# A CONTROL STRUCTURE FOR MULTI-TASKING WORKSTATIONS

**Richard J. Norcross**

Reprinted from PROCEEDINGS OF THE 1988 IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION,
Philadelphia, Pennsylvania, April 24-29, 1988

# A Control Structure for Multi-Tasking Workstations

Richard J. Norcross

Robot Systems Division
National Bureau of Standards
Gaithersburg, MD 20899

## Abstract

Manufacturing control modules, which are based on hierarchical control theory, decompose commands from a supervisory controller into elementary tasks to be performed by subordinate systems. The ability to simultaneously manage coordinated and independent functions of subordinates, while also processing new commands from the supervisory controller, is beneficial in advanced implementations of these controllers. This paper describes a control structure, based on computer operating system principles, which provides the desired capabilities. Utilizing concurrent processing and coordinating tasks via resource allocation provides extensive modularity which simplifies integration, gives a multi-processing environment, and produces the aforementioned capabilities.

## Introduction

The Automated Manufacturing Research Facility (AMRF) at the National Bureau of Standards utilizes a five layer control hierarchy to manage production within an automated factory [1,2]. Although the upper levels have not been implemented, the AMRF operates through the hierarchy as a small batch machine shop with six workstations. The Cleaning and Deburring Workstation (CDWS) is the newest workstation at the AMRF and represents a significant departure from other workstations. While most workstations in the AMRF use a single robot or AGV to assist autonomous machines in performing a single function, the CDWS (figure 1) incorporates two robots to assist and perform three distinct operations: deburring, washing, and buffing[3]. The robots can work cooperatively, independently, or in some combination of the two. There are also tasks which either robot can perform and for which there is no preference prior to execution. The workstation's tray stations provide a simple parts buffer and enables the workstation to act on new commands while performing the aforementioned tasks. These distinctions require the CDWS workstation controller to have significant flexibility.

A manufacturing control module within a hierarchical control scheme decomposes input tasks into subtasks, assigns subtasks and allocates resources to subordinates, analyzes subordinate feedback, initiates appropriate corrective actions, relays status to the supervisory controller, and is highly interactive and easily extensible [4,5]. The CDWS requires its controller to have additional capabilities. First, to utilize the input buffer, the controller must process multiple independent commands from the supervisor. Second, to utilize the redundant components (the two robots), the controller must incorporate alternatives in executing tasks. Finally to maximize the throughput of the different workstation functions the controller must blend independent and coordinated tasks.

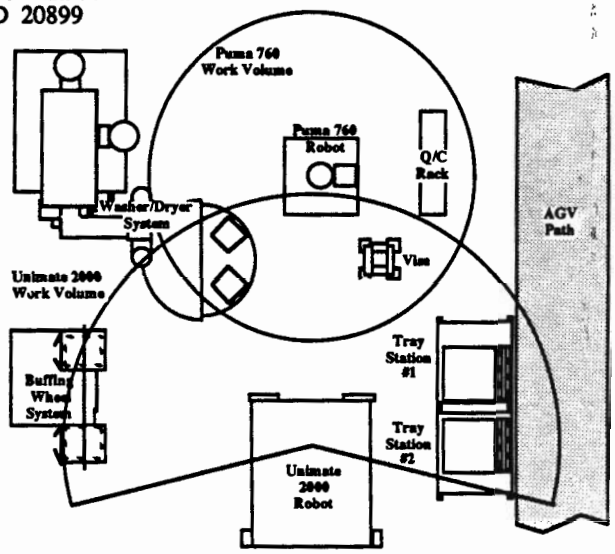


Figure 1. Cleaning and Deburring Workstation Layout

The AMRF's controllers are designed to be general purpose and data-driven. The controllers use two forms of data: task specific data (e.g., a machining lot's contents) and instructions on how to perform the task (i.e., programming). Thus, a controller provides the environment which executes the programs with the task specific data. Hereafter this environment is referred to as the *execution engine*. Although implemented through various coding schemes, the workstations in the AMRF generally use a Finite State Machine (FSM) execution engine with task instructions represented in either state tables or "process plans". The state table representation provides enormous flexibility, but a combinational explosion results when handling multiple commands. Conversely, the "process plan" representation readily handles multiple commands, but system changes require substantial modifications to the execution engine.

In computers, the operating system provides a multi-processing environment for the computer's tasks. Duplication of basic principles in a controller's execution engine produces a multi-tasking controller. Furthermore, by combining portions of state tables and process plans, the controller has the maintainability and flexibility desired. Thus, the CDWS's controller combines the desirable aspects of the controllers in the AMRF with an execution engine based on a computer operating system.

This work is partially supported by funding from the Navy MANTECH Program and was prepared by U.S. Government employees as part of their official duties and is therefore a work of the U. S. Government and not subject to copyright. Equipment listings in this paper do not imply a recommendation by NBS nor that the equipment is the best for the purpose.

## Execution Engine

"Generic" controllers have an execution engine which runs the system's programs. The engine does not reflect the particular activities of a system as much as its capabilities, i.e., the execution engine provides the environment for executing the programming which defines the system's tasks. Similarly, a computer's operating system supplies the environment for the execution of programs[6,7]. The kernel provides process management, synchronization, communications, and device management functions of an operating system[6]. The control structure described in this paper uses basic computer operating systems principles to implement its execution engine similar to the kernel of a small operating system. Modeling a controller on an operating system kernel results in a controller with multi-processing capabilities and highly modular programming.

Process management in a kernel includes the creation, manipulation, suspension, resumption, and termination of processes[7]. Following the operating system concept of a hierarchical process structure, the CDWS's control structure creates tasks hierarchically. The processes form a directed rooted tree whose nodes correspond to processes with one parent and zero or more children. A process definition (the programming) specifies the precedence relationships which direct the creation of the child processes.

The control system creates a new process (task) for every executed step in the function definition. "Creation" means the system generates a *job control block* (JCB) for the task and adds that block to an active queue. The JCB is a data structure which defines the current state of the task. Like computer operating systems, this information includes the task's process relationships, status, and resource allocations.

The active queue is a list of tasks waiting for an opportunity to execute. Only a portion of the tasks are in the active queue. The system suspends (i.e., removes from the active queue) all tasks which are blocked by a child task or resource allocation conflict. Tasks resume execution when the resource conflict is resolved, a child task completes, or when directed by a separate task. Tasks execute concurrently with their parent and no task may execute unless it is a member of the task hierarchy.

A job control block's process relationships include pointers to the task's parent, to it's offspring, and to a hierarchical data structure which parallels the hierarchical process structure. The child JCB's data.pointer points to the parent's data list when spawned. The parent's entries, made before spawning the child, are shared by the parent, the child, and by the child's children. Since the data is shared the hierarchical data structure provides inter-process communications and some synchronization between the various tasks of a sub-tree.

While inter-process communication provides some synchronization, the primary mechanism for task coordination is resource allocation. The system's programs (the decomposition plans) contain a list of required resources. The system allocates resources to a task if the resource is free or held by the task's parent. If the resource cannot be allocated, the task is blocked and the system adds the task to a waiting list for that resource. When the resource becomes available, the system returns the waiting tasks to the active queue and the tasks compete for the resource. The first task, whose resources are all available, becomes active while the others return to the waiting list. In the current implementation, in the CDWS, starvation and deadlocks are not possible so the active queue and wait lists are combined. However, the potential for starvation and deadlocks could exist in other applications and proven solutions from operating systems should be successful.

Since system coordination is based on device management, programming the system is highly modular. The programming of conflicting tasks may address their resource requirements individually, greatly simplifying programming. The use of modular components in decompositions also simplifies on-line planning. The advantage of this execution engine is that execution of separate tasks proceed independently while intertwined tasks remain coordinated. The engine is easily extensible and may incorporate other operating systems solutions as problems arise. Most importantly, this execution engine fulfills the requirements of a multi-tasking workstation with redundant components.

## System Programming

The execution engine functions as a simple operating system. Thus, it does not directly control the workstation but provides an environment for the execution of programs. These programs specify how commands decompose into instructions to subordinate systems. A command from the supervisor forms a task which decomposes into multiple sub-tasks. These tasks decompose further until the decomposition forms a command for a subordinate controller. At each level, the decomposition is guided by a program, called a *decomposition plan*.

### Decomposition Plan Format

Decomposition plans contain three groups of data: parameters, resources, and steps. This follows the AMRF Process Planning Format's parameters, requirements, and procedure sections [8], however, differences exist in the use and form of each section. Particularly, the steps contain alternatives and prerequisite functions similar to state tables.

Name and value pairs define parameters. The name is the entry into the hierarchical data structure and the value is the default value for that parameter. When spawned, a task adds those of its parameters which are not already members of its branch of the data structure to the data structure. If the decomposition call does not reference a given parameter and that parameter is not already a member of the data list then the default value is used.

Resources are essentially scheduling and coordination flags. They form the control system's primary method to synchronize tasks. The resource list contains the hardware and software items the plan needs for decomposition. Alternative resource sets are listed when applicable. Tasks do not decompose until all resources, from one alternative set, are allocated and no resources are allocated unless all are available.

The execution steps are the heart of the decomposition plan. Each step contains predicate and action elements. The predicate elements test the state of the system and are a list of prerequisite steps and system query functions. Through the predicate elements, process relationships are constructed. The action elements are commands with arguments. A command is either a function call or a plan to decompose.

A plan's execution is a review of the decomposition steps. This review consists of comparing the steps' prerequisites to the system's current state. The system must not change during the review. Thus, the review is a critical section and must be kept short. This execution and programming format avoids interrupt requirements and simplifies implementation, maintenance, and troubleshooting. Thus the system segments tasks and permits multi-task decomposition and control.

### On-line Planning

On-line planning avoids the requirement to maintain several similar decomposition plans. Based on the system's modularity, functions can be written which produce decomposition plans which decompose to existing plans. The planning approach should be reasonably quick so as to not impede the controller. Also planning functions may be implemented on a separate processor.

### Error Recovery

The hierarchical task structure also simplifies automatic error recovery. Recovery from an error has four steps: isolation, investigation, correction, and continuation. While the investigation and correction are not time critical, the error must be isolated quickly to avoid being compounded. Pruning the decomposition tree automatically isolates the resources affected by the error since the controller would issue no commands affecting the resources held in the pruned sub-tree.

Error recovery programs consist of functions which replace the command in an ancestor's JCB with one which performs the recovery. This technique necessarily requires the recovery function to know the decomposition hierarchy. However, this is not a loss of generality since investigation for recovery requires the determination of the original intent. A very robust procedure is required if the "prune and replace" procedure reaches the root of the hierarchy (orderly shutdown and request for help).

## Example

Consider moving a workpiece from the VISE to the WASHER in the CDWS (fig 1). Based on the decomposition plan in figure 2, the command is "**move-part P52 WASHER**". "$$part" and "$$goal" are assigned the values "P52" and "WASHER" respectively. Part P52's current position and description are in the world model and thus not specified in the command. Although a robot will be required to move the workpiece, the decomposition of **move-part** does not require any resources.

**name:** **move-part**
**parameters:** ($$part nil) ($$goal nil)
**resources:** nil
**steps:**

| # | predicates | action | |
|---|---|---|---|
| 1 | (0) (set-data $$move-plan) | create-movement-plan | |
| 2 | (1) | execute | $$move-plan |
| 3 | (2) (=? $$goal (location $$part)) | remove-plan | $$move-plan |
| | (2) (reset 1 2) | nop | |
| 4 | (3) | report | |

Figure 2. Sample Decomposition Plan Definition

Steps 1 and 2 above form a sequential process construct. The first step calls a routine which develops a plan for moving the part, then Step 2 executes that plan. Since the "$$move-plan" data entry is created prior to creating the plan, all of the task's offspring have access to the movement plan's name. Step 3 is a conditional loop. If the movement plan leaves the part at its ultimate goal, the plan is removed from the plan table and the loop is exited. However if the ultimate goal was not reached (i.e., part taken to a buffer), steps 1, 2, and 3 are re-executed.

Figure 3 shows the plan created in **move-part**. Since there may be several **move-part**'s executing in the system, the name is any unique symbol. There are three alternatives of resources which reflect the alternatives in the "xor" construct in Step #1. In the first alternative, the Unimate 2000 moves the workpiece via the given sequence to the WASHER ("2" is the WASHER).

**name:** **g#00654**
**parameters:** nil
**resources:** (VISE C2000 WASHER)
(VISE C760 WASHER)
(VISE C2000 TRAY11)
**steps:**

| # | predicates | action | |
|---|---|---|---|
| 1 | (0) (resource? C2000 WASHER) | C2000-move-part | (546 553 623 2) |
| | (0) (resource? C760) | C760-move-part | (546 876 562 2) |
| | (0) | C2000-move-part | (546 553 623 11) |
| 2 | (1) | report | |

Figure 3. On-line Generated Decomposition Definition

The second alternative commands the Puma 760 to do the same via a different sequence; while the third option sends the part to temporary storage.

Step 4 of figure 2 and step 2 above, signal completion of the decompositions. Upon completion the system terminates the task by releasing the resources, removing the JCB, and returning the parent task to the active queue.

If an error occurs during the part movement, the resources held by **g#00654** define the affected area of the workstation. Unrelated activity would not be affected by the failure. The command which replaces **g#00654** specifies the recovery and may effect more areas, but would wait for them to be available.

## Summary

This paper outlines a controller structure based on computer operating system principles. Specifically, the structure uses Job Control Blocks, an active queue, critical sections, hierarchical task structure, inter-process communications, and resource allocation to implement an execution engine which segments tasks and provides for control of multiple independent tasks and coordination of multiple actors. The advantages are explained in terms of modular programming which improves flexibility, on-line planning, and error recovery. The principles used in this controller are applicable to any workstation controller which performs multiple functions, utilizes on-line planning, or implements redundant components. These activities will become more important as designers improve workstation utilization rates and reliability.

## References

1 J.A. Simpson, R.J. Hocken, and J.S. Albus, "The Automated Manufacturing Research Facility of the National Bureau of Standards", Journal of Manufacturing Systems, Vol. 1(1) pp17-32, 1982.

2 J.S. Albus, A.J. Barbara, R.N. Nagel, "Theory and Practice of Hierarchical Control", Proc 23rd IEEE Computer Society International Conference, Sept. 1981.

3 H. McCain, R.D. Kilmer, K.N. Murphy, "Development of a Cleaning and Deburring Workstation for the AMRF", Proc DSC '85, 2:26-43.

4 A.T. Jones, C.R. McLean, "A Production Control Module for the AMRF", Proc of ASME Computers in Engineering, Aug 1985.

5 H. Scott, K. Strouse, "Workstation Control in a Computer Integrated Manufacturing System", Proc Autofact 6, Oct 1984.

6 J. Peterson, A. Silberschatz, Operating System Concepts, Addison-Wesley Publishing Co. Inc., Reading, MA, 1983.

7 H. M. Deitel, An Introduction to Operating Systems, Addison-Wesley Publishing Co. Inc., Reading, MA, 1984.

8 D. Gordon, "Process Plan Flat File Format", AMRF-Internal Report, NBS, March 1986.