Complex Control System Design and Implementation Using the NIST-RCS Software Library*

Mathew L. Moore Veysel Gazi Kevin M. Passino
The Ohio State University
Department of Electrical Engineering
2015 Neil Avenue
Columbus, Ohio 43210

Will P. Shackleford Frederick M. Proctor
The National Institute of Standards and Technology
Intelligent Systems Division
Gaithersburg, Maryland

Introduction

Due to demands for higher levels of automation from systems, the use of hierarchical and distributed control systems that integrate estimation, prediction, control, fault tolerance, adaptation, learning, etc. into one framework is becoming widespread. Specific applications that utilize complex control of this sort include robots [1], [2], [3], [4], [5], [6], [7], manufacturing and automation systems [8], [9], [10], [11], automated highway systems (AHS) [12], [13], [14], [15], [16] and other intelligent vehicles [17], [18]. Here, we study the use of the Real-Time Control System (RCS) software library, that was developed by the Intelligent Systems Division (ISD) at the National Institute of Standards and Technology (NIST) for the design and implementation of such complex control systems. We examine both the theory behind the development and implementation, and provide one physical implementation example (a process control problem) and a conceptual example of how to use RCS in the design of a hierarchical controller for an AHS.

Designing a control algorithm based only on isolated conventional control methods will not suffice to achieve autonomous or even proper operation of a large control system that needs to perform many complex tasks in real time. For such control systems it may be difficult or even impossible to develop an analytical model that describes the overall operation. For many years researchers have been trying to develop a systematic approach for design and implementation of these types of control systems. The usual approach for overcoming the complexity is to break down the problem to smaller and easier to solve sub-problems (and conventional control approaches are sometimes quite useful for these). The resulting control algorithms often use a "hybrid" or coordinated combination of control methods and address the control of both the continuous and discrete-event components of the plant. This has led to the introduction of several functional

*This work was supported by the Intelligent Systems Division of the National Institute of Standards and Technology (NIST) and the Center for Intelligent Transportation Systems at OSU. Please address all correspondence to K. Passino ((614)292-5716, k.passino@osu.edu). NIST: No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

architectures that integrate a variety of methods to try to achieve autonomous operation [2], [19] (to be addressed briefly in the next section). In general, these architectures have a hierarchical structure and provide coordination of the physically distributed subsystems to achieve system-wide goals (e.g., in AHS there is a controller resident on each vehicle for vehicle guidance and a need for coordination between the vehicles to ensure safe and efficient operation of the system, or simply to minimize accidents and maximize throughput [13], [14], [15], [16]).

Challenging practical issues surround the implementation of complex real-time control systems. One example is the development of communications between multiple, separately operated subsystems whose behavior may be interdependent. Moreover, different subsystems may be running on different and incompatible platforms making the problem even more difficult. It is desirable to develop a control algorithm which is system independent, so that it can be operated on different computer hardware and software. This saves the designer the overhead of reprogramming the whole algorithm if there is a change in the system. Moreover, we want to be able to port and reuse software developed in one application in another application.

Other important issues in development and implementation of complex control systems are the ease of maintanence, modification, and operator interface. For example, assume that in a system with hundreds of sensors, one sensor fails. We want to develop the control system so that the operator can locate the failure in a short time. As another example, assume that a new subsystem is added to the overall system. We want to implement the initial control system so that to make the necessary modification we do not need to reprogram the whole control algorithm. As the complexity (and size) of the plant grows, issues such as these quickly become more complex and challenging.

The RCS methodology discussed here is based on a task decomposition analysis of the overall operation of the plant. This helps the designer to decompose the problem into several simpler subproblems to be performed by different subsystems at different time periods. Usually the resulting control system consists of several hierarchical layers of functional control *modules* that can be distributed over multiple computer systems or platforms. Each module (often referred to as an "RCS module") can contain its own sensory processing, decision-making/state-table execution, and actuation components. At the modules in a higher level of the hierarchy, the tasks are decomposed into smaller task sequences and passed as commands to the lower subordinate layers. RCS places no limitation on the physical location of these modules. RCS provides the communications tools, via the Communication Management System (CMS) and the Neutral Message Language (NML), that allow different RCS modules, which can be placed on separate computer systems running on different platforms, to "talk" with each other. This allows for the distributed and hierarchical control of an arbitrary number of subsystems by linking several modules across multiple backplanes.

RCS provides a general controller architecture without specifying implementation details. As a result, it does not limit the possible control applications—both conventional, single-input, single-output controllers and complex "intelligent" autonomous controllers can be implemented in RCS (see [2], [12], [20] and Chapter 2 of [19]). The user determines the layout and use of the RCS modules to minimize controller complexity and maximize its performance. Moreover, since the development of controllers in RCS is uniform, code developed for one application can easily be ported to another. This porting can even occur across different operating systems, as the RCS library is supported under numerous platforms. Furthermore, the RCS library has a diagnostics tool that provides an operator interface. In other words, using this tool a human operator can monitor the system operation from a remote host and send sophisticated commands to the system. Another important feature of the RCS library is the RCS design tool which allows the application designer to easily layout the modules in the controller hierarchy and automatically generate the skeleton of the application including almost all of the application independent code.

The need for tools for designing and implementing real-time complex control systems has lead to development of several software packages for this purpose including Network Data Delivery System (NDDS) from Real-Time Innovations (RTI) (http://www.rti.com) which is a package with similar features to NML; the Control Shell, also from RTI, and LabView from National Instruments (http://www.natinst.com) which have similar features to the graphical RCS design and diagnostics tools; Open Robot Controller Computer-Aided Design (ORCCAD) [3], which is based on Esterel synchronous programming language, and developed by INRIA (http://www.inria.fr) in France; Onika [8], which is based on Chimera real-time operating system, and developed by the Advanced Mechatronics Laboratory at Carnegie Mellon University; and Teja by Teja inc. (http://www.teja.com). Moreover, new software packages are being developed every day, and each of these is continually having new versions implemented; hence a valid comparison of the packages today may be inaccurate tomorrow. Furthermore, we find that evaluation of different packages, especially along the lines of ease of use (which is very important), tends to be rather subjective. Therefore, we will not attempt a comparison to other packages. However, it is important to stress once more some of the important features of the RCS library including platform independence, support to a number of communication protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Remote Procedure Call (RPC), graphical automatic design and code generation tools, graphical user interface, taking advantage of the hierarchical system topologies and finally and perhaps most importantly the fact that RCS is free, which provides big advantage for this package, both for universities and for many parts of the industrial sector. To download the RCS library just visit the the anonymous ftp site `ftp://ftp.isd.mel.nist.gov/pub/emc/rcslib/`. For detailed information on the RCS library you can visit `http://www.isd.mel.nist.gov/projects/rcs_lib/` or consult [21].

During the past few decades RCS has been implemented and tested in many applications including mining [9], manufacturing systems [4], [5], [6], [10], [22], autonomous undersea and land vehicles [17], [18], space station telerobots [7], post office automation [11], etc. With the development of the RCS software library, the design and implementation of similar applications becomes much easier and faster. It can be used in different industrial, military or educational applications. The industrial applications may range from large scale chemical plants, manufacturing lines, etc., to small conventional control systems. In the universities it can be taught as a class and used in laboratory experiments [23]. This article (which is a significantly expanded version of [23], minus components on the educational program), introduces the RCS tools and examines the use of RCS as a controller architecture for complex systems by providing both the general structure and theory and several examples of its implementation. We begin by introducing the relation of the RCS software library to some functional architectures, then we discuss the RCS design methodology and the RCS design tool, following which we describe two of the important features of the RCS library, the RCS generic controller nodes and the RCS communication tools, which are described in more detail in [21]. Finally, we introduce two RCS controller designs, one lab experiment implementations (a tank experiment) and one on an automated highway system.

RCS and Intelligent Control Architectures

An intelligent control system is one that is highly autonomous and able to take appropriate control actions, even in an uncertain environment. One can obtain an idea of how an intelligent system should be constructed by observing intelligence in nature. At the very least, intelligence requires the ability to sense the environment and take appropriate control actions. At higher levels, intelligence involves the ability to analyze and understand, learn, generate plans for the future, and decompose complex tasks into simpler steps that can be efficiently carried out. Certainly, it is also advantageous to have the ability to adapt to changing conditions (both in the plant and in the environment). Higher degrees of intelligence also incorporate evolution, in which more successful behavior is retained and less successful behavior dies out [2], [12], [19], [20].

More and more research in control system theory is incorporating this idea of intelligence. One of the main hurdles in constructing autonomous systems is the fact that it is difficult to understand and model intelligent behavior completely, and thus implementing intelligence in digital control algorithms becomes a large concern. Clearly, however, basic levels of intelligence involve certain concrete components, including sensory processing, modeling, analysis and evaluation, decision-making, learning, and actuation. Thus, it is possible to develop an *architecture* that models, at a simple level, the intelligence that occurs in nature. Several such architectures exist [2], [19], [20], and these are continually being evolved as we begin to understand intelligence more clearly. The creators of these architectures tend to use hierarchical levels in the model of complex systems, as shown on the right side of Fig. 1 (a simplified version of the architecture in [20]), which allows the possibility of subsystems that may operate at different rates, and for the decomposition of tasks. Subsystems, referred to as *modules*, at the highest level take high-level commands and decompose them into smaller tasks that are sent to the modules below. They also maintain a model of the system itself and the environment surrounding it. The left side of Fig. 1 shows the detailed components of a module. Note that structures used for sensory processing, world modeling, value judgment and behavior generation are typical in intelligent systems [20]. The lowest level modules simply gather sensor information from a data acquisition card (sensory processing) and implement simple single-loop feedback control algorithms (value judgment and behavior generation at the simplest level) in response to the evaluation of the problem or command coming from higher levels. They may also maintain a model of the physical subsystem they are controlling (a world modeling structure that contains more detail but at a smaller scale than the higher modules). Higher level modules may incorporate long-term planning and evaluation in the behavior generation and value judgment operations.

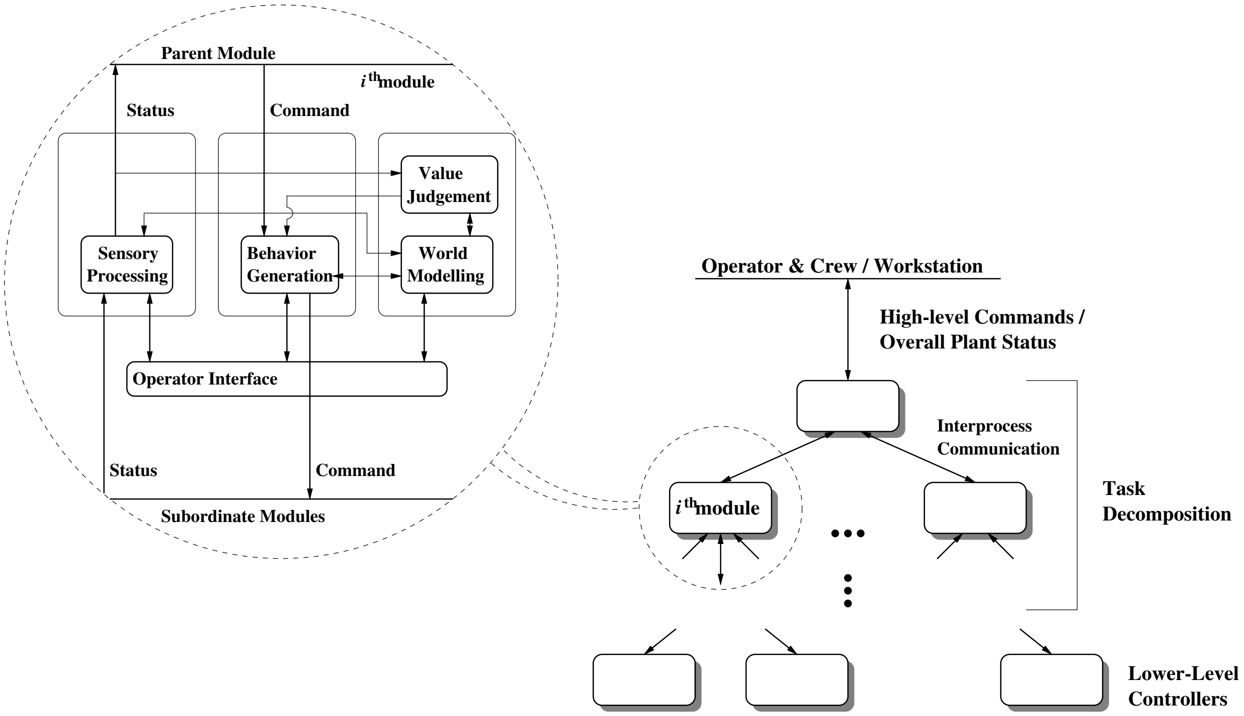

Fig. 1. Example intelligent architecture.

A significant concern with complex systems is the communication between modules. Communications set the infrastructure for intelligent and complex designs. Since high-level commands are decomposed throughout the levels, it is necessary for higher modules to pass commands to lower modules so that tasks can be carried out. Furthermore, for system modeling purposes, long-term planning, and fault detection, it is necessary for lower modules to be able to communicate status information to those at the higher level. Communications become an even larger concern when algorithms may be running across several computers. Clearly, intelligent and complex systems require two important components—communications between modules and the modules themselves.

The RCS library provides the software tools that allow us to model this general complex architecture simply and efficiently by providing basic generic controller modules and the necessary tools for connecting these modules to each other in a hierarchical structure. This will become clear to the reader through the next several sections, where the RCS library and its components are introduced in detail and application examples are provided.

Designing an RCS Application

The RCS Methodology

As with most of the design methodologies for complex control systems, the RCS design methodology is based on decomposing the complex control problem to smaller and relatively easy to develop, maintain, and modify subproblems. Therefore, a typical RCS design starts with task decomposition analysis of the system. First, the designer identifies the physical subsystems of the overall system, the corresponding sensors and actuators to these subsystems, the tasks subsystems can perform, which subsystem performs which tasks, which tasks are related to each other, what is the information or data flow between the subsystems, and which tasks sequences lead to desired performance. This helps the designer to define the task parameters and simplifies the control problem since every task can be considered as a separate small scale control problem. The designer can continue to break down the tasks to even smaller subtasks until the tasks are reduced to simple control problems that can be solved using conventional control methods.

To illustrate, consider a task called "Move Part J from location A to location B" to be performed by an autonomous robot. This task can be broken down to "Move to location A," "Grab part J," "Move to location B" and "Release part J." These tasks can be broken even further to simpler subtasks. For example, one can compose the task "Grab part J" from "Open the gripper," "Align the gripper with part J" and "Close the gripper." Note also that each of these subtasks may be performed by different subsystems of the robot. For instance, "Grab part J" will be performed by the gripper, whereas "Move to location A" will be performed by the motion subsystem.

Having performed task decomposition analysis of the control system, the designer defines the controller hierarchy. It is a good practice to start by assigning a control module for each actuator and its corresponding sensors at the lowest level of the hierarchy. Then the modules in the higher levels are defined by grouping the lower modules based on physical and functional relation and assigning a supervisor to them.

After task analysis and controller architecture definition, the designer knows which operations can be performed by the system, which subsystem or module will perform which task, which task sequence scenarios lead to a desired performance, and what information the modules in the hierarchy need to share. To make the design even simpler, the designer breaks down each task into state tables using state table analysis. State table analysis can be done by breaking down the task to a sequence of operations in time which represent different "states" of the system. First, the state tables of the bottom modules are defined and then the ones for the upper modules are based on the chosen controller hierarchy and upper level task decomposition. In general, one iterates and redefines the task decomposition structure and the controller hierarchy as the state tables become better defined. After completion of the procedure, since the designer has determined the task knowledge and information to be shared between the modules, the command and status message vocabulary is also defined.

Having finished the analysis of the system and the design of the RCS controller the engineer faces the challenge to convert the control algorithm to a computer code, or in other words, to implement the RCS application. The RCS design tool, which is a Java-based graphical tool, is used to help the designer at this stage and to generate most of the related code automatically.

The RCS Design Tool

The RCS design tool is a Java-based graphical program that can be used from any web browser which supports Java, or can be run as a Java Applet or as a stand alone application provided that the Java Development Kid (JDK) is installed in the system. It provides a graphical user interface (GUI) through which the programmer can layout and build a hierarchy of controllers for an RCS application and automatically generate most of the application code. The tool provides the ability for easily adding to or removing modules from the application. It can modify the hierarchy, set up communication channels between modules, define command, status, or auxiliary messages to be passed between the modules, etc. Using it the designer can assign different cycle times for the modules based on their timing requirements. It also generates the code of the NML servers (to be discussed later) and command line scripts needed for compiling and running the application together with all of the application independent code. In other words, using the RCS design tool a "skeleton" code of the RCS application is generated and only the application-dependent parts are left for the programmer to fill in. To illustrate, consider the task "Close gripper" to be performed by the gripper module of the autonomous robot mentioned earlier. In the design tool we define a command for this task, `CLOSE_GRIPPER`, to be passed by the superior of the gripper module. This command can contain data variables to specify how much to close the gripper, which control algorithm to use, any variables that are needed to specify the conditions that are needed for the task to be initiated or finished, etc. Note that the RCS designer specifies which information needs to be passed. Once the programmer defines the command `CLOSE_GRIPPER` in the RCS design tool, the tool generates all the code related to this command together with an empty function for the actual control algorithm since the control algorithm can be application-dependent and user determines which algorithm to use. Then the programmer can easily insert within the body of the function (or import form another application) the actual low-level control algorithm for closing the gripper. The control algorithms are based on the state-table analysis of the given command and are, in general, implemented in `if-then-else` form. This allows the current state of execution to be tracked within the RCS diagnostics tool. Note that the state tables for the commands are application dependent and the RCS design tool does not generate them.

The tool handles establishing communications between modules that lack a supervisor-subordinate relation (auxiliary communication channels). It also provides the ability to generate code for different platforms including DOS, Windows 95/NT, Linux, LynxOS, SunOS, VxWorks, IRIX. Thus, the user can build an RCS application using the RCS design tool and generate most of the code for it, and then "fill in" the application-dependent control or estimation algorithms. This frees the programmer form dealing with the interprocess communications, which may involve low-level network programming, so that you can concentrate on the control of the system. Consult [21] for detailed description of the RCS design tool.

The RCS Library Components

As we discussed earlier, to develop a controller composed of a hierarchy of control modules, which do not necessarily run on the same computer system, we need the control modules themselves and means for communication between them. The RCS library provides these tools. In this section we will discuss the two important components of the RCS library, the RCS communication tools and the RCS control modules.

The RCS library is composed of a set of C++ program segments. It takes advantage of the programming language's structures and abilities, including classes, inheritance, and virtual functions. A C++ class is simply a structure that groups together aggregate data types that are typically related due to application (rather than type). The components of a class are called its *members*. Members can range from data variables necessary for handling information or data to functions that operate on the members of the class. The idea of inheritance allows the user to derive new classes from a base class. This effectively produces a new structure that contains both the members of the base class and the members of the new class, grouping them together as one object. RCS uses this idea to develop a base class containing functions and variables for communications between RCS modules. The RCS module itself is a class that is equipped with all the tools for communication with its superior and subordinates and also with additional functions and data variables commonly used in RCS control routines. Although it is helpful to have some exposure to either C or C++ programming to understand the details of the RCS software library, this article will focus on the higher-level design only and include low-level RCS code only where it is crucial for understanding the overall design.

RCS Communications

The keys to the portability and standardized architecture of RCS are the use of the CMS and NML. CMS is a library that contains system-dependent operating system calls that are contained in the CMS base communications class. These are crucial in establishing communications along a network. Communications are provided through a series of shared memory buffers. Communication between RCS modules (which may be running on the same or separate computers) occurs in a message-based format by having one process write a message to the buffer and a second process read the resulting message from the same buffer. The

CMS class provides the functions that perform the reading and writing of the data messages to the memory buffers, as well as many additional functions. The passing of information across networks occurs using Internet Protocol (IP) protocols—TCP, UDP, and RPC are currently supported. RCS passes information across different operating systems by encoding data in a neutral format before writing the message to the buffer. Several different encoding structures are available, including ASCII (American Standard Code for Information Interchange) and Xdr (eXternal data representation). The location, size, and other attributes of the memory buffers and communication processes can be user-specified in a configuration file.

NML provides a set of classes that provide a higher-level interface to CMS, so that the user need not be concerned with the low-level operating system processes contained in CMS. Thus, using NML, we obtain a uniform interface to the RCS communications. NML also provides the base classes for setting up the messages that are communicated between RCS modules and processes, as well as those for producing the RCS modules themselves.

An example communications structure for an RCS application using NML is shown in Fig. 2. This figure represents processes running on three separate computers (or backplanes). Each process could represent a module within a single hierarchical structure or contain a complete structure within itself. The design of the system is left to the developer; RCS merely provides the mechanisms for producing a complex architecture. In the example communication structure, all memory buffers are located on a single computer (Computer 1) and are directly accessed by Processes 1, 2, and 3. These processes are labeled as LOCAL to the memory buffers. Processes 4, 5, and 6 are considered REMOTE to the buffers. The need for the NML servers is discussed later. Note that this structure is completely arbitrary. The user can set up an RCS system in any fashion, with any placement of buffers. Time-critical tasks, however, should generally be placed such that they access only LOCAL buffers. This can result in a significantly faster operation, since the use of remote buffers often involves network communication delays.

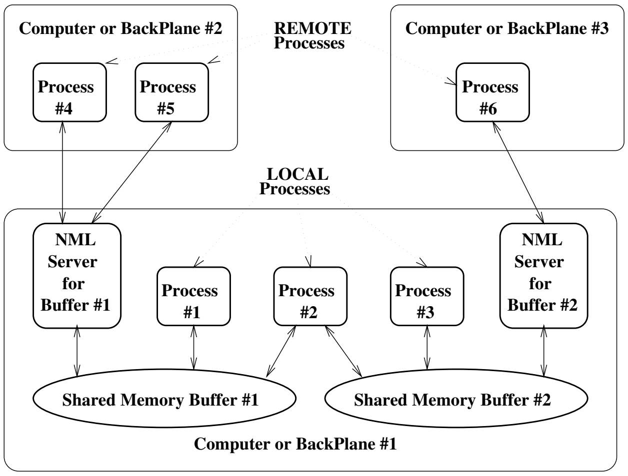

Fig. 2. Example RCS communications structure.

Communication Channels

A communication channel exists between modules that need to communicate with each other. For communications between processes to exist, two items must occur—the memory buffer must be established, and the process must set up access to this segment of shared memory. Each shared memory buffer must have a *master* (i.e., an RCS process that is responsible for creating that buffer). Communication channels between buffers and processes are set up using functions available in the base NML class (`class NML`).[1] Which RCS

[1] We use the tele-type font throughout this paper to refer to computer code.

processes access which buffers is established using an NML configuration file. This is an ASCII file that lists the buffers and processes and contains the desired size of the shared memory buffer, location of the buffer, communication methods used to access buffers, which processes access which buffers, the buffer's master, etc. When an object of `class NML` is instantiated so that Process A can access Buffer A, the NML constructor (function for creating a new object) is called, which reads in the designated configuration file, determines if Process A is the master of Buffer A (if so, the buffer is created), and establishes a communication channel so Process A can access Buffer A in the method that the configuration file specifies. An NML class object must be established for each process-to-buffer communication channel, since it contains the variety of functions used to perform reads and writes between the process and the buffer.

For processes to access a buffer that is physically located on a different computer, an NML server must run on the same computer as the memory buffer. The NML server encodes and decodes the messages from that buffer on behalf of the remote process. Using servers to provide remote access to the shared memory buffers frees local processes from being slowed down by the communications with remote processes. The RCS library provides the means for running the NML servers as well.

Communication channels usually connect a module with its subordinates or superiors, as the architecture discussed earlier suggests. Therefore, the RCS library has classes for establishing specific communication channels, command channel class, `RCS_CMD_CHANNEL`, and status channel class, `RCS_STAT_CHANNEL`. These classes are derived from the NML base class and have some additional command or status channel utilities. Moreover, RCS also provides the ability to set up auxiliary communications so that two modules on the same level or separated by several levels may directly communicate with each other.

Messages

In RCS applications, the designer needs to specify the types of messages that will be communicated to different modules. A set of C++ classes acts as the message vocabulary that is used to create message classes. A typical message contains several members—data variables to hold the message size and unique identity number of the message, data variables that will be application specific, and a function that tells NML how to update, or encode/decode, the members of the message class from the encoded format so that a proper read/write can occur. This last item is referred to as the `update()` function. The data variables placed in the class are the components of the message that are written to and copied out of the memory buffer during a read and write cycle. As such, the NML message the user develops should contain the data that needs to be passed to other applications. NML provides a base class `NMLmsg` for producing a message. Application specific messages are created by deriving a new class from `NMLmsg`. `NMLmsg` contains the member function for storing the size and unique ID number of the message and a function for producing the message (called a C++ *constructor*).

Although NML classes contain `read()` and `write()` functions for communicating messages, these functions generally are used as a high-level interface to the CMS communication tools. The actual writing/reading of the messages to and from the memory buffers is accomplished by the low-level CMS communication methods. One of the advantages of using shared memory buffers for communications is that each member of a message can be written to the buffer, or read from the buffer, individually. In CMS methods, the message is encoded in a neutral format using the `update()` functions. The unique ID number is also encoded. However, it is handled differently than the remaining items of the memory so that other processes can access this variable directly without reading in the entire message. Since all messages contain the common data member for the unique ID of the message, this variable is first encoded separately from the rest of the message. CMS methods will always write these common variables to the same places in the shared memory, regardless of the type of messages, so all other RCS processes know where to look in the memory buffer to find the ID number of the message. Once the ID number is known, the type of the message is known, and CMS methods call the appropriate `update()` function to read/write and decode the data members of that particular class. Since message contents are application specific, the developer needs to create an `update()` function specific to every additional message that is created. This function simply calls the CMS `update()` function for each data variable in the class. The RCS library already contains `update()` functions for most of the basic C data types, including floating-point numbers, integers, characters, etc., and as a result, writing an `update` function for the user-specified message simply involves calling the RCS library `update()` functions for each data variable in the message.

Although messages of any type can be constructed, RCS provides additional tools for producing two specific types of messages—command messages (`class RCS_CMD_MSG`) and status messages (`class RCS_STAT_MSG`). Command messages are those that are passed from higher modules to lower modules. They request that an action be performed. Status messages are passed from lower modules back to their parents, transferring data that characterizes the current status of the lower module. `RCS_CMD_MSG` and `RCS_STAT_MSG` are two classes derived from `NMLmsg` with some additional data fields. Note that the user can define the messages of an application within the RCS design tool. Moreover, all the communication channels between the modules will be established automatically by the design tool once the controller hierarchy is defined. Therefore, all these do not bring any extra programming overhead to the RCS designer.

RCS Control Modules

Developing a complex hierarchical system using the individual NML communications and messaging tools would be tedious. NML is provided as the communications workhorse between applications and multiple processes. It does not provide the tools for actually producing modules of a complex hierarchical control structure. To do this, RCS provides additional software tools based on a set of classes that combine the communication abilities of the NML classes with standard functions used in RCS control routines to allow for the production of the hierarchical structure. The RCS library contains a general *controller module*, also referred to as an *RCS module*, *NML module*, or *RCS template*, which takes care of many of the cyclic processes needed in a hierarchical structure. The hierarchy is established through the NML configuration file, which lists the buffers and processes of a particular RCS application and tells which processes access which buffers. In general, RCS applications have a static structure. In other words, their structure does not change during run time. However, the RCS library can handle some dynamic structures, too. Each NML module is associated with at least two buffers—a command buffer (readable to the module) and a status buffer (writable to the module)—and has both parent modules and subordinate modules. Parents have access to the subordinates' command and status buffers. Generally, the parents obtain status information from the subordinate's status buffer and maintain the ability to command the subordinates with the ability to write command messages to the command buffer. This is illustrated in Fig. 3. Module 1, at the top of the hierarchy, has the ability to command modules 2 and 3 by sending command messages to modules 2's and 3's command buffers. Likewise, module 1 can gather status information by reading module 2's and 3's status buffers. Each node of the hierarchy behaves in this fashion. The operator interface can send commands to the upper level module, where initial task decomposition occurs.

When each module is "activated" (usually on a cyclic interval), the module first checks for commands from the parent and reads the statuses of the subordinates, then calls the user-written preprocess, decision process, and postprocess functions in that order. The preprocess function (`PRE_PROCESS()`) could be used, for example, to obtain sensory data, or implement estimation procedures. The decision process function (`DECISION_PROCESS()`) calls one of the command functions of this module based on the command to be executed this cycle time. If there is no request for a new task then the module continues the operation on the last received command. The command functions (which the programmer needs to develop) may (in higher modules) decompose tasks into a series of actions that are passed as commands to subordinates or (in lower modules) calculate low-level controller outputs based on its incoming command using a predefined control method. NML modules provide functions that can write command messages to the buffer of a subordinate. The postprocess function, `POST_PROCESS()`, is reserved for updating the module's status variables, which are passed to its parent, and possibly for sending out actuation signals. The NML module completes the cycle by updating its output buffer, which generally includes writing its current status to its status buffer, and possibly commands to its subordinates.

The operation of a single cycle of an NML module is shown in Fig. 4. Items boxed in a dashed line are functions already available in the RCS library; they do not need to be written by the user. Those items

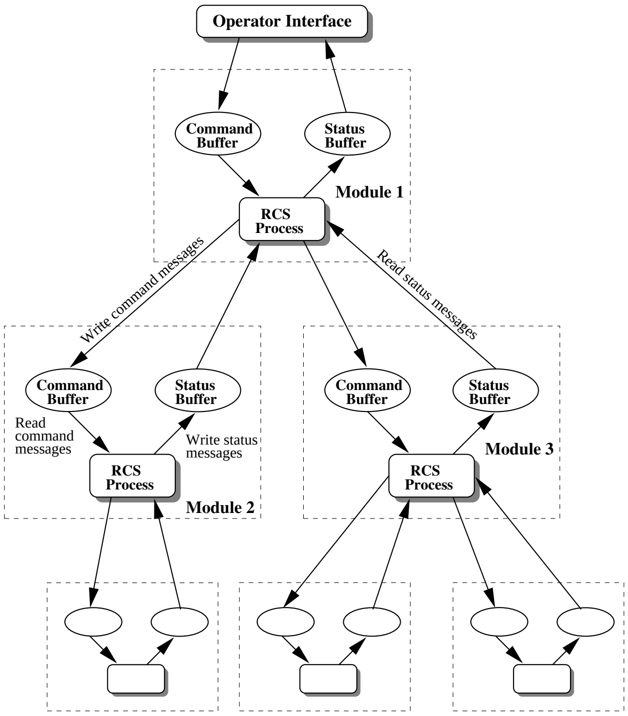

Fig. 3. General RCS hierarchy.

in solid lines are application-specific code that the user needs to develop. (A skeleton of these functions is generated by the RCS design tool.) The main loop is implemented once every sampling period of this particular module. To "activate" the controller module, a simple call to the NML module's `controller()` function is necessary. This function in turn calls functions that start the timing for the cycle (for diagnostics purposes) and read the input buffers of the particular module, for example, the command buffer and the subordinate status buffers. Then, application-specific code is implemented in the order given above, for the preprocessing, decision processing, and postprocessing functions. The `controller()` cycle ends with the writing of any necessary messages to the output buffers, where, for example, commands may be passed to subordinate modules. Note that different commands will cause different functions to be called in the `DECISION_PROCESS()` resulting in different operation.

User Interface via the RCS Diagnostics Tool

The RCS diagnostics tool is a package developed by NIST to provide an operator interface to an executing RCS application. Similar to the RCS design tool, it is a Java-based program that can be used from any web browser that supports Java, or can be run as a Java Applet or as a stand alone application. It allows a human operator to interact with an RCS application. A user can view the status of the modules in the hierarchy and send commands to them using the diagnostics tool. The files needed by the diagnostics tool can be generated automatically by the design tool; therefore, no additional programming overhead is required.

It has different views through which you can view the status and command messages or the hierarchy of the application, send commands to the modules in the application, view the state tables, plot status variables, etc. These options which are as follows:

- `Login` This option is used to prevent some users from having complete access to the NML buffers.

- `Details` This option is used to view the available command and status messages, modify the fields of

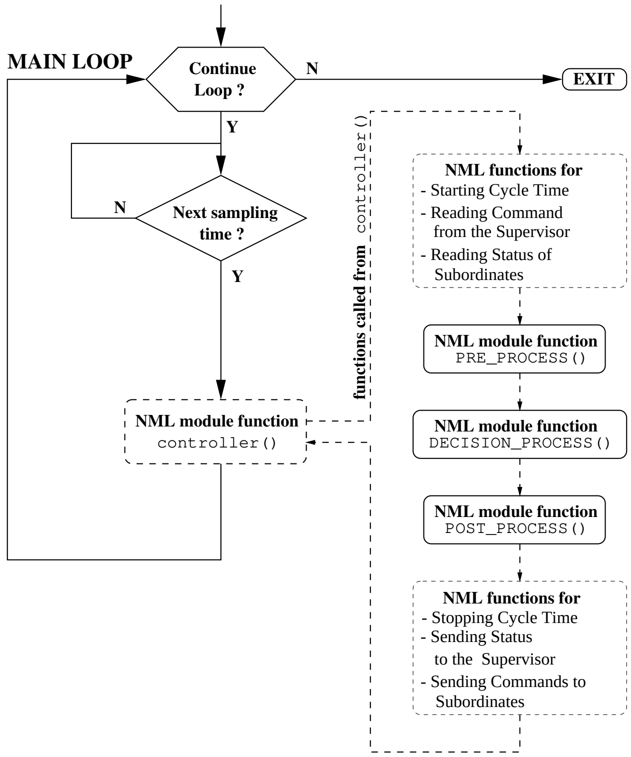

Fig. 4. Operation of the NML module.

the command messages and send any commands if needed. Here, we can also choose which fields of which status or command messages will be plotted.

- `Auxiliary Channels` This option is similar to the `Details` option except that it shows the auxiliary messages.
- `Hierarchy` We can use this option to view the controller hierarchy. The modules appear color coded based on their current status of operation.
- `Graph` This option is used to view the plots of the variables that were chosen to be plotted in the `Details` option.
- `Error Log` This option is used to view the error messages (in case of any) logged to the special buffer called `errlog`.
- `State Table` This option shows the place within the state table which is currently executing.
- `Debug Flags` This option is used to set on or off some debug flags.

You can find more detailed information on the RCS diagnostics tool in [21].

RCS Application Examples

In this section, we introduce the use of RCS as a design method using an actual implementation of the RCS library on a laboratory experiment. The simplest application is for the implementation of a single controller to achieve a single task on a single experiment. We have also implemented such a system in

RCS for a rotational inverted pendulum in our laboratory, and some details on this implementation are provided in [23]. In this section, we will provide a description of a more interesting RCS application for a process control experiment where there are multiple objectives and subsystems. In the next section, we give a conceptual view of an RCS design for an automated highway system. Since an AHS is a complex system that requires large computation power as well as all the aspects of the intelligent autonomous architecture (including fault tolerance), it provides a good testbed introduction for the RCS design architecture.

A Process Control Experiment

A process control experiment can provide a good illustration of the power of the RCS design tools, since it is a practical application that can vary in order of complexity. A simple process control experiment could consist of simply two tanks, one for mixing and another for storage, as shown in Fig. 5. With each tank, we have a pump, heater, and mixer, as well as sensors for determining temperature and level of tank contents. Possibly the easiest way to implement such an experiment in which temperature and level values are being controlled is to run two controllers serially (that is, one at a time) in one program. However, this quickly becomes a waste of resources. First of all, the required sampling times are different—temperature control can operate at a relatively slow rate compared to level control. Also, the two tanks could be located at different locations (across a plant floor, for example), making it more efficient to separate the controllers for each of the tanks. As the complexity of the plant grows, physical constraints may require the use of a distributed control structure. The addition of several chemicals, each of which must be accurately heated to a specific temperature, forces us to have accurate control of the rate at which several chemicals are mixed. As we increase the number of actuators, the creation of a single program to control all processes becomes inefficient and perhaps even costly in terms of recovery from faults.

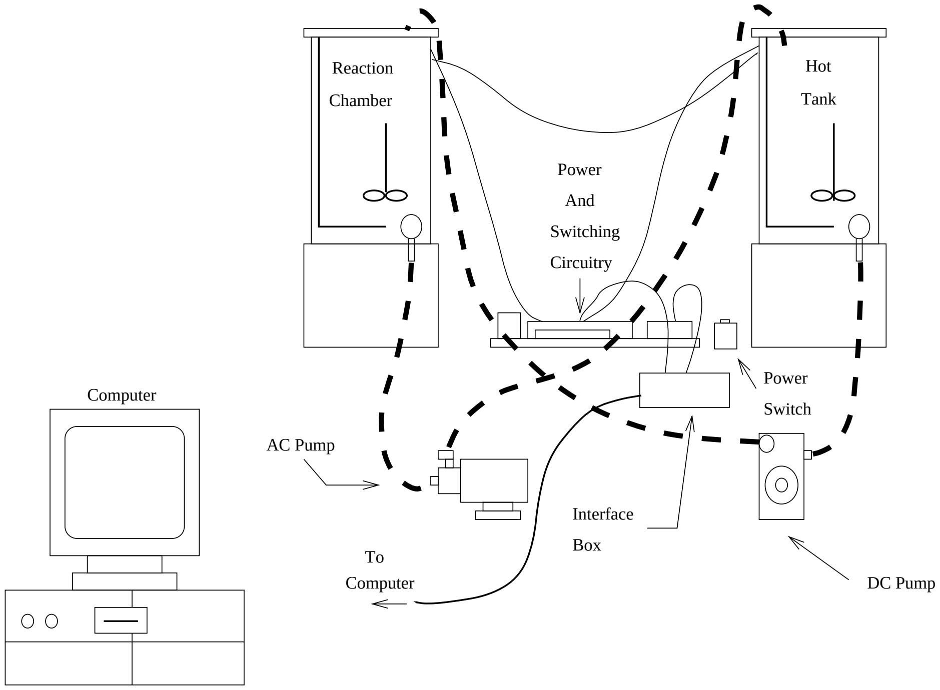

Fig. 5. The process control experiment (figure drawn by Scott Brown).

In a process control experiment where everything is interdependent, it is necessary to share information about each tank with the other processes controlling the mixing. With an RCS design, the process control experiment can be broken down into sets of subprocesses and modules without eliminating the ability to communicate and share information between processes. Processes can act independently while they are still linked together, sharing information that is crucial for successful production. With RCS, temperature and level control for each tank can be operating close to the tank while communicating with the other processes to which it is linked. These tank controllers can be spread out throughout a plant, and all information could be shared with a supervisor located local to a plant operations panel. A plant operator could then have the power to change the mixing process and set points of all tanks from a single location.

RCS Analysis and Design for the Tank

We set up the simplified control problem where we regulate the level and the temperature of the liquid in the reaction tank at specified reference values that can be set by an operator from (possibly) a remote host.

In this simplified control problem we do not have to use all the actuators and sensors available in the system. In fact, in the simplified system we have only two sensors and four actuators. The sensors are for sensing the level and temperature of the reaction tank and the actuators are the heater, two pumps for filling and emptying the reaction chamber, and the stirring mechanism in the tank. The stirrer is used in order to counteract the disturbance due to the high pressure pumping (this allows for a more accurate reading in our level sensor) and also to mix the liquid so that there is no temperature difference in different sections of the tank.

This set up immediately suggests that on the first level of the hierarchy we have four lower-level modules, one each for the heater, the two pumps and the stirrer, and a supervisor on the second level to supervise or coordinate the actions of the four lower-level modules. This is a fairly simple hierarchy, however, we simplify it even more by combining the two pumps into one module. Moreover, we let the stirrer always be on as long as the program is running and hence remove its control module from the hierarchy.

Simple task analysis shows that the module for the two pumps (which we will call the "level module") can either pump the water in the reaction chamber (i.e., increase the level) or pump the water out of the chamber (i.e., decrease the level). These two actions are similar, because in either case we increase or decrease the level to a predefined reference level (we do not allow the tank to overflow or to become empty). Therefore, these two tasks can be referred to as two different states of a single task of setting a reference level. Another two states of this task are the idle state and the error state which occur respectively if the reference level is reached or some problem, such as failure of a sensor or actuator, arises.

The task analysis for the heater is even simpler because the heater is either on or off, hence it is either heating or not. The task of controlling temperature can be in either the state of running the heater, idle state or error state.

In order for the supervisor to be able to coordinate the actions of the level and temperature control modules, it needs the values of the current level and temperature in the tank. Therefore, these values need to be passed from the lower modules to it. Based on this information it will send commands to its two subordinate modules for changing set points and stopping and starting the control algorithms.

Messages that are passed between RCS processes are application specific, and therefore the developer needs to define the messages for each particular application. After determining the tasks that each module can perform and what data from one particular module is needed by another module or a human operator, the message vocabulary is also determined. From the task analysis above we know that the level module will accept a command for setting a reference level `LEVEL_SET_REF`, which will provide the value of the reference level, and the heater module will have a command for setting the reference temperature `HTR_SET_REF` which will provide the value of the reference temperature, as well as additional commands for initializing and halting the process. The supervisor, on the other hand, may accept a command which specifies both the reference level and temperature, `SUPV_SET_REF`, and then may distribute them to the heater and level modules. Similarly, commands for starting and shutting down the system are also necessary. To supply the current level to the supervisor from the level module we provide a variable in the level module's status message, `LEVEL_STATUS`, that holds the value for the current level. This message, when written to the status buffer at the end of the controller cycle, can then be viewed by any RCS module or process that has access to the level status buffer. Thus, it is accessible to both the supervisor and the interface computer. Likewise, for the heater module, we provide a field to hold the current temperature to its status message, HTR_STATUS.

The use of RCS here provides us the ability to quickly and easily set up a relatively complex system. In fact, adding additional tanks to the process control system is trivial, since RCS code will already be developed for one tank, it simply would need to be ported over and applied to a second tank setup. Coordinating activities through the supervisor allows this easy upgrade. The only major change to the code would be to update the supervisor to handle the additional tasks of the additional tank. Furthermore, following the RCS architecture described earlier, we could add additional algorithms to handle fault tolerance or prediction and estimation.

Fig. 6 shows one possible layout for the process control application. The mixing tank modules are all located on a single backplane, and each individual module has two local memory buffers associated with it. We can design this structure by using the RCS design tool in very short time. We only have to specify the names of the modules, assign their hierarchy, define the commands accepted by each of the modules, and add the needed variables to the command and status messages. Then, all the related RCS code (excluding the implementation of the actual control algorithms) can be automatically generated by a mouse click.

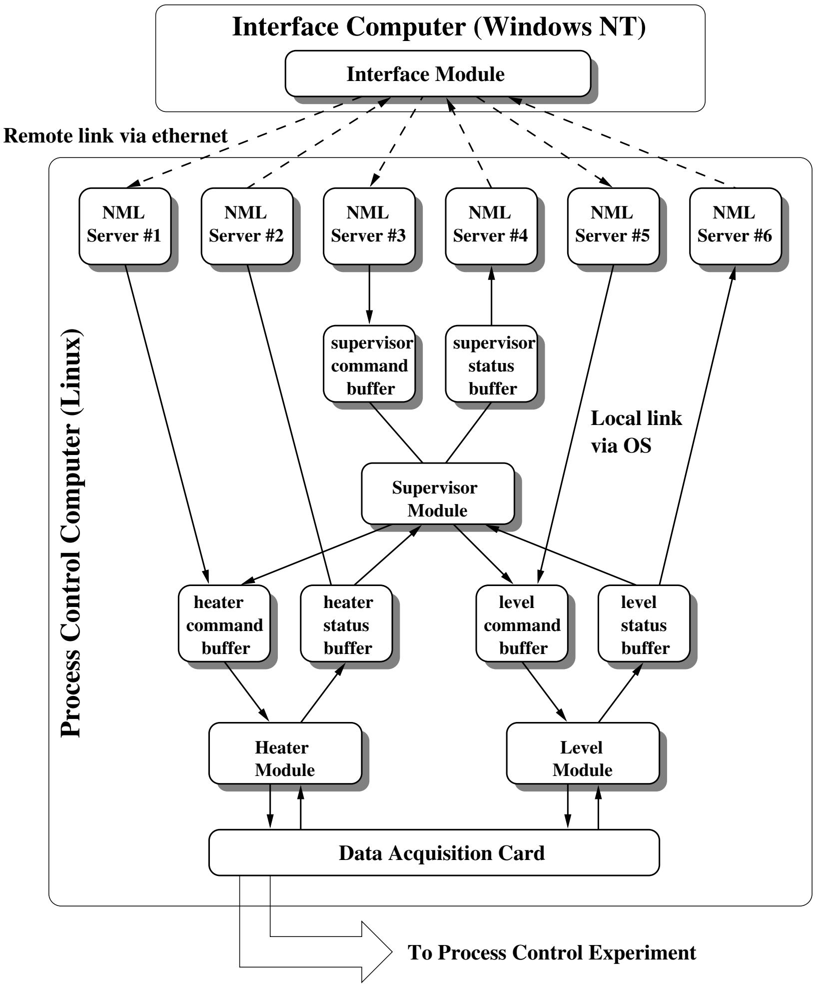

Fig. 6. Design layout for a tank controller.

The human operator can change attributes of the mixing tank system through the interface computer, which runs diagnostics on the system and has access to send commands to each of the modules. Note that the interface computer must access each buffer through an NML server. The mixing tank algorithms are running on a PC (under the Linux OS), which has access to the plant equipment through a data acquisition card. The interface computer is running under the Windows NT platform.

Single Tank Process Control Operation

All the modules of the single tank process control experiment run as separate programs, each containing its own "main" function. The main function creates a cyclic looping structure that activates each module once during a set sampling period. The overall operation of the supervisor module is illustrated in Fig. 7. Because all of the processes are separate, different sampling rates could be used for each module. Note that in the process control case, the temperature module could be operated at a slower rate, since the dynamics of heating and cooling are much slower than those of adding or subtracting fluid from the tank. In this case, for example, we could set the sampling time of the level module to 1 second and that of the heater module to 10 seconds. The supervisor module should run frequently enough to calculate diagnostics of the lower modules (in case of fault detection or modeling), although it will generally run slower than its subordinates.

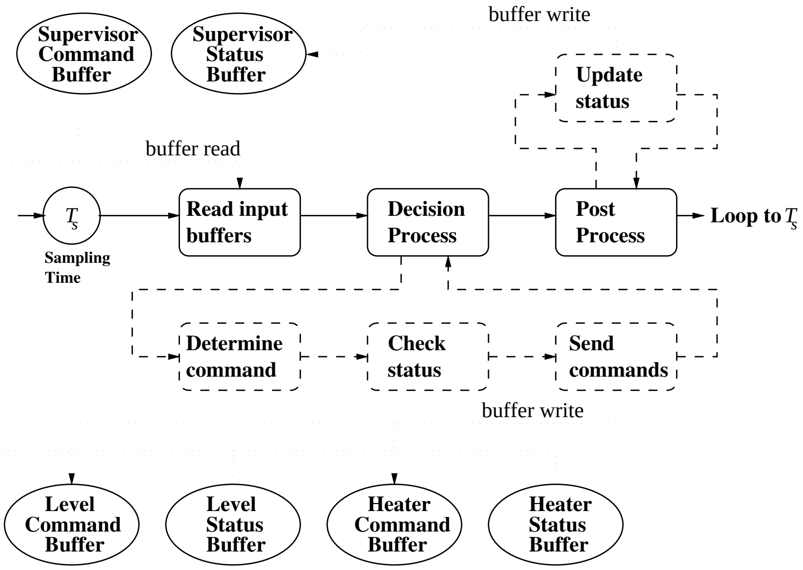

Fig. 7. Overall operation of the supervisor module in the process control experiment.

The task decomposition for the system is primarily accomplished by the supervisor. When the message is written to the supervisor's command buffer, a flag is set, indicating that the buffer contains a new message. On the next cycle through, the supervisor module reads the contents of that buffer. The unique ID of the message lets the supervisor module process know what type of message has been sent and what data variables it expects to read from the buffer. In the case of a `SUPV_SET_REF`, for example, the data variables of the message include the new reference levels for the level and temperature modules. The supervisor's decision process function responds to the message written to the buffer by calling appropriate actions. In this case, the supervisor calls a function `SET_REF()`. This function decomposes the tasks of updating the controllers by sending the appropriate commands to the subordinates that are actually responsible for implementing the control algorithms. It writes the `LEVEL_SET_REF` message to the command buffer of the level module and the `HTR_SET_REF` message to the command buffer of the heater module. Recall that each of these messages contain variables to hold the desired reference values, so these are passed down to the actual control algorithms. During the next cycle of the level (heater) module, the command buffer is read and the `LEVEL_SET_REF` (`HTR_SET_REF`) command is available to the subordinate module. The appropriate action is taken in the subordinate's decision process routine; in this case, the reference level (temperature) is changed to the value contained in the message via a simple feedback control algorithm.

Automated Highway System

As an additional application example, we turn to the idea of the automated highway system, a project that has seen increasing attention due to the large amount of highway traffic congestion in major metropolitan areas [13], [14], [15], [16]. AHS offers numerous benefits to society, including reduction in traffic congestion and increased highway safety. The development of such a system is an extremely complex endeavor due to complex vehicle dynamics and the large number of vehicle interaction possibilities (passing vehicles, multiple lane roads, traffic entering/exiting highways, etc.). Communications plays a key role in an AHS vehicle platoon (a group of vehicles closely spaced together)—the cars must be able to communicate with each other or a platoon leader so that the coordination of activities can take place. It is understood that with only limited or no communications, relatively high performance car-following can be achieved. Here, we consider an AHS with a variety of communications based on the predefined AHS protocol (see [13] for an example of such a protocol). One example of how to implement AHS using RCS can be found in [12] where the authors deal with the complexities of the vehicle itself and suggest that each vehicle has his own RCS structure and the road control mechanism is organized as another RCS structure. Certainly, this is not the only way to implement AHS using RCS and in this example we will consider the three layered hierarchical structure in [13]. We emphasize that our objective here is not the development of a realistic AHS, but simply to show that the RCS library is not limited to a particular architecture and can be used in variety of different applications.

The AHS Problem

In general, in RCS applications we start the design of the controller with the task analysis of the overall system. In this case however, we will assume that the controller hierarchy is prespecified. We will consider the AHS setup shown in Fig. 8, which is essentially the one suggested in [13]. At the top of the hierarchy is the platoon coordinator module, which is responsible for overall long-term planning and coordination of the highway traffic (e.g., for congestion alleviation). This module can command the individual platoon supervisor modules by sending command messages to them and can gain high-level access to their status information. The platoons consist of their own RCS hierarchy, which is controlled by the platoon supervisors. These supervisors take the commands from the platoon coordinator and translate them into actions to be performed by each individual vehicle in the platoon. The platoon supervisor can command its vehicles by writing messages to the command buffer of the appropriate vehicle. The vehicles communicate important status information necessary for long-term planning to the supervisor through the status buffers of the vehicles. Finally, the complexity of AHS is broken down further by representing each intelligent vehicle as an RCS structure, which itself consists of a vehicle supervisor for high-level decision processes local to the vehicle and sensing and controller modules for gathering sensor data and performing speed and lane-change control.

For illustration purposes, we will focus on the description of the intelligent vehicle only, and leave out the details of the platoon structures and the platoon coordinator module. See [13] for an analysis of platoon operations and interactions. In the following sections, we provide more detail about the high-level RCS design of an intelligent vehicle.

RCS Design for an Intelligent Vehicle

Each intelligent vehicle consists of three subsystems—one for higher level decision processes, one for sensor data gathering and local processing, and a third for controller actuation (see Fig. 8). In RCS design, each of these subsystems can be designed as an RCS module. Communications between the modules follow the RCS architecture, with the establishment of two shared memory buffers for each of the three modules. The vehicle supervisor passes commands to the sensor and controller modules through their respective command buffers. The resulting status of the subordinate modules are passed back to the supervisor through the

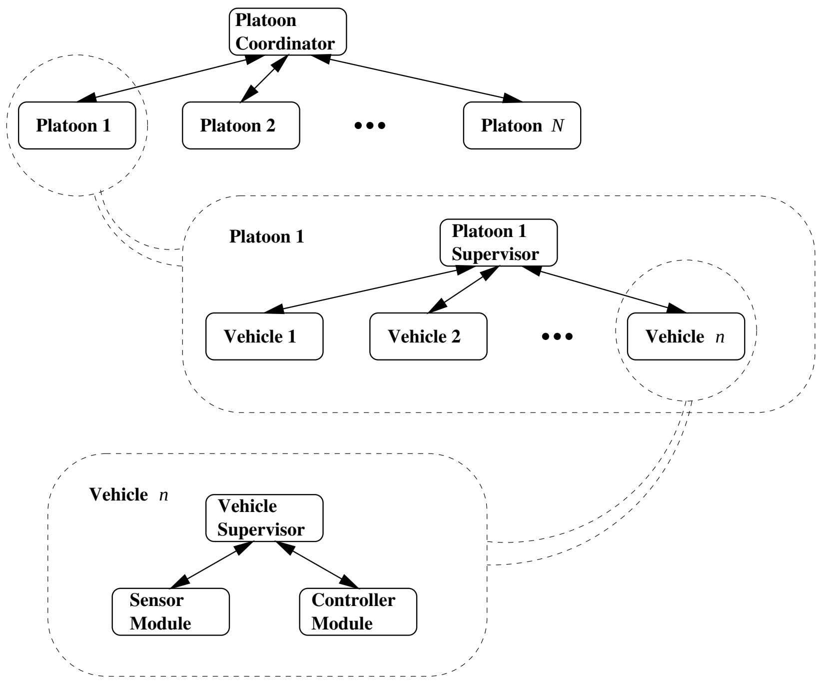

Fig. 8. An RCS implementation of a simple AHS.

respective status buffers. The vehicle supervisor responds to two separate actions—high-level commands from the platoon supervisor module *and* the resulting status of it's subordinates.

Simple task analysis for the vehicle shows that it can perform the following tasks: "Move forward," "Move backward," "Speed up," "Slow down," "Move with constant speed," "Change lane," "Turn to right," "Turn to left," and "Make U turn." (Note that this is a simplified set of actions that a vehicle can perform.) Therefore, these will form the set of command messages or command message vocabulary of the vehicle supervisor. In other words, we will define one command message for each of these tasks (using the design tool). To these messages we add data variables (if needed). For example, the command "Move with constant speed" will require a data field that will hold the required speed.

Based on the command received (or its internal decision mechanism) the vehicle supervisor may decide to perform one of these actions and send appropriate commands to the subordinate control and sensor modules. For example, the task "Change lane" can be broken down to: "Check front traffic," "Check rear traffic," and "Check side lane traffic" (to be performed by the sensor module in this order), and the task "Change lane" (to be performed by the control module, possibly as a simple feedback loop, if it is safe). Based on the information obtained from the first three tasks, the decision mechanism can decide whether it is safe to perform the next operation and send a "Change lane" command to the controller module, or to wait. Note that this may involve not only sensor readings but also some communications with the nearby vehicles [13] (through the sensor module). In a similar fashion one defines the other tasks of the vehicle.

As discussed earlier, each module reports status to its superior. The designer decides which information each module needs to report. For example, we may decide that the vehicle supervisor's status information will consist of the following: "Current position in the highway," "Current longitudinal speed," "Current lane of travel," "Current lateral speed," "Current distance to the front vehicle," and "Current task." Some other designer may add "Distance to the destination," "Total travel time elapsed," etc., depending on the underlying protocols and the overall AHS system. We combine these in a single structure for the status message (which can be generated by the design tool).

For each individual vehicle, the two low-level modules are responsible for determination and implementation of the sensing and control algorithms. The sensor module controls the gathering of data from each of the intelligent vehicle sensors (we assume there are sensors located in the front, on the sides and on the rear of the vehicle and one for determining its own speed) as well as communicating with the near vehicles based on some predefined protocol.

This data needs to be communicated back to the parent module so that coordination can occur. The front sensor of the vehicle determines the existence of a vehicle in front (if within the sensor range), its speed, the distance between the two vehicles, and the lane (left or right) the vehicle is on. The side sensors return the spacing between the carrying vehicle and the vehicles on the side lanes. The rear sensor returns the spacing between the vehicle itself and the vehicles in the rear of the vehicle (including the ones on the adjacent lanes).

By analyzing the sensor data, the higher level decision process of the vehicle supervisor can determine if the needed task is safe. Whether or not the vehicle supervisor actually proceeds with the task (e.g. lane change) depends on the level of safety of the action gained from analysis of sensor module data.

The vehicle controller module performs the low-level control algorithms necessary in implementing the tasks. Therefore, it will accept commands similar to the ones of the supervisor.

By analyzing the tasks of the sensor module we can decide that it will respond to the following commands: "Check front traffic," "Check rear traffic," "Check side lane traffic," and "Return the vehicle speed." Each of these commands may require multiple readings and processing. For example, the command "Check front traffic" may require determining whether there is a vehicle in the front, its speed (if there is one), and its intention to maneuver (e.g. to change lane), which can be obtained by a communication with that car. The other commands can be defined similarly.

We let the sensor module and the control module to communicate through an auxiliary channel so that in case there is a break in communication with the supervisor, the control module can continue its operation, via preprogrammed emergency routines and safely bring the vehicle to the side of the road, or until the driver takes the control.

Proceeding with the analysis, we can define all the possible sequence scenarios (within the AHS protocol) and develop the decision algorithms together with the low-level control algorithms for all the modules of the vehicle and in the whole hierarchy.

This example shows that the RCS library can be used for implementation (or upgrade) of predefined (developed by other means) control hierarchies even if the system is fairly complex.

Conclusion

In this article, we studied the fundamentals behind implementing complex intelligent controller designs using the NIST-RCS software. The tools and components of the RCS library were shown to provide functions such as communication abilities, task decomposition, functional decomposition using subsystems and modules, etc., which promote the implementation of intelligent controller architectures. The primary purpose of the software library is to ease the development of complex control systems. The actual development tools are generic and, as a result, can be used in virtually *any* control problem (both simple single-loop systems and more complex autonomous control architectures).

We began by introducing an intelligent controller architecture and the theory behind RCS controller design. Then we described the RCS design methodology and the RCS design tool. After that, the components of the RCS software library were introduced. The actual library consists of a set of prewritten C++ program classes and segments which contain functions that take care of handling message communications, basic controller functions, etc. By setting up communication channels between controller modules, the RCS hierarchical structure can be implemented. We stressed that although this is the general pattern RCS uses, the developer is free to set up any other auxiliary communication channels to facilitate module interaction.

We gave two design examples—a process control experiment and an automated highway system. Although they were only higher level descriptions (we avoided including low-level coding details to facilitate the understanding of RCS *design*), the examples helped show that the RCS design architecture can be applied to almost any application and that the design methodology is both functional and systematic. Because of its modular and functional design, incorporation of more advanced systems can be accomplished simply by adding more modules in a task decomposition structure. We showed that RCS design provides the power of introducing advanced high-level control methods that can deal with high-level decision making incorporating (possibly) long-term planning, adaptation, fault detection, together with simple feedback loops.

The RCS library is free of charge and can be downloaded from the anonymous ftp site
`ftp://ftp.isd.mel.nist.gov/pub/emc/rcslib/`.
Detailed information on the RCS library can be found on
`http://www.isd.mel.nist.gov/projects/rcs_lib/` or in [21].
Consult [23] for a description of an example educational program on RCS.

Acknowledgments

The authors would like to thank the reviewers and editor for their helpful suggestions. Moreover, we would to thank Profs. Ümit Özgüner and Stephen Yurkovich for their support and extensive laboratory development efforts over many years that have helped establish the infrastructure for the laboratory component of this work.

References

[1] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Transactions on Robotics and Automation*, vol. RA-2, pp. 14–23, March 1986.

[2] K. Valavanis and G. Saridis, *Intelligent Robotic Systems: Theory, Design, and Applications.* Norwell, MA: Kluwer Academic Publishers, 1992.

[3] D. Simon, B. Espiau, E. Castillo, and K. Kapellos, "Computer-aided design of a generic robot controller handling reactivity and real-time control issues," *IEEE Trans. on Control Systems Tech.*, vol. 1, pp. 213–229, Dec. 1993.

[4] A. J. Barbera, J. S. Albus, and M. L. Fitzgerald, "Hierarchical control of robots using microcomputers," in *Proceedings of the 9th International Symposiumon Industrial Robots*, (Washington, DC), March 1979.

[5] J. S. Albus, C. McLean, A. J. Barbera, and M. L. Fitzgerald, "An architecture for real-time sensory-interactive control of robots in a manufacturing environment," in *4th AC/IFIP Symposium on Information Control Problems in Manufacturing Technology*, (Gaithersburg, MD), October 1982.

[6] E. W. Kent and J. S. Albus, "Servoed world models as interfaces between robot control systems and sensory data," *Robotica*, vol. 2, January 1984.

[7] J. S. Albus, H. G. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," Technical Note 1235, National Institute of Standards and Technology, Gaithersburg, MD, April 1989.

[8] M. W. Gertz and D. B. Stewart, "A human-machine interface to support reconfigurable software assembly for virtual laboratories," *IEEE Robotics and Automation Magazine*, vol. 1, Dec. 1994.

[9] H. M. Huang, R. Quintero, and J. S. Albus, *A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real-Time System Control for Coal Mining Operations.* Advances in Control and Dynamic Systems, Academic Press, July 1991.

[10] J. A. Simpson, R. J. Hocken, and J. S. Albus, "The automated manufacturing research facility of the National Bureau of Standards," *Journal of Manufacturing Systems*, vol. 1, no. 1, 1983.

[11] J. S. Albus, E. Barkmeyer, and A. Jones, "Approach to a system architecture for post office automation," in *Proc. USPS 4th Advanced Technology Conference*, (Washington, DC), November 5-7 1991.

[12] J. S. Albus, M. Juberts, and S. Szabo, "RCS: A reference model architecture for intelligent vehicle and highway systems," in *Proceedings of the 25th Silver Jubilee International Symposium on Automotive Technology and Automation*, (Florence, Italy), June 1992.

[13] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya, "Protocol design for an automated highway system," *Discrete Event Systems: Theory and Applications*, vol. 2, no. 3/4, pp. 183–206, 1993.

[14] Ü. Özgüner, C. Hatipoğlu, A. İftar, and K. Redmill, *Hybrid Control Design for a Three Vehicle Scenario Demonstration using Overlapping Decompositions*, ch. IV, pp. 294–328. Hybrid Systems, Springer Verlag, 1997.

[15] Ü. Özgüner, C. Hatipoğlu, and K. Redmill, "Autonomy in a restricted world," in *Proc. of I. IEEE ITS Conf.*, (Boston, USA), p. 283, November 9-12 1997.

[16] K. Redmill and Ü. Özgüner, "The ohio state university automated highway system demonstration vehicle," in *International Congress and Exposition*, (Detroit, MI), February 23-26 1998.

[17] J. S. Albus, "System description and design architecture for multiple undersea vehicles," Technical Note 1251, National Institute of Standards and Technology, Gaithersburg, MD, September 1988.

[18] S. Szabo, H. Scott, K. Murphy, and S. Legowik, "Control system architecture for remotely operated unmanned land vehicle," in *Proc. of 5th IEEE Int. Symp. on Intelligent Control*, (Philadelphia, PA), September 5-7 1991.

[19] P. J. Antsaklis and K. M. P. (eds.), *An Introduction to Intelligent and Autonomous Control*. MA: Kluwer Academic Press, 1993.

[20] J. S. Albus, "Outline for a theory of intelligence," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 21, pp. 473–509, May/June 1991.

[21] V. Gazi, M. L. Moore, K. M. Passino, W. P. Shackleford, F. M. Proctor, and J. S. Albus, "Hierarchical distributed real-time control systems: Software for design and implementation." in preparation; see website http://www.isd.mel.nist.gov/proj/rcs_lib.

[22] A. J. Barbera, J. S. Albus, M. L. Fitzgerald, and L. S. Haynes, "RCS: The NBS real-time control system," in *Proceedings of the Robots 8 Conference and Exposition*, (Detroit, MI), June 4-7 1984.

[23] V. Gazi, M. L. Moore, and K. M. Passino, "Real-time control system software for intelligent system development: Experiments and an educational program," in *IEEE Int. Symp. on Intelligent Control*, (Gaithersburg, MD), pp. 102–107, September 1998.