REAL-TIME CONTROL SYSTEM SOFTWARE
Some Problems and an Approach

L. S. Haynes - A. J. Wavering

Real-Time Control Group
National Bureau of Standards

Abstract:

The National Bureau of Standards Center for Manufacturing Engineering is currently conducting research to help define interface and data format standards necessary to support the factory of the future. In support of this goal, NBS has implemented an experimental factory, called the Automated Manufacturing Research Facility (AMRF).

The evolution of the AMRF has included the development of a Real-Time Control System (RCS) to direct robot motions according to high-level commands received from a supervisory workstation controller and environmental data obtained by local sensors. During the development of RCS, it became clear that large-scale software for real-time control is fundamentally different, and in some aspects, more complex than that intended for scientific or business applications.

Specifically, software designed for control purposes must contend with a large number of non-deterministic states which complicates program synthesis and makes testing and debugging extremely difficult. This paper discusses these problems, and how they are addressed by RCS.

I. Introduction

The National Bureau of Standards Center for Manufacturing Engineering is currently conducting research to help define interface and data format standards necessary to support the factory of the future. In support of this goal, NBS has implemented an experimental factory, called the Automated Manufacturing Research Facility (AMRF).

The AMRF is currently composed of six workstations, six robots, an automated material handling system, buffer storage, a networking system for communication, a database system to support the needs of the facility, and over 100 sensors throughout the facility to provide continuous monitoring of all processes. One of the major challenges of the AMRF is to implement a hierarchical control structure for the facility such that facility-level production orders are decomposed first into shop-level schedules, then to cell-level batch lists and route sheets, down to work station-level instructions to robots and machine tools, and finally into robot and machine tool-level motion commands.

The basic philosophy of task decomposition and modularization pervades the AMRF. Each component of the hierarchy has been designed to integrate fully with the levels above and below as well as with common resources such as the database. A Real-time Control System (RCS) has been developed which embodies this philosophy in a control system for robots. RCS is now being used to provide control of a robot which tends a horizontal machining center. The control system receives commands from the workstation controller, which coordinates robot and machine tool activities. These commands are decomposed by RCS into path control points to be presented to a commercial robot controller. The commercial controller performs the actual joint servoing of the robot.

During development of the AMRF and RCS, the character and some particular problems of real-time control software have been identified. Three main characteristics of control software which delineate it from other applications and cause special problems are:

1) In real-time control the number of possible states increases exponentially with the number of elements to be controlled, resulting in complexity which is difficult to handle.

2) The command and sensory inputs to physical systems such as industrial robots are non-deterministic. Events which change the state of the environment and the goal state of the control object may happen at any time.
3) By definition, a real-time control system must process the available input data and produce a response soon enough for that response to be effective. These characteristics cause several serious problems:

1) It is difficult to understand how to design the system, and the effect that modifications will have on existing code is hard to determine.

2) Because the sequence of external events cannot be predicted, conventional sequential programming techniques are difficult to apply.

3) It is only possible to test a small percentage of the situations which may be encountered. Debugging is also difficult because it is often not possible to recreate the situation that caused a particular problem.

This paper discusses these problems and shows how they have been addressed in the design of RCS.

II. Description of the NBS Real-time Control System (RCS)

RCS employs an organizational and processing architecture directed at controlling the complexity of real-time tasks. References 1 and 2 describe the underlying theory of RCS, and references 3 and 4 describe the details of current applications of RCS.

The architecture has two main components - processing and structural. The processing architecture defines the component modules of the system, how the internal functioning of each module is accomplished and how and when information is exchanged between modules. The structural architecture describes how each of the modules interrelates in order to have the system exhibit the desired real-time sensory-interactive behavior.

II.1 The Processing Architecture

It has been found through experimentation at NBS over a number of years, that certain basic techniques have greatly aided the development of complex control systems. The first is the use of the input-process-output structure to create well-bounded functional modules with specified data interfaces. The second is the use of generic structures wherever possible within the components of the system. The third technique is the use of a common memory-based communication mechanism to move data among all the system components, and the last is to execute all components in the system on a repetitive cycle.

II.1.1 Input-Process-Output

The input-process-output structure (Figure II.1) is the fundamental building block upon which all the processing within the system is based. The basic model is that of a functional module that processes its input data set to generate a resulting set of output data. The function that each module performs should be independent of other processing that might occur within the system, with information required between modules being passed only through the interface data sets. New capabilities may be added to the system - new sensors, new algorithms for directing robot trajectories, new end-effectors - all by providing modules to handle these components, and integrating them with the other components of the system through their data interfaces. The robot controller itself is a component defined in this manner, and may be integrated with a higher level controller, such as a workstation, by meeting the interface data requirements that are defined for it.

A great deal of care has to be given to the design of the overall system based on these type of modules. It is not difficult to generate a system of modules that are so completely interwoven in their function that the modularity is lost and it is impossible to clearly define their interfaces. It is extremely important to define well-bounded functional modules which are as cleanly separated as possible from all other modules in the system, so that the function of the module can be understood by simply looking at its input data and output data and description of its processing. In addition, this technique facilitates module testing and debugging as, by supplying input data sets to a module, it may be tested in isolation, either before being added to the system or afterwards, to diagnose a problem.

II.1.2 Generic Processing Structures

Having structured all the components of the system into the input-process-output format, a problem develops in that the number of functional modules themselves becomes large making it difficult for the user to interact with the system. A primary method of dealing with this problem is to develop generic processing structures. That is, attempt to make the
operation of the modules look the same, so that they appear to be replications of one processing format.

In RCS, the primary generic control structure is called a control level. A control level consists of a preprocessing function, decision processing, and a post processing function (Figure 11.2).

The decision processing is the major decision mechanism used to identify the appropriate procedures that will execute based on a few high-level variables. State tables are used to specify the various test conditions and the corresponding output procedures to be executed if those test conditions are satisfied. Preprocessing evaluates, scales, reduces, and transforms the input data into the more appropriate set of variables used by the decision processing. Postprocessing picks up the extraneous, but necessary, additional processing required. It performs such functions as saving internal variables or reformatting algorithm results for output or transfer to other modules.

Control levels are combined to form hierarchical structures. High level commands are decomposed by each level into simpler commands to be executed by the levels below. This hierarchical structure is described more completely in section II.2.

II.1.3 Common Memory Based Communications

The use of the above generic structures creates a system of separate components that interact only through their input and output data interfaces. To connect any two components, the output data of one must become the input data of the other. This implies that there must be some method to transfer data between these components. A communications mechanism has been defined that uses a common memory area. A copy of the output data is moved to a duplicate buffer area in common memory and from there moved into the input buffer of the other module (Figure II.3).

This type of communication provides the system with a number of advantages. First, the addition or deletion of processes or components in the system is simplified. Data is transferred between components through an independent link, therefore each component may be developed in isolation by supplying appropriate test values to its input data set. Once the modules have been tested, they may be integrated by allowing the communications mechanism to supply their data. This means that even when the system is integrated, each of the processing modules executes as an isolated process bounded by its data interfaces. Also, the buffering of data in common memory is a convenient structure because it eliminates any need for synchronization between sending and receiving modules. Data is moved from the first module to common memory whenever that module is ready to send it, and moves from common memory to the second module whenever that module is ready to receive it, totally asynchronously with the first module's execution. This helps simplify the implementation of these processes on multiple, distributed computers.

Moreover, the existence of the duplicate copy of these system buffers in common memory makes available to an additional process, such as a diagnostic process, the present values of all of the critical variables in the system. If the communications process is executed periodically, then there is an interval when the data in the common memory buffers are not being altered. During this time, a diagnostic process can capture and archive or display this data for user evaluation of the real-time internal behavior of the system.

II.1.4 Repetitive Cycling Execution

The real-time aspect of control is based on the concept of producing a response to changes in input data soon enough for that response to be effective. If a control cycle (in which the input data is sampled and the output response generated) is repeated at a sufficiently fast rate, the system will provide apparent continuous control in real-time.

The communication mechanism described above can be used to define this repetitive process through the use of a periodic synchronization pulse. The communication process can move all the available data to all the processes that are ready to process. After the data transfer is done, those processes will execute acting on the data presented in their input buffers. The communications process itself runs periodically off a real-time clock. This defines the basic control cycle. Every cycle, after communications has transferred the data, each process that is ready samples its data and generates its output response (Figure II.4). All of the system processing occurs each cycle based on the current input data set. This means that the system is not event driven in the sense that an event generates an interrupt which modifies the control flow. All of the relevant variables are sampled each cycle and the information is processed through totally deterministic programmed algorithms. The system
therefore offers a very deterministic, well-defined view of a control algorithm that executes in real-time and further, can be executed offline for diagnostic testing by supplying the data to its interface.

II.2 The Structural Architecture

A structural architecture for a robot controller that is a component part of a larger system architecture for an entire automated factory has been developed at NBS. The robot controller is one module between a workstation controller and a robot joint controller, with access to sensory processing and world model data relevant to the type of task that it will handle (Figure II.5). Several implementation techniques have been developed which, when applied with the methods of the processing architecture, organize complex information processing into a flexible system which will effectively provide sensory-interactive real-time control. These include the use of generic processing levels that perform hierarchical task decomposition, and the specification of interfaces between control levels.

II.2.1 Generic Task Decomposition Levels

The overall architecture of RCS is based on a set of generic levels, that perform the hierarchical task decomposition of the input commands for the robot controller. Figure II.6 shows a robot controller defined as a set of three processing levels. The concurrent operation of these control levels provides a stepwise decomposition of high level tasks into successively simpler and simpler component subtasks. By only requiring each level to decompose the task a little further in the next lower set of subtasks, it is relatively simple to comprehend and manage the control function of each level. Making control levels as generic as possible helps the user, as well as the system designer, to create systems that are reliable and modular, and to develop plug compatible interfaces through the data sets that bound each of these modules. The generic control level is defined as a process that will do a partial task decomposition of its input command into a sequence of not more than seven or eight subcommands. The level is thus structured in the input-process-output format.

The major decision processing that occurs within a level is to take the input command, relevant data from the sensory processing/world model, and status from the level below, and decide on the next output command and output status. Each of the different control levels can be described by the generic type of processing described previously. The functional processing within each level identifies for the user where appropriate information should be contained. This increases the ease with which the user can upgrade the capabilities of the system, such as deciding where to add additional sensors. The generic structure of each of the major components (the levels) provides the consistent framework for the user to know where each module belongs within the system.

For example, the robot controller in Figure II.6 shows a three level task decomposition. The TASK level takes input commands from the workstation and generates subcommands that define major end-point for the robot motions. The ELEMENTAL-MOVE (E-MOVE) level generates trajectory segment goal points that define a complete path to an end point. The PRIMITIVE level generates each of the intermediate path points along the trajectory segments required to move the robot to the goal points.

Consider the addition of a vision system that is capable of providing ranging, feature detection and object recognition. Identification of where these different types of information should be added to the system is made straightforward through the system structure. Ranging information is appropriate at the PRIMITIVE level since it is doing high speed servoing of the robot motion. Full object recognition is most appropriately added at the TASK level since it is concerned with major motions of the system. The actual algorithms for handling this sensor data at each level are easily included in the code because of the well-defined framework of the processing structure.

II.2.2 Control Interfaces

Each generic control level is connected through data interfaces to three other components within the system. There is an interface to a control level above that performs a higher level decomposition. This interface is composed of two buffers, a command buffer from and a status buffer to the higher level. In like manner, there is a similar interface to a control level below that consists of a command buffer down and status buffer back from this lower level. There is an interface horizontally to the sensory processing/world model system, which includes the request buffer to this system and the feedback response buffer.

These three major interfaces surround
each of the generic control levels. Viewed differently, they act as a set of input data that is processed to generate a set of output data. The input data is the command in from the level above, the sensory processing/world model feedback, and the status coming in from the level below. The function of the control level is to take those three major data interfaces and generate the output data which consists of a command out, a sensory processing/world model request out, and a status out to the level above.

These interfaces offer the potential for providing plug compatibility with many different types of sensors or other components with the controller. If the controller consists of functional modules bounded by clearly defined interfaces, and if a proper communication structure is set up such that these are not directly coupled with the elements that are describing the information on the other side of the interfaces, then these blocks can become different components that can be plugged together in different ways without resulting in a bundled system.

III. Problems Addressed by the RCS Architecture

The RCS architecture has been designed to help solve problems caused by the complexity and non-sequential nature of real-time control software, including testability.

III.1 Complexity

The basic problem of control systems is not the number of lines of code involved in the programs, but the number of states the system has to test for and react to. Millions of lines of code can be written solving complex sets of mathematical equations. Though many bugs may be introduced, they may eventually be discovered and corrected by proper testing using data sets. These programs are characterized by many lines of mathematical operations but not a large number of test and branch conditions.

Large real-time systems seem to be the opposite. There is a very limited number of mathematical operations that are applicable (the control problem is far too complex for us to represent it in mathematical formulations.)

In the case of real-time control, most of the processing is in the form of tests on various states of input variables (from both sensory systems and internal status operations) and branching to appropriate calculations which include further test and branch conditions. The number of possible states gets very large, and changes of state are frequent. Each different value of every variable tested could result in a different path through the code. As the number of variables and the number of values they can take on increases, the number of possible control paths through the code grows at a much greater rate such that it becomes impossible to test and evaluate the system's response to all states.

The primary goal of RCS has been to provide a superstructure which helps a designer cope with the complexity of real-time systems. The generic processing structures and task decomposition levels which have been developed partition the problem of understanding the system as a whole into understanding the operation of a number of small modules, each with a well-bounded function and clearly defined data interfaces.

This enables the user to better understand how to build the system to make it do what he wants it to do and to understand how it is working. Moreover, he will be able to modify or add pieces to the system in a structured manner that will ensure that those changes will still maintain the correct operation of the system.

III.2 Non-Sequentiality

Control software must deal with non-deterministic inputs. User inputs, changes in sensor readings, and communications from other systems can happen at any time and often require action within absolute time constraints. With conventional programming techniques, a particular event might have to generate an interrupt in order for the system to respond. Since many sets of conditions may require the use of interrupts, this technique of exception-handling becomes a major cause of unpredictability.

RCS does not use interrupts to modify control paths. If there are conditions which require specific handling, they are listed as explicit conditions in a state table. Each state table lists the input conditions to be considered by that module during every control cycle. At the beginning of each cycle, the input conditions are examined, and one and only one line of the state table must match the current status. For each line of the state table, there is a set of procedures which are executed when that line matches. The state tables provide
control flow; therefore the system is not required to respond at random times.

III.3 Testability

The large number and unpredictability of inputs to control software make testing such systems especially difficult. Only a small portion of the possible sequences of events may be tested, and many unforeseen situations may arise. The testing problem for software used to control a physical system is particularly critical, since errors have the potential to cause a great deal of damage.

The modularity of RCS allows components of the system to be tested and debugged individually. Individual testing, however, does not ensure that the components will behave as expected when assembled into a large system. A feature of RCS that helps debug the operation of the system as a whole is that during most of the control cycle, the data in common memory is stable and may be used to generate diagnostics during execution.

All of the state variables stored in the common memory can be logged out onto a log file every cycle. This provides the capability to look back at previous system states to analyze what has occurred. In addition, the state variables can be changed interactively, and the system may then be run forward with the altered variables. Also, a real-time graphics display system has been developed to help pinpoint the source of execution errors.

RCS has been implemented as a highly interactive system with support utilities that allow the user access to the code at many levels of detail. Any or all levels can be single-stepped, with any or all of the other control levels running at normal speed. Furthermore, only the lowest level of the control system is actually connected to real actuators, valves, etc. None of the other levels of control are affected whether the lowest levels are real hardware, software stubs, or component simulators. RCS can be switched from moving a real robot to moving the graphical image of a robot by simply redirecting output data generated by the lowest level to a different device.

Another key point is that if an executing control level reaches a point where a set of conditions occur which have not been programmed into the system, the control system enters an error state or a recovery state. The error state can be coded to try to handle the immediate problem, and it is straightforward to later add the appropriate condition to the state table along with the procedures to handle the new condition. When conventional implementations of real-time control systems reach a point where a condition has occurred which the programmer had not considered, the algorithm simply does the wrong thing. Often, the error is not even detected during testing, but may have disastrous ramifications in specific cases. RCS is a system which embodies a mechanism for gracefully adding the code to handle situations not initially anticipated. In the prior art of real-time control system design, major re-writes may have been required to handle new conditions.

IV. Conclusion

In reference 5, Dr. David Parnas discussed the conventional sequential nature of programming. Dr. Parnas writes:

The easiest way to describe the programming method used in most projects today was given to me by a teacher who was explaining how he teaches programming. 'Think like a computer,' he said...

One can follow the 'think like a computer' method with programs that have neither branches nor loops. As soon as our thinking reaches a point where the action of the computer must depend on conditions that are not known until the program is running, we must deviate from the method by labeling one or more of the actions and remembering how we would get there. As soon as we introduce loops into the program, there are many ways of getting to some of the points and we must remember all of those ways...

As we continue in our attempt to 'think like a computer,' the amount we have to remember grows and grows. The simple rules defining how we got to certain points in a program become more complex as we branch there from other points...

When there is more than one computer in a system, the software not only appears to be doing more than one thing at a time, it really is doing many things at once. There is no sequential program that one can study. Any attempt to 'think like the computer system' is obviously hopeless...

For the reasons Parnas describes, RCS focuses on conditions, and actions to be
performed in the event of those conditions. State tables contain no sequence information. The application programmer must think in terms of what must be done in each case. Since the control flow is separate from the computation, it is easier to think in terms of events. The relevant events are explicitly listed as part of the state tables. Since there are no interrupts, there is no way that given a particular event, the specific processing specified in the state table will not execute to completion. Finally, events are easy to add to a state table, and if an unexpected situation occurs, the situation is noted and the system will either stop, or execute default routines. On the negative side, it is difficult to use RCS to implement sequential operations, and it is difficult to determine the normal sequence of operations.

V. Future Plans

We intend to expand the capability of RCS in several directions.

One of the additions which will help simplify state tables is to augment each line of the state tables with an expected next state entry, and a pointer to a new secondary state table to use in the event that the indicated state is not the next state. The secondary state tables would deal with exceptional conditions. The benefit of this additio is that it would reduce the size of state tables, and would provide the ability to more easily deduce the normal sequence of operations, given a certain current state of the system.

Currently, procedures in RCS must be written in a language called SMACRO, designed and implemented as an integral part of RCS. We believe we can expand RCS to permit procedures to be written in any of several common languages such as C, Ada, Pascal, etc.

RCS provides tools and a superstructure which help a user encode actions which must be taken to ultimately complete a task. However, RCS requires the user to consider every case, and enter the relevant task knowledge manually. We would like to experiment with, and eventually develop approaches whereby the system could use a task knowledge data base to perform geometric reasoning, and task planning. There are languages such as Prolog (ref 6) which are promising for performing "reasoning" about a situation, and deriving a solution about a problem without the necessity for explicitly entering the solution for each case. While Prolog rules could be encoded in a state table format, Prolog rules and the mechanism for their interpretation are a significant generalization of the RCS concept of matching one state table line each cycle. We plan to extend RCS into the area of planning.

Languages such as Smalltalk, (refs 7 and 8), provide an object oriented structure which permits objects and operations on those objects to be hierarchically defined. Objects inherit properties from their parents, if their properties are not explicitly defined. This capability provides a natural structure for defining objects and their relations. RCS's underlying operating system can provide the necessary capability, and we would like to incorporate the concepts of SMALLTALK into RCS.

There has been recent research into how to implement control systems which accumulate data as they operate, and hence "learn" the expected response of the system (ref 9). Robot line following errors can be significantly reduced by accumulating data on previous traversals of the line (ref 10). Instead of reducing line following errors, robot speed could be increased based on previously experienced path errors. As an example related to higher levels of control, a robot could use a general purpose path planner to determine paths, but keep a database of these paths to help expedite path planning for future similar paths. We intend to generalize the structure of RCS to permit control decisions to be made based on accumulated error.

The National Bureau of Standards Automated Manufacturing Research Facility is partially supported by funding from the Navy Manufacturing Technology Program.

VI. References


The Input-Process-Output Structure is the Fundamental Building Block of the Architecture.

II.1

A Standard Processing Format of Preprocess, Decision Process and Postprocess is Used for Each Functional Module

II.2
Communication of data between modules through a Common Memory eliminates process to process interactions and provides a snapshot of system status in the duplicate buffers in common memory for diagnostic evaluation.

II.3

Real-time behavior requires the continuous repetition of a control cycle that samples inputs and generates outputs.

II.4
Robot Controller is integrated with other components in an Automated System through well-defined data interfaces.

II.5

Robot Controller is composed of a set of generic levels that do hierarchical task decomposition. Task specific data is kept separate from the programs and is accessed in real-time from a knowledge base maintained in Common Memory.

II.6
A Generic Control Level with interfaces to higher and lower level controllers and a World Model/Sensory Processing System is a generic processing structure.

II.7