NISTIR 5142

An Intelligent Control System for a Cutting Operation of a Continuous Mining Machine

John A. Horst
Unmanned Systems Group

and

Anthony J. Barbera
Advanced Technology and Research Corporation
Laurel Technology Center
14900 Sweitzer Lane
Laurel, MD 20707

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

NIST

based on definitions in the NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM) [Albus 89]. RCS modules at these levels address the solution of problems that suit a rule-based as opposed to continuous control based approach. The type of rule-based problems well served by this implementation are characterized by tasks that span time intervals between 0.5 seconds and 5 minutes and spatial distances between 5 centimeters and 50 meters. Such rule-based problems are characterized by an abundance of 'conditional' code, *i.e.*, code that contains large amounts of `if then else` type decision making. For example, delivering a missile to a target is largely a continuous control based problem whereas following a recipe to bake a cake is more of a rule-based problem. RCS has often been used to solve large-scale conditional[1] sequencing and coordination problems where the continuous control part of the problem is very simple. A good example of a conditional sequencing and coordination problem is the mining machine control problem at hand. This is even more true due to the fact that continuous mining machines have on-off or 'bang-bang' style control of appendages. Nonetheless, traditional control algorithms can still operate within this implementation of RCS. Other implementations of RCS have been specifically designed for dynamic and kinematic generation of trajectories and servo control of position, velocity, force, and mechanical impedance. These levels of control are particularly addressed at the PRIMITIVE and SERVO levels of NASREM [Fiala 87, Wavering 88].

The key focus of this paper is the current status of an implementation of the RCS methodology for the control of a continuous mining machine used in underground coal mining. Some RCS design and implementation tools, those in use and proposed, are also identified and discussed and their exploitation for mining machine control system design is detailed.

Two recent trends are critical to large scale control system design:

1) hardware is cheap
2) software development is expensive.

How can these trends be exploited for better, cheaper systems? Under the RCS paradigm, involving more memory and processors (at little extra cost) than required can bring significant improvements in maintainability and extensibility leading to significant overall cost savings.

Control under the RCS paradigm provides robust real-time control that is maintainable and extensible and requires a short design and test cycle time. Established and emerging software design methods for handling complexity, *e.g.*, managing complexity through abstraction, encapsulation, and inheritance [Coad 91] are utilized. Tasks are encapsulated within controller modules based on common functionality and level of abstraction. RCS can be thought of as a real-time operating system specifically tailored to large-scale decision oriented real-time intelligent control applications.

Parallel processing is inherent in the approach and therefore, distributed control, simulation, and animation across a hybrid of hardware and software is allowed. The current implementation for mining machine control demonstrates just such distributed computing. Distributed parallel processing is allowed largely because of a simply defined multiple buffering scheme, coupled with carefully timed cyclic execution of all software modules on each central processing unit (CPU). For this paper, a software module is defined as a process that has clearly defined inputs and outputs. The types of software modules we will consider are controllers, simulators, and animators.

This work has been performed by NIST and ATR in support of the US Bureau of Mines (BOM).

1. "Conditional" meaning code that contains large amounts of `if then else` type decision making.

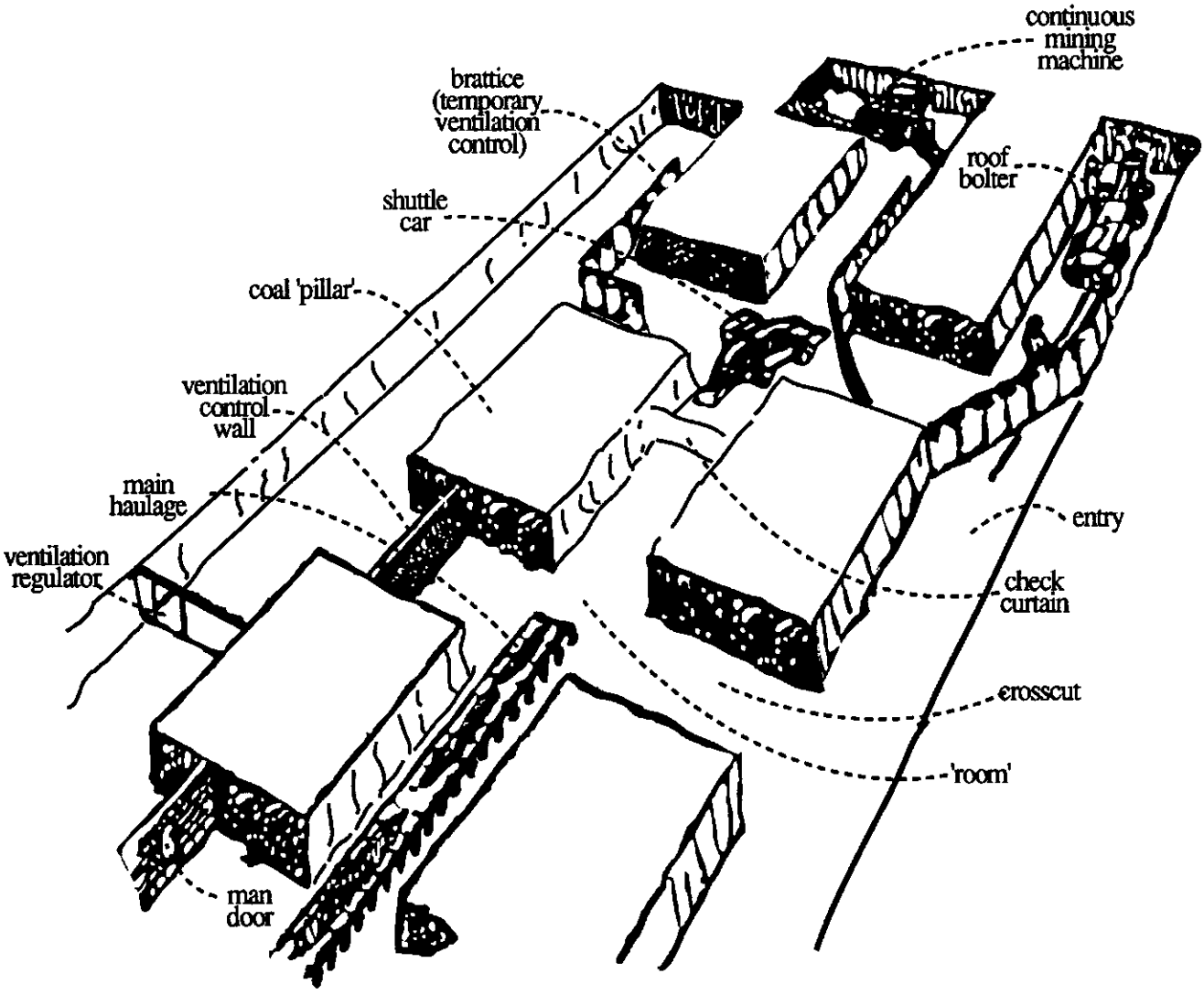


Figure 1: A 'room and pillar' coal mining operation

A box cut is ready to be performed when the CM is within a few feet of the coal face. When in this position (assuming no other errors), the cutting drum is raised to the proper height and turned on. The CM trams forward toward the coal face. During this initial tram forward, the current load on the cutting drum motor is monitored. If this load exceeds threshold, the CM is declared to be at the face[4]. In addition, the orientation of the CM is closely monitored for deviation from the proper orientation[5]. This is called the initial_approach_face task.

4. The face is defined as the front of the coal seam where cutting operations occur.

5. Since it is difficult to recover from a cut that is skewed at the start, the initial orientation of the CM for a box cut is important.

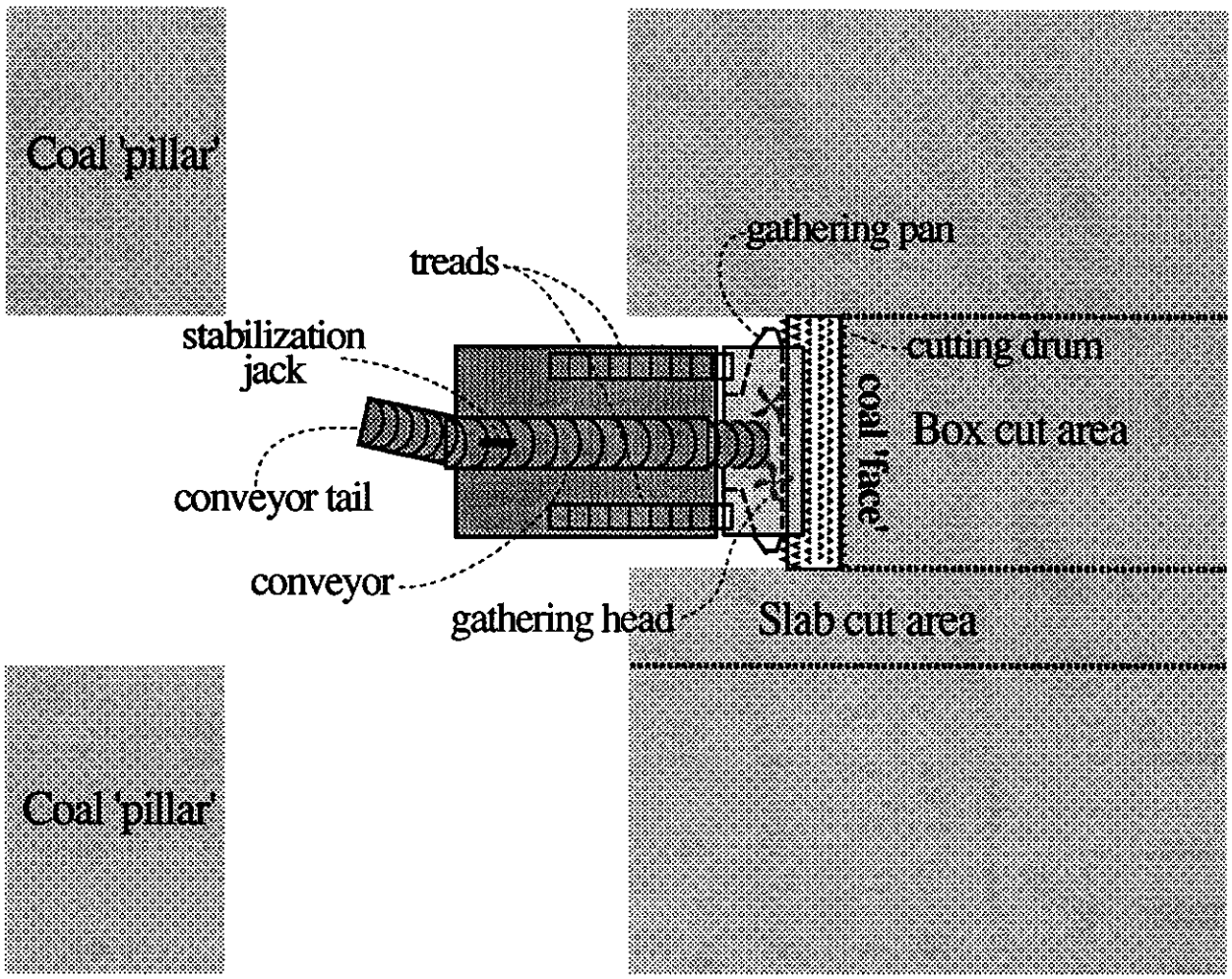


Figure 2: A continuous mining machine executing a box cut

Within a box cut operation, the initial_approach_face task precedes the first execution of a sump_shear_cusp task. After completion of the initial_approach_face task, the sump begins with the drum at the ceiling and rotating. The gathering pan is put in the float position. A command is sent to an operator to put the tail of the conveyor belt into the correct position for deposit of coal into a shuttle car or continuous haulage conveyance. The CM trams forward for about half the diameter of the cutting drum. The completion of this tram forward event is detected by a position and orientation sensor[6] and triggers the start of the shear operation.

During a shear, the following occurs:

6. The US BOM Pittsburgh Research Center (PRC) and NIST have exerted significant effort on position and orientation sensing and have developed several such systems including length and angle measurement systems and the MAPS inertial guidance sensing system [Sammarco 92].

1) the stabilization jack is lowered,
2) the cutting drum remains on and is lowered, and
3) the gathering head and conveyor are turned on.

The latter is done only if enough coal has piled up to warrant turning on the conveyance systems and a status message is received from the operator interface module stating that the positioning of the conveyor tail has been accomplished. There are a couple of reasons for a shear to be halted:

1) A shear has progressed to such a point that the loose coal needs to be removed and a signal has not been received from the operator that the tail of the conveyor system on the CM is in position over the haulage system.
2) During a shear, the cutting drum occasionally pops out of the cut[7]. The control code watches the pose sensor for a significant deviation and appropriate corrections are made and actions taken[8].

When the cutting drum boom reaches the appropriate angle down near the floor of the mine, the shear task is complete. The choice of an appropriate angle is dependent on whether the coal seam is level or sloped at that point.

Due to the cylindrical shape of the cutting drum, the sump and shear operations leave a residue of coal, called the cusp, on the mine floor between the cutting drum and the gathering pan. The cusp is removed by raising the stabilization jack and tramming in reverse with the cutting drum remaining on. This completes the first sump_shear_cusp cycle. The next task is the approach_face command which in most respects is the same as the initial_approach_face command described above. Most importantly, during approach_face, the exact location of the face relative to the CM is known which was not true during the initial_approach_face. Therefore, after raising the boom as before, the CM need only tram forward a specified distance after which contact with the face is guaranteed; no cutter current load monitoring is required, however, in a final system it would be useful to monitor cutter motor current load as well as CM position during the approach_face command. This would assure robust performance of each approach_face command. Then another sump_shear_cusp cycle is executed exactly as before. At this point in the box cut task, a sequence of approach_face and sump_shear_cusp commands are executed until the total cut distance (specified by an operator) is reached. The box cut task is now complete.

2.2 Task tree

It is useful after developing the scenario to generate the explicit decomposition of tasks, based on that scenario. Graphically, this task decomposition can take the form of a tree as in figure 3 on page 8. This is a useful exercise in clarifying the scenario. In addition, with the help of the task tree, tasks of similar type and at the same level of task abstraction are grouped into controllers. For example, in figure 3 on page 8, each task is prefixed by a two letter mnemonic where each mnemonic corresponds to a particular controller module.

---

7. Such a deviation can occur for various reasons, for example, due to an anomaly in the coal seam, to slippery floor conditions, and/or failure of the stabilization jack.

8. The logic for handling such an error is in the control code but has not been tested.

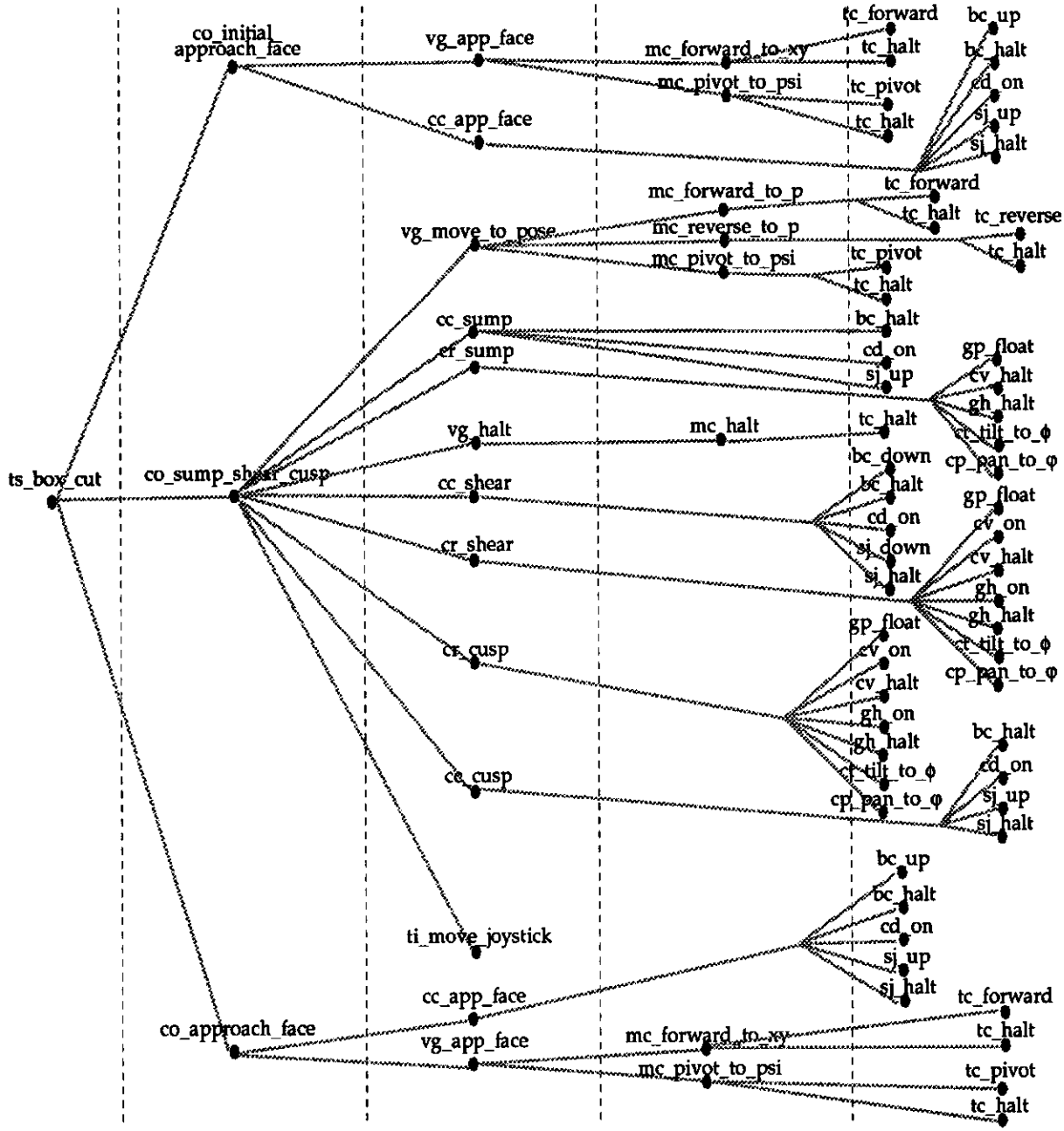


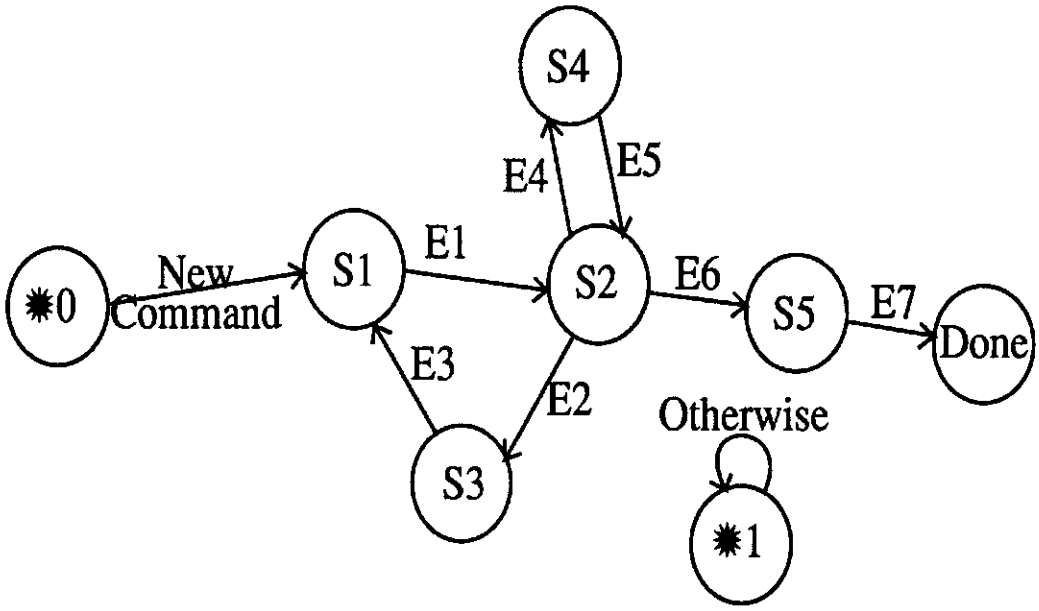
Figure 3: Task tree for a box cut of a continuous mining machine

2.3 State machine example: a sump_shear_cusp command

In section 3.5 on page 18 we explain that RCS adopts a finite state machine model. Here we give an example of such a finite state machine as used in our code.

While performing a box cut, a fundamental operation of the CM is the sump_shear_cusp command for removing coal (as described in section 2.1.2 above). This simple command is decomposed to lower level commands and delivered to its three subordinate control modules within a

decision state table. This decomposition can be seen in the task decomposition tree of figure 3 on page 8. However, the task tree is limited in that it only reveals the set of tasks that are derived from each higher level task. The task tree neither reveals the conditions that trigger the execution of each subordinate task, nor does it specify the precise order in which each subordinate task is executed. Therefore, a decision structure of some sort is required beyond a simple task decomposition. We have chosen to use finite state machines, though this is not the only option. The state graph and state table for this specific command is given in figure 4. 'C' code for this state logic is listed in appendix C on page 36.



Conditions | Actions

Event	Current State	Next State	Jobs	Commands to subordinates: Coal removal (cr)	Vehicle guidance (vg)	Coal cutting (cc)
New Command	✸0	S1	set pose for sump	cr_sump	vg_move to pose	cc_sump
E1:sump distance reached	S1	S2	none	cr_shear	vg_halt	cc_shear
E2:popped out of shear	S2	S3	none	cr_halt	vg_halt	cc_shear correct
E3: shear correct done	S3	S1	set pose for sump	cr_sump	vg_move to pose	cc_sump
E4:too much loose coal	S2	S4	none	cr_halt	vg_halt	cc_halt
E5:tail in position	S4	S2	coal removal	cr_shear	vg_halt	cc_shear
E6:shear complete	S2	S5	set pose for cusp	cr_cusp	vg_move to pose	cc_cusp
E7:cusp complete	S5	Done	none	cr_halt	vg_halt	cc_halt
Otherwise	✸1	Same	NOP	NOP	NOP	NOP

✸i => don't care condition Sn => n^{th} state En => n^{th} event

Figure 4: A state graph and associated state table for a sump_shear_cusp command (task)

We note that if the cutting drum pops out of the coal during a shear, it is assumed that no correction in orientation or position is needed. The shear correction done by the coal cutting controller might simply involve raising the cutting boom followed by a repeat of the sump task.

During the sump and shear, the coal removal controller specifies that commands can be received from the operator interface module responsible for getting the conveyor tail position from a human operator.

2.4 Design of the controller hierarchy

We choose to have the continuous mining machine (CM) perform all aspects of a box cut. A detailed scenario for a box cut task has been defined (section 2.1.2 on page 4). A task tree showing the decomposition of tasks into subtasks is complete (figure 3 on page 8). However, throughout this process of scenario development and task decomposition, tasks of the same level of abstraction are grouped into what are called controller modules[9] or simply, controllers. At the lowest level, the controllers are matched with the appropriate actuators. At all levels, sensors, world model values and functions, and operator interfaces are matched to the appropriate controller modules.The relationship between the task tree and controller hierarchy can be seen by comparing the task tree of figure 3 on page 8 and the controller hierarchy of figure 5 on page 11.

Controllers (and, similarly, simulators and animators) are the key 'objects' in the RCS methodology. Tasks are encapsulated within each controller and the communications interface to its supervisor and subordinate(s) is of the same form for all controller 'objects.' The internal structure of these controllers is discussed in section 3.6 and section 3.15 and the nature of the communications interface in section 3.8 and section 3.9. Each controller is responsible to synchronize and coordinate the tasks of all its subordinates.

Tasks at similar levels of abstraction are grouped into controllers. These are the principles guiding this grouping:

1) the relatedness or common functionality of the tasks in each controller
2) keeping the number of tasks in each group small enough for human perspicuity.

For example, we grouped all cutting related tasks at a certain level into the coal cutting controller and similarly grouped tasks relating to coal removal into the coal removal controller. The sum of this design effort up until now produces the controller hierarchy shown in figure 5 on page 11. Of course, this design is mutable which is just as we would want it to be. In addition, as the number of tasks grows in a particular controller, the controller can be split up into two or more separate controllers.

Before coding can begin, state logic (state graphs) must be developed for all the tasks, both high level (*e.g.*, box cut) and low level (*e.g*, left tread motor on)[10]. From these state graphs, any functions required by each state machine are identified and coded. Now we are ready to begin to use generic 'C' code (called 'C' templates, as described in section 4.1 on page 27) allowing the designer to fill out these templates and thereby specify the controller's activities.

Simulation and animation are required for our implementation at NIST, since we had no CM available for use. However, even if equipment is available for testing, the development of simulation and animation is well advised, since it is a safe way to perform debug and refinement of task knowledge and since CM operation is costly. Coding the software modules and mapping these modules onto specific hardware is complete for the mining implementation. The current mapping done for CM control, simulation, and animation is shown in figure 8 on page 14. All simulation

9. We define module as a generic term including grouping of like tasks into controllers, sensor and actuator simulators, animators, operator interfaces, and communication handlers (see figure 8 on page 14).

10. An example of a state graph for the sump, shear, and cusp task is given in figure 4 on page 9.

(control, sensor, actuator, and environment) is chosen to run on PC#1 (see figure 8). This choice simplifies the simulation of CM appendages since the simulation code on the PC can be made to run without interruption. On the other hand, animation is chosen to reside on a Silicon Graphics IRIS™ under a non-deterministic operating system (namely, UNIX™). For animation to be effective, all that is required is that the CM and its appendages move in a manner realistic to the human observer.

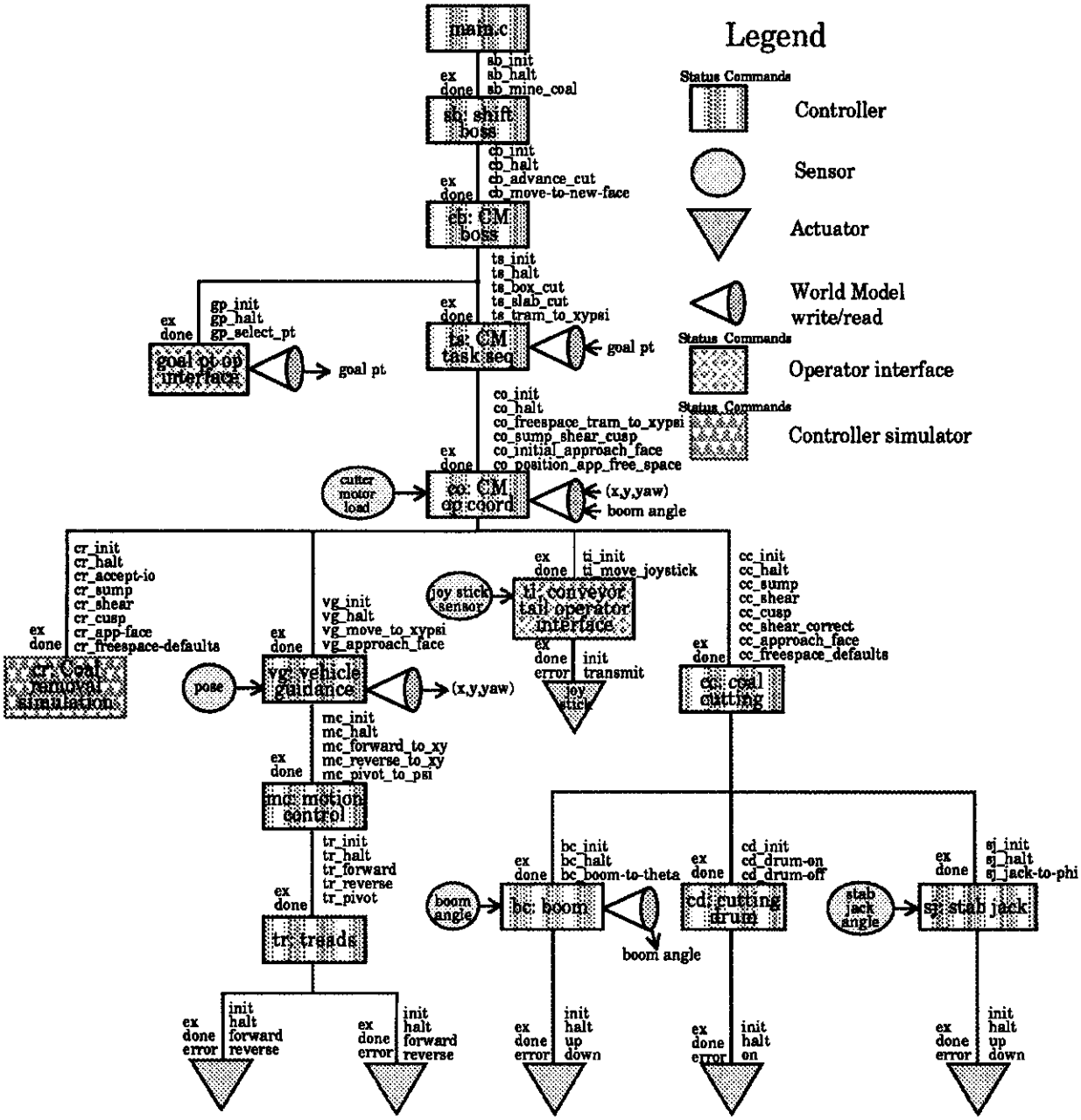


Figure 5: Example of RCS control system hierarchy (without sensor or actuator simulators)

Animation is a useful debugging tool. However, our software tools offer additional debugging capabilities available in the 'C' code on the PC. For instance, values relating to system status can be displayed. Each software module within the hierarchy has its own list of values that the debugger monitors. For example, the maximum execution time of the module is computed and can be displayed. In addition, single stepping and slower execution are available.

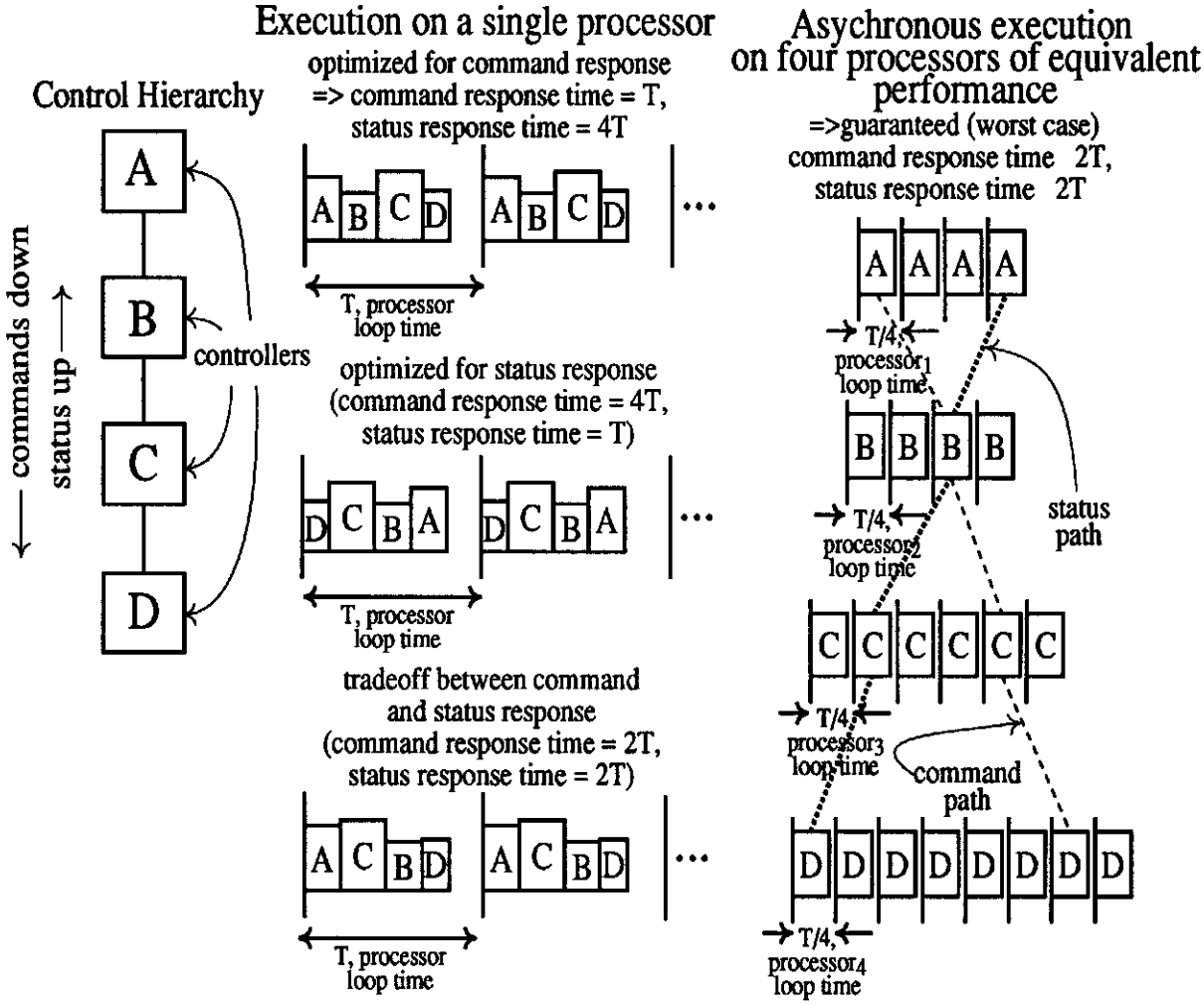


Figure 6: Serial and parallel execution of controllers

With the relatively small amount of processing done by all modules, we were easily able to map all controllers and simulators to a single CPU. Because of the cyclic execution pattern in RCS, these particular modules are executed serially. We have chosen an ordering for CM control such that the command execution goes from top to bottom in one system cycle. An illustration of how execution ordering was done for CM control, simulation, and animation is shown in figure 7 on page 13. Our ordering is optimized for command response, since, for example, 'halt' commands need to be executed immediately. Other applications may not have this requirement and a designer could optimize the execution ordering of software modules for command or status[11] or anything in between. However, when optimizing for maximum command response, a status message will take *n* system cycles to travel from the bottom level to the top level if there are *n* super-

visory levels in that particular thread in the control system hierarchy. Figure 6 illustrates the comparison between serial execution of modules on one CPU versus asynchronous parallel execution on multiple CPUs. In summary, serial execution performs at least as well as asynchronous parallel execution and can be optimized for very fast command or status response.

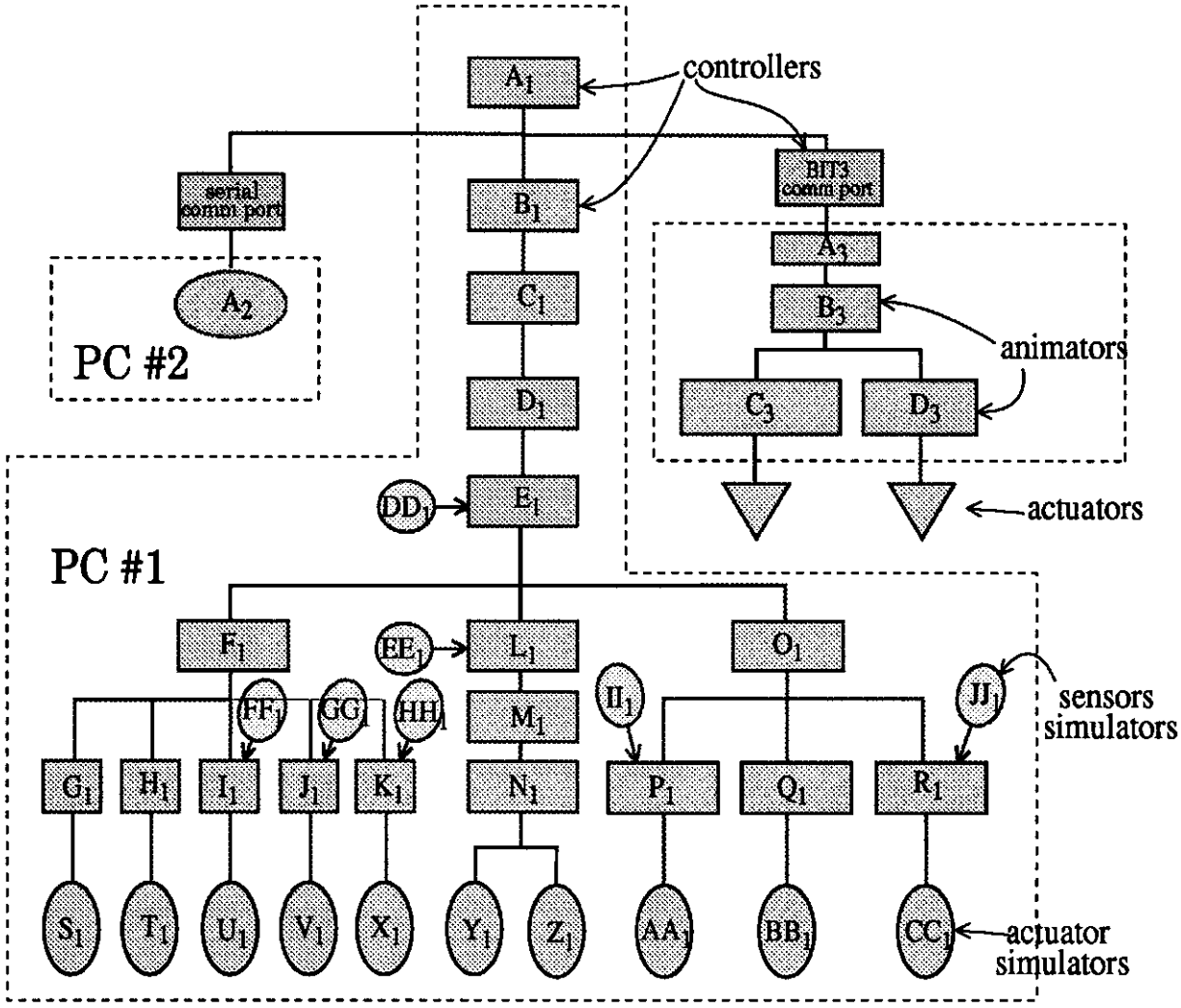


Figure 7: Module execution order for the CM implementation where the execution sequence for the i^{th} processor is: A_i, B_i, ..., AA_i, BB_i, ..., A_i, B_i, ...

The technique of module execution ordering just described can be thought in terms of process scheduling, such as performed by an operating system. A typical multiprocessing operating system such as UNIX™[12] performs dynamic process scheduling. Such operating systems schedule the execution of processes by allotting certain fixed quanta of time to each process. When the process releases the processor (by exiting or blocking) or the time limit has passed, another process

11. Status is the explicit response of a subordinate controller to commands from its supervisory controller. Status is typically 'executing', 'done', or 'error'. See figure 5 on page 11.

12. Reference to product or company names is for identification only and does not imply government endorsement.

that is ready to run is selected to execute. Process priority is a commonly used technique for ordering the runnable processes. In addition, there is no fixed system cycle time. This indeterminate characteristic of UNIX™ is why it is not often used in RCS implementations of the type described in this paper. In contrast, we use a method of scheduling that could be described as static process scheduling, since the processes are executed according to a fixed schedule. Each process is designed to execute repeatedly, and on each invocation, it will execute to its conclusion. Each process must be executed at a specified frequency. If the resources of the system are insufficient to meet the required schedule, it can be discovered by the scheduler using the debugging tools provided.

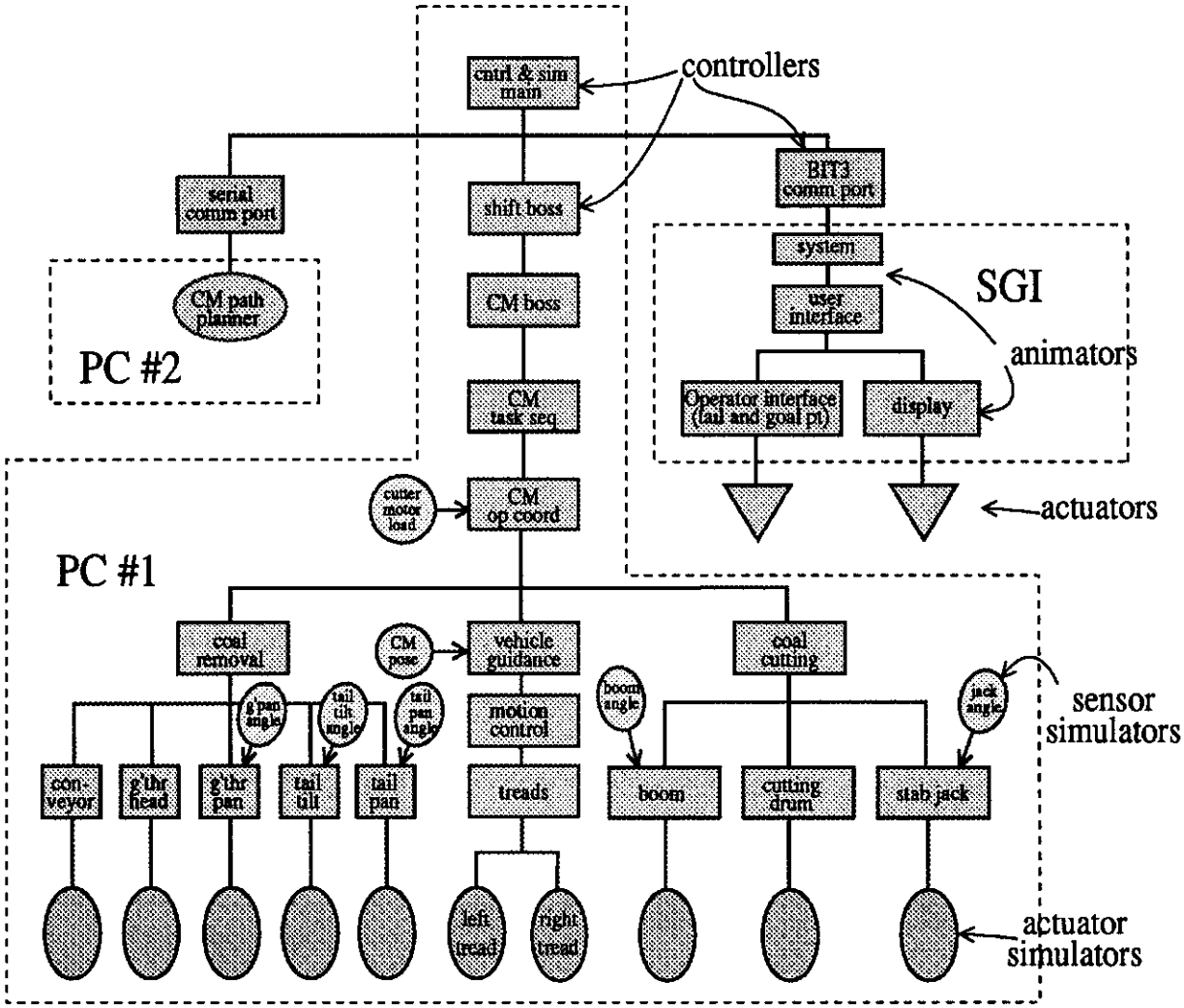


Figure 8: Continuous mining machine control, simulation, and animation hierarchy

2.5 Simulation

We implement four types of simulation: sensor, actuator, environment, and controller simulation. Sensor, actuator, and controller simulators used in the CM control implementation are illustrated in figure 8. Note that not all actuators have sensors directly associated with them. For

example, as we currently have specified the system, both left and right treads operate open loop. A single sensor, the pose sensor, discovers the composite effect of the activity of both actuators. Note, as well, that the pose value enters the hierarchy at a higher level than, for example, the sensed value of boom angle.

The dynamics of the real actuator are specified in the actuator simulator. The sensor simulator filters the output of the actuator simulator to model the performance of the real sensor system. For example, the boom actuator simulator receives a command from the boom controller to move to a particular angle. The simulator contains the parameters, such as angular velocity of the boom, relating to the mechanical response of the boom and its motor. These output values are then filtered by the boom angle sensor simulator based on the particular dynamic characteristics of the real sensor system.

The environment simulator in our current implementation is a simple coal/no-coal two dimensional map of the mine. We might later add things like slope and slipperyness (as parameters of the mine floor), moving obstacles, failure of ventilation and/or high methane gas levels, etc. As a tool for general robotic applications, a map with obstacles can easily be simulated.

2.6 Animation and operator interface

Much of the animation code used in the CM control implementation is portable. Similar code has been written for other RCS applications following the RCS design philosophy. The animation hierarchy is illustrated within figure 8 on page 14. The existing animation code is in 'C' on the Silicon Graphics IRIS™ [13] (SGI) using 'GL,' a SGI specific 'C' library of predefined graphics functions. As is illustrated in figure 8, the animation code resides on a separate hardware platform from the simulation and control code.

The display screen is considered an actuator to be controlled. The user interface is similarly structured. The user interface allows the user to change the viewpoint and scene through a mouse.

Also available on the SGI are the two operator interface capabilities for:

1) control of the position of the conveyor tail and for

2) entry of a goal point for the movement of the machine in the mine (either for cutting or tramming in free space).

The following are required to convert the 'generic' animation code (currently in use for several other applications) into the CM control implementation.

1) Draw the continuous mining machine, i.e., specify the coordinates of its body and all its appendages in three dimensional space.

2) Get the SGI to read appendage position values from the PC via the common memory (as in figure 9 on page 16) and write operator interface values to the same common memory.

3) Adjust existing animation code to draw the mining machine on the screen cyclically based on both the raw values and the user specified viewpoint.

4) Integrate operator interface as required. For the CM control implementation, we have control of the conveyor tail and the goal position for CM free space navigation and cutting.

---

13. Reference to product or company names is for identification only and does not imply government endorsement.

5) Allow user interface to give user ability to change scene and viewpoint with mouse or keyboard.

6) Incorporate the display of all common memory values.

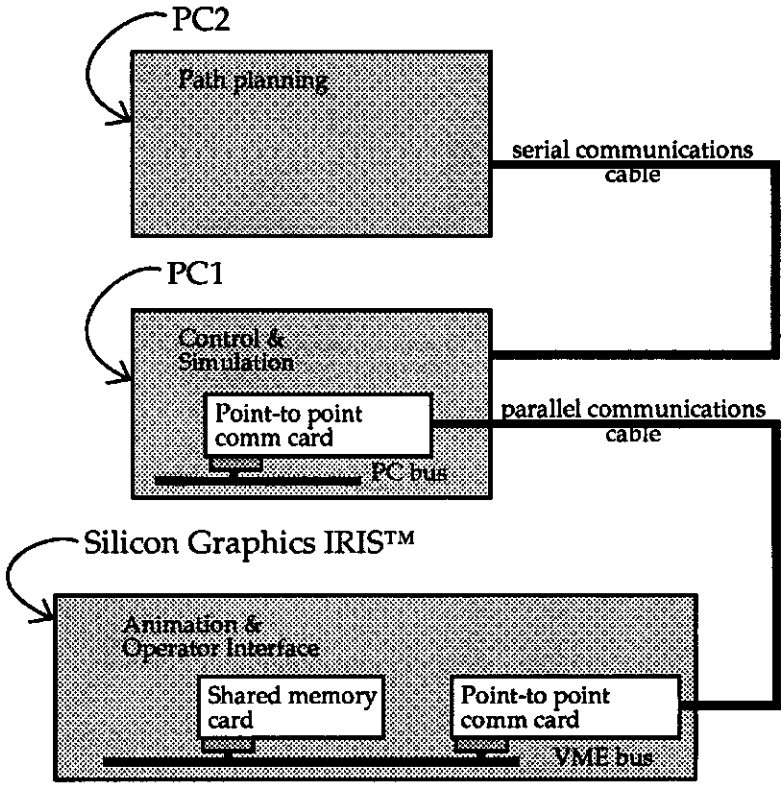


Figure 9: Hardware tools

3 Principles of the RCS methodology

This section presents more detail on the RCS methodology in the form of important characteristics that clearly define the method. Each subsection will present and examine some key aspect. Most of these key design concepts will be related to the mining machine control problem at hand.

3.1 Problem analysis through task decomposition

Fundamental to the control of electro-mechanical devices is understanding the task it is to perform. For tasks that involve complicated sequences of steps and the complex coordination of many operations, the exact nature of that task becomes critical to solving the control problem. The task drives the use and sychronization of sensors and actuators, the choice of computing hardware, and the nature of the world model. Effective design of control systems for complex decision oriented problems is well accomplished through careful task analysis and decomposition. For example, the task, mine_coal (as in figure 5 on page 11), is quite complicated due to the many subordinate commands which come from it, *e.g.*, box_cut and slab_cut.

3.2 Controllers encapsulate tasks

Tasks of similar type and similar level of temporal and spatial abstraction are grouped into software controllers (illustrated in figure 5 on page 11). Controllers can be thought of as objects encapsulating tasks that support a particular system function. For example, all tasks specifically and exclusively relating to cutting operations are grouped into the coal cutting controller as can be seen in figure 5 on page 11.

Controller 'objects' encapsulating functionally related tasks allow generic and orderly communications between those controllers. In other words, the interfaces between controller 'objects' are standardized. In addition, a common structure is defined for all controllers (described in section 3.15 on page 24) which helps simplify the design of these interfaces.

3.3 Hierarchical with strict chain of command

RCS is perhaps best known for offering a system design architecture that is hierarchical [Albus 89]. This hierarchical breakdown of complexity is specified based on space and time horizons that are natural to the tasks that need to be performed. For example, the task, "drive to the mall," has longer time and space horizons than the task, 'turn the key in the ignition' and is logically placed at a more abstract level in the hierarchy. Tasks of similar type and of similar time and space horizons (*e.g.*, 'depress the brake pedal,' 'release the brake pedal,' 'tap the brake pedal,' *etc.*) might form part of the same controller.

Why have a hierarchical structure? Common automation tasks are logically organized in a manner such that large control tasks decompose (*e.g.*, 'drive to the mall') into the more detailed tasks at lower levels of abstraction (*e.g.*, 'turn the key in the ignition'). In addition, hierarchical structures for computer controlled systems closely model 'real-time' human organizational structures (*e.g.*, military). In contrast, systems designed with, for example, mutually cooperating agents involving complex, uncertain decision making and planning are arguably not the most effective model for real-time control.

RCS specifies that at any instant of time, a controller lower in the hierarchy (a 'subordinate') will have only one immediate supervisory controller.

3.4 Rule-based

The decision-making structure at the heart of RCS is similar to the `if` *condition*$_1$ `then` *action* `else if` *condition*$_2$ `...` structure associated with rule-based architectures. Therefore, RCS consists of a set of expert rules since the designer, in concert with a domain expert, generates these rules (in the form of state machines) prior to execution of the code.

Being rule-based, RCS is especially suited to handle control problems that involve a complex sequencing and coordination of many operations and where the continuous control part of the problem is very simple. The continuous mining machine problem easily fits into this class of problems. Problems where the continuous control part is complicated and there are very few operations that need to be sequenced and coordinated are better served by traditional single level control, for example, PID control. Nonetheless, RCS can accommodate the smooth integration of both rule-based and traditional control.

3.5 Finite state machine model

A finite state machine has finitely many states and responds to input conditions by transitioning to a new state. An example of a state machine is the state graph and state table of figure 4 on page 9. The decision structure of RCS adopts the finite state machine model. Why? A well constructed decision table or state graph of manageable size is easy to comprehend. It also works easily with well established discrete time control for low level control. The concept of generic controllers containing manageably sized state machines means that controllers at high levels of abstraction are structurally identical to lower level servo controllers. Adopting a finite state machine model means that the task and all its subtasks are each executed using repeatable read, compute, and write operations. It has been our experience that RCS, with the finite state machine model, can exhibit complex and intelligent dynamic responses to uncertain stimuli even though all aspects of the system are completely deterministic and verifiable.

3.6 Generic controllers

Under RCS, controllers at every level in the hierarchy are formed from the same underlying structure (see figure 13 on page 25). Each controller contains sensory processing, world modelling, and behavior generation functions. RCS controllers are organized around a triptych of processing: pre-processing, decision processing, and post-processing. This generic structure can greatly simplify system design for control, simulation, and animation modules as we argue in section 4 on page 27. Use of these templates also enhances perspicuity and maintainability.

3.7 Determinism

Using RCS for control system design assures determinism by requiring, at least, the following: 1) avoid controller process interrupts, 2) enter task knowledge (rule plans) explicitly into the system, 3) that the designer know the maximum execution time of controllers, and 4) that the order of execution of controllers on each CPU be explicit at any system cycle.[1]

---

1. Determinism in RCS does not mean that random search or Monte Carlo style algorithms are prohibited. Nor does it imply that every permutation of possible states of the world need be tested and investigated.

It is common to see real-time systems designed around interrupt driven prioritizable processes. However, for such systems of reasonably large scale, the designer is burdened with a forbidding task of determining beforehand whether each process can recover successfully from an interrupt at any point during its execution. In practice, many designers simply code the processes and adjust the priorities of processes in a somewhat random manner until system performance appears to be acceptable. We argue that this leaves too much to chance. Avoiding controller interrupts keeps the system in known states. Since extra processing power is abundantly and cheaply available, RCS requires that all processes execute in a cyclic manner (effectively in parallel with multiple processors). No one process interrupts another.

In RCS, the decomposition of high level commands into lower level commands is made explicit in the system design phase. In contrast, we argue that design methodologies such as mutually cooperating agent models and systems with complex planners (planners that are expected to create new rule plans[2] during system execution) may be less effective at solving real-time control problems partly because of the complex and virtually indeterminate interaction between such processes. RCS maintains that 'intelligent' tasks can be performed in uncertain environments when virtually all the task knowledge is entered *a priori* by the designer in the form of decision logic state graphs or tables. The latter is not intuitive and is the focus of continuing research.

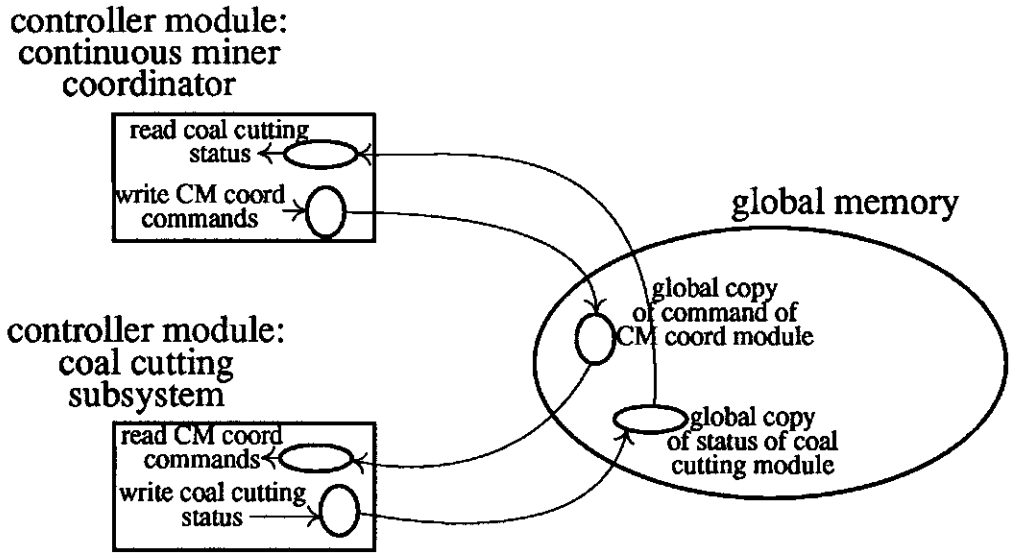


Figure 10: Example of command/status copying

3.8 Data integrity through multiple buffering

RCS specifies that communication between controllers be done in the following manner. Within each control cycle, each controller reads commands from its supervisory controller and status from its subordinates. After all processing is completed in the current cycle, the controller writes commands to its subordinates and status to its supervisor. This scheme can be seen in figure 10. In order to maintain data

2. Though, typically, one finds path plans being generated during execution than rule plans.

integrity, each controller works with local copies of all this data while doing its initial read from and final write to a global copy. This assures data integrity. In recent years, semaphores have become a common method to assure the integrity of shared data during communication between processes. However, since memory is abundant and cheaply available, we argue that maintaining multiple copies is reasonable and may be more perspicuous.

For sensor values we have a multiple buffering scheme as in figure 11 on page 20. The sensed value is written only by one controller. Any other module that seeks to use this value has read-only privileges.

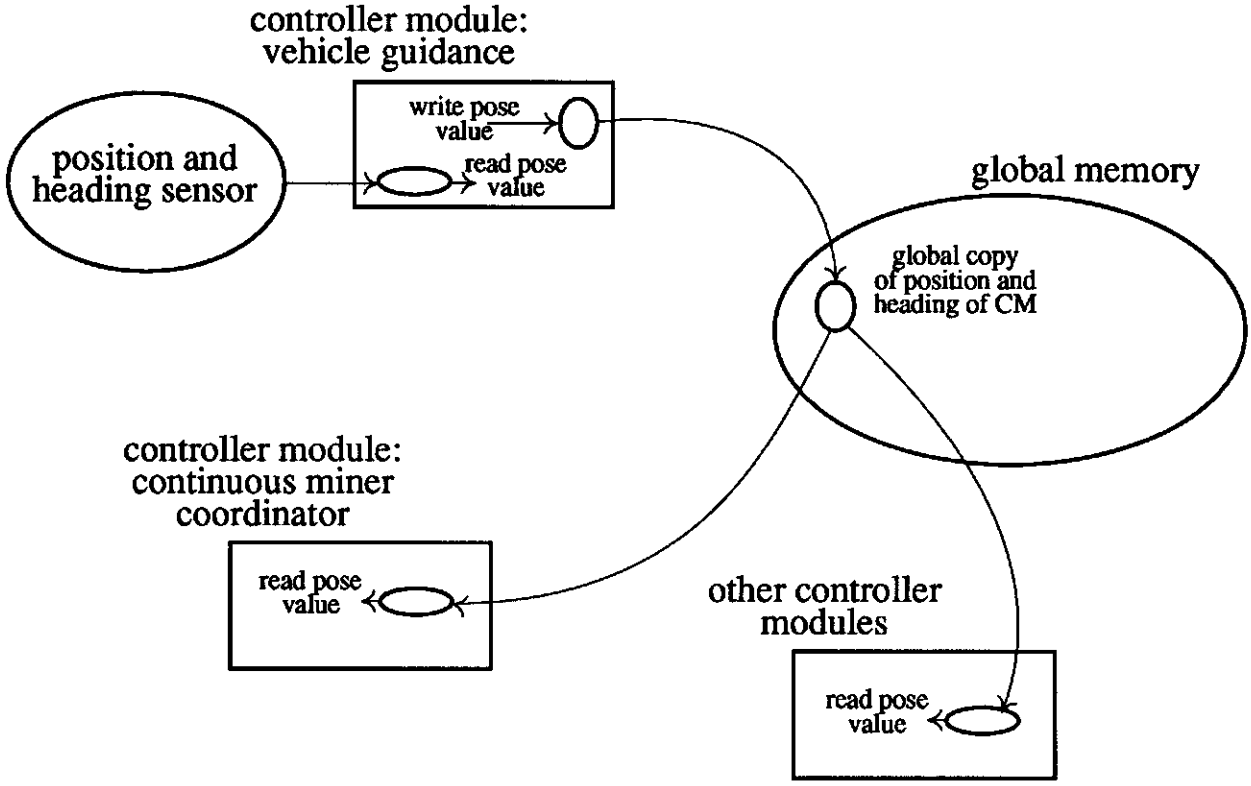


Figure 11: Example of multiple buffering

3.9 Handshaking between controllers by command numbers

RCS requires that a serial number accompany each command from supervisor to subordinate. The handshake is completed when the same number is returned along with status by the subordinate controller. This indicates that a particular command from the supervisor was received. Each serial number is unique to a particular controller and increments only at each new command from that controller. This also serves as a useful debugging tool, since it can be used to detect if a controller is not responding to commands. See figure 13 on page 25 to see how this handshaking fits into the execution of a generic controller.

This form of handshaking is not in conflict with parallel processing of controllers. Even if controllers

are distributed across several processors and running at different system cycle rates, a controller needing command data from a supervisory controller on the 'slower' processor, simply executes its cycle, performing its normal execution until the command data are received. The supervisory controller can be made aware that the subordinate has received the command by examining the command number and status whenever it becomes available.

3.10 Real-time execution through cyclic processing

We have mentioned that the method of handshaking described in the previous section allows for distributed parallel processing. Nonetheless, within a single processor, the controllers mapped to that processor are sequentially executed in an infinite loop. Real-time execution is gained when the RCS designer ensures that all controllers can respond to changes in the environment within the time required by stability considerations. On a given processor this translates into a constraint in which the designer is required to ensure that all controllers will execute within a certain time frame called the system cycle time. The latter is defined as the time it takes[3] for all the controllers in the control hierarchy to execute on a single processor.[4] This type of cyclic execution helps to bring to a manageable limit the number of possible states of the system. This cyclic pattern of execution is illustrated in figure 12 on page 23. On-line conflict resolvers and complex schedulers are avoided, since they are done off-line by the system designer.

This method contrasts sharply with the currently popular use of prioritized interrupts and time slicing. Both interrupt driven methods and RCS can achieve real-time execution. However, the use of interrupts tends to complicate the code beyond ease of human understanding, particularly, as the number of real-time processes increases. If one allows a real-time task to be interrupted, the state of the system or world may have changed in an unknown way after servicing the interrupt. One needs to handle a prohibitive amount of exceptions in order to determine what an interrupt would mean at any point in the control code. Typically, this type of code is difficult to maintain and, when it changes hands, a long time is spent understanding it. Often, in the end, the code is rewritten.

Since processor costs are comparatively low, we see little reason to employ complex software constructs, like time slicing and process interrupts, which were created to optimize processor usage. Cyclic execution of all controllers helps achieve determinism, allowing for an assured real-time response as long as the maximum execution times of controllers are known and there is never any looping within processes causing the rest of the system to wait. Additionally, through multiple buffering and handshaking by command numbers, RCS is a parallel processing architecture, even though there is serial execution on each CPU.

The controllers on a given processor are executed sequentially. What is the rule guiding the order in which they are executed within each cycle? It should be noted that the order one chooses to execute these controllers will not adversely effect the communication between supervisor and subordinate controllers because the method of handshaking used prevents such problems. However, the order of sequential execution of controllers will effect how quickly commands (down) and status (up) will ripple through the hierarchy. This could hamper or eliminate real-time response of the system. We typically choose an

---

3. The system cycle time is typically on the order of tens or hundreds of milliseconds for electro-mechanical systems.

4. Multiple processors, each executing controllers cyclically and serially, are often required to assure execution of all controllers within a specified time frame.

ordering such that command execution goes from top to bottom in one cycle. The disadvantage of this ordering is that status will take n cycles to ripple to the top level if there are n supervisory levels in the control hierarchy (see figure 6 on page 12). However, in general, one can trade off status and command response times, and the needs of the particular application will dictate the optimum order of execution.

3.11 Straight-through execution of controllers; no conditional looping

Another feature of RCS is to require that there be no indeterminate looping on condition or polling loops within a controller responsible for control. This is required because an executing controller is never interrupted. The execution of that controller would have to be interrupted, if conditional looping were allowed within a controller, in order to ensure real-time operation. Forbidding conditional looping and controller interrupts means that all controllers are guaranteed to execute within a known period of time, ensuring deterministic execution of the entire system.

For example, certain controllers are responsible for reading the sensed value of pose (position and orientation) of the continuous mining machine (CM). A common way of handling this is for certain processes (controllers) to poll sensors until new values become available. Other processes (controllers) are allowed to interrupt the polling based on carefully determined priorities. RCS is distinctly different in that the controller checks the sensor at regular intervals and, if a new value is not available, processing continues with no interrupts or polling. In this sense, a RCS control system samples sensors, commands, and status at deterministic (not necessarily uniform) intervals.

3.12 Multiprocessing inherent

Large-scale designs quickly consume processor time, particularly since the method requires execution of controllers until completion on each cycle. Therefore, in a typical design it is not long before the entire set of controllers in many large scale designs cannot execute serially within the system cycle time that is required for stable control. This requires that some subset of controllers be executed in parallel on another processor. RCS specifies a uniform method of communication between controllers (through the use of multiple buffering and command numbers). As a result, the processing of each controller can proceed simultaneously, independently, and asynchronously from all other controllers as long as real-time response can be assured. Assured real-time response is possible only when worst case response times are known for each process. The careful mapping of controllers to various processors can greatly influence the time it takes commands to travel down the hierarchy or sensed values and status to go up.

3.13 Execution order of controllers

RCS allows distributed multiprocessing of the controllers through the use of multiple buffering and handshaking by command numbers. However, when should one use additional processors and in what order should the controllers on a single processor be executed?

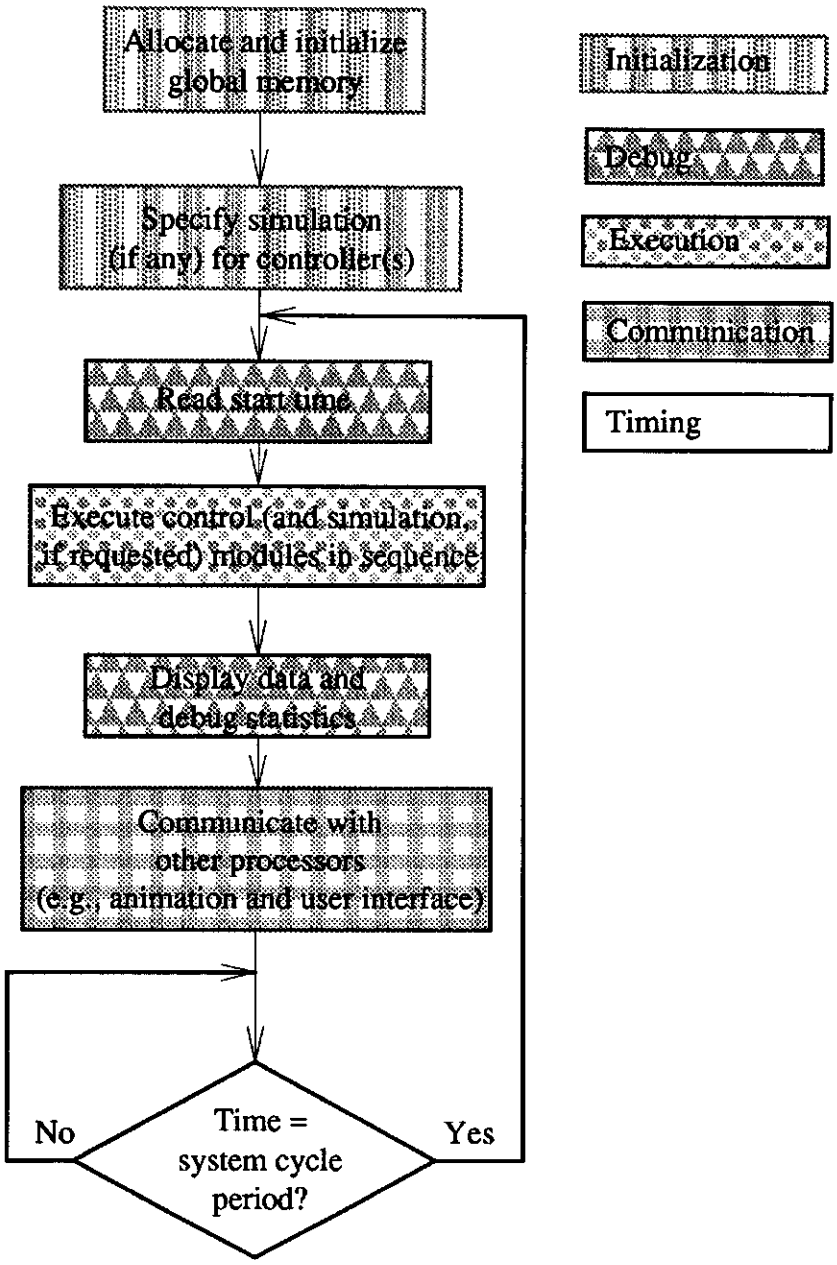


Figure 12: Main loop for cyclic execution of all controllers mapped to a single processor

Consider a control system simple enough so that all controllers and simulators can be executed on a single processor[5]. In figure 6 on page 12 it is shown that one can allow command response from the top of the control hierarchy to the bottom within one system cycle, in which case, the status response from bottom to top takes n system cycles, if there are n levels in that thread of the hierarchy. Alternately, one can allow status response from the bottom to the top

5. This is true in the current mining machine control implementation. However, we are now at the limit of processor performance and have added a second processor.

within one system cycle, in which case, the command response from top to bottom takes n system cycles. Trade-offs between command and status response can be accomplished based on the ordering of controller execution. Therefore, for a single processor, one can choose an order of controller execution that suits that system's needs for command and status response.

Multiple processors are required when the worst case execution time of all controllers exceeds maximum system cycle time. In this case, the several processors execute independently and asychronously.

In figure 7 on page 13 we show the current execution order for controllers in the CM control example. Two processors (PC#1 and PC#2) are used because the path planning controller was best executed in parallel while all the other controllers and simulators executed serially on another processor (PC#1). Note that the order of execution on PC#1 is optimized for command response.

3.14 Problem domain understanding critical

Most control system design methodologies stress the importance of problem domain understanding for effective control system design. RCS is no exception. However, because RCS stresses determinism (*i.e.*, all knowledge is explicitly entered into the system in the form of the finite state machines), domain knowledge is of paramount importance. The system designer creates a more effective control system to the degree the task and what is to be controlled are understood. For control of the continuous mining machine, we found that the process of RCS control system design forced a comprehensive understanding of the operation of the mining machine.

3.15 Generic processing pattern in all controllers

Within each controller, whether for control, simulation, or animation, a consistent pattern of processing is maintained. This processing currently involves the following sequential steps: pre-processing (including debug-related processing, sensory and world model processing, and planning), decision processing (including plan specific sensory processing, world model processing, planning, and decision making), and post-processing (including more debug-related processing, sensory and world model processing). This is illustrated in figure 13 on page 25.

In pre-processing, appropriate command and status values from supervisors and subordinates are copied into internal buffers in order to maintain data integrity. All inputs are collected and distilled into an appropriate format for use by the decision process. Sensor values are filtered and fused. World modelling functions like system state prediction based on past measurements is performed. Pre-processing is followed by decision processing which uses these distilled versions of the state of the system and the environment to decide on the action to perform in the current system cycle time. Any plan specific algorithms are executed. Appropriate commands are given to subordinate controllers. Finally, the post-processor writes appropriate command and status values from internal to external buffers in order to maintain data integrity. It is manifest that the use of generic controllers expedites the process of controller design. This total pattern makes the decision processing (the state tables) as english-like (*i.e.*, perspicuous) as possible.

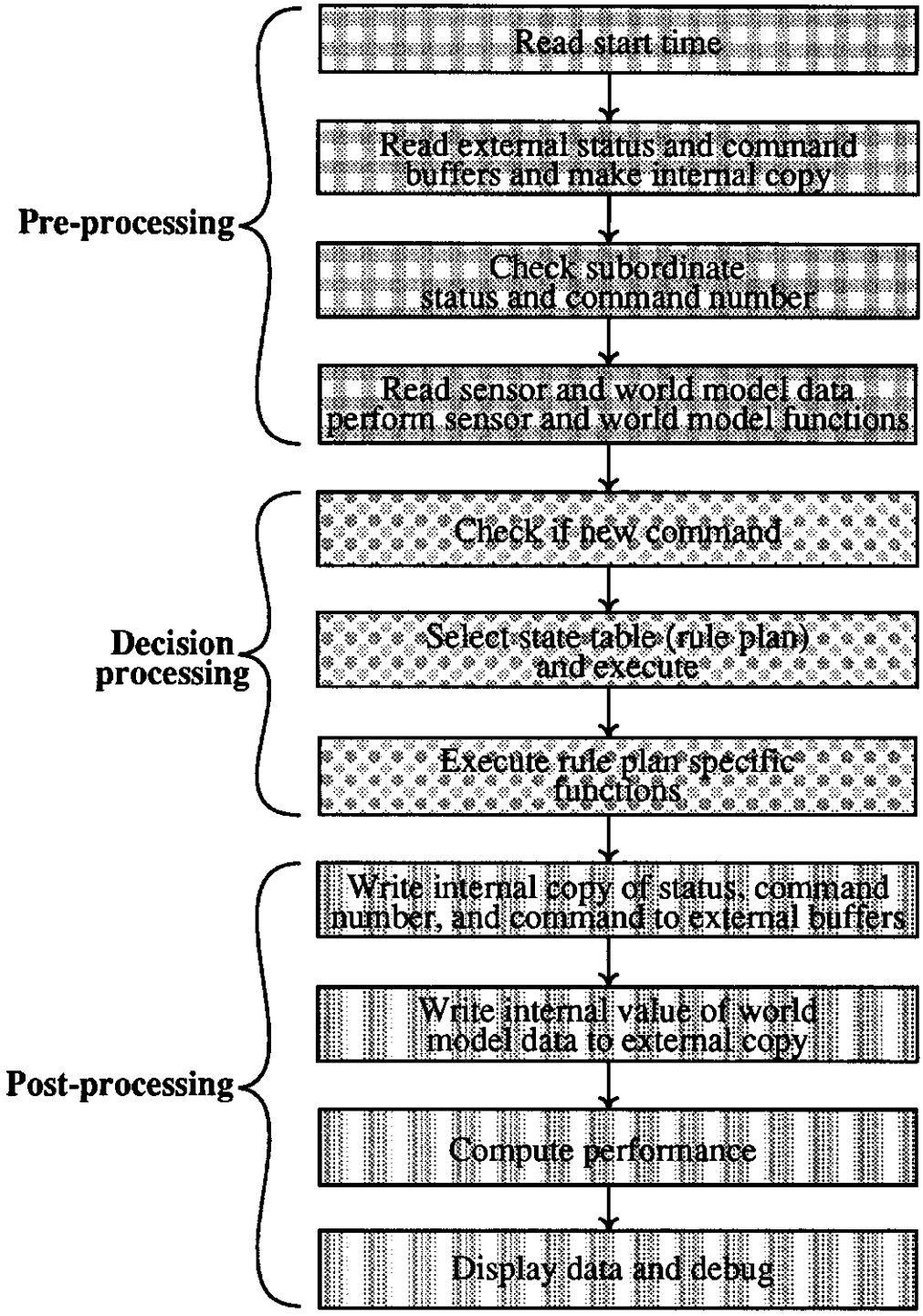


Figure 13: Generic processing pattern for all RCS controllers

3.16 Controllers small enough for human understandability

For the sake of understandability, the amount of decision logic in each controller is kept within certain limits. These limits can be defined by the user. If the design logic for a controller becomes too complex, under RCS, the controller can be broken up into two or more controllers.

3.17 Long and short range planning

How do long range processes like planners and computationally costly algorithms (*e.g.*, vision algorithms) fit into the system under the RCS design philosophy -- a philosophy which encourages cyclic, deterministic execution of controllers? RCS is an attempt to try to make motion planning a hybrid of "keep moving toward the goal in the next system cycle" and "plan the entire trajectory prior to execution." This is because the type of planning needed is highly application dependant. For example, navigating a mining machine, through a mine which contains independently moving objects like miners and other mining machines might require a "keep moving toward the goal in the next system cycle" approach, whereas, navigating a mining machine when the only obstacles are the pillars of coal might suggest a "plan the whole path first" type approach. Under RCS, it is suggested that the long range process be 1) divided into hierarchical pieces at various levels of abstraction and 2) build the software control of the long range processes so that these processes can be redirected at any given system cycle. Therefore, RCS allows a union of both long range and short range planning.

3.18 Tools

Effective software tools that are easy to understand are essential to any design methodology. A FORTH™ based programming tool called SMACRO and a set of generic software templates in the 'C' programming language[6] have been developed for RCS. In addition, we are currently developing CASE tools that will even more efficiently allow RCS style control system design. In this paper, we will primarily focus on the 'C' language templates, what they are and how are they used. In figure 14 we illustrate the history and proposed future for RCS tools development.

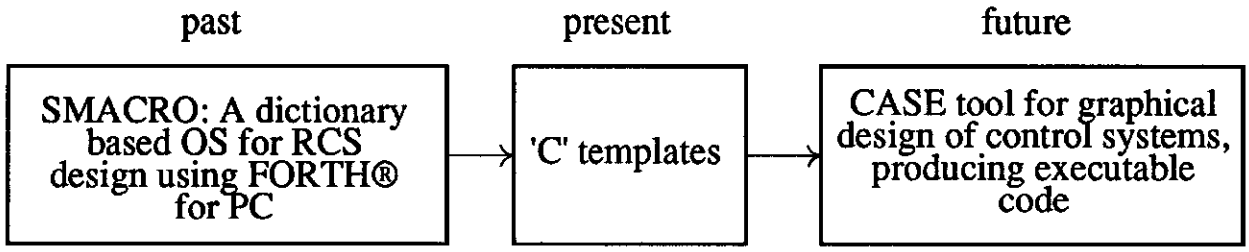


Figure 14: History of RCS tools development

6. These both are currently available only for PC compatible computers.

4 The hardware and software tools for RCS

We believe that the RCS methodology would be valuable to many industries, (*e.g.*, mining, construction, chemical process control, automotive assembly) and the development of software design tools for doing RCS designs will help disseminate the methodology. Therefore, we are currently developing such tools to better facilitate RCS design and implementation[1]. In addition, this implementation of RCS for coal mining machine control is in concert with our belief that control system design methodologies are best advanced by testing the method against real control problems. In this section, the current progress on such tool development is described.

Though we list hardware as a tool for RCS design and implementation, we stress that the software tools are the centerpiece and are designed to work on virtually any hardware platform. However, certain hardware configurations and constraints better suit the RCS design philosophy as it was outlined in section 3 on page 17. Just what these configurations and constraints are will be the topic of section 4.2 on page 31.

Software tools for RCS control system design are currently being written in the 'C' language in order to reach a wider audience of programmers. Software templates of various types have been and are being developed that allow the system designer to more easily design according to the method. These templates will be discussed in the software tools section.

4.1 Software tools

Essentially two approaches to tool development are being used: 1) a FORTH based environment called SMACRO and 2) a 'C' language based set of software 'templates.' The templates allow the user to write 'C' code easily for the particular application by making simple alterations to the existing template. Both approaches have been successfully used to accomplish design and implementation of control, simulation, and animation in the RCS style. One of the tasks of the current effort has been to convert existing control and simulation code from SMACRO into 'C.' Therefore, we will focus on describing the work done in 'C.'

We are in the process of developing CASE tools which support the user in much the same way as the aforementioned tools, but with a simpler graphical user interface. We intend to have the tool perform data management and code generation tasks while still allowing the designer to change source code as may be required.

4.1.1 Software tools for control

The 'C' language allows the programmer to break up large computing projects into distinct modular components called files. This is a natural modularization inherent in the language with the only restriction that a function named `main` be called in one and only one of the files that constitute the total computing project. Furthermore, in 'C,' the function called `main` calls all other functions defined and/or called in this and all other files.

RCS specifies that all control modules mapped to a single processor will be executed sequentially and

1. We also have made and are making efforts to examine existing CASE and other types of software design tools to see if they will fit well with RCS style system design. However, no satisfactory tools have been found.

cyclically. Therefore, `main` is the logical place where these modules will be called in sequence. As we've defined it, the file containing the function, `main`, is called `main.c`.

In order to be a transferrable tool set, it is important to separate functions and variables unique to a particular hardware/software platform from those functions and variables that are portable across various platforms. This is particularly important as we attempt RCS design on a greater variety of platforms.

4.1.1.1 Operation of `main.c`

Main.c essentially runs some setup and initializing routines and proceeds to execute sequentially each of the modules that are mapped to that particular CPU. In figure 12 on page 23 the operation of `main.c` is described.

During initialization, `main.c` allocates global and world model memory based on the structures defined in the header file, `global.h` (described in section 4.1.1.2). After that, `main.c` calls the following functions:

`main_init_global`: initializes global variables

`kbd_init_arrays`: loads in the information into arrays relating to the files command.dat and level.dat which associate mnemonics with numbers.

`timer_initialize_interrupt`: Setup up pc clock for 0.211ms

`draw_screen`: draws bkgnd for debug screen

`main_transmit_init`: causes INIT command to ripple through.

`read_write_BIT3`: control/simulation code on the PC that writes values (needed by the animation code) each cycle to memory common to the PC and the SGI.

`main_simulation_query`: allows designer to specify simulation at any part of the hierarchy; generally to speed up execution for debugging efficiency.

`read_write_comport`: allows simple serial communication with another PC (see figure 8 on page 14).

After initialization, `main.c` enters an infinite while loop[2]. The most important operation performed within this loop is to run each controller assigned to its CPU. However, if some modules were specified at any time to be simulated by the designer, these are also executed and control modules subordinate to these are skipped (since the simulator module simulates all its subordinates). In addition, the global and world model data structures have memory allocated to them during initialization. A few other functions are called within the loop, namely,

`kbd_evaluate_board_input`: if a key is stroked the input is processed by this function.

`timer_check_for_timer_pulse`: the infinite while loop concludes with this while loop that waits until this function returns TRUE.

The control system designer can specify the presence of simulation. All types of simulation used will be dealt with in section 4.1.2 on page 30.

---

2. Note that in RCS, infinite loops are allowed for processor cycling. However, they are not allowed within software modules (neither controllers, simulators, nor animators).

4.1.1.2 The templates for the generic software module: QQ1.c and QQ1.h

A goal for these software tools is to allow the designer simply to copy a generic file in the 'C' programming language and then "fill in the blanks" as required for that particular controller. QQ1.c is the generic file in 'C' that executes the following functions.

qq1_pre_process: all inputs are collected into an appropriate format for use by the decision process; reads appropriate command and status values from external to internal buffers

qq1_check_if_new_command: checks to see if the command serial number has incremented since the last cycle and if so the command the number accompanies is a new command.

qq1_decision_process: uses the distilled versions of the state of the system and the world to decide on the action to perform in the current system cycle time.

qq1_post_process: writes appropriate command and status values from internal to external buffers in order to continue to maintain data integrity.

The file QQ1.h is the header file that contains the generic data structures definitions (called QQ1_BUFFER) that contain the commands, status, and debugging information for each module. The debugging information consists of the execution mode (*e.g.*, single stepping, etc.) and some performance parameters like module execution times. QQ1.h for each module is included in the global.h header file which in turn is included in QQ1.c for each module. Global.h further defines a structure that is a collection of the structures QQ1_BUFFER for each module. Global.h also defines the structure that is a collection of the relevant data about the state of the world (called WORLD_DATA).

The characters 'QQ1' in QQ1.c and QQ1.h refer to the mnemonic describing that particular module (*e.g.*, in the coal mining implementation, the shift boss module was contained in the files sb.c and sb.h). One then can do a simple find and replace while converting the generic template to a particular software module.

4.1.1.3 Other important generic files

Timer.c contains functions needed in order to set up the computer clock in fine enough increments for the application. It contains the following functions:

timer_initialize_interrupt: called in main.c and sets to zero the timer_counter variable in the world model data structure.

timer_check_for_timer_pulse: called in main.c which checks if the cycle time is reached.

timer_restore_interrupt: restores the timer interrupt before terminating the interrupt

timer_interrupt: function called in response to interrupt from the processor and increments the variable, timer_counter, contained in the world model data structure.

Draw.c and dprintf.c contain functions needed in order to draw debugging information on the control and simulation computer monitor. These functions are highly hardware/software platform specific. Some implementation details will be discussed subsequently.

NonPortable.h contains definitions for hardware/software specific functions. It also makes similar variable definitions making explicit the lengths of type int, etc.

4.1.2 Software tools for simulation

Computing hardware continues to plummet in cost. Software development, though costly, is often less costly than system development using real equipment. Therefore, the use of simulation and animation can often reduce system development costs and time. We have identified several areas that might require simulation: 1) controller, 2) actuator, 3) sensor, and 4) environment. We will look at each of these types of simulation in this section.

It is expected that throughout and beyond the design cycle, the designer will want to switch back and forth between simulation and the actual system. Using environment simulation in the mining example, we might want to operate the actual mining machine above ground without obstacles expected in a mine, e.g., pillars and roof. Given this requirement, one would want an environment simulator, which might be a simple two dimensional static mine map. As an example of another type of simulation, one might want to test some new decision-making logic quickly without having to wait for the entire control system to execute. This can be accomplished by simulating controllers. Another example is that a sensor may not be working on a given day and one could design and use a sensor simulator which could allow experiments to continue. Similar things can be said about an actuator simulator.

We have specified in the generic `main.c` file that a user will be able to choose easily which entities will be simulated. We accomplished this by writing the code such that if a particular controller was chosen to be simulated, all its subordinate controllers would not be executed at run time.

4.1.2.1 Controller simulation

Each controller in the RCS hierarchy is a collection of specific tasks. When a large control system is running, tasks within control modules at the higher levels of the hierarchy can take minutes or hours to complete. For example, in figure 5 on page 11, the controller, shift boss, execution of a Cut-two-passes task can take hours. During the design and maintenance phases, one will want to change state tables and then test those changes at a faster than normal execution rates. Therefore, software modules that simulate controllers subordinate to the controller of interest are useful if they realistically accelerate the execution of each task. Another reason for having a controller simulator is that one may want to postpone the design of a certain thread of the hierarchy until later. For example, in figure 5 on page 11, we show the coal removal section simulated by a single controller simulator, although we have developed controllers for the entire coal removal section.

4.1.2.2 Actuator simulation

For actuator simulation we have provided a very simple template file which specifies the following:

1) instructs the designer to enter the necessary header files (`global.h` is assumed to be needed),

2) provides placeholder for entering external variables, for example, commands from controller supervisor and variables for feedback to the same controller and any dynamic parameters like speed,

3) contains directions to define, in `global.h`, any new world model variables defined in this simulator file,

4) directions instructing the designer to enter dynamic equations of motion with appropriate uncertainty (random error) of the actuator being simulated.

4.1.2.3 Sensor simulation

We have provided a very simple template file for sensor simulation which specifies the following:

1) instructs the designer to enter the necessary header files (`global.h` is assumed to be needed),

2) provides placeholder for entering external variables, for example, parameters from the appropriate actuator or actuator simulator module,

3) contains directions to define, in `global.h`, any new world model variables defined in this simulator file,

4) directions to enter code to read parameters from relevant actuator simulators and processes and couple these with appropriate sensor dynamics (e.g., lags) and noise.

4.1.2.4 Environment simulation

Currently, we provide no generic software template for environment simulation.

4.1.3 Software tools for animation

Strictly speaking, we provide no generic software templates for robot animation. However, the animation code that is used in the mining machine implementation can easily be converted to any other application, since much of the existing code is portable. Our graphics code is written using basic drawing routines that draw fundamental shapes such as cylinders, circles, and the like. This greatly simplifies and streamlines the graphics code. These basic drawing routines are useful for a variety of applications. The code specific to the mining machine implementation is described in section 2.6 on page 15.

4.2 Hardware tools

We have mentioned that the RCS philosophy discourages interrupts of modules (either controllers, simulators, or animators) in section 3.10 on page 21. Internal looping within modules is also avoided as described in section 3.11 on page 22. In addition, RCS specifies that all modules execute[3] within the response time required by the application (see section 3.10 on page 21). In a typical large scale control system design, it isn't long before one has created so many software modules that they cannot execute within a single system cycle on a single CPU. As discussed in section 3.13 on page 22, one must map some modules to one processor and some to another to allow parallel execution. However, we do not want to choose a communication setup between the two (or more) processors such that one processor waits an indeterminate amount of time to receive communication from the other processor. Therefore, as a standard testbed, a point-to-point communications link was chosen that allows asynchronous reading and writing of a specific block of shared memory.

3. Each module performing the following: read, decide, write

As seen in figure 9 on page 16, we have as a simple testbed for RCS development a PC, a Silicon Graphics IRIS™ (SGI), two point-to-point communication cards, and a common memory card.

The current usage of the point-to-point communications link is to allow orderly asynchronous writing of dual-port memory locations by the control and simulation modules on the PC, and reading of the same memory locations by the animation and operator interface code on the SGI. The control and simulation code on the PC write to the dual-port shared memory every system clock cycle and the animation and operator interface code on the SGI reads whenever it can. This latter case is because all the code on the SGI runs under the UNIX™ operating system and therefore can be interrupted by other software processes. We maintain that animation is primarily a diagnostic and demonstration tool and hasn't the real time requirements that the control and simulation code have. Therefore, a strict separation between control/simulation and animation/operator interface has been utilized in order to assure deterministic control and simulation.

5 References

[Aberdeen 91] Unpublished minutes of the *Workshops on Architecture for Real-Time Intelligent Control of Unmanned Vehicle Systems*, Aberdeen, MD, January 15-17, 1991, Workshop Chairman: Colonel Robert Harper, U.S. Army Missile Command, ATTN: AMSMI-RD-UG, Redstone Arsenal, AL 35898-3060.

[Albus 89] Albus, J. S., McCain, H.G., and Lumia, R., "NASA/NBS Standard Reference Model for Telerobotic Control System Architecture (NASREM)," NBS Technical Note 1235, National Institute of Standards and Technology, U.S. Department of Commerce, April 1989.

[Albus 91] Albus, J. S., "A Theory of Intelligent Machine Systems," *Proc. of IEEE Intelligent Robots & Systems 1991 -- Intelligence for Mechanical Systems*, Nov. 1991

[Albus 92] Albus, J. S., "RCS: A Reference Model Architecture for Intelligent Vehicle and Highway Systems," *ISATA 92*, Florence, Italy, June 1992.

[Brooks 86] Brooks, R. A., "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, March 1986.

[Coad 91] Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Yourdon Press, 1991.

[Fiala 87] Fiala, John C., Lumia, R., Albus, J. S., "Servo Level Algorithm for the NASREM Telerobot Control System Architecture," SPIE Vol. 851, Proceedings of SPIE 1987 Cambridge Symposium of Advanced Robotic Systems, Cambridge, Massachusetts, November 2, 1987.

[Horst 92] Horst, J. A., "An application of measurement error propagation theory to two measurement systems used to calculate the position and heading of a vehicle on a flat surface," *NISTIR* 90-4434, National Institute of Standards and Technology, U.S. Department of Commerce, 1992.

[Huang 92] Huang, H. and Hira, R., "Applying the NIST Real-Time Control System Reference Model to Submarine Automation: A Maneuvering System Demonstration," *NISTIR*, to be published, National Institute of Standards and Technology, U.S. Department of Commerce.

[Lyons 89] Lyons, D. M.and Arbib, M. A., "A Formal Model of Computation for Sensory-Based Robotics," *IEEE Transactions on Robotics and Automation*, Vol. 5, No. 3, June 1989.

[Nilsson 89] Nilsson, N., *Action Networks*, Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems, Tenenberg, J., *et al.* (eds.), University of Rochester, New York, 1989

[Quintero 92] Quintero, R. and Barbera, A. J., "An RCS Methodology for Developing Intelligent Control Systems," *NISTIR*, to be published, National Institute of Standards and Technology, U.S. Department of Commerce.

[Sammarco 92] Sammarco, J., Anderson, D., and Jobes, C., "Overview of the U. S. Bureau of Mines Research in Position and Heading Systems for Mine Machine Guidance," *SME Annual Meeting*, 1992.

[Schiffbauer 92] Schiffbauer, W. H., "Flexible control networks in mining machines to improve worker safety," *IEEE Control Systems Magazine*, June 1992.

[Simmons 90] Simmons, R., Lin, L., and Fedor, C., "Autonomous Task Control for Mobile Robots" *Fifth IEEE International Symposium on Intelligent Robotics*, Philadelphia, PA, Sept., 1990.

[Skillman 89], Skillman, T. L. and Hopping, K., "Dynamic Composition and Execution of Behaviors in a Hierarchical Control System," SPIE Mobile Robots IV, Philadelphia, Nov. 1989.

[Wavering 88] Wavering, A. J., "Manipulator Primitive Level Task Decomposition," NIST Technical Note 1256, September, 1988.

6 Appendices

A Overview of the underground coal mining environment

A typical underground coal mining environment is described for those readers less familiar with the common terminology.

Advance cut: The action that the mining machine takes to excavate a block of coal in a direction perpendicular to the coal face.

Brattice: A moveable system of curtains that provide temporary ventilation to the area immediately surrounding the coal face.

Coal face: The front of the coal seam where cutting operations occur

Crosscut: The action that the mining machine takes to excavate a block of coal in a direction parallel (or nearly parallel) to the coal face.

Cut: The action that the mining machine takes to excavate a block of coal. Usually the mining machine cuts in two passes to achieve a desired width of a cut (typically the width of an entry).

Entry: Region remaining following a series of advance cuts allowing 'entry' into the mine. See 'Room and Pillar mining'.

Haulage: As the CM is cutting the coal, some method must be employed to move the coal (which spills off the tail of the conveyor on the CM) from the CM to the main haulage system outby (see figure 1 on page 5).

Inby: In the direction of the coal face of the underground mine.

Outby: In the direction of the entrance to the underground mine.

Man Door: A door allowing human passage through permanent ventilation control wall.

Panel: A large block of coal (usually rectangular) to be extracted which is separated from the next panel by leaving a long rectangular pillar of unextracted coal between panels. The long unextracted pillar is a safety precaution to prevent the collapse of the coal roof over any more than one panel in the event of a cave in.

Pillar: Generally means a small (compared to panel) block of unextracted coal, 36 meters or less in length and 4.5 to 18 meters in width. As long as they are of sufficient size and separation from one another, pillars (along with roof bolts) help keep the roof of the mine from caving in (see figure 2 on page 6).

Roof: Seams of rock are typically located immediately above a coal seam. Once the coal is removed this rock becomes the roof of the mine. Certain methods are used to keep the roof from caving in. One is to control the size of remaining pillars in relation to widths of entries. Another is to use roof bolts.

Roof Bolter: This is a piece of equipment that uses various methods with the same goal; namely, to keep the roof from falling down on miners and equipment. It places roof bolts of various

kinds into the rock strata after the coal is removed from an entry. It's operation is closely coordinated in space and time with the continuous mining machine.

Room-and-Pillar Mining: A mining method that also features the development of main entries at both sides of a panel in preparation for long wall mining. Coal is extracted forming rooms with pillars left. Pillars may be extracted at a later stage in a retracting operation. Sizes of pillars and width of entries may vary depending on the roof support and the transportation support requirements. See figure 1 on page 5.

Shuttle Car: A vehicle specifically designed for transporting the coal mined by the CM from the CM to the main haulage.

Tramming: Motion of the CM as a whole along the floor of the mine.

Ventilation Regulator: A structure placed in an entry that controls airflow for ventilation.

B A continuous mining machine

Continuous mining machines (CM), as in figure 15 on page 36, are used in room-and-pillar mining We now describe key components of a particular CM, illustrated in figure 15, called the JOY14CM[1]:

A tramming subsystem: Tram motors driving each tread exist allowing forward and reverse motion as well as turns and pivots.

A cutter drum: A hydraulic actuated cutter boom extends out at the front of the machine. Attached to the front of the boom is an electrically operated cutter drum. Replaceable cutting bits are installed at the surface of the cutter drum which fracture the coal as the drum is pushed into the coal face while turning.

A gathering head subsystem: This subsystem is located at the bottom of the front end of the machine. The gathering pan can be set to float on the floor. The gathering head, using a rotary motion, scoops the coal inward onto the conveyor. The conveyor moves the coal to the rear of the machine.

A conveyor subsystem: The conveyor extends from the gathering head to the rear of the machine. An adjustable position conveyor boom forms the end of the conveyor system. It can move from right to left as well as up or down. Coal is dumped from the conveyor boom onto a haulage unit behind the CM.

A stabilization jack: This hydraulic jack provides a stabilizing force to counter-balance the cutting force.

The continuous mining machine has ten tram control commands: slow/fast speed forward, slow/fast speed reverse, pivot left/right, turn left/right forward, and turn left/right reverse. These are open-loop commands. Execution of any of these command can be terminated by either a stop command (implying the tram control loop is closed at a higher level where the sensory information is processed), or by a con-

1. Reference to product or company names is for identification only and does not imply government endorsement.

dition that some maximum time has expired (a safety time-out condition).

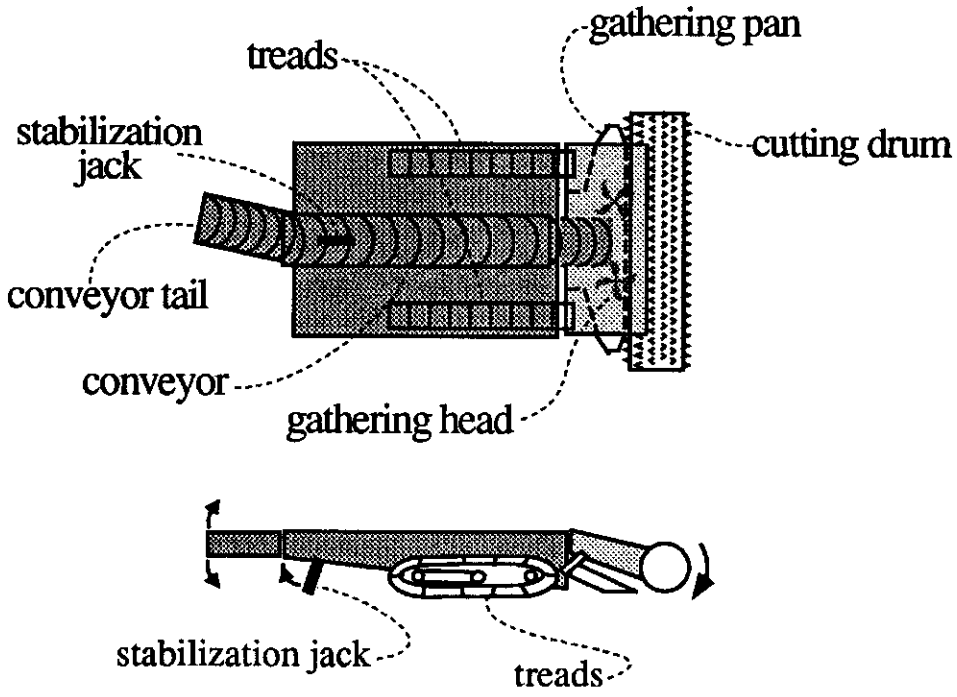


Figure 15: A continuous mining machine

The U. S. Bureau of Mines has been implementing a computer control system testbed [Schiffbauer 92]. This testbed is a distributed network linking the continuous mining machine, various sensor systems (length and angle measuring systems and a laser ring gyro [Horst 92] [Sammarco 92]), and an operator console which are all nodes on the network.

C State table in 'C' for sump-shear-cusp command

In this section we give the 'C' code for the state table logic as specified in figure 4 on page 9. This code uses an `if then else` code structure for the decision logic. Using a 'case' statement in 'C' is an alternate method for performing the state machine logic.

```
/* The following code is a function in 'C' contained in the controller named 'continuous miner operation
coordinator' (see figure 5 on page 11). It is an example of a command (co_sump_shear_cusp) from a su-
pervisory controller that is decomposed both spatially (to three subordinate controllers) and temporally
(sump, shear, and cusp commands, in order, to the subordinate controllers) */
static void co_sump_shear_cusp(void)
   {
```

```
if (co_cur_state == S0) /* If this is a new command */
{ /* do a sump operation */
co_cur_state = S1;
co_position_for_sump();
 cr_co.command = CR_SUMP;
 cr_co.command_num++;
 vg_co.command = VG_TRAM_TO_XY;
 vg_co.command_num++;
 cc_co.command = CC_SUMP;
 cc_co.command_num++;
dprintf(CO_DEBUG_LINE,19,"busy     ");
}

else if (
(co_cur_state    == S1)&&
  (vg_si.status == VG_DONE)&&
  (cc_si.status == CC_DONE))
{ /* do a shear operation */
co_cur_state = S2;
 cr_co.command = CR_SHEAR;
 cr_co.command_num++;
 vg_co.command = VG_HALT;
 vg_co.command_num++;
 cc_co.command = CC_SHEAR;
 cc_co.command_num++;
} /* check for the error that the CM popped out of the shear*/
else if (
(co_cur_state == S2)&&
  (W->popped_out_of_shear))
{
co_cur_state = S1;
co_position_for_sump();
 cc_co.command = CC_SHEAR_CORRECT;
 cc_co.command_num++;
} /* check for the error that the operator hasn't positioned the conveyor tail over the haulage, in which
case too much coal is built up and one must pause the shear operation */
else if (
(co_cur_state == S2) &&
too_much_loose_coal())
{
 cc_co.command = CC_HALT;
 cc_co.command_num++;
}
```

```
else if (
(co_cur_state == S2) &&
!too_much_loose_coal() &&
(cc_co.command == CC_HALT))
{ /* continue with shear*/
 cc_co.command = CC_SHEAR;
 cc_co.command_num++;
}
else if (
(co_cur_state    == S2)     &&
  (cr_si.status == CR_DONE)  &&
  (vg_si.status == VG_DONE)&&
  (cc_si.status == CC_DONE)&&
  (!W->popped_out_of_shear))
{ /* do a cusp operation */
co_cur_state = S3;
 cr_co.command = CR_CUSP;
 cr_co.command_num++;
 cc_co.command = CC_CUSP;
 cc_co.command_num++;
}
else if (
(co_cur_state    == S3)     &&
  (cr_si.status == CR_DONE)  &&
  (vg_si.status == VG_DONE)&&
  (cc_si.status == CC_DONE))
{
co_cur_state = S4;
co_position_for_cusp();
 vg_co.command = VG_TRAM_TO_XY;
 vg_co.command_num++;
}
else if (
(co_cur_state    == S4)     &&
  (cr_si.status == CR_DONE)  &&
  (vg_si.status == VG_DONE)&&
  (cc_si.status == CC_DONE))
{
co_cur_state = NOP;
co.so.status = CO_DONE;
dprintf(CO_DEBUG_LINE,19,"         ");
```

D Generic controller template in 'C'

In this section we give the 'C' code for generic controller template, QQ1.c, and associated header file, QQ1.h. The general ordering of the generic controller template is illustrated in figure 13 on page 25.

D.1 Generic controller template in 'C'

```
/*
* New File - QQ1.c
FILE-DESCRIPTION: QQ1.C is a generic version of a controller module and is intended to be used as a
template. The function qq1_controller is called from the main routine within the main event loop. Each
time it is called it calls a pre_process routine, a check_if_new_command routine, a decision process rou-
tine, and a post-processing routine. All control modules should have at least these. The state tables are the
primary means of making the module unique. The appropriate state table function is called from the deci-
sion process.

INSTRUCTIONS-FOR-PROJECT-CREATORS: The first step in creating the control units is copying this
file to the name of that control unit. Then a pair of initials should be chosen for the module and all qq1's
should be replaced with those initials and all QQ1's should be replaced with the initials in capitals. If the
module is to have only one subordinate the qq2's and QQ2's can simply be replaced with the subordinate's
initials. If there will be more than one subordinate the lines that deal with subordinates need to be replicat-
ed in each position and the initials for each subordinate replacing the qq2's in one line of each set. These
lines include:
 1. declaring the command out and status in for the subordinate
 2. copying buffers in the pre and post processing routines
 3. sending the init and halt commands down in the state tables.
Furthermore, one will have to add state_tables to a controller. This can be done by adding the function pro-
totype, adding a case statement and copying one of the supplied state tables (init or halt) and renaming it.
Then it can be altered to perform the needed task.
*/

/*
* Include Files
*/
#include "global.h"    /* global definitions */

/*
* Definitions and Macros
*/
#define QQ1_DEBUG_LINE     1      /* line used for debug screen */

/*
* Variables internal to this file, but
* remaining in tact between calls to
```

```
* functions within this file using these variables.
*/
static unsigned qq1_cycle_start_time;

static STATE qq1_cur_state = S0;
static QQ1_BUFFER       qq1;    /* command from above */
static QQ2_COMMAND      qq2_co; /* command to subordinate */
static QQ2_STATUS       qq2_si; /* reply from subordinate */
/* End private global variables (this comment needed for create level) */

/*
* Function Prototypes
*/
void qq1_controller();

static void qq1_decision_process    (void);
static void qq1_init                (void);
static void qq1_halt                (void);
static void qq1_go                  (void);
static void qq1_pre_process         (void);
static void qq1_post_process        (void);
static void qq1_check_if_new_command (void);
static void qq1_print_in_data       (void);
static void qq1_print_out_data      (void);

/*
* Functions not defined but used within this source file
*/
extern int dprintf(int x, int y,char *fmt, ...);

/*QQ*****************************************************************
RTNS:  None.
PRPSE: External Routine used by main to execute this task.
This is the only entrance to this task.
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************/
void qq1_controller()
{
```

```
qq1_pre_process();
qq1_check_if_new_command();
qq1_decision_process();
qq1_post_process();
}

/*QQ*****************************************************************
RTNS:  None.
PRPSE: This routine's function is to route the control to a specific
 state table based on the command in as well as other inputs and
 sensor data.
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************/
static void qq1_decision_process(void)
{
/*
* If run flag in not set return
*/
if (qq1.mode.dont_run == TRUE) return;

/*
* If single step flag in set but single step number has not changed
*    return
*/
if (
(qq1.mode.single_step == TRUE) &&
(qq1.mode.single_step_num == qq1.perfo.single_step_num))
return;

/*
* If single step flag in set and single step number has changed
*    cycle once.
*/
if (
(qq1.mode.single_step == TRUE) &&
(qq1.mode.single_step_num != qq1.perfo.single_step_num))
qq1.perfo.single_step_num = qq1.mode.single_step_num;
```

```
switch (qq1.ci.command)
{
case QQ1_INIT:
dprintf(QQ1_DEBUG_LINE,8,"init     ");
qq1_init();
break;
case QQ1_HALT:
dprintf(QQ1_DEBUG_LINE,8,"halt     ");
qq1_halt();
break;
default:
dprintf(QQ1_DEBUG_LINE,8,"INVALID  ");
break;
}
}

/*QQ*****************************************************************************
RTNS:  None.
PRPSE: This is the state table for the init command of this level.
ATHR:  David G. Quigley
CRTD:  10/14/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************/
static void qq1_init(void)
{
if
(qq1_cur_state == S0)
{
qq1_cur_state = S1;

qq2_co.command = QQ2_INIT;
qq2_co.command_num++;
dprintf(QQ1_DEBUG_LINE,19,"busy     ");
}

else if (
(qq1_cur_state    == S1)     &&
  (qq2_si.status == QQ2_DONE))
{
qq1_cur_state = NOP;
```

```
qq1.so.status = QQ1_DONE;
dprintf(QQ1_DEBUG_LINE,19,"        ");
}
}

/*QQ*****************************************************************
RTNS:  None.
PRPSE: This is the state table to halt all actions or movement controlled
 by this level.  It also passes the halt command down to all of
 its children.
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************/
static void qq1_halt(void)
{
if
(qq1_cur_state == S0)
{
qq1_cur_state = S1;

 qq2_co.command = QQ2_HALT;
 qq2_co.command_num++;
dprintf(QQ1_DEBUG_LINE,19,"busy     ");
}

else if (
(qq1_cur_state    == S1)     &&
  (qq2_si.status == QQ2_DONE))
{
qq1_cur_state = NOP;
qq1.so.status = QQ1_DONE;
dprintf(QQ1_DEBUG_LINE,19,"        ");
}
}

/*QQ*****************************************************************
RTNS:  None.
PRPSE: This routine handles all of the preprocessing this level.  It
 includes:  Reading the global buffers,
```

```
Checking executing status of children;
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**********************************************************************/
static void qq1_pre_process(void)
{
dprintf(QQ1_DEBUG_LINE,1,"QQ1");

/*
* Start cycle timing
*/
qq1_cycle_start_time = W->timer_counter;

/*
* Get incoming commands from superior module.
*/
COPY_BUFFER(&qq1,&(G->qq1_buf),sizeof(QQ1_BUFFER));

/*
* Get status back from subordinates.
*/
COPY_BUFFER(&qq2_si,&(G->qq2_buf.so),sizeof(QQ2_STATUS));
COPY_BUFFER(&qq2_co,&(G->qq2_buf.ci),sizeof(QQ2_COMMAND));
/* End get subordinate status (this comment need for create level) */

/*
* If the subordinate status back echo number does not
* match its command number, assume it is still executing.
*/
if (qq2_si.status_num != qq2_co.command_num)
qq2_si.status = QQ2_EXECUTING;

/* End check subordinate status (this comment needed for create level) */

/*
* Run print in data
*/
qq1_print_in_data();
}
```

```
/*QQ*****************************************************************
RTNS:  None.
PRPSE: This routine handles the writing of all global data as well as
 any other required post processing.
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************/
static void qq1_post_process(void)
{
/*
* Calculate performance information
*/
qq1.perfo.last_cycle_time = (W->timer_counter - qq1_cycle_start_time) * 21;
if (qq1.perfo.min_cycle_time == 0) qq1.perfo.min_cycle_time = 0xFFFF;
if (qq1.perfo.last_cycle_time > qq1.perfo.max_cycle_time)
qq1.perfo.max_cycle_time = qq1.perfo.last_cycle_time;
if (qq1.perfo.last_cycle_time < qq1.perfo.min_cycle_time)
qq1.perfo.min_cycle_time = qq1.perfo.last_cycle_time;

/*
* Write status back to superior module.
*/
COPY_BUFFER(&(G->qq1_buf.so),&qq1.so,sizeof(QQ1_STATUS));
COPY_BUFFER(&(G->qq1_buf.perfo),&qq1.perfo,sizeof(QQ1_PERFORMANCE));

/*
* Write commands to subordinates.
*/
COPY_BUFFER(&(G->qq2_buf.ci),&qq2_co,sizeof(QQ2_COMMAND));
/* End write subordinate commands (this comment needed for create level) */

/*
* Run print out data
*/
qq1_print_out_data();
}

/*QQ*****************************************************************
```

```
RTNS:  None.
PRPSE: This routine checks to see if the command receieved from above is
 a new command.  It checks this by comparing the current command
 in number with the last command in number it received (stored
 in status out number).
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************/
static void qq1_check_if_new_command(void)
{
if (qq1.ci.command_num != qq1.so.status_num)
{
dprintf(QQ1_DEBUG_LINE,5,"NC");
qq1.so.status_num = qq1.ci.command_num;
qq1_cur_state = S0;
qq1.so.status = QQ1_EXECUTING;
}
else dprintf(QQ1_DEBUG_LINE,5,"  ");
}

/*QQ**************************************************************************
RTNS:  None.
PRPSE: This routine prints the command and command number received this
 cycle to the screen if the debug for this level is active.
ATHR:  David G. Quigley, John Horst, Will Shackleford
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************/
static void qq1_print_in_data(void)
{
dprintf(QQ1_DEBUG_LINE,35,"%5.5u",qq1.ci.command_num);
dprintf(QQ1_DEBUG_LINE,30,"%4.4d",qq1.ci.command);

dprintf(QQ1_DEBUG_LINE+20,1,"QQ1");
if (qq1.mode.dont_run == TRUE) dprintf(QQ1_DEBUG_LINE+20,10,"STOP");
else                           dprintf(QQ1_DEBUG_LINE+20,10,"RUN ");
```

```
if (qq1.mode.single_step == TRUE) dprintf(QQ1_DEBUG_LINE+20,16,"SINGLE");
else                              dprintf(QQ1_DEBUG_LINE+20,16,"AUTO  ");
if (qq1.mode.simulate == TRUE) dprintf(QQ1_DEBUG_LINE+20,23,"SIMU");
else                           dprintf(QQ1_DEBUG_LINE+20,23,"REAL");
}

 /*QQ*****************************************************************
**
RTNS:  None.
PRPSE:This routine prints the status, status number, and state matched
for this cycle to the screen if the debug for this level is active.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*********************************************************************
*/
static void qq1_print_out_data(void)
{
dprintf(QQ1_DEBUG_LINE,48,"%5.5u",qq1.so.status_num);
dprintf(QQ1_DEBUG_LINE,43,"%4.4d",qq1.so.status);
if (qq1_cur_state == NOP)
{
dprintf(QQ1_DEBUG_LINE,55,"NOP  ",qq1_cur_state);
}
else
{
dprintf(QQ1_DEBUG_LINE,55,"S%d  ",qq1_cur_state);
}
if ((qq1.mode.dont_run == TRUE) ||
(qq1.mode.single_step == TRUE))
{
dprintf(QQ1_DEBUG_LINE+20,35,"                                  ");
}
else
{
dprintf(QQ1_DEBUG_LINE+20,35,"%5.5u  %5.5u  %5.5u",
qq1.perfo.last_cycle_time,
qq1.perfo.min_cycle_time,
qq1.perfo.max_cycle_time);
}
```

```
}
/* Extra Support Functions */

/*
```

D.2 Generic controller header file template in 'C'

```
/*
* New File: QQ1.h
*/

/*
* QQ1 buffer definitions:
*/
enum qq1_commands {             /* qq1 commands */
QQ1_INIT = 100,              /* initialize */
QQ1_HALT                    /* pause */
};

enum qq1_responses {          /* qq1 responses */
QQ1_NOT_READY = 0,              /* not initialized */
QQ1_EXECUTING,                /* command executing */
QQ1_DONE,                   /* command done */
QQ1_ERROR                   /* error occured */
};

/*
* Module buffer interface structures:
*/
typedef struct
{                   /* qq1 command structure */
unsigned              command_num;
enum qq1_commands       command;
} QQ1_COMMAND;

typedef struct
{                   /* qq1 response structure */
unsigned              status_num;
enum qq1_responses      status;
unsigned              error_num;
} QQ1_STATUS;

typedef struct
{                   /* qq1 mode structure */
boolean               dont_run;
boolean               single_step;
boolean               simulate;
unsigned              single_step_num;
```

```
} QQ1_MODE;

typedef struct
{               /* qq1 performance */
unsigned        last_cycle_time;
unsigned        min_cycle_time;
unsigned        max_cycle_time;
unsigned        single_step_num;
} QQ1_PERFORMANCE;

typedef struct
{               /* qq1 command-response buffer */
QQ1_COMMAND         ci;
QQ1_STATUS          so;
QQ1_MODE            mode;
QQ1_PERFORMANCE     perfo;
} QQ1_BUFFER;
```