To be presented at the 8th IEEE International Symposium on Intelligent Control, Chicago, IL, August 25-27, 1993

A Task Oriented Approach for Developing Complex Large-Scale Intelligent Control Systems Using Software Templates

Richard Quintero
Robot Systems Division
National Institute of Standards and Technology
and
A. J. Barbera
Advanced Technology and Research Corporation

ABSTRACT

This paper presents a task oriented systems engineering approach for developing complex, large-scale, integrated, intelligent machine systems using software templates. This methodology can be applied to a wide variety of real-time machine applications. The method presented here is built upon the *Real-Time Control System (RCS)* Reference Model Architecture being defined by researchers in the Robot Systems Division at the National Institute of Standards and Technology (NIST). We believe that the reference architecture and methodology described here could form the basis for defining an open-systems architecture for intelligent control systems applications.

INTRODUCTION

This paper is a condensed version of [Qu 92]. The methodology presented here is based upon a task oriented systems engineering approach originally conceived by Barbera [Ba 84]. The methodology also complies with the *Real-Time Control System (RCS)* Reference Model Architecture published by Albus [Al 89a, Al 89b]. Researchers at the National Institute of Standards and Technology (NIST) are exploring several RCS implementation approaches in addition to this one. Each of these approaches are generally optimized for a particular class of RCS applications.

Real-time intelligent machine control systems applications cover a very broad spectrum. For example: 1) Controls engineers often deal with robotic or machine tool applications with requirements for high-speed servo control of machines with multiple joints and/or several axis of motion. These problems become even more interesting and demanding when real-time closed-loop compensation is introduced to achieve very high accuracies or compliant motion. 2) Systems engineers are interested in coordinating the control of several machines or in the case of large vehicles (e.g., ships, submarines, aircraft), several major subsystems in order to accomplish a set of desired system goals. Such systems usually require some degree of human interaction. Closed-loop control is introduced in these applications in order to deal with uncertain and noisy input data, and in order to be able to function in unstructured environments. 3) At a higher level of abstraction, systems engineers become concerned with the enterprise model in manufacturing, traffic management in highway or other traffic control problems, or battle coordination in military applications. Communications networks, human interface, and knowledge management receive increased attention in these applications. Sensory-feedback is used in these applications for the same reasons discussed above but at a higher level of abstraction.

The method described here is particularly well suited to applications which are rule driven (i.e., they employ strategies, tactics, and process knowledge). Such applications (types 2 and 3 above) are characterized by a need to monitor sensory input to detect events and objects in the intelligent system's environment. These systems use sensed events to trigger desired system activities and to react to exception conditions. The methodology also accommodates control systems problems (type 1 above) dealing with path planning, trajectory generation, and control law algorithms.

PRODUCT ENDORSEMENT DISCLAIMER

References to specific brands, equipment, or trade names in this document are made to facilitate understanding and do not imply endorsement by the National Institute of Standards and Technology.

BACKGROUND

Early work by Albus [Al 81] and Barbera [Ba 84], in the Automated Manufacturing research Facility (AMRF) [Si 83], gave rise to the first definition of a Real-Time Control System (RCS) systems engineering approach focusing primarily on software design. This approach was derived from a control systems engineering perspective rather than a data processing perspective. The Robot Systems Division has refined and evolved these techniques by applying the RCS approach to a number of robotic problems in manufacturing as well as robotic applications intended for unstructured environments including the Army Field Material-Handling Robot (FMR) project [Mc 86], the Defense Advanced Research Projects Agency (DARPA) Multiple Autonomous Undersea Vehicles (MAUV) project [Al 88], the Army Tech-based Enhancement for Autonomous Machines (TEAM) project [Sz 88], the U.S. Bureau

of Mines Coal Mining Automation Project [Al 89b], [Hu 92], and [Hu 91] as well as others.

One of the more well known NIST architecture definition efforts was the development of the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Al 89a]. NASREM was adopted by NASA to provide a software control system architecture guideline for use by development contractors charged with building the Flight Telerobot Servicer (FTS) control system as part of the Freedom Space Station project.

A number of versions of the NIST control system have been implemented over the last decade. RCS versions have been implemented using the FORTH, C, and ADA languages and running on 680x0 series processors as well as on 286/386 machines and on Multibus and VME backplanes. Applications have been built using real-time operating systems such as: GRAMPS, pSOS, and VxWorks. RCS applications have also been hosted within the DOS operating system on personal computers (PCs).

The approach presented in this paper is based on our most recent work in applying RCS techniques to the automation of submarine maneuvering control, under DARPA sponsorship, and in demonstrating the automation of a continuous coal mining machine, under the sponsorship of the U. S. Bureau of Mines. This work is being carried out by a team of researchers from NIST and the Advanced Technology and Research (ATR) Corporation.

WHAT IS A CONTROL SYSTEM ARCHITECTURE?

The Random House College Dictionary, [Ra 82], defines ***architecture*** as "the character or style of building; the structure of anything". The RCS Architecture is a style of building real-time intelligent control systems. These systems generally include software, hardware, machines, people, communications, information repositories, information/knowledge models and real-time execution models as shown in Figure 1. The RCS Architecture defines a highly structured, modular organization of these control system components which can serve as a standard reference model for an open-system architecture.

The RCS methodology presented here follows the Barbera approach to building control systems. The Barbera approach is characterized by defining a small number of building blocks with which one can build large complex intelligent control systems. The basic building block is a ***controller module***. A controller module does not contain any submodules but it may encapsulate any number of functions, subfunctions, or processes. A controller module can be viewed as a systems integration "wrapper" which is implemented as a template. We encapsulate software within this wrapper to ensure compliance with our integration rules. Every module built using the controller module template inherits a software execution model, a communications mechanism, performance measurement capabilities, and debug mechanisms. All of these properties are discussed in detail in [Qu 92].

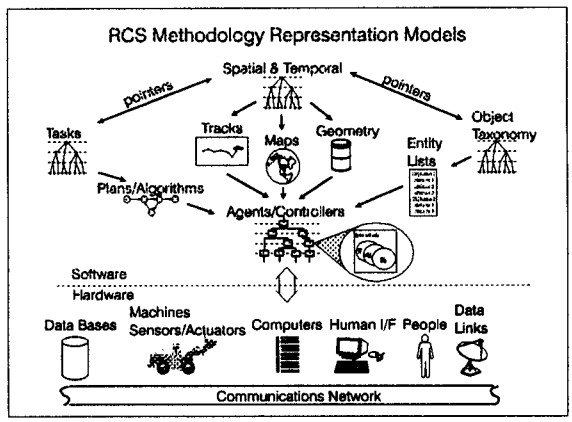

Figure 1.

THE RCS PROBLEM DOMAIN

RCS specifically addresses intelligent machine control systems problems. We define ***intelligent machines*** to be machines designed to perform useful physical work while employing in situ knowledge (sensory input data), and a priori knowledge, tactics and strategy. Intelligent machines use feedback from the physical environment to manifest "intelligent behavior" in real-time via computerized real-time control of the machine's electro-mechanical actuators and sensors. In addition, we believe that practical intelligent machines almost always require some level of human interaction. The definition given above is intended to include: ***automation systems, embedded systems,*** and ***robotic systems*** ranging from factory floor robots to space vehicles and planetary exploration robots. In priority order, our objectives in developing RCS are to:

1) Improve human understanding of the design.

2) Manage software complexity.

3) Provide for robust, verifiable, efficient, coordinated, real-time performance.

4) Provide for extensibility, portability, and software reuse.

Two Robot Systems Division papers [Al 91a] and [Qu 92] elaborate on these objectives.

Developing an RCS Methodology involves: 1) establishing a comprehensive set of integration rules, 2) identifying information models and real-time software execution models which explicitly highlight critical components of the RCS problem domain, and 3) selecting software engineering implementation techniques which are compatible with the integration rules, the models and the RCS Methodology objectives presented above. An architecture implementation which results from applying such a systems engineering methodology includes all of the people, machines, sensors, actuators, and computing hardware as well as a communications network and the software needed to manifest appropriate systems behavior. The critical design components to be explicitly addressed include:

- real-time task behavior and software execution models
- interfaces and communications methods between software modules, hardware resources and human operators
- information models and knowledge base management
- allocation of resources (assignment of software modules to computing hardware)
- rules for decomposing the design spatially and temporally

THE RCS ARCHITECTURE REFERENCE MODEL

The RCS Methodology described in this paper complements and is based upon the theoretical RCS Reference Architecture principles developed by Albus, Barbera and other NIST researchers, since the mid 1970s. A comprehensive treatment of the RCS Architecture is contained in the following publications: [Al 91b], [Al 90a], [Al 90b], [Al 89a], [Al 81], [Ba 84]. The RCS Reference Architecture is described briefly below.

Albus models the most primitive form of an intelligent machine as a closed-loop control system (see Figure 2.). A closed-loop system is formed in the machine by inputting sensory data to ***Sensory Processing (SP)***, passing the processed information off to the ***World Modeling (WM)*** function, which maintains the machine's best estimate of the state of its world, and finally closing the loop through ***Behavior Generation (BG)*** which plans and executes actions to be performed through the machine's actuators. An intelligent machine must also utilize a value system in order to judge the "goodness" of the results of its actions within the context of the tasks it is expected to perform. The value system, or the ***Value Judgment (VJ)*** function, is used in goal selection to direct Behavior Generation in selecting alternative plans and actions.

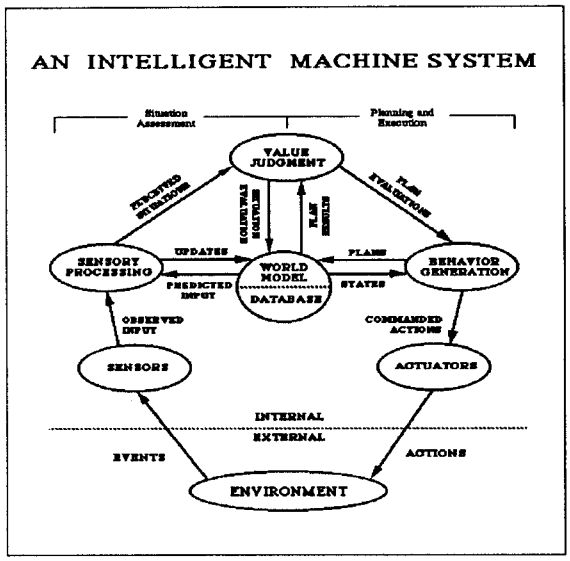

Figure 2.

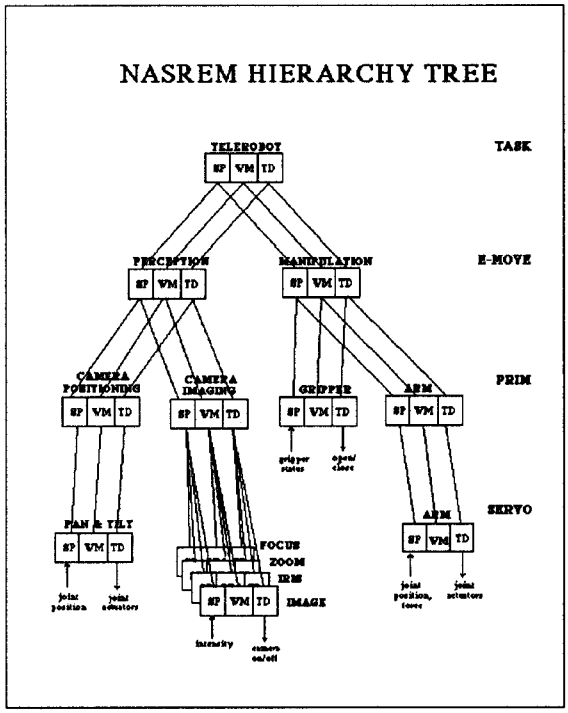

Figure 3.

The RCS Reference Architecture, defined by Albus, extends the notion of an intelligent machine design containing the basic SP, WM and BG functions by creating a hierarchy (see Figure 3.). The basic SP, WM, and BG functions are grouped as ***controller nodes*** and distributed in a hierarchically organized, integrated set of nodes. In an RCS

implementation a node is a collection of one or more software modules. Each controller node is assigned a set of tasks at an appropriate level of abstraction and each has a limited range of authority and responsibility within the chain-of-command formed by the RCS hierarchy (much like a human military command structure would be organized).

The terms, ***World Model or World View***, are used to describe the intelligent machine's collective capability to perceive the world in which it functions (both external and internal). When we use these terms we are referring to algorithms for understanding the world, WM server functions, and the information stored in the Global Memory.

Global Memory (GM) is the complete collection of globally defined variables in an RCS application. It may be thought of as the repository or knowledge base where controller nodes store information to be shared with other controller nodes. In many applications Global Memory is implemented as a distributed database. GM can also be viewed as a combination of the communications mechanisms and the repository necessary for implementing an RCS application.

The ***Behavior Generation (BG)*** function is further decomposed into ***Job Assignment (JA), Planner (PL), and Executor (EX)*** functions. ***Job Assignment*** involves commanding subordinates to carry out concurrent tasks. Stored or generated plans temporally decompose tasks into sequences of sub-goals to be accomplished, to the limit of the appropriate planning horizon for a level. ***Planners*** are responsible for selecting pre-stored plans and/or generating new plans to be instantiated by the Executors. ***Executors*** instantiate the next step in the current plan based on the current state of the world as viewed via the World Model. Executors pass instantiated task commands to the next lower level JA, where this pattern is repeated, down the hierarchy, in a pipelined refinement of task detail. In general, subordinate levels deal with less abstract task details, at faster sub-goal completion rates.

Temporal decomposition in an RCS design deals with planning horizons, memory spans and goal completion rates which are subdivided by roughly an order of magnitude in time between levels. Planning horizons and memory spans increase as we move to higher levels of the architecture while sub-goal accomplishment rates increase as we traverse the hierarchy towards the lower levels.

In RCS, there is a notion of decomposing the control system design into layers or levels of abstraction. The RCS Reference Model Architecture describes the types of tasks carried out at each level of abstraction, starting at the bottom of the hierarchy. See [Al 89a] and [Qu 92]. In some implementations these levels have been labeled as follows: **Level 1 - Servo, Level 2 - Primitive or Prim, Level 3 - Elementary Move or E-Move, Level 4 - Individual Machine or Task, Level 5 - Group(1) or Workstation, Level 6 - Group(2) or Cell, Level 7 - Group(3) or Factory, Level 8 and Higher Levels**. There is no upper limit on the number of levels in an RCS hierarchy. The number of levels of coordination and abstraction are strictly a function of the demands of the application (i.e., the organization of people, machines, communications links and tasks to be coordinated).

RCS Method Tenets

We use the word tenet, here, to mean guidelines and engineering rules of thumb which characterize this RCS Methodology approach. Together the RCS Architecture definition and these tenets form a basic set of rules or systems integration standards for building real-time control systems. Tenets 1) through 5) are generally applicable to all RCS Methodology approaches while tenets 6) through 10) are expressed in terms of the Barbera approach. An in-depth discussion of these tenets is presented in [Qu 92].

1) Use task oriented decomposition (driven by scenarios)
2) Use hierarchical organization and assign responsibility and authority
3) Organize the control hierarchy around tasks top-down and equipment bottom-up
4) Partition by an order of magnitude between levels (spatial and temporal resolution) and roughly ten decisions or less per plan
5) Use seven + or - two subordinates per supervisor and only one supervisor at a time
6) SP/WM/BG functions are distributed throughout RCS and assumed to exist in each node
7) Allow human I/F at any node
8) Controller modules are finite state machines communicating through Global Memory
    * Use a controller template as the basic building block
    * Use cyclic sampling rather than interrupts for context switching
    * Surround everything with data buffers
    * Use non-blocking input/output (I/O)
    * Implement Global Memory using a One Writer, Many Readers Paradigm

* Match the control cycle time to the demands of the control application

9) Design for concurrent processing
   * Measure execution time performance
   * Allocate sufficient computing resources
10) Use synchronous control at the lowest levels, transitioning to asynchronous control at the highest levels

RCS PLANS

An RCS control system can be viewed as an integrated collection of finite state machines which are capable of selecting or generating and executing RCS plans in real-time. This requires a method of representing task knowledge in plans and a set of integration and decomposition standards for distributing those plans within the RCS hierarchy. The RCS Methodology described here uses both rule plans and path plans.

RCS Rule Plans are uninstantiated plans (or plan schemas) which can be represented using some form of *If-Then-Else* construct. Rule plans specify branching conditions and they can be represented by state transition diagrams. ***RCS Path Plans*** are ordered sets of instantiated poses, knot points, commands, or other variables specifying a sequential order of execution. Path plans do not specify branching conditions. A rule plan is required to specify conditions to be monitored in order to interrupt the execution of a path plan for branching (i.e., out-of-tolerance condition branching). A path plan can be generated by fully instantiating a rule plan (assuming or given a time sequenced set of input conditions) and storing the result. In addition to rule and path plans, RCS also accommodates goal-point generators or control-law algorithms. Such algorithms (typically mathematical) are used to generate the next goal-point for an actuator movement at the Servo-Level or the next goal-command for a subordinate module at other levels of an RCS hierarchy.

We use state graphs and state tables to represent RCS rule plans as illustrated in Figure 4. These plans embody strategies, tactics, and process knowledge. A rule plan (also called a plan schema) is a set of uninstantiated rules for accomplishing some task. Rule plans are used throughout the RCS Architecture. Rule plans are often very simple at the low levels and more complex at the higher levels.

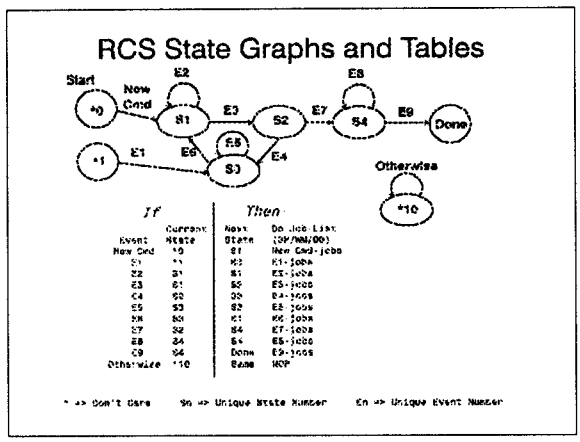

Figure 4.

Edges or arcs in a state graph indicate both the prerequisites for transition from state to state upon the occurrence of the next state clock cycle and the activities to be performed if those prerequisites occur. Nodes in a state graph define and name the allowable states a finite state machine may enter. Normally each edge in a plan will initiate a list of jobs to be accomplished as a result of satisfying the specified transition conditions. Upon initiating these jobs, which are typically task commands to subordinate controllers and Sensory Processing and/or World Modeling interim calculations, the machine steps to the next state. The edges emanating from the next state are subsequently evaluated on the next control cycle (state clock cycle).

By mapping an RCS state graph to a state table our representation becomes one step removed from actual software code. An RCS state table is in the form of a list of *If-Then* rules. The *If-Then* rule table can be implemented using a number of software language constructs such as if-then-else statements or case statements. The table is organized in precedence order for evaluating event conditions (edges). Each event is logically *anded* with the current state in precedence order from the top of the list. A match (logical true condition) causes the plan to transition to the next state and initiate (run) the job list associated with the triggering event. In our approach only the first match found is acted upon during a control cycle. This process is repeated on each control cycle until the plan reaches the "Done" state. After reaching "Done" a no-operation (NOP) is executed thereafter.

Implementing a Controller Template, the Basic RCS Building Block

Building understandable large systems requires defining systems integration standards. If we can define a small set of primitive generic building blocks which can be replicated and integrated using a concise set of integration standards, we are likely to have more success in building even more complex systems structures. The ideal situation is to be able to build a complex structure using a single standard type of building block. The RCS implementation approach described here requires only two basic building blocks: An RCS Controller Template and a Main Program Template.

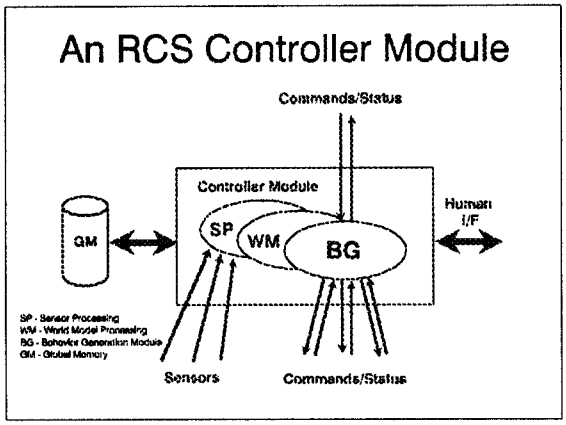

Figure 5.

A generic RCS controller module template complies with the RCS Reference Model Architecture, presented earlier, in that it provides a software execution structure within which the basic functions of Sensory Processing, World Modeling, and Behavior Generation may be implemented. In addition this model begins to address the interface definitions required to integrate a set of controllers to form an RCS hierarchy. The controller must be able to accept task commands from its superior and send commands to its subordinates or to actuators if the controller is at the lowest level of its branch in the RCS hierarchy (see Figure 5.). It must be capable of accepting status from its subordinates and sending status to its superior. A controller module must be capable of accommodating a human interface and it must have the capacity for communicating with other controllers and the knowledge base through some set of Global Memory communications mechanisms. A controller must also be capable of directly accepting sensor data for processing.

Since a controller module is a finite state machine, its response to stimulus is deterministic for any given execution cycle. Its output is only a function of its current state and its input event space. Furthermore, its execution time can be measured or calculated for every event-state pair in a given plan. Any algorithm implemented within a controller module must be designed to execute in a cyclic manner, with a definite execution time for each execution cycle. Such an algorithm could, however, be allocated any number of execution cycles in order to produce a solution or a set of more and more optimal solutions on each subsequent execution cycle. An example might be an algorithm that requires a number of sample points in order to converge on a solution within a reasonable margin of error. Such algorithms often produce increasingly accurate results with each new data sample processed.

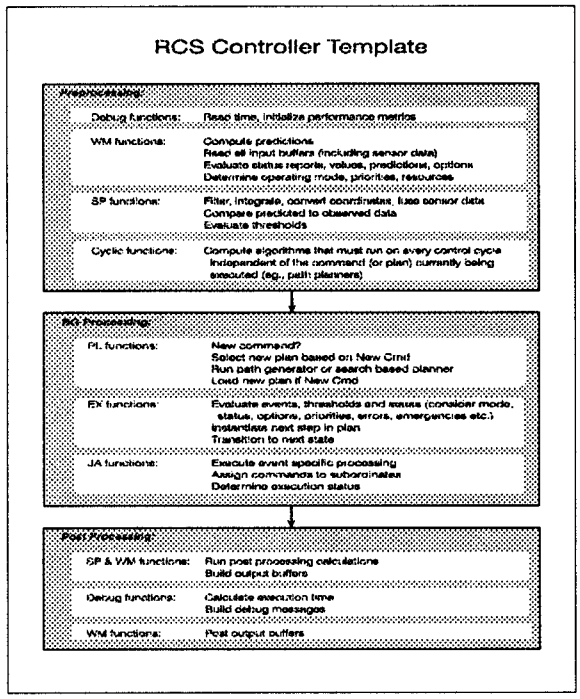

Figure 6.

Every controller module must be capable of communicating with other modules in a manner which complies with the RCS Methodology tenets. One way to ensure compliance is to replicate a standard RCS Controller Template incorporating all of the necessary "hooks" for integration. This overhead structure is included in every module to make it easy for humans to integrate modules and to understand and maintain the design. This is done

even though some of the overhead may not be needed in every controller module instance. A controller module built from an RCS Controller Template performs Preprocessing, then Behavior Generation (also referred to as Decision Processing), followed by Post-Processing on each control cycle (see Figure 6.).

Preprocessing includes four types of subfunctions: Debug or overhead functions, World Model functions, Sensory Processing functions, and other cyclic functions. These subfunctions are cyclical, meaning they are performed on every control cycle no matter which plan the controller module happens to be executing.

The Behavior Generation (BG) function consists of three sub-functions: planning, execution, and job assignment. When the BG function is implemented within a controller module it contains one Job Assignment (JA) function, one Executor (EX) function and one Planner (PL) function (see Figure7.).

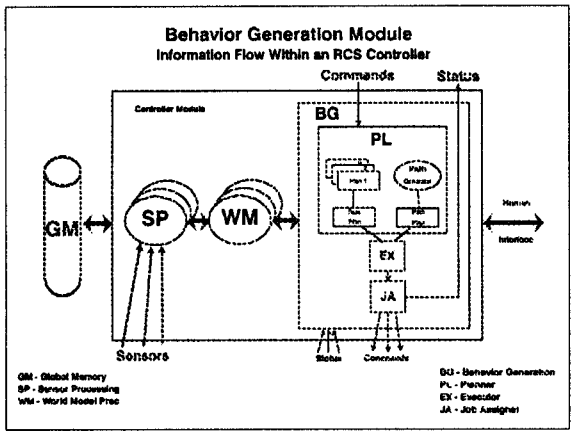

Figure 7.

All controller modules build output buffers and post them to Global Memory during Post-Processing. They also update local variables/pointers in preparation for the next control cycle (e.g., current plan pointer, current state pointer, etc.). All controllers compute any required performance measures like execution time during Post-Processing. In addition, any cyclic SP or WM functions which need to be computed after Behavior Generation runs are performed during Post-Processing.

MULTI-TASKING ON A SHARED CPU, USING A MAIN PROGRAM TEMPLATE

Multitasking within a shared CPU is implemented in RCS with a CPU Main Program Template (Figure 8.). The Main Program allows initialization of CPU parameters, such as declaring global variables (externals), and the loading of starting values in Global Memory.

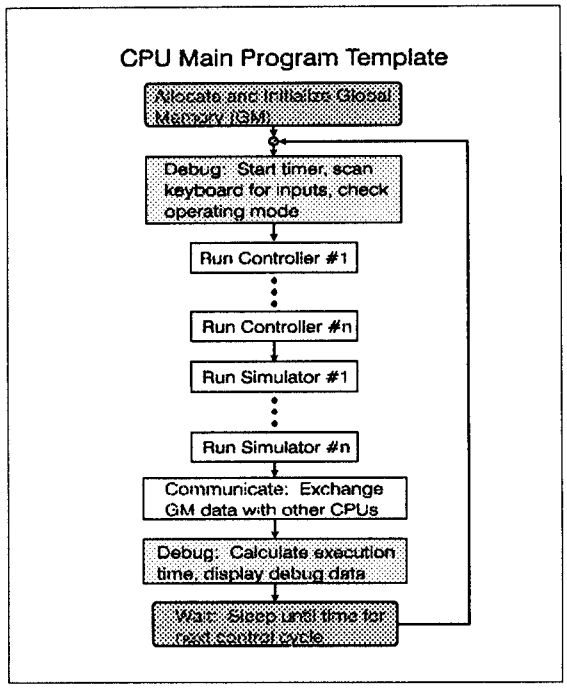

Figure 8.

After initialization the Main Program begins running the heartbeat control cycle for the CPU. First, a Debug function runs to check for operator inputs indicating a change in operating mode (e.g., debug single step, normal run mode, etc.) and to start a timer to measure control cycle execution time. Once that completes each RCS controller module (including simulation modules) runs in sequence according to the precedence order of the execution schedule established by the programmer. Of course, in compliance with the RCS tenets, all of the controllers must be able to complete their execution within the established heartbeat control cycle time. If the controllers overrun the cycle time then the RCS designer must reassign one or more to different CPUs or increase the cycle time.

At the end of each control cycle the communications controller modules are executed to exchange Global Memory data with other CPUs within the backplane and over any networks being used. The total execution time is calculated as a last step to be sure that the heartbeat control cycle time has not been exceeded (a debug error is posted if it ever does). At that point the Main Program enters a wait loop (or goes to sleep) until time for the next control cycle.

RCS Target Hardware and Operating Systems

Any multiprocessor backplane hardware suite may be used (e.g., VME, Multibus, Nubus, etc.) for implementing an RCS control system. Many standard commercial boards including communications interfaces are available today for such systems at very reasonable prices. Software operating systems and device drivers are also widely available for these systems. Using this type of hardware suite in an RCS implementation makes it very easy to tailor the hardware selection and the communications network to meet the real-time requirements of the application. It is also very easy to extend such systems as the system evolves. In addition, if the RCS software is written using standard software language compilers such as "C", we can improve the prospects for portability of the code to newer generations of hardware.

Real-time applications must respond to the physical environment within some set of time limits. This means that real-time applications (like RCS) must be able to guarantee that they exist in the computing environment (memory-locking) at all critical times and that they can respond to environmental stimuli within some "worst case" time limit. Generally this means that the RCS application must be able to preempt and/or take over complete control of the computing platform during real-time operation. Single user operating systems such as DOS can meet these requirements as well as most real-time operating systems. Unix operating systems and other non-real-time, time sharing, multitasking, operating systems, on the other hand, usually don't allow an application (e.g., RCS) to take over control. Therefore, only operating systems with real-time (RT) extensions (e.g., VxWorks, Lynx OS, RT-Unix, RT-Posix, etc.) should be considered as host environments.

RCS Methodology Development Steps

The methodology described in this paper should be interpreted as an iterative, "rapid prototyping", real-time software development method. The steps listed in Table 1 are roughly in the sequential order of a first pass through the method to achieve a skeleton of the overall RCS architecture to be implemented. Once a skeleton is developed, the developer(s) should iterate within the steps to develop executable controller modules in a bottom-up process. As these controller modules are developed (as executable code) their behavior can be studied and their performance measured to further refine the control hierarchy. This process should involve revisiting and revising the original problem description and requirements as well as the organizational structure and definition of the RCS controller modules. As the running system evolves it should be used as a tool to enhance the dialogue with the domain experts and sponsors. By demonstrating the evolving system, developers, experts and customers are better able to refine requirements and explicitly capture the domain expert's knowledge.

Table 1. Summary of the RCS Methodology Steps

1) Concept Development
   - A) Gather domain knowledge.
   - B) Develop the problem description / scenario
   - C) Conceptualize the application in terms of:
     - the RCS Controller Hierarchy,
     - the Operator I/F System,
     - the Data Management System, and
     - the Communications Management System.

2) Design the RCS Hierarchy using Task Decomposition
   - A) Develop a task tree (hierarchical decomposition of tasks).
   - B) Choose a "thread" of tasks which span the task tree from the highest node to the bottom of the tree.
   - C) Design the controller hierarchy by iteratively adding task threads in a rapid prototyping fashion.
   - D) Design controller software by adding design detail using generic RCS templates in a bottom-up process.

3) Coding and Testing RCS Software
   - A) Incrementally develop code for each controller using generic RCS coding templates in a bottom-up fashion.
   - B) Incrementally develop simulators to drive each controller in a closed-loop fashion.
   - C) Incrementally develop simulators for the human interfaces required.
   - D) Measure the performance of each controller in terms of execution time.
   - E) Map the controller modules (software processes) onto the computer hardware (processors).

4) Port the RCS software to the target hardware system.

5) Incrementally integrate and test the RCS controllers with the robotic system's sensors and actuators.

A) Perform lab tests.

B) Perform field tests.

6) Develop a simulator to animate the robotic system in the envisioned physical environment (workspace).

7) Design, code and test the Operator I/F System, Data Management System and the Communications Management System

8) Integrate the RCS Controller Hierarchy with the Operator I/F System, Data Management System and the Communications Management System.

9) Produce final documentation for the system version or release.

10) Iterate all of the steps above extending the RCS system, in a "rapid prototyping" fashion, by adding new controllers and/or processing modules to execute additional task threads.

CONCLUSIONS

In this paper we have attempted to begin to define a consistent set of systems engineering rules for building, evolving, and maintaining large, complex, intelligent control systems. In our approach:

1) We build on the work of Albus, Barbera, and others over the last two decades.

2) We use task scenarios in the knowledge engineering process in order to capitalize on the human associative memory capacity.

3) We have emphasized hierarchical organization as a powerful method of complexity management.

4) We have selected cyclic sampling and the finite state machine as our execution model in order to ensure our designs are deterministic and verifiable.

5) We have emphasized rule plan knowledge models (state graphs and state tables) which are compatible with and can be directly executed by finite state machines.

6) We have presented a primitive communications mechanism (Global Memory) which is compatible with cyclic sampling and provides for non-blocking I/O.

7) We have defined generic RCS Controller Module Templates and the RCS Main Program as our basic systems integration wrapper mechanism to simplify the development and integration process.

8) We have presented an outline of a set of rapid prototyping steps which can be used as a systems development life cycle approach.

Of course there are many areas we haven't discussed here. They are the subject of our ongoing research in intelligent machines. The NIST Robot Systems Division is currently conducting a long term research program, called the Intelligent Machines Initiative, which is focusing on Sensory Processing and World Modeling for machine vision in particular as well as many of the other issues not addressed here in detail.

REFERENCES

[Al 91a] J.S. Albus, R. Quintero, R. Lumia, M. Herman, R.D. Kilmer, K.R. Goodwin, *A Reference Model Architecture for ARTICS*, ASME and IIE, Manufacturing Review Volume 4, Number 3, September 1991.

[Al 91b] J.S. Albus, *Outline for a Theory of Intelligence*, IEEE Journal, Transactions on Systems, Man and Cybernetics, Volume 21, Number 3, May/June 1991.

[Al 90a] J.S. Albus, *The Role of World Modeling and Value Judgment in Perception*, Proceedings of the Fifth IEEE International Symposium on Intelligent Control, Philadelphia, PA., September 5-7, 1990.

[Al 90b] J.S. Albus, *Hierarchical Interaction Between Sensory Processing and World Modeling in Intelligent Systems*, Proceedings of the Fifth IEEE International Symposium on Intelligent Control, Philadelphia, PA., September 5-7, 1990.

[Al 89a] J.S. Albus, H.G. McCain, and R. Lumia, *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NIST (formerly NBS) Technical Note 1235, April 1989 Edition.

[Al 89b] J.S. Albus, R. Quintero, H. Huang, M. Roche, *Mining Automation Real-Time Control System Architecture Reference Model (MASREM)*, NIST Technical Note 1261 Volume 1, May 1989.

[Al 88] J.S. Albus, *System Description and Design Architecture for Multiple Autonomous Undersea Vehicles Project*, NIST Technical Note 1251, September 1988, p. 126.

[Al 81] J.S. Albus, *Brains, Behavior and Robotics*, BYTE/McGraw- Hill, Petersborough, NH, 1981.

[Ba 84] A.J. Barbera, J.S. Albus, M.L. Fitzgerald, and L.S. Haynes, *RCS: The NBS Real-Time Control System*, Robots 8 Conference and Exposition, Detroit, MI, June 1984.

[Hu 92] H. Huang, J. Horst, R. Quintero, *A Motion Control Algorithm for a Continuous Mining Machine Based on a Hierarchical Real-Time Control System Design Methodology*, Journal of Intelligent and Robotic Systems 5: 79-99, 1992, Kluwer Academic Publishers, Dordrecht, The Netherlands.

[Hu 91] H. Huang, R. Quintero, J.S. Albus, *A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real-Time Control System for Coal Mining Operations*, Control and Dynamic Systems: Advances in Theory and Applications Volume 46: Manufacturing and Automation Systems: Techniques and Technologies, Part 2 of 5, Edited by C. T. Leondes, Academic Press,1991.

[Mc 86] H. G. McCain, et al., *A Hierarchically Controlled Autonomous Robot for Heavy Payload Military Field Applications*, Proceedings of the International Conference on Intelligent Autonomous Systems, Amsterdam, The Netherlands, December 8-11, 1986.

[Qu 92] R. Quintero and A. J. Barbera, *An RCS Methodology for Developing Intelligent Control Systems*, NISTIR 4936, October 1992.

[Ra 82] Random House College Dictionary, 1982, Revised Edition

[Si 83] 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, pg. 17, 1983.

[Sz 88] S. Szabo, H. A. Scott, R. D. Kilmer, *Control System Architecture for the TEAM Program*, Proceedings of the Second International Symposium on Robotics and Manufacturing Research, Education and Applications, Albuquerque, NM, November 16-18, 1988.