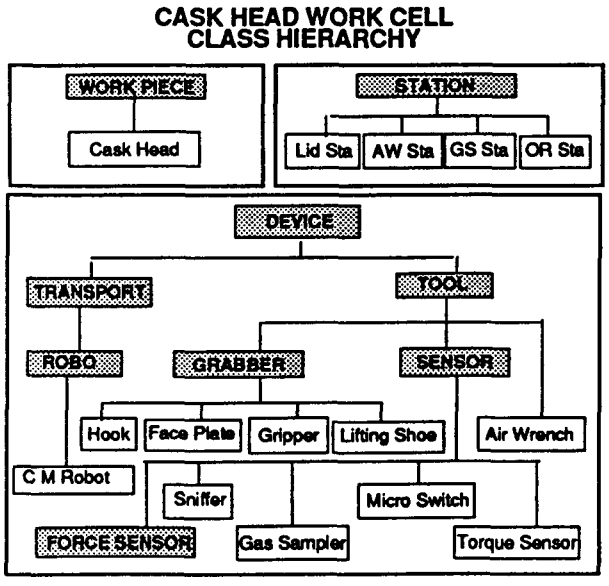


Figure 1

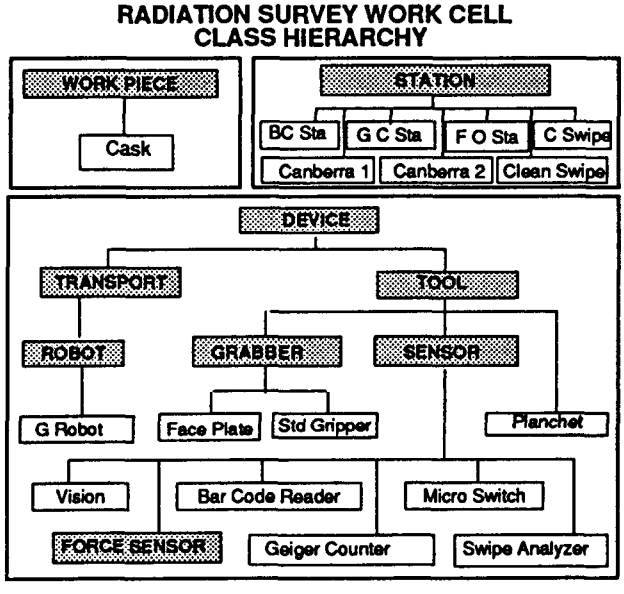


Figure 2

expressiveness, extensibility, and reusability but also on compatibility with FAC-SIM [5], a simulation system developed at Sandia for the analysis of robot systems. The partitioning of a system into classes is fairly straightforward since most classes reflect the physical objects of the application. The software classes which do not represent physical objects are termed "virtual objects" and include *CommunicationHandler, WorldModeler, ErrorHandler. TrajectoryGenerator* and *PathPlanner.*

### 3.1 The Generic Objects

The class inheritance hierarchies in RIPE are designed to allow the programming of tasks using generic classes. Figures 1 and 2 illustrate two example system hierarchies. In the Cask Head Work Cell, a Cincinnati Milacron robot performs leak detection and gas sampling operations on the head of a cask containing nuclear waste. In the Radiation Survey Work Cell, contamination surveys of the cask are performed by a CIMCORP gantry robot. The division of a robot system into the three basic classes of *WorkPiece, Station* and *Device* 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. The definition of the class hierarchy for *WorkPiece* and *Station* is specific to an application, but each work cell has several kinds of devices, e.g. robots, sensors, grippers, and other tools. All devices which carry out actions are derived from the parent class *Device.* Instead of, or in addition to a manipulator, a system might employ other devices such as an NC machine, a conveyor, a remotely controlled fork lift, or a mobile robot. These have the property of being able to move or transport a work piece or a tool and thus, derive from the *Transport* class. *Tool* is the parent class of any object used by the robot to perform a task. *Grabber* has the attribute of being used to pick up work pieces or other tools. Grippers, hands, face plates, and hooks are instances of the *Grabber* class. A *Sensor* is a *Tool* which provides data for the performance of the task. Besides force sensors, vision systems, and proximity sensors, examples of the *Sensor* class would be a bar code reader used for identification of a work piece, a contact switch which verifies the presence of a tool in its station, or a gas sampling device. A tool, such as an air wrench, which is not an instance of the *Grabber* or *Sensor* class, derives directly from the *Tool* class.

Each subclass of *Device* which is also a generic parent class is highlighted in Figures 1 and 2. Although not shown in Figures 1 and 2, virtual objects such as *ErrorHandler* and *CommunicationHandler* are also generic parent classes. The routines that define the user interface to a generic class, along with other attributes and routines common to all instances of that class, are defined at the generic parent level of the hierarchy.

Figure 3 shows the definition of the *Robot* class. The attributes are defined in the data structures such as *current* and *home.* The actions that are common to all robots are found in the definitions of the routines. The keyword *virtual* at the beginning of each function declaration indicates that the routine is to be defined by a device-specific subclass. All robots can be commanded to *move,* but the definition for executing a move is specific to a particular robot. Note the presence of default values for some parameters in the virtual routines, for example *speed* in the *move,*

**ROBOT CLASS DEFINITION**

```
class Robot: public Transport

{ protected:
    point     home,
              current;
    int       current_coordinate_type,

    double    speed;
    int       current_speed_attribute,

    double    accel;
    int       current_accel_attribute;

  public:
    // Abstract class so constructor is empty
    Robot();
    virtual ~Robot();

  virtual int move (point loc, int motion_attributes,double speed=0.0);
    // Absolute move to loc.

  virtual int move_rel (point delta, int motion_attributes,
                        double speed=0.0);
   // Move to a position delta away from current position.

  virtual int move_home ();
    // Move to home (ready) position.

  virtual void approach (point loc, int axis, double dist);
    // Move to a position dist away from loc along axis.

  virtual int depart (int axis, int direction, double dist,
                      double increment=0.0);
    // Move along the specified axis distance dist.

  virtual int move_react (point loc, Sensor * FS_ptr);
    // Move to position loc under force control.

  virtual int move_comply (point loc, int numincrs, Sensor * FS_ptr,
                           double fmax = F_MAX);
    // Move to position loc with compliance specified in f.

  virtual int path_move (path_point * p, int motion_attributes);
      // Move to each point in path p.

  virtual int path_move_rel (path_point * p, int motion_attributes);
    // Move relative to each point in path p.

  virtual void stop ();
   // Cause the robot to stop.

  virtual int set_speed (double s, int speed_attributes);
   // Set the robot member speed to s.

  virtual double get_speed ( int speed_attributes);
    // Return the value of robot member speed.

  virtual int open_gripper ();
    // Open the gripper

  virtual int close_gripper ();
    // Close the gripper

  virtual int get_effector (Device * t_ptr);
   // Move to station and pick up effector.

  virtual int put_effector (Device * t_ptr);
    // Stow end effector at station.

  virtual int perform (int task);
   // Execute a taught sequence

  virtual void where (point cur_loc, int coordinate_type);
    // Set cur_loc to the Cartesian position of the robot.

  virtual int report_status ();
    // Print the current status of the robot.
};
```

Figure 3

*move_rel*, and *path_move* declarations in Figure 3. Including optional parameters and default values provides flexibility in the subclass definition and in the application. For example, if the user wants to specify a speed during a move, he invokes the desired *move* routine with the *speed* parameter set. If he does not set the *speed* parameter, the default speed, which is established earlier by a call to *set_speed*, is used.

### 3.2 Derived Objects for Specific Applications

Figures 1 and 2 show how objects in specific work cells are derived from their generic parent classes. The bottom level of the hierarchy enumerates the software representation of the physical objects in the work cell. The Cask Head Work Cell, as represented in Figure 1 for example, required programming of the *CMRobot* class as well as other types of subclasses representing the different devices and work pieces. The interface to *CMRobot* is already defined in parent class *Robot*. The interface serves as a kind of template so that the programming is a matter of "filling in the blanks." For example, the code for the *move* routine consists of translating the command into the format the Cincinnati Milacron controller expects and invoking the *CommunicationHandler*'s routine *send_msg*.

If a subclass has more capabilities than the parent, the user interface to the subclass is the set of routines defined for the parent class plus additional ones defined in the subclass. The *CMRobot* class has been extended for research into oscillation-damped movement of a flexible beam. Routine *flex_beam_damping* uses the *move_comply* command and torque feedback to actively damp vibrations of a cantilevered beam. When only the routines specified in the generic parent class are used in programming an application, a different derived object representing a different physical device can be substituted, and the commands in the application code remain the same. These routines define the primitives for RIPL.

Figure 2 shows the derived objects for the Radiation Survey Work Cell. A comparison of the two work cells shows that the same set of generic classes can provide very different software object systems. The tasks performed in the application determine which class routines will be invoked in the supervisory code. Thus, changing the way the supervisor uses these class routines can result in the implementation of an entirely different task even though the class definitions remain unchanged.

## 4. APPLICATION OF RIPE

The Cask Head Work Cell and Radiation Survey Work Cell were constructed as part of the Advanced Handling Technologies Project (AHTP) at Sandia National Laboratories [4]. The AHTP includes efforts to automate cask handling operations at nuclear waste facilities. These work cell prototypes serve as proof-of-concept systems to demonstrate cask handling operations that might be performed robotically.

### 4.1 Cask Head Work Cell Example

The AHTP consists of several subprojects, one of which is the Cask Head Operations (CHO) project. The CHO project investigates robotic performance of cask head operations required before and after nuclear fuel bundle unloading. The Cask Head Work Cell was designed as a prototype system for cask head operations which include leak detection, gas sampling (port cover removal/replacement and coupling/decoupling of the sampling apparatus at the port), and bolting and unbolting operations. Robust algorithms were required for mating the torque wrench to various bolt heads on a cask head mock-up using force feedback. The requirements for the work cell operations illustrate the application of the RIPE environment.

One of the premises on which object-oriented design is based states that designers should avoid as long as possible describing and implementing the specific tasks of a system[12]. Rather, they should produce a high level design that defines only a set of classes which characterizes the behavior of the objects in the system. We followed this principle by designing and implementing the necessary Cask Head Work Cell classes, as discussed above, independent of any application to which they would be applied. As anticipated, implementation of the actual cask head operations was fast and straightforward. All that was required was to create and manipulate the work cell objects to perform the specific tasks of the system. Also, other applications of the work cell such as flexible beam oscillation damping research [15] were easily implemented because the classes had been designed completely independent of any particular work cell activity.

#### 4.1.1 Computing Environment

Figure 4 illustrates the computing architecture that is used to control the Cincinnati Milacron work cell doing cask head operations. The primary components include a Sun 3/60 workstation, a VME bus with two Force 68020 CPU's, global memory, an 8-port serial I/O card, a Cincinnati Milacron series

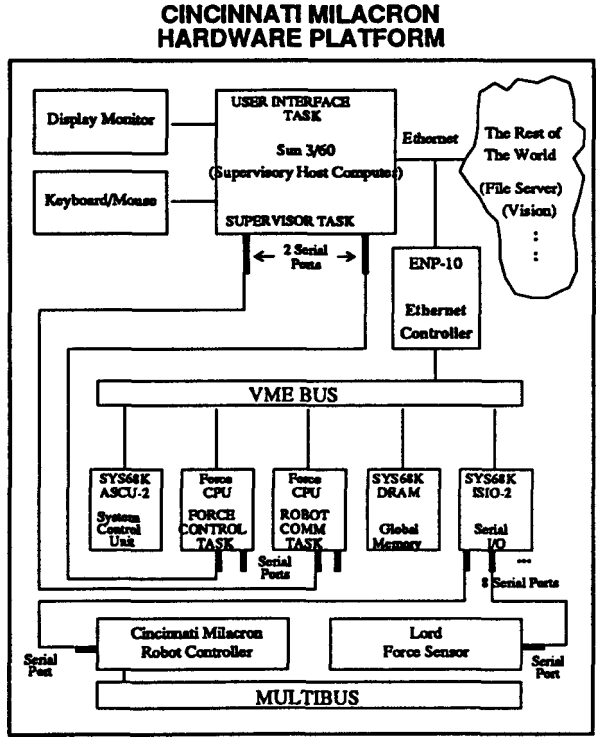


Figure 4

T3-786 robot, and a Lord force sensor. The computing elements, robot, and force sensor are all commercial subsystems. Special end effectors and grippers were designed and built at Sandia.

The distributed VME multiprocessors coordinated by the Sun workstation allow individual CPUs to control each subsystem in the work cell and provide support for continuous tasks, concurrency, synchronization, data sharing, communications, real-time control, and sensor-based activity. In addition, this architecture reflects the hierarchical layered approach to hardware which corresponds with the different levels of robot software (task, manipulator, servo). However, the design of RIPE allows the software to be mapped onto multiple layers of the hardware, depending upon the application. For example, the servo level normally resides at the robot controller, but whenever compliant motion is performed, some of the servo software functions are executed on the VME bus CPUs. Similarly, the manipulator level software may reside on either the robot controller, VME bus, or the Sun workstation. The task-level software will normally be at the workstation level. Finally, model-based control requires that knowledge about the work cell and its contents be distributed among the software objects that logically represent their physical counterparts, and these objects may reside at any level of the hardware.

### 4.1.2 Software

To perform cask head operations, four software tasks were required. All of the tasks are implemented in C++ and utilize the communication and device class libraries discussed above to perform their functions. A UNIX environment exists on the Sun workstation, and a VxWorks environment controls the VME hardware. Figure 4 shows how the tasks are distributed among the work cell CPUs for the Cask Head Operation task.

The first task allows the operator to interact with work cell devices. The current implementation uses SunView [17], but future interfaces will be built with a recently developed object-oriented package called InterViews [10]. The second task, which also resides on the Sun, serves as the work cell supervisor. It accepts commands from the operator through the first task and carries out these commands by initiating appropriate work cell actions (which may be performed by other hardware components).

The remaining two tasks reside on the VME bus CPUs. One task monitors the Lord force sensor (mounted on the wrist of the Cincinnati Milacron robot) and computes position updates to control robot movement whenever the torque wrench has to mate with a bolt. The other task provides the communications to the robot controller, utilizing the DDCMP protocol [3]. Both tasks use a serial I/O card for message transmission to the force sensor and robot controller.

Figure 5 illustrates the objects that are created by these tasks whenever they are executed. The work cell supervisor creates a *LORDForceSensor* object and a *CMRobot* object. These two objects are, in a sense, distributed over multiple environments. The way they are created (argument list specification) determines how they are distributed and how they communicate with the actual devices. In figure 5 the shaded boxes indicate the communication objects created by the device objects.

For example, if the force sensor were to be controlled directly

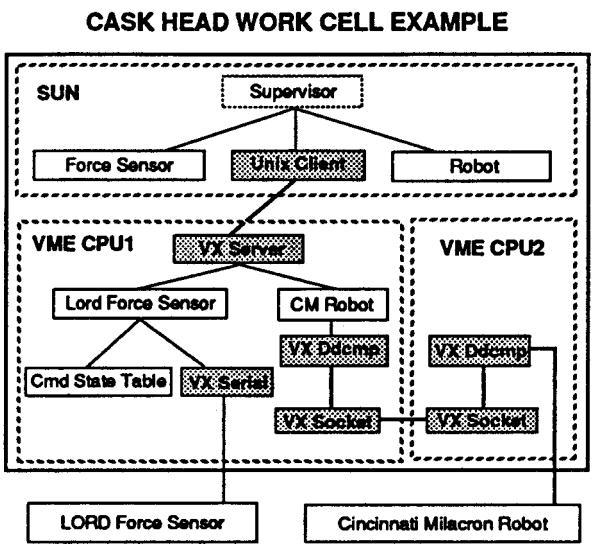

Figure 5

from the Sun workstation, a *LORDForceSensor* object could be created with a parameter list that would cause the creation of a *UnixSerial* object for direct communication to the force sensor device through a Sun serial port. In our implementation for cask head operations, the *LORDForceSensor* object and *CMRobot* object are distributed across both the Sun workstation and a VME CPU due to the real-time requirements of force servo control. They communicate through *UnixClient /VxServer* objects over the Ethernet between the UNIX environment on the workstation and the VxWorks environment on the VME bus. The *LORDForceSensor* and *CMRobot* objects on the VME CPU, in turn, create *VxSerial* and *VxDdcmp* communications objects respectively which allow them to talk to the actual hardware in the work cell. To achieve the update rate necessary for force control, an additional *VxDdcmp* object is created on a second VME CPU to handle the low-level protocol and message transmission to the robot controller. The two distributed *VxDdcmp* objects communicate with each other over the VME bus through a *VxSocket* object.

Finally, the *LORDForceSensor* object also creates a *CmdStateTable* object which reads a configuration file that defines the specific behavior of the Lord force sensor device. For example, using the *CmdStateTable* information, the *LORDForceSensor* object knows that it must send an "OA<CR>" (Output ASCII) command to the force sensor in order to obtain ASCII readings of the current forces being sensed. By isolating device-specific attributes and commands into files that are managed by the *CmdStateTable* object, methods that control the device's behavior are written generically and can reside in the parent *Tool, Sensor,* or *ForceSensor* classes rather than in the *LORDForceSensor* class. These methods therefore can be used by other types of force sensor classes derived from the parents (such as the *JR3ForceSensor* class) which have their own configuration files.

Figure 6 illustrates one portion of code for the Force Control Task residing on the first VME CPU. It shows how the objects in

**FORCE CONTROL CODE EXAMPLE**

```
//Defines and includes for class definitions
   :
// Create server object for communication with host
func ptr_testserver = testserver ;
p_func ptr_panicfunc = panicfunc ;
VxServer MyServer(SERVER_NUM, ptr_testserver,
                  ptr_panicfunc, TABLE_LEN, msg_table[0]);
   :
int testserver(VxServer* TestServer)
              // Spawned by MyServer
{
   LORDForceSensor* TestSensor ;
   force* fptr ;
   CMRobot* ptr_robot ;
   point new_loc;

// Create force sensor and robot objects
   TestSensor = new LORDForceSensor("LORDStateTbl",
                                    4, 19200, DEBUG_OFF) ;
   fptr = new force ;
   ptr_robot = new CMRobot("cm1", 1401, DEBUG_OFF) ;
   :
   :
// Wait for a command from the client
   len = TestServer->receive_msg (line, MAX_SOCKET_MSG+1,
                                  LU_MONITOR);
   :
// Initiate robot communication & configure sensor
   ptr_robot->perform (WAIT_FOR_BEGIN_REMOTE);
   TestSensor->set_bias () ;
   TestSensor->set_output_mode (BINARY_MODE) ;
   table_entry = ptr_robot->report_Var_entry (HOST_ENTRY);

// Select a work cell activity
   switch(table_entry)

// Feel for bolt with torque wrench until they mate
      case 1:
       while (i < 22) {
          new_loc[Z] = -0.150;  new_loc[R] = 0.0 ;
          printf("MOVE DOWN -0.150 INCHES\n") ;
          ptr_robot->move_rel (new_loc);
          TestSensor->take_reading (fptr) ;
          if ( fptr->t_gain[Z] > -10.0)  {
             printf("SUCCESS IS TRUE\n") ;
             success = TRUE ;          break ;
           }
          else {
             new_loc[Z] = 0.150;
             printf("MOVE UP 0.150 INCHES\n") ;
             ptr_robot->move_rel (new_loc);
             new_loc[Z] = 0.0 ;     new_loc[R] = -5.5 ;
             printf("ROTATE 5.5 DEGREES\n") ;
             ptr_robot->move_rel (new_loc);
             i++ ;
           }
        }
   :
    }   // End this work cell activity
```

Figure 6

the work cell are created and used to perform a simple bolting operation. Messages are sent to the *LORDForceSensor* and *CMRobot* objects to obtain force readings and initiate robot motion until the torque wrench is properly seated on a bolt.

### 4.2 Radiation Survey Work Cell Example

The Radiation Survey Work Cell was the first experimental system to be built for AHTP. Its initial application, the Robotic Radiation Survey and Analysis System (RRSAS), was completed in August, 1987 [13], and included operations to locate a half scale cask mock-up in the work cell using stereo vision, identify cask contents by reading bar codes, and perform both non-contact and contact radiation surveys. Key technologies such as automatic motion planning and programming of the CIMCORP XR100 gantry robot and force sensor integration to maintain constant force contact with the cask surface during contamination surveys are demonstrated by RRSAS. RRSAS was completed prior to the development of RIPE.

Some subsequent studies currently under development include the Impact Limiter Handling project and the Cask Tiedown project [4]. These projects will investigate robotic removal, handling, and replacement of cask impact limiters and tiedowns. They also require technologies similar to those developed in RRSAS, including machine vision and force control. These new projects are being executed using RIPE.

Although RRSAS was originally implemented in C from a function-oriented top-down design, its highly modular structure and generic functions for robot control make it possible to use some of the existing code for the methods of the C++ gantry robot class (*GRobot*). Also, because the generic *Robot* class had already been defined and much had been learned during development of the *CMRobot* class, implementation of the *GRobot* class was fairly automatic. Similarly, the Radiation Survey Work Cell employs a JR3 force sensor rather than a Lord force sensor, so a *JR3ForceSensor* class also had to be implemented. This again was facilitated by the existence of generic parent classes (*Tool, Sensor, ForceSensor*).

#### 4.2.1 Computing Environment

Aside from the specialized subsystems required by RRSAS, the primary hardware components of the Radiation Survey Work Cell are the same as those found in the Cask Head Work Cell (Sun workstation, VME bus, robot, sensors). There is of course a different robot and a different force sensor, and the VME bus uses Heurikon 68020 CPUs rather than Force CPUs (transparent to the software). As stated earlier, this hierarchical distributed approach is our standard architecture which provides the power, compatibility, flexibility, and extensibility needed to implement complex work cell environments.

#### 4.2.2 Software

The first application using RIPE in the Radiation Survey Work Cell performs force controlled movement of the robot arm for the random contact swipe survey. The original force servo control system in RRSAS consisted of a PDP/11 with an RT-11 environment [13]. This is replaced by a much more powerful VME based 68020 CPU and a VxWorks environment. The new swipe survey software (*Swipe Server*) is a C++ application which creates and manipulates a *GRobot* and *JR3ForceSensor* object to monitor a JR3 force sensor mounted on the wrist of the gantry robot and make real-time trajectory corrections to the robot arm. The corrections are based on the contact force detected between the robot's end effector (swipe planchet) and the cask. The control structure maintains a $4.0 \pm 1.0$ pound normal contact force during the swiping motion.

Figure 7 illustrates the objects that are created by the *Swipe Server*. If this figure is compared with Figure 5, it can be seen

**RADIATION SURVEY WORK CELL EXAMPLE**

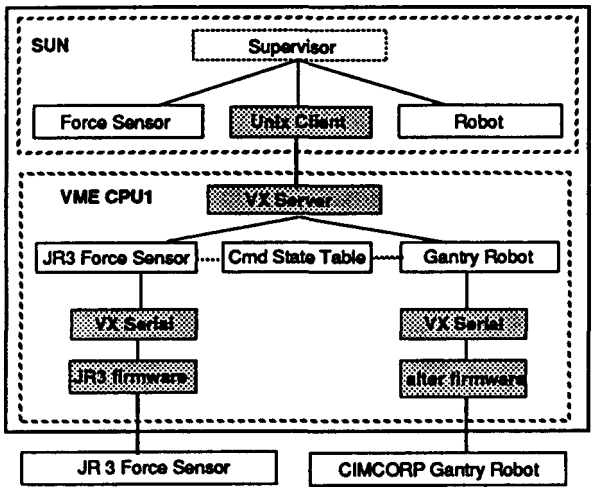


Figure 7

that the object hierarchies are nearly identical. The primary differences reside at the communication and servo level, where additional state machine firmware is utilized on the VME serial I/O card to handle the JR3 packet protocol and to synchronize position updates with a special trajectory card in the robot controller. At the manipulator level, the interface is identical to that found in the Cask Head Work Cell application. Again, the objects behave according to how they are created. The work cell devices can be directly controlled by objects residing on the Sun workstation whenever there are no real-time requirements, or they can be controlled in real time by objects distributed across VME CPUs.

Two additional comments can be made about the *Swipe Server* application. First, it illustrates the ability of our object-oriented environment to coexist with more traditional function-oriented environments. Rather than rewrite all of the RRSAS supervisory software, which is over 15,000 lines of C code, it was only necessary to replace a handful of modules which interfaced the supervisor to the PDP/11. The new modules create a *UnixClient* object which allows the RRSAS supervisor to communicate with the new VME-based *Swipe Server* through a *VxServer* object over the Ethernet. Everything else in the supervisor remains unchanged. Second, the *Swipe Server* utilizes *GRobot* methods which illustrate an object-oriented implementation of a task-level capability that allows the supervisor to ask the robot to "swipe the surface of a designated work piece", in this case the cask. The code segments in Figure 8 show how the generic manipulator level methods of the *GRobot* class are pieced together to create this task-level function. Similar implementations could be used in other work cells with other robots to provide this highly useful capability.

## 4.3 Error Handling

Frequently, extensive error detection and recovery procedures for complex robot systems are needed to ensure reliability. It is of particular importance for remote systems which must perform autonomously in hazardous environments, such as the cask

**SWIPE OPERATION CODE EXAMPLE**

```
int GRobot::swipe_operation (Sensor* FSensor, float inc)
{
  int numincs;
  int stat = 0;

  stat = move_till_touch (FSensor);
  if (stat) return(stat);
  numincs = SWIPE_DIS / INC_MAG;
  stat = make_swipe (FSensor, inc, numincs);
  if (stat) return(stat);
  depart (FORCE_AXIS, BACK_DIST, BACK_INC);
  returnto_preswipe_location ();
  clear_offsets_and_zero_alter ();
  return(stat);
}

int GRobot::move_till_touch (Sensor* FSensor)
{
   point loc;
   int stat = 0;

   FSensor->desired_values[Y] = TOUCH_SET;
   loc[X] = 0.0; loc[Y] = APPRO_INC; loc[Z] = 0.0;
   loc[D] = 0.0;   loc[E] = 0.0;    loc[R] = 0.0;
   stat = move_react (loc, FSensor);
   return(stat);
}
int GRobot::make_swipe (Sensor* FSensor, float inc, int numincs)
{
  point loc;
  int stat = 0;

   FSensor->desired_values[X] = 0.0;
   FSensor->desired_values[Y] = SWIPE_FORCE;
   FSensor->desired_values[Z] = 0.0;
   FSensor->desired_values[D] = 0.0;
   FSensor->desired_values[E] = 0.0;
   FSensor->desired_values[R] = 0.0;
   loc[SW_AXIS] = inc;
   stat = move_comply (loc, numincs, FSensor);
   return(stat);
}
```

Figure 8

handling operations presented earlier. When a robot system contains many distributed components which all interact with one another, the number of unique system states can be very large. It is difficult to anticipate all of these states when designing control software, and therefore, it is difficult to guarantee that the software will provide appropriate automated responses to them. In addition, the interaction between the software environment and the actual work cell it controls makes error recovery difficult to implement without continuous knowledge feedback regarding the current physical state of the work cell. This implies that error handling be sensor-based as well as model-based.

From a software point-of-view, there should be uniform, standardized methods for handling failures. These methods should be relatively transparent and not clutter mainline code. Furthermore, they should be easily tailorable to new subsystems and applications by providing flexible recovery options and logic flow control. Finally, error recovery procedures should be modular and localized whenever possible, and they should be implemented in parallel with normal work cell control code.

The object-oriented approach used in RIPE applies well to error handling and satisfies the requirements just enumerated. Each intelligent device in the work cell has its own set of conditions or assertions which determines whether an action involving that device was successful or not. This implies that error handling code for a device is naturally associated with the RIPE class that logically models that device. This satisfies our requirements for modularity and localization. To achieve transparency and to avoid combining device object methods with error recovery code, we create a new error handler software class hierarchy which parallels RIPE's device class hierarchy. In other words, a generic *ErrorHandler* class is defined, and then device-specific error handler classes are derived from this generic parent class. The two hierarchies are then tied together by each device class initializer which automatically creates an instance of its corresponding error handler class whenever an instance of the device class is created.

Having the two class hierarchies results in RIPE error handler methods being implemented separately but in parallel with its device class methods. RIPE also tailors each error handler for the specific device it corresponds with by using the model-based information for that device which resides in the device's class definition. Error handling capabilities are added incrementally to support new subsystems and to make old subsystems more robust as new error conditions and recovery mechanisms are discovered during work cell operations. This is done without modifying existing code and risking adding new bugs into the system. The generic *ErrorHandler* defines uniform, standardized ways for the device-specific error handlers to react to failures. This includes wrappers created as inline functions or macros which will generate calls to an error handler's interrupt service routine whenever an abnormal condition is detected.

To tie all of this together, a *Global_ErrorHandler* class has also been defined, which is instantiated whenever the main supervisory applications code begins execution. This class is derived from the *Task* class provided with the C++ environment. It utilizes UNIX signals to generate global interrupts in order to control the logic flow of the high level application whenever an error occurs. Another set of wrappers are supplied with this class which utilizes the UNIX *setjmp/longjmp* environment to provide options such as retry, continue, start over, skip, or abort, depending upon how well the local device-specific error handlers respond to a failure.

Currently, the *Global_ErrorHandler* and generic local *ErrorHandler* are in their initial stages of development and testing. RIPE supports evolution towards a long term goal for intelligent execution monitoring tasks. Such tasks will fully utilize the world model and sensors to maintain an accurate representation of the work cell in real-time as part of the overall software environment.

## 5. Conclusions and Future Work

The two completed implementations demonstrate that the design of RIPE has resulted in modular, reusable, extensible, and portable robot system software, and therefore has increased software development productivity and reliability of robot applications. The layered object-oriented software environment reflects the physical system. This simplifies the work for robot software developers by allowing them to construct control software environments in much the same way that the hardware system developers integrate the actual physical devices into a working system. This in turn facilitates communication between hardware and software engineers during system integration. Systems can be implemented faster due to the reusability and portability of the software. Also, RIPE can be used on most commercially available computing equipment because the development is based on a standard language and off-the-shelf operating systems. RIPE's device hierarchy and its communication interfaces which are inherent to object-oriented programming contribute to the development of a standardized Robot Independent Programming Language (RIPL) which is used to program different intelligent robot systems.

The ability to create RIPE objects in different ways by supplying different parameter sets provides for a very flexible system with an open-ended architecture having three levels of generic interfaces. An application normally will directly create objects representing specific devices and work pieces in the work cell, whereas the communication objects will be created internally, with the details of the message transmission hidden from the applications code. This provides generic interfaces that result in architecture independence. In other words, an application can be modified to run either in a single CPU environment or a distributed environment simply by changing the way it creates its objects, and consequently the way it communicates with work cell devices.

The second level of generic interfaces exists at the device level. Because there are generic device classes for robots and sensors (*Robot, Sensor, ForceSensor*), whose attributes and methods are inherited by specific robot and sensor classes (*CMRobot, LORDForceSensor*) derived from the generic classes, the application interface to a device will look the same no matter what device is used. In other words, if the Lord force sensor is replaced in the Cask Head Work Cell application by a JR3 force sensor, the application remains unchanged except for the way it creates its force sensor object (one line of code). This, in turn, leads to the third level of generic interfaces, the user level. Through inheritance and polymorphism, the same messages are sent to a *CMRobot* as are sent to a *GRobot*. Likewise, the same messages are sent to a *LORDForceSensor* as are sent to a *JR3ForceSensor*. This is illustrated by our two different work cell examples.

As a result, RIPL begins to develop as a natural consequence of RIPE. The code sequence already discussed in Figure 6 contains routine calls such as *receive_msg* for communications, *perform* and *move_rel* for robot control, and *set_bias, set_output_mode,* and *take_reading* for force sensor control. These calls, as well as the *Robot* declarations in Figure 3, form the basis for RIPL. Currently, RIPL is an intermediate manipulator level language, upon which a task-level language is being constructed. An example of this is the swipe command used in the Radiation Survey Work Cell.

We are currently enhancing the class hierarchies for the Radiation Survey Work Cell to perform new tasks such as mating a storage cask to a storage facility door, manipulating impact

limiters, and securing tiedowns. In addition, we are implementing the RIPL primitives for GMF and PUMA robot classes which will be used in future glovebox and inspection applications. We acknowledge that RIPE and RIPL must be evolutionary to be successful.

## Acknowledgments

The authors wish to acknowledge the contributions of the following: W. Davidson for his communications software and system integration support, B. Petterson for robot force control, and M. Griesmeyer, C. Selleck and J. Werner for their helpful discussions in the course of this work.

## References

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

[2] Blaha, J.R., J.P. Lamoureux, and K.E. McKee, "Higher Order Languages for Robots,' MTIAC Report AD-A193 796, October 1986.

[3] Cincinnati Milacron, *Communcations Manual Ver. 4.0 Robot Control*, Part No.5010321-134, Cincinnati, OH 45209.

[4] Griesmeyer, J.M., W.D. Drotning, A.K. Morimoto, and P.C. Bennett, "Cask System Design Guidance for Robotic Handling," *SAND89-2444*, (in preparation).

[5] Griesmeyer, J.M., "Generalized simulation environment for factory systems." *Tools for the Simulation Profession 1989*, 20-28, 1989.

[6] Hayward, V., and Paul, R., "Robot Manipulator Control under UNIX RCCL: A Robot Control 'C' Library," Int. J. of Robotics Research 5(4):94-111, Winter 1986.

[7] LaLonde, W.R., D.A. Thomas, and K. Johnson, "Smalltalk As a Programming Language for Robotics?," *Proc. 1987 IEEE Int. Conr. on Robotics and Automation*, 1456-1461, March31-April 3 1987.

[8] Latombe, J.-C., "Toward Automatic Robot Programming," *Proc. 1983 Int. Conf. on Advanced Robotics*, 203-12, Vol. 1, 1983,

[9] Lieberman, M.I., and M.A. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," *IBM Journal of Research and Development*, 21:4, July 1977.

[10] Linton, M.A., J.M. Vlissides, and P.R. Calder, "Composing User Interfaces with InterViews," *Computer*, 8-22, February 1989.

[11] Lozano-Perez, T., and P. Winston, "LAMA: A Language for Automatic Mechanical Assembly," *Proc. 5th Int. Joint Conf. on Artificial Intelligence*, 710-716, Aug. 1977.

[12] Meyer, B., *Object-oriented Software Construction*, Prentice Hall, 1988.

[13] Miller, D.J., "Supervisory Control for a Complex Robotic System," *Robots 12/Vision '88*, June 6-9 1988.

[14] Nackman, L.R., M.A. Lavin, R.H. Taylor, W.C. Dietrich, and D.D. Grossman,"AML/X: A Programming Language for Design and Manufacturing," *Proc. 1986 Fall Joint Computer Conf.*, 145-59, November 2-6 1986.

[15] Petterson, B., R. Robinett, and C. Lennox, "Lag-Stabilized Force Feedback Damping," in preparation.

[16] Stroustrup, B., The C++ Programming Language, Addison-Wesley Pub. Co., 1987.

[17] Sun Microsystems, *SunView Programmer's Guide*, Part No. 800-1345-10, Mountain View, CA 94043.

[18] UNIMATION Inc., *User's Guide to VAL-II, Programming Manual*, Version 2.0, #398AGI, Shelter Rock Lane, Danbury, CT. 068100.

[19] Van Brussel, H., D. DeWinter, P. Valckenaers, and H. Claus, "A Universal Programming Structure for Multi-robot Assemble Systems," *Proc 8th Int. Conf. Assembly Automation*, 209-226, March 1987.

[20] Volz, R.A., and T.N. Mudge, "Robots Are (Nothing More Than) Abstract Data Types," *Robotics Research: The Next Five Years and Beyond*, August 1984.

[21] Volz, R.A. "Report of the Robot Programming Language Working Group: NATO Workshop on Robot Programming Languages," *IEEE J. of Robotics and Automation*, 4:1, February 1988.

[22] Wind River Systems, Inc., *VxWorks Reference Manual, Ver 4.0*, Emeryville, CA 94608.