REPORT AD-A258 866

Form Approved
OMB No 0704-0188

1

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (*Leave blank*) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED<br>~~THESIS~~/DISSERTATION |

4. TITLE AND SUBTITLE
Emulation Framework for Testing Higher Level Control Methodology

5. FUNDING NUMBERS

6. AUTHOR(S)
Harold W. Ennulat, Captain

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student Attending: Virginia Polytechnic Institute and State University

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA- 92-018D

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/CI
Wright-Patterson AFB OH 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW 190-1
Distributed Unlimited
ERNEST A. HAYGOOD, Captain, USAF
Executive Officer

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

DTIC
ELECTE
DEC 0 8 1992
S B D

92-31040

14|pr

| | |
|---|---|
| 14. SUBJECT TERMS | 15. NUMBER OF PAGES<br>131 |
| | 16. PRICE CODE |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18

# EMULATION FRAMEWORK FOR TESTING HIGHER LEVEL CONTROL METHODOLOGY

by

Harold W. Ennulat

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Industrial and Systems Engineering

APPROVED:

[illegible]

Dr. M. P. Deisenroth, Chairman

[illegible] [illegible]

Dr. O. K. Eyada Dr. R. J. Reasor

May 1992

Blacksburg, Virginia

# EMULATION FRAMEWORK FOR TESTING HIGHER LEVEL CONTROL METHODOLOGY

by

Harold W. Ennulat

Michael P. Deisenroth, PhD., Chairman

Industrial and Systems Engineering

(ABSTRACT)

Emulation is defined as an intermediate stage of simulation where the model represents the "as specified" mechanical plant and equipment, but not the control logic required to drive it. This thesis investigates the utility of providing a computer representation of the functional elements to be controlled by system control programs. These representations or "emulators" mimic the behavior of the system, or factory being controlled. The advantages of such a scheme are that developers of control software, are able to test out new control methodologies withoit actually connecting to the hardware system under control.

This thesis investigates system control for automated manufacturing systems and identifies how emulation can be used as a valid tool in reducing the implementation time of such systems. The functions and characteristics of system control are identified as well as the problems associated with their implementation. The problems are then categorized to identify where emulation is a valid tool for problem resolution. This thesis is concluded by a description of a software demonstration which validated the concept of using emulation to solve system control problems.

## ACKNOWLEDGMENTS

First, I thank God and my parents for providing me the direction that has allowed me to succeed in all facets of my life to date. Next I thank my girlfriend, soon to be fiancé, Carol Walker. Without her continuous support and proofreading, this document would not be a reality.

At Virginia Polytechnic Institute and State University, I owe many thanks to numerous individuals. First, I thank my advisor, Dr. Michael P. Deisenroth for his invaluable advice and counsel. Throughout my stay at Virginia Tech he accepted me not only as a student but as a friend, making this an enjoyable learning experience.

Next, I thank my other committee members, Dr. Osama K. Eyada, and Dr. Robert J. Reasor for their patience and support throughout my research. Their inputs and critiques added greatly to the quality of this research. I would also like to thank Dr. Dennis Kafura for helping me with the software demonstration, his explanations of UNIX helped clarify many ambiguities.

Lastly, I thank my many fellow graduate students and friends that helped out on numerous occasions. Without the help of Jeff, Simon, Malay, Jai, K.K., and especially Joni this document would never have been completed on time. I also want to thank many other friends I made here, Andy, Mike, Jin, Janice, Angela, Carol, Nei, Dan, Tina, Katti, Allen and others I am sure I forgot. These friends supported me through the low points and helped celebrate the good times, thanks.

TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## 1.0 INTRODUCTION

With the arrival of low cost microprocessors and the quest for increased productivity from manufacturing enterprises, automation of manufacturing systems is receiving increased attention within the manufacturing community. Many companies view Flexible Manufacturing Systems (FMS) and Computer Integrated Manufacturing (CIM) as keys to increased productivity. However, research has shown that implementation of these systems presents many obstacles [40]. Most of these can be traced to the following points:

1. Software development is a major technical problem and many difficulties arise related to the software implementation.
2. Implementation time is almost always considerably longer than expected.
3. Training is of vital importance when applying these advanced technologies.

As a result, a high percentage of companies attempting to implement advanced automation encounter major difficulties. More often than not, the final results are "islands of automation." One area where many companies encounter problems is in the control, coordination and communication between devices to make them act as an integrated system.

## 1.1 BACKGROUND

Software development, being a major hindrance in implementing automated systems such as CIM and FMS, has been the focus of much research in the past decade. The design of control software for FMS and CIM systems is a challenging task, and is hampered by a number of factors which cause the development of such software, by its very nature, to be a time consuming and iterative process. Among these are:

1. large data requirements,
2. randomness in the environment,
3. multiple interface requirements,
4. multiple level hierarchies,
5. multiple communication protocols, and
6. event timing and sequencing.

In most cases, companies do not even attempt to write this software themselves; instead, they enlist the services of third party system integrators.

Most successful control schemes are structured in a hierarchical manner [8,25,34]. An example of how this structure might look is shown in Figure 1.1. At the highest hierarchical level, the planning horizon may be three years in length. Command inputs at this level are decomposed into more detailed commands and passed down to the next level. Lower hierarchical levels are concerned with shorter planning horizons, in the order of seconds or fractions of seconds. Modules at each level make decisions based on upper level commands and lower level feedback [19]. This control scheme is chosen most often because of its modularity and reliability [50].

Software developers for automated systems, such as CIM and FMS, are most concerned with the middle and upper levels of the control hierarchy. This research is

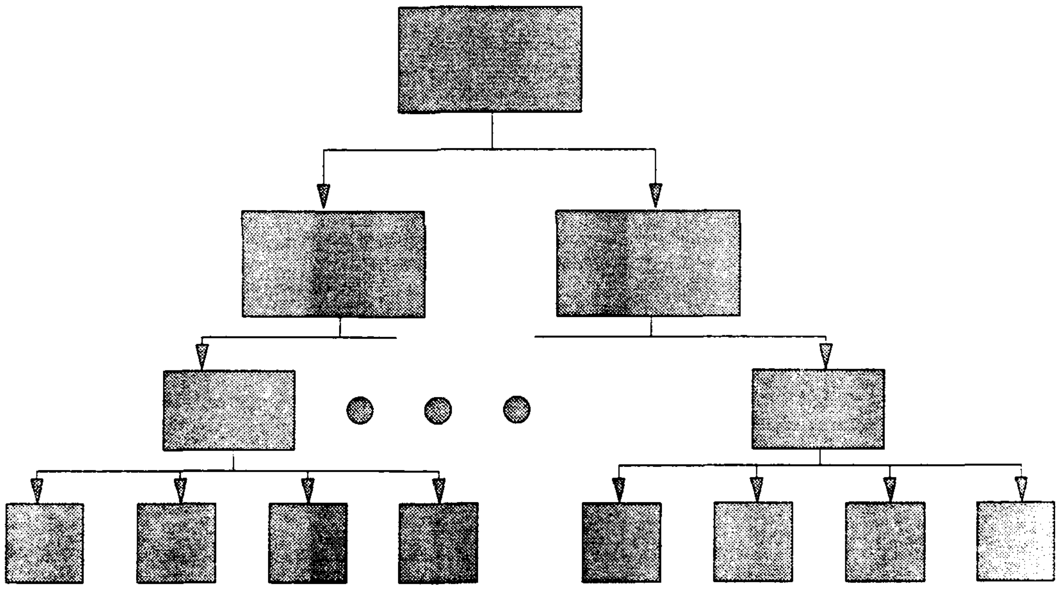

FIGURE 1.1 Hierarchical Control Structure

focussed on the system control level, and provides details of an emulation framework which can be used as a tool for the testing and debugging of control software.

## 1.2 PROBLEM STATEMENT

Typically, it takes two and one half to three years of development time to fully implement an FMS system [47]; development of the control software is a major part of this effort. Few companies have the resources to allow equipment to remain idle on the factory floor while the control software is being debugged. This dictates the need for the software to be tested and validated, to the fullest extent possible, before it is connected to the hardware in the production facility.

Presently, control software developers lack adequate means to test and validate their software without connecting it to the hardware on the factory floor. It is desirable to have a user friendly tool which would interact with system control software, and mimic the behavior of the factory floor. Such a tool would allow software test and validation to be done off-line. This would significantly reduce the development time for automated manufacturing systems.

## 1.3 OBJECTIVE

The concept of modelling with computer software is known as simulation. Although, it is now widely used in the manufacturing industry simulation modelling does not allow the testing of actual control logic used on the factory floor. This is because the model already is comprised of an abstraction of the control logic and the physical facility. The control logic is inseparable from the model, and being an abstraction of the actual control logic, it may not be transferrable to the actual physical system. The objective of this research was to provide a framework for testing actual system control software in the form it is being used on the factory floor, not an abstraction thereof.

## 1.4 APPROACH

The approach that was pursued in this research is known as emulation. Emulation is defined as an intermediate stage of simulation where the model represents the "as specified" mechanical plant and equipment [29], and outside control logic is used to drive the model. In emulation, inputs to the control logic are generated by the emulation model. These inputs match the inputs normally seen from the factory floor. The goal of this research was to investigate the applicability of emulation to system control, and recommend a framework for providing a computer representation of the elements to be controlled by system control programs. These representations or "emulations" mimic the behavior of the system being controlled.

There are many tangible benefits to such a scheme. Developers of control software will be able to test new control methodologies without connecting to the hardware system under control. This will save both time and money, as developers will not have to tie up expensive equipment to debug the control software.

Emulators provide system response feedback to control logic. These responses are generated, based on the behavior that should occur for the control decision made, by the control scheme. The control logic can not discern the difference between the emulation model and the real system. This provides system control designers with the capability to analyze how their software will control the system. Emulation's current major function is to provide a tool for software developers to use in testing and validating control schemes. Eventually, the scope of emulation may evolve to provide debugging and optimization capabilities for control schemes. Currently, validation still requires personnel with knowledge and experience to identify malfunctioning of the control software.

The use of emulation in the test and design of control software has been addressed by numerous individuals. The majority of the literature available on this subject leads one to believe that emulation is only being utilized in cell control applications. A number of authors mention that it can be used to test and validate control software at and above the system level. However, there is little mention of emulation being applied at the system control level. One reason for this may be that there are technical barriers preventing the further application of this technology at the system control level. This research investigated the use of emulation in relation to automated system control, with a focus on providing a tool for the testing and debugging of control software prior to implementation. Recommendations are provided for the structure and composition of emulators to be used for system control test and validation.

## 2.0 LITERATURE REVIEW

The first portion of the literature review provides a history of the development of simulation and emulation as a tool for developing and debugging control software. The second portion reviews some of the major developments in system control concepts and system control software.

## 2.1 EMULATION

Johnson, Milligan, Fortmann, Bloom, McLean, and Furlani [33] of the National Institute of Standards and Technology (NIST), formerly the National Bureau of Standards (NBS), first utilized emulation as a software development tool at NIST's Automated Manufacturing Research Facility (AMRF) in 1982.

Approximately 50 different control systems were implemented simultaneously at the AMRF. Emulation was used as a management tool to integrate all the interfaces between the different control modules. The hierarchical control system emulator (HCSE) was developed to accurately reflect the hierarchical structure of the AMRF. In this way, it could be used as a real-time interface to the individual control modules, allowing each to be tested when desired. The emulation modules were constructed as finite state-machine tables, translated into the high-level PRAXIS language, compiled, and combined into processes according to their emulated process, location, and function. The modules interacted through a shared time-sliced synchronized common memory. This process required an emulator to be developed for each component in the system, and was a lengthy, tedious process.

Courvoisier, Bigou, Valette, Desclaux, and Benzakour [16] used a Petri net modelling approach to develop emulators to integrate the behavior of machine tools and robots in their Specification, Emulation, and Conception of an Integrated Automation (S.E.CO.I.A.) project. Their approach executed a discrete event simulation based upon the utilization of timed Petri nets. Their approach did not allow the interaction of actual control software with the emulators. They modelled the control structure with Petri nets also, and linked the two models .

Quinn [46] used a simulation based software system to automatically develop and test automated guided vehicle (AGV) control software. He created a simulation model of the AGV system in the GPSS/H based AutoMod simulator. He then linked the outputs, which represented discrete control interface events, from the simulation, to his emulator software. The emulator software then translated these events into discrete signals, or messages, that were transmitted to the control software. The emulator was also capable of receiving control commands and translating them into future event logic for the AutoMod simulator to execute.

Quinn's emulation model made use of the trace file generated by the AutoMod simulation software. Trace files provided a very comprehensive means of indicating change in system status, because they were used by graphics software to generate animation of the model. The source of the trace file was the event file within the GPSS/H simulator. The emulator processes traced files and searched for events that would cause the physical system to send discrete signals or messages to the controller. When these events were identified, the emulator sent the appropriate signal or message to the controller. While this was occurring, the event was delayed in the simulator, until a return message was received by the emulator from the controller. The event then was modified, if necessary, and placed back on the event calendar.

Bell, Roberts, Shires, Newman, and Khanolkar [4] created a suite of data driven software for manufacturing system design and assessment. This included creating detailed simulation models with actual values, rather than assumptions and samples from distributions. They called these detailed simulations, emulations. However, the data was only used for statistical analysis and an animation of the proposed design. The research did not include a means for testing control logic.

Godio and Vignale [22] gave the best description of the emulator concept. They related the use of emulators for control software development and testing, as it related to automation of discrete parts manufacturing at Sandretto Industrie in Italy. They identified the major drawback of the emulator approach was the effort required by the development of the software. Each application differed from the previous one, due to different machines and/or configurations. Hence, the effort required replication correspondingly. They confirmed the need for a set of user friendly tools that would allow the user to easily specify and implement the emulator required by each individual application.

The developed emulator mimicked the facility from both the communication and the application point of view. The application emulator dealt only with symbolic addresses. Specifically, data transmission duration was considered negligible and always successful. In the application view, simulated time could be compressed to a fraction of actual time, if the processing time was disregarded. This viewpoint only tested the message response portion of the control software.

The communication emulator was more complex. It accounted for all other previously disregarded parameters. The communication application addressed simulating transmission errors, transmission duration, and volume of data to be exchanged. In this mode, simulated time equaled actual time.

Erickson, Vandenberge, and Miles [20] described a means of testing planned control logic for a specific manufacturing system. Their application involved linking a simulation directly to a programmable logic controller (PLC), providing a means for testing the control logic of the PLC. This was not considered emulation, which was defined as real-time data used to drive an animation, providing a display of the current status of the manufacturing shop floor.

Harmonosky and Barrick [28] discussed the use of simulation logic for real-time control in a CIM environment. In particular, they were concerned with the computer communication structures. They modelled the actual system components and the interactions between them. The logic used in the actual system communication paths, which triggered system activities, was the same logic used to control the simulation. The simulation emulated the communication signals which triggered activities, facilitating the critiquing of communication logic employed in the system. The model utilized the SIMAN simulation language.

The research concluded that the communication time to download programs or machining data was a significant contributor to part flow time. In addition, when different computers attempted to communicate, bottlenecks and computer deadlocks sometimes occurred. Therefore, the communication structure and its associated interactions should be included in the simulation or emulation for proper and accurate reflection of the real system.

Co and Chen [13] discussed an FMS emulator project at Case Western University. The FMS emulator used was an educational tool to help operations management students to understand the potential of computers and operations research techniques in the manufacturing environment. Programmable controller hardware for computer control, and IBM AT-compatible microcomputers were used for the emulation of the signals emanating from simulated machine centers. The simulation interacted with PLCs to feed commands to physical models of the system. Processing times on the models were simulated with manual toggle switches.

Hitchens and Ryan [29] of HEI Corporation discussed the use of simulation modelling throughout the life cycle design of automated manufacturing systems. They used detailed simulation, which was called emulation, to test control logic by direct connection to the controller. One of the primary distinctions between simulation and emulation was noted. In simulation, the process model and control logic are usually located in the same processor. However, in an emulation model, the process model and the control logic often are located in two separate computers. HEI's emulator was an entity based, time/speed/distance bounded model, which allowed interaction between the physical elements influenced by the control devices.

Clark and Withers [11] discussed the Computer Integrated Manufacturing - Open Systems Architecture (CIM-OSA). Within the context of CIM-OSA, integration criteria was defined for a decision support system, which was implemented through simulation.

Simulation models were used to produce emulated data for CIM applications to support "try-for-fit" and "what-if" analysis. The simulation model was supported by the same CIM architecture as the physical system it was modelling.

*Siggard and Alting [53] discussed the development of a test bed of emulators for test* and validation of shop floor control systems, at the Technical University of Denmark (TUD). The emulators provided a methodology for testing shop floor control systems' dynamic behavior in an operating environment before implementation. The emulators, which they developed, worked at the cell control level, imitating the Distributed Numerical Control (DNC) communication interfaces between the shop floor equipment and the cell controller.

By utilizing the same protocol, software was written which communicated in the same way as specific machine controllers. TUD developed a PC based software package which allowed them to define and emulate the communication interfaces of four computer controlled devices on a single PC. If more than four machines were contained in a cell, they had to link multiple emulations on multiple PCs.

The software was composed of three modules, an emulator creation module, an emulator execution module and a log book analysis module. The creation module allowed them to create or edit the definition of communication protocols which were to be emulated. The protocols could be supplemented with additional features, such as the simulation of part programs, fault messages and other spontaneous messages that the machine controller might send. The protocol definitions were stored as data files and could be edited any time. The execution module performed the emulation. It read the data files containing the protocol definitions and emulated up to four protocols simultaneously. A detailed history file of all communications between the host and the emulators was generated for later analysis. The log book analysis module allowed the user to view the history file. This file showed the byte for byte communications, and was a useful debugging tool.

Cloud [12] described work done at the Jet Propulsion Laboratory (JPL) of the California Institute of Technology. JPL was one of the contractors working on developing the Strategic Defense Initiative (SDI). The complexity of SDI forced developers to pursue the use of concurrent processors for the most computationally intensive algorithms, including the tracking and optimization functions. The work discussed by Cloud focused on developing a software framework to emulate multiple simultaneously executing strategic defense processes and their inter-process communications. By utilizing this emulation capability, developers were able to test multiple algorithms for strategic defense systems under realistic scenarios. This provided a means to determine the feasibility of different algorithms prior to implementation.

The JPL emulators were implemented on Mark III hypercube concurrent processors. The testbed, called Simulation89, operated in a closed loop mode. The SDI architecture was specified, as were the elements of the environment, with which the architecture interacted. Since the environment was so complex, JPL devised a method of operating the elements in either high or low fidelity. The architecture was tested against smaller subsets of the environment, primarily operating in high fidelity. The remainder of the environment was utilized, when necessary, but was operating in the background in low fidelity mode.

## 2.2 SYSTEM CONTROL

The control of automated manufacturing systems can be approached in a number of manners. Most control structures decompose the system into hierarchical levels [7,17,21,30,33,34,36,43,46,49]. Jones and McLean [34], along with colleagues at NIST, designed a hierarchical control model for the AMRF in Gaithersburg, Maryland. This control model is shown in Figure 2.1. It is decomposed into five major levels: facility, shop, cell, workstation, and equipment. Control modules at each level are broken down into one or more further detailed modules or sublevels, in the classic tree structure.

At each level, there are certain control functions associated with the modules that comprise that level. Figure 2.2 depicts the control levels with their associated functions. The facility level comprises three major functional areas: manufacturing engineering, information management, and production management. Manufacturing engineering provides user interfaces for the computer-aided design of parts, tools, and fixtures, as well as the planning of production processes. Information management provides interfaces and supports for the administrative functions of cost and inventory accounting, order handling, and procurement. Production management tracks major products, generates long-range schedules, identifies capital investment requirements, determines excess production capacity, and tracks quality performance data. The production planning data generated at this level is released to the shop control system at the next lower level in the hierarchy.

The shop control level is responsible for the coordination and allocation of resources to the production and support functions on the shop floor. It is comprised of two major functions, task management and resource management. The first schedules job orders,

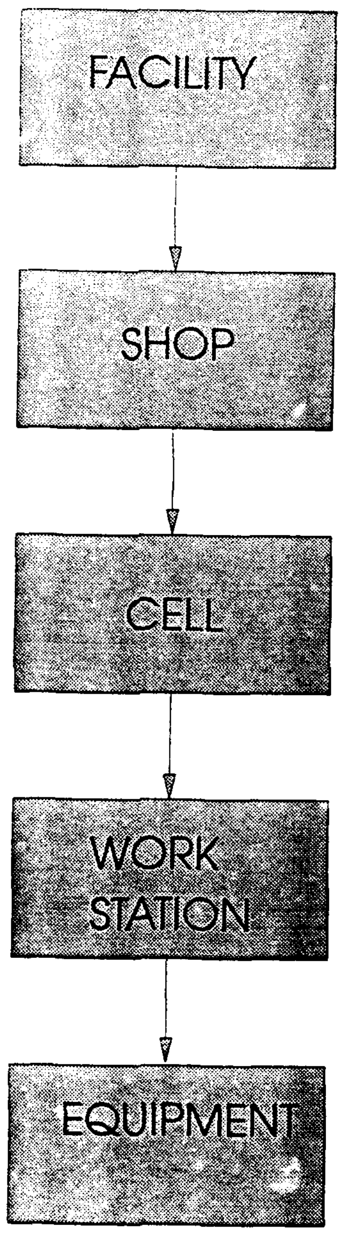

**FIGURE 2.1 AMRF Control Model**

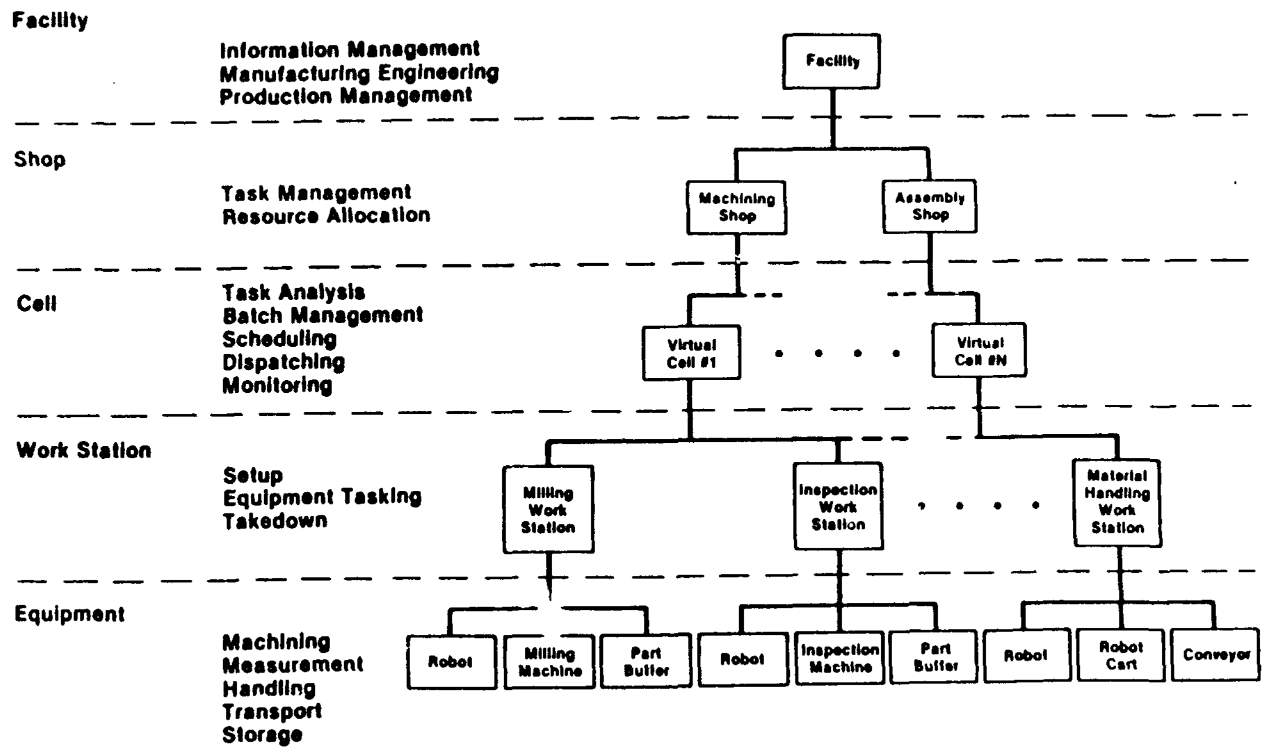

FIGURE 2.2 NIST Control Levels and Functions

equipment maintenance, and shop support services. The task manager also tracks equipment utilization, handles capacity planning, does batch scheduling, tracks orders to completion, and schedules preventive maintenance for all equipment in the facility. The resource manager allocates work stations, storage buffers, tools, and materials to cell control systems for particular production jobs.

The cell control level is responsible for managing the sequencing of batch jobs through the workstations, and supervising support services, such as material handling or calibration. Modules within the cell control scheme decompose tasks, analyze resource requirements, report on job progress to shop control, route batches, schedule tasks at assigned workstations, and monitor the progress of those tasks.

The workstation control level coordinates and directs the activities of equipment groups on the shop floor. It sequences robots and machine tools through job setup, part fixturing, cutting process, chip removal, in-process inspection, job take-down, and cleanup operations.

The equipment control level is tied directly to all pieces of automated equipment on the shop floor; these may be robots, numerical control (NC) machines tools, coordinate measuring machines, material handling systems, or storage/retrieval devices. The functions of the equipment controller are to translate the commands from the workstation controller into a sequence of simple tasks, that can be understood by the vendor supplied controller, and to monitor the execution of these tasks via sensor feedback.

Nof, Whinston and Bullers [42] developed a different approach for control of automatic manufacturing systems. The concept addressed the problems arising from unstructured and ill-defined decisions when controlling automated manufacturing systems. Unstructured decisions are those which cannot be formulated a priori because the problem has not arisen before, is unusually complex, or is so important that it requires special treatment. Many current automatic manufacturing systems cannot handle unstructured decision problems, since they do not implement a systematic model of system states and goals. To satisfy these unstructured and ill-defined decision requirements, Nof, etal., have developed a computerized decision support system called the Manufacturing Operating System (MOS). Its primary objectives are to implement automatic control for structured decisions, and provide, what has been termed "decision support", for unstructured decisions.

In developing the MOS, a number of the control limitations in current automatic manufacturing systems were outlined. Principle to these limitations is that automatic control systems are comprised of multiple functional areas, such as inventory control, physical distribution, scheduling, and capacity planning. Often the data required for a particular decision is contained in multiple program modules. Such disjoint modules may not have common representation for system attributes. Thus, even if the relevant data can be identified, attribute value retrieval is complicated by differing attribute representation among the functional modules.

Kochlar [38] discussed some of the advancements that have been made in software packages for manufacturing control. These packages are increasingly being used in the production planning and control capacities. Many of the initial packages available on the market were rigid, and customizing by the individual manufacturing organizations proved difficult, time consuming, and expensive. As a result, many companies had to develop and implement their own systems over long periods of time. The advances in computer software packages mean that many of the packages are flexible, easy to customize and implement. Kochler outlined the features of these advanced automated manufacturing control systems, including on-line real-time updating, ad-hoc and exception report generation, data validation based upon user specified rules, automatic system scheduling and recovery, and operation in distributed processing/local area network environments.

O'Grady [45], in his book, *Controlling Automated Manufacturing Systems*, discussed the production planning and control aspect of automated manufacturing systems. He tied together the work of these two diverse fields. In production planning and control, there is a large body of work completed in analytical modelling, computer structures, and overall systems. Equally, in the area of detailed hardware control, extensive studies have also been completed. His book stressed the important elements of both areas that are vital to effective production planning and control of the whole automated manufacturing system.

He defined automated manufacturing systems, and outlined a few of their most prevalent features and useful application areas. This was followed by a description of the particular requirements that these systems impose on production planning and control. He then provided the background against which a production planning and control system could be implemented. Included was an overview of master production scheduling, materials requirements planning, and job shop scheduling.

The remainder of the book was devoted to the structure of the production planning and control system for an automated manufacturing system. The control system was divided into four hierarchical levels for production planning and control purposes. They are factory, shop, cell, and equipment levels. Control functions associated with each level were presented and discussed.

Overall, the book presented a viable production planning and control scheme for an automated manufacturing system. It demonstrated how this structure ties in with more traditional production engineering and production management approaches. Feasible and effective approaches were described, and their application and implementation was discussed.

The staff of the Charles Stark Draper Laboratories wrote a book entitled *Flexible Manufacturing Handbook* [10], which is a good reference guide to companies planning to implement FMS technology. The book was designed to answer the following questions:

1. Why an FMS?
2. Will an FMS best serve your application?
3. What problems might be encountered?
4. How do you design an appropriate system? and
5. What is required to operate a system?

The book contains detailed descriptions of the subsystems that make up a typical FMS, as well as descriptions of several operational FMSs.

The book has a fairly detailed section on FMS operation. This section discusses FMS control from the perspective of a three level view of organizational operation. The first level is dedicated to long-term decision making. This involves establishing policies, production goals, economic goals, and making decisions that have long-term effects. The second level involves medium-term decisions, such as setting the production goals of the system for a specified period of time, perhaps the next month. The third level involves short-term decisions, such as which work order should next be introduced into the system. A summary of the three decision levels, and the associated software, hardware, and management tasks are shown in Figure 2.3. The book discussed the tasks involved at each level of control, and task implemention in a typical FMS.

Bakker [1] introduced a new control structure for FMS which departed from the traditional hierarchical structure. Bakker's view of hierarchical control structures was that they are implemented on a single central computer. While not always true, this presented problems that were the basis for the development of a new distributed FMS (DFMS) architecture. An argument was made that implementation of a control system on a single computer has some disadvantages. The system is vulnerable; if the central control system breaks down, the whole system will become inoperative. It is difficult to adapt the capacity of the central computer system when the FMS is extended. It will be difficult to add control power in small steps on an "as needed" basis.

DFMS is a system architecture that can be implemented on a number of small control units, referred to as station managers. Advantages of the DFMS concept include reliability, simplicity, and extensibility. Bakker contends that station managers can be implemented on programmable computerized numerical controls (CNCs) or on separate PCs connected through a serial link to the CNC it controls. In a DFMS system, the operations on each machine-tool are controlled by its station manager. The station managers behave as agents for the machine-tools. They negotiate which operation will be performed by each machine-tool.

| Time Horizon | Management Level | Typical Tasks | Typical Decision Support Software Used | Hardware Used |
|---|---|---|---|---|
| Long Term (Months/Years) | Upper Management | •Part-mix changes<br>•System modification/ expansion | •Part Selection Program<br>•Queueing models<br>•Simulation | •Mainframe Computer or DSS computer |
| Medium Term (Days/Weeks) | FMS Line Supervisor | •Divide production into batches<br>•Maximize machine utilization<br>•Respond to disturbances in production plan/ material availability | •Batching and Balancing Programs<br>•Simulation | •DSS Computer or FMS Computer |
| Short Term (Minutes/Hours) | FMS Line Supervisor (exceptions only) | •Work order scheduling and dispatching<br>•Tool management<br>•React to system failures | •Work order dispatching program<br>•Operation and Tool Reallocation Program<br>•Simulation | •FMS Computer |

FIGURE 2.3 Draper Lab Decision Levels

In the DFMS, concept schedules are not made before the actual production starts. The system is scheduled by manipulating operational queues for the separate machine-tools. This system precludes the necessity for an upper control level to track global data for determining scheduling and sequencing. After the completion of an operation, station managers check the product to see if more operations are needed, since sequences travel with the individual products. If another operation is necessary, the station manager negotiates with other station managers to determine which will get the next operation. The only information necessary for these decisions is the machine's capabilities and the length of the operational queues. The station managers also communicate with function modules, which perform general services, such as pallet transport, tool transport, and NC program storage.

Bakker provided detailed descriptions of the DFMS concept. He included the functions performed by the station managers and function modules, and discussed system implementation. He concluded with the advantages and disadvantages of DFMS.

Duffie, Chitturi, and Mou also pursued a more localized control structure [19]. They described a control architecture and set of fundamental design principles for developing and implementing fault-tolerant manufacturing systems. Their approach identified a number of innovative ideas. These included the concept of "intelligent manufactured parts", the use of a "flat" heterarchical architecture, as opposed to the widely accepted hierarchical architecture, the combination of simulation and control in system development and operation, and achievement of implicit modifiability and fault-tolerance.

Duffie, etal., developed an experimental, heterarchically controlled manufacturing system, consisting of a robotic machining and assembly cell. Global information was reduced to a minimum in this system, as were the complex relationships between entities. Parts to be manufactured were programmed as "intelligent" entities that interacted in a cooperative manner with "intelligent" robots and processing machines to satisfy system production requirements. They contended that localized or heterarchical control architectures offer prospects for reduced complexity by localizing information and control, reducing software development costs by eliminating supervisory levels, improving maintainability and modifiability through improved modularity and self-configurability, and improving reliability by taking a fault-tolerant rather than a fault-free approach.

The basic premise of fault tolerance was included in the author's design. Autonomy enforces localization of information, isolating each module from other modules in the system. A system with autonomously functioning components will not collapse when one or more of the components malfunctions or fails. Increased autonomy reduces the need for a highly centralized governing body.

## 2.3 LITERATURE SUMMARY

As the literature review demonstrates, control of automated manufacturing systems is a complicated problem. There are many interrelated decisions which have effects throughout the control hierarchy. As these complex control systems are being designed, software developers invariably make structural and schematic errors which do not become evident until implementation. The result is often a lengthy and costly delay in implementation of these systems.

Emulation, a hybrid of simulation, is an emergent technology, which has evolved in the last ten years. Control software developers, particularly at the cell control level, have utilized emulation to assist in the test and validation of cell control software prior to implementation. The concept applies to system control as well. Definition of a generic framework, for the use of emulation at the system control level, is the first step necessary in providing a user friendly tool for the testing and debugging of control software.

# 3.0 METHODOLOGY

In order to develop the use of emulation at the system control level, this research followed a structured approach. The research proceeded according to the following steps:

1. define the characteristics of system control,
2. show how system control interacts with the system,
3. define the form and functions of the emulators,
4. identify problems in developing system control,
5. relate emulation to problem solution, and
6. provide a software demonstration.

## 3.1 SYSTEM CONTROL

The initial portion of this research had a twofold purpose. The first objective was to define and identify the problems hindering the development of system control. Second, it helped define the boundaries of the research. System control, as it was pursued in the context of this research, is the computer control and coordination of a number of automated manufacturing workcells, functioning together as a system for the batch manufacturing of discrete parts. There are many categories of automated manufacturing systems; this research focused on system control as it relates to flexible manufacturing systems and computer integrated manufacturing systems in discrete parts manufacturing.

Initially, the functions comprising system control were identified. These functions were broken down into two categories. The first category consisted of those major functions which were absolutely necessary to control an automated manufacturing system. The second category comprised those other functions which could be added on at the system control level, but were not necessary to control a basic system.

## 3.2 EMULATION

The second major portion of the research was to pursue the further development of emulation concepts in system control applications. Current use of emulation within the manufacturing industry is limited mostly to cell control applications. Emulation is being used at a higher level by the Defense Department in the test of major systems, such as the SDI [12]. This supported the direction of the research, which utilized emulation at a higher level to test and validate the behavior of systems control. It showed that emulation is not restricted to a low-level tool for mimicking the behavior of individual pieces of equipment within workcells. Rather, emulation could be utilized for more complex, high level applications, such as resolving control and integration issues of automated manufacturing systems.

Initially, the different forms of emulation utilized at the cell level were investigated. This helped to establish the best emulation structure to utilize at the system control level. Generic classes of emulators were identified as a means of providing a user-friendly, easily configured emulator structure at this level. In order to develop the form of the emulators, the tasks performed by emulators in each class were identified, leading to the categorization of major functions. In order to provide modularity and reusability for different system configurations, the major functions were transformed into generic emulator functions. These functions could easily be configured to any number of emulators to account for different system control functions.

This portion of the research concluded with the identification of system control problems. In addition, it provided a discussion of the uses of emulation in solving some of these problems.

## 3.3 SOFTWARE DEMONSTRATION

The final portion of this research was a software demonstration to demonstrate the viability of emulation, with respect to automated system control. The demonstration showed how emulation modules could interact in a passive manner with system control software. The emulation modules incorporated functions to:

1. receive messages from control software,
2. simulate error conditions,
3. induce delays representing applications, and
4. send messages back to the control software.

A hypothetical system was used as the basis for developing the software demonstration. Figure 3.1 depicts the hypothetical system. This system also served as a basis in identifying and defining the major system control functions. These functions are discussed in Chapter 4.

The hypothetical system is an FMS consisting of three machining workcells. Two of the workcells have fixed processing capabilities, while the third is more flexible and can perform the operations of the other two workcells. Work pending and finished part queues exist at all workcells. In the software demonstration portion of the research, it is assumed that work pending queues have an infinite capacity. The FMS has a load and unload station where new parts are introduced into the system, and finished parts are removed for packaging and delivery. All the machined parts for the finished product are on one pallet when they arrive at the assembly workcell. Pallets are transported through the system on AGVs. There are three AGVs servicing the system.

Each workcell, including the AGVs, has its own controller that can communicate with the system controller through a messaging system on a local area network. Figure 3.2 depicts the communication network servicing the system.

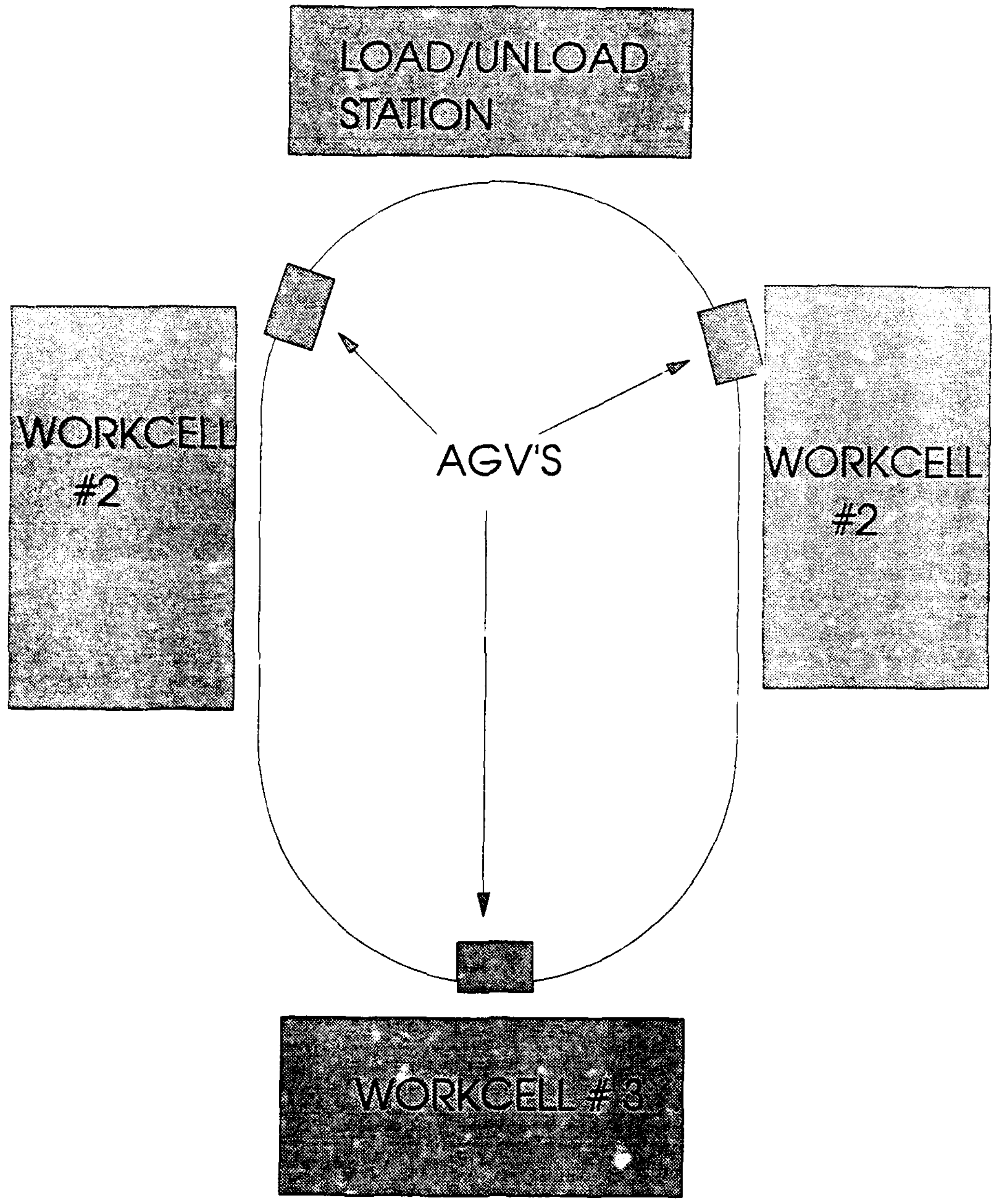

FIGURE 3.1 FMS Demonstration Model

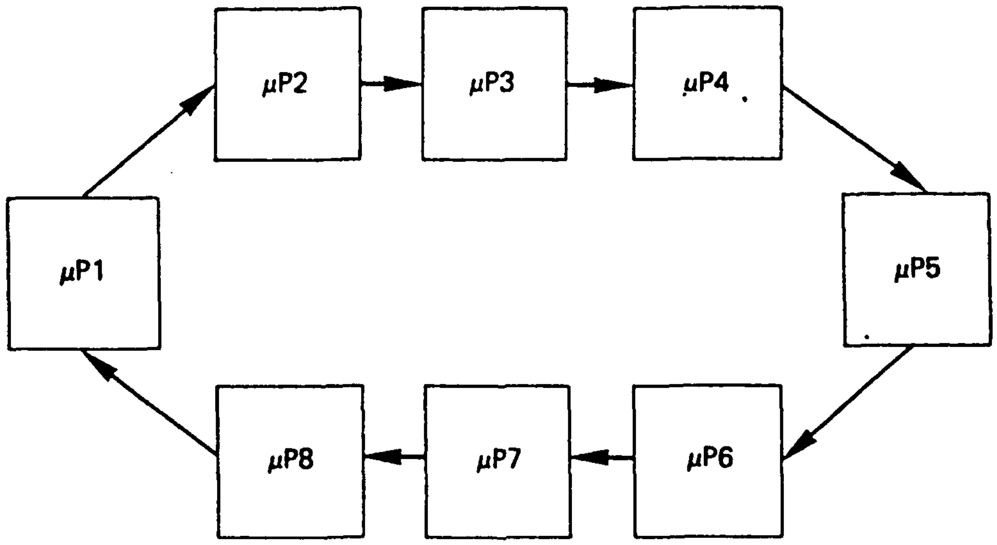

FIGURE 3.2 FMS Communication Network

# 4.0 SYSTEM CONTROL

## 4.1 STRUCTURE

A complicated issue in developing an automated manufacturing system is the integration of a diverse spectrum of components into a functionally homogeneous system [19]. As the system is being built up, the control structure and the level of control distribution must be established early in the system development cycle. Control structures vary from very centralized systems, with all the computing power residing on one centralized machine, to very distributed systems, where all the entities in the system have a high degree of local autonomy [1]. Figure 4.1 shows the spectrum from which the choice can be made.

Each potential control structures has advantages and disadvantages. This research is based on a hierarchical control structure, since it is the most commonly applied architecture. Hierarchically controlled systems are constructed using the levels of control concept. These systems contain a number of control modules which are arranged in a pyramid structure. Commands from the highest level are concerned with longer planning horizons than those at the bottom of the pyramid. Higher level commands are decomposed into more detailed commands for the lower levels. Modules at each level make decisions based on the input from the level above, and feedback from the level below. The number of levels, and the decisions made at each level, vary from system to system. However, the basic philosophy of command distribution is the same. Figure 4.2 depicts the levels used in the context of this research.

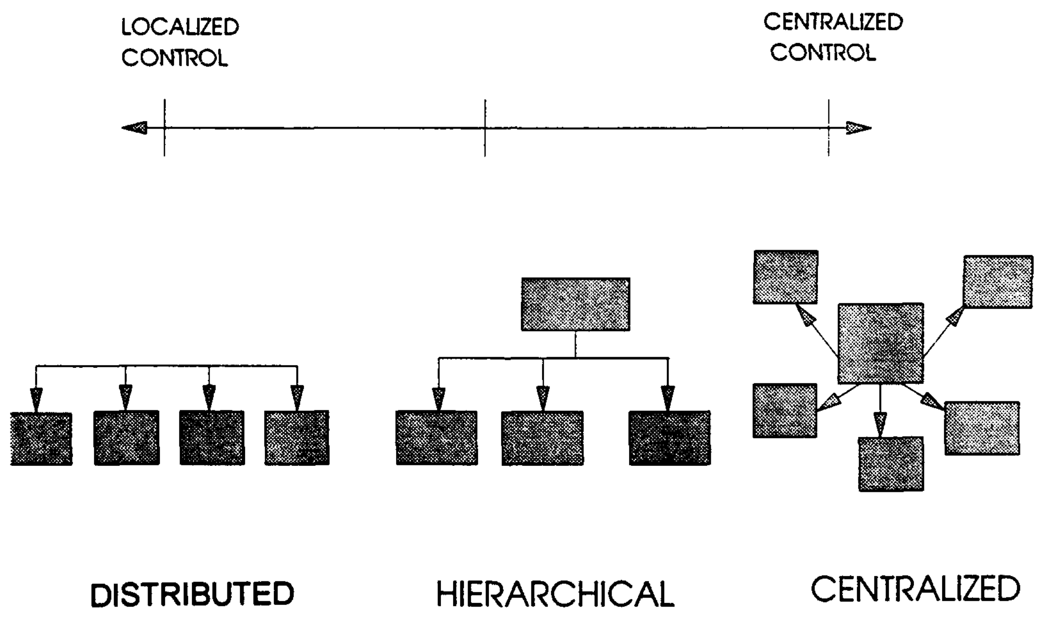

**FIGURE 4.1 Control Spectrum**

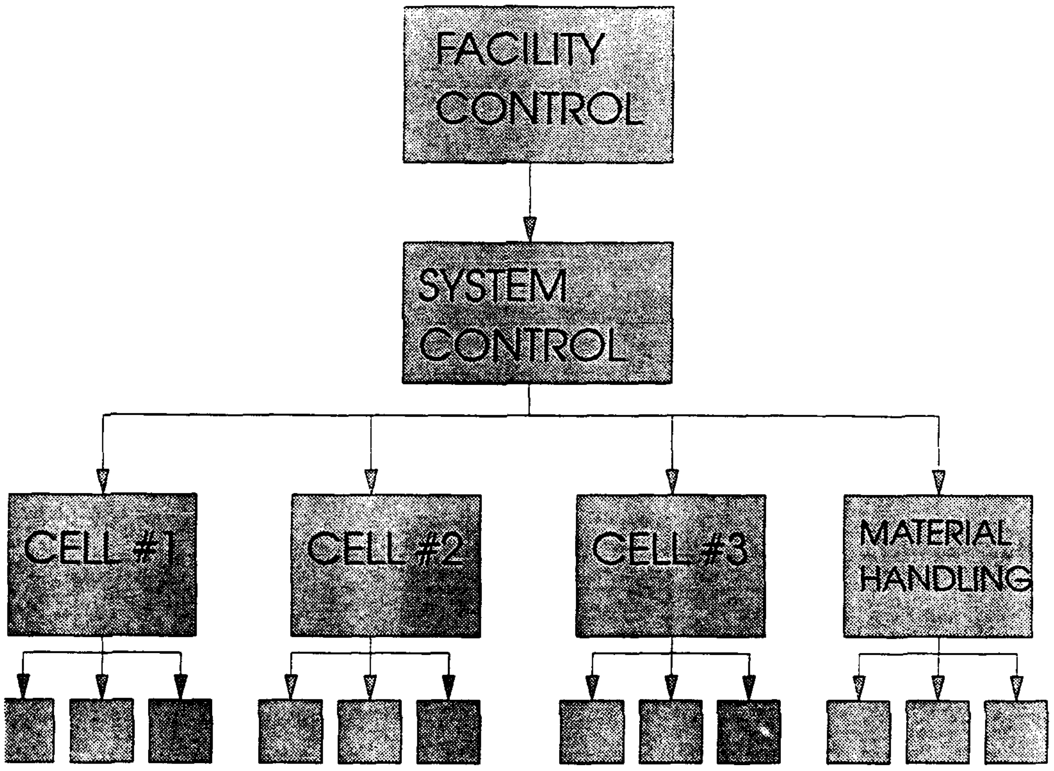

FIGURE 4.2 Control Levels

The highest level is the facility level; it takes into account all the different inputs from the environment to develop work to be done by the system. These inputs include demands, forecasts, material availability, finances, and factory capacity. The functions performed at the facility level, such as process planning, master production scheduling, and materials requirements planning, provide the next lower level, the system control level, with the number of units of a particular product to produce in the form of a "work-to" list.

This research was focused on the system control level which is mainly concerned with taking the requirements given by the facility level, and translating them into commands or goals for each cell. The functions performed at the system control level will be elaborated upon in Section 4.2.

Directly below the system control level is the workcell control level. The workcell controller takes commands from the system controller, and translates them into specific commands or operations to be conducted on the workpiece by the workcell equipment. The workcell controllers pass status information back to the system controller, so that workpieces can be efficiently sequenced through the system. In a hierarchical architecture, the control structure is suitably decentralized, such that the workcell controllers can have adequate decision making capabilities, and thus only require fairly simple messages to carry out their functions.

At the lowest level in the hierarchy are the equipment controllers. These usually reside on the actual equipment provided by vendors. Equipment controllers accept commands from cell control to perform the actual processing operations on the workpieces. The type of equipment usually present at the equipment level include robots, NC machines, coordinate measuring machines, AGVs, and conveyors. These different equipment types have very diverse languages and communication protocols, resulting in severe strains on the workcell controllers that must communicate with all the different types of equipment in their cell. Unfortunately, due to the large number of equipment manufacturers, achieving a common communication protocol has been a long, slow process. It was hoped that General Motors' Manufacturing Automation Protocol (MAP) would be adopted as the standard for the industry. However, it now appears that this effort has failed, due to high cost and lack of vendor compliance.

## 4.2 MAJOR SYSTEM CONTROL FUNCTIONS

Hierarchical control of most automated manufacturing systems can be decomposed into functional elements. Some of these elements are essential requirements to control the system, while others are added features which can just as easily be done off-line or at another level in the hierarchy. The major system control functions discussed in this section are not all inclusive. However, they provide a sound representation for the purposes for this research. They are:

1. Status Monitoring [2] [5] [14] [23] [24] [36] [45] [54] [56]
2. Control Enforcement [10] [19] [20] [24] [36] [45]
3. Traffic Coordination [1] [2] [3] [10] [14] [23] [24] [34] [36] [45] [54] [58]
4. Work Order Scheduling [1] [2] [3] [10] [14] [17] [23] [24] [34] [36] [42] [45] [58]
5. Status Reporting [1] [10] [23] [24] [38] [58]
6. Operator Interface [1] [3] [14] [36] [42], and
7. System Maintenance [10] [19] [38]

The following sections define each function and its relationship to the other major control functions. In addition, problems relating to the development and application of each function are discussed. Lastly, a description of the application of each function in the hypothetical model, discussed in Chapter 3.3, is provided.

### 4.2.1 STATUS MONITORING

The status monitoring function is a near real-time function that tracks the status of the entire system by communicating with the workcells which make up the system. The status monitoring function consists of four sub-functions:

1. Read
2. Write
3. Watchdog Timer, and
4. Status Tracking

The read sub-function polls workcell communication medium looking for messages. Once messages are read, they are erased by system control to avoid repeating action on a workcell just serviced. The type of messages that system control receives from the workcells can be classified as action messages or status messages. Action messages are initiated as a result of a change in state within the workcell. Examples of messages, which are self initiated by the workcell controller, are workorder complete and alarm messages. The presence of an action message indicates a change of status, requiring system control to take action with respect to the cell sending the message. This necessitates that the status monitoring function must interact with other major control functions. Action messages are acted upon by the traffic coordination, control enforcement, and/or work order scheduling function.

Status messages are also usually displayed on an operator interface for monitoring purposes; examples of these include number in queue, workcell busy, workcell idle, and workcell down. Whether status display messages are sent as a result of a request, or independently, is dependent upon the system configuration. Status messages are also necessary for status reporting. This is a separate function, which interacts with the status monitoring function, for this data.

Problems with the read sub-function arise when more than one workcell sends a message since the most recent poll. System control must determine which cell requires the highest priority action and act accordingly. Another potential problem is determining how frequently the workcells are to be polled. It is assumed that system control knows the processing time of the different workpieces for all operations. Workcells should definitely be polled at least as often as the shortest of these processing times. However, the possibility of a failure alarm at one of the workcells also exists. Therefore, how often system control should check for errors needs to be determined. In addition, system control may try to read a message at the same time that a workcell is writing a message. This type of error is known as deadlock, and could result in an unrecoverable error. Allowances need to be made to prevent these timing problems, from impeding upon system performance.

The second sub-function in system monitoring is a write command. It supplies the workcell controllers with messages which dictate their next action, or perhaps requests them to supply the system controller with status information. A problem associated with this sub-function is assuring that system control's messages are received. This can be accomplished by having the workcell controller acknowledge receipt of the message.

The next sub-function, watchdog timers, is an error checking type of operation, but, since it involves reading and writing information from cell control, it is listed as part of status monitoring. Watchdog timers are simple functions which check if the workcell controllers are still in an active state. They do not check the status of the workcell. They simply verify the read/write capability of the workcell controller. Watchdog timers are also used in error recovery. They are utilized as the first step in responding to a time-out error, during communications with workcell controllers. A prudent idea is to schedule watchdog timers to test all workcell controllers on a periodic basis, in order to avoid sending workpieces to a cell which is inactive. If a watchdog timer test fails, the status of the cell can be changed to down, and pending work can be rerouted, if necessary, while repair operations are being conducted.

The final sub-function within status monitoring is the task of status tracking. Status tracking applies to workcells (busy, idle, down), material handeling devices (breakdowns), workpieces (the next step in the routing), and possibly raw materials and labor. When the status of workcells change, system control needs to note this status change, and preserve it. When workcells are down for repair, system control reroutes work to other cells, if possible. Maintaining status within system control eases the problem of over burdening the system, since system control can track when a part can be entered into the system. Tracking the status of workpieces overlaps one of the other major functions, traffic coordination which will be discussed in Chapter 4.2.3.

Tracking the status of broken down material handeling devices is closely related to another major function, control enforcement, discussed in Chapter 4.2.2. When material handeling devices break down, the alarm message from the material handling controller informs systems control. Once the material handeling device has been repaired, system control can then inform the material handling controller to reactivate it. Raw materials and labor tracking are areas that fall into other functional elements, which are outside of the group of major system control functions.

The communication between entities, within an automated manufacturing system, almost always take place over a local area network. Providing that there is a sufficient level of decision making capability within the cells of the system, these communications can take place through the use of a messaging format. Different ways of passing messages between control entities exist; the most common used method is the use of mailboxes in a common area of memory, accessible to all controllers [7]. Messages are written into mailboxes by one controller, where they are read and acknowledged by the receiving controller. After they are read, it is most efficient if messages are erased by the receiving controller, so that operations are not repeated. In order to insure that messages are received, system control awaits an acknowledgement from the cell controller after each message transmission. If no acknowledgement is received after a set time period, a time out error is assumed. Communication between system control and the cell controller is verified, and the message is then retransmitted.

Messages from system control to workcell controllers are general in nature. They do not contain specific processing information. A generic message format would contain the part type, the part number for tracking purposes, and the operation type in a numerical code or descriptive type message. From this information, the cell controller ascertains the configuration of the part to load into the cell, the operation sequence along with specific processing parameters, and the part number for tracking and status messaging back to system control. The specific processing information resides within the cell controller. The cell controller needs to know when a part has been delivered to it, and some basic characteristics of the part. Again, once the message is read by the cell controller, it is erased to facilitate the tracking of incoming parts.

## 4.2.2 CONTROL ENFORCEMENT

Control enforcement is the second of the major control functions. Control enforcement is a real time function, which provides system control with the ability to deal with unpredictable events that occur in one of the cells, to the material handeling mechanism, or in the communication link between the system controller and one of the cell controllers. It is viewed as a set of emergency contingency plans, which is only enacted when one of the elements of the system breaks down. The control enforcement function interacts directly with the status monitoring function in learning when an error or alarm has occurred. Once it has established that action is necessary, it interacts directly with the part of the system requiring attention.

The major problem associated with this function is determining what to do with work already in queue at a broken workcell. Work that must be removed and rerouted must be fit into the work schedule. If the repair time is relatively short, then work in queue need not be rerouted. Similarly, when material handeling devices breakdown, the disposition of the part that was being transported must be determined.

How the control enforcement function handles different emergencies in the system is dependent upon the system configuration and the capabilities of the workcell controllers. Since control enforcement is a real-time function, it should be able to react to an error or alarm signal immediately. In order to have this quick reaction time, system control should have contingent plans in the event of various errors, prior to their actual occurrence. Relating this function to the hypothetical system described in Chapter 3, duplication of processing capabilities exists for some operations. The workcell controllers have the capability to determine the nature of the error. Workcell controllers send a specific error message to system control identifying the type of error that occurred within the cell. System control alerts the operator of the condition via the system control operator interface function, which is detailed later in Chapter 4.2.6. System control ascertains the nature of the alarm, and determines the average repair time from tables or a database to which it has access. Some error conditions, such as a time-out, may not require the operator to interfere with the system. In the event that the error requires an operator to attend to it, the system controller changes the status of the cell to down until the operator notifies it that the error has been corrected. If an error requires the cell to be down for any length of time, then work going to that cell is rerouted.

Rerouting of work requires identifying the process necessary, determining which cell has duplicate capabilities, and routing the work to the new cell. If either duplicate capabilities do not exist for the particular operation, or the duplicate cell is not in an active state, workpieces will need to be rerouted to a work-in-process (WIP) storage buffer. In this case, it is necessary for system control to request more work, which does not require the services of the nonfunctioning workcell, from the facility controller. Once the cell has been repaired, system control reactivates it, and the rerouted work is returned to its original routing. WIP that was put into storage is cleared back into the system.

If the breakdown occurred to the material handeling system, the recovery scenario is somewhat different. Since material handeling systems are much larger than workcells, upon receipt of an error message, system control notifies the operator which part of the material handeling system is in need of repair via the operator interface. System control waits for the operator to confirm that repair has been made, before instructing the material handling controller to reactivate the material handeling device.

### 4.2.3 TRAFFIC COORDINATION

Traffic *coordination* is one of the more integral functions in system control. This function controls the movement of workpieces between workcells. Tracking the routing of different workpieces is one of the most important aspects of this function. Routings are determined by facility control, and downloaded directly to system control or located in a common database accessible by both system and facility control. The implementation of the traffic coordination function varies, depending upon the material handling mechanism in the system. In functioning, it interacts with the work order scheduling and status monitoring functions.

The traffic coordination function faces several potential problems. First, a balance is necessary between workpiece prioritization and sequencing. Multiple requests for transportation can occur. Work order determination is required, dependent upon existing prioritization and workcell status and capacity. Second, a workpiece pileup at a workcell can occur. Traffic coordination must know that the next workcell in the routing has the capacity to accept the work order being delivered. The workcell may have a full queue or be in a down state.

The traffic coordination function loads parts into the system according to the schedule provided by the work order scheduling function. The traffic coordination function sends the material handling controller a message indicatinڋ part number, priority, pickup site and delivery destination. The material handling controller responds to the traffic coordination function, and indicates when the part has reached its destination. Upon completing the processing of the workpiece in the workcell, the status monitoring function relays the status back to the traffic coordination function. The next cell in the routing is then determined, and the cycle is repeated.

### 4.2.4 WORK ORDER SCHEDULING

The work order scheduling function determines the processing schedule of work orders based on the "work-to" list originated by the facility control. It takes into account the capacity and status of the system before determining schedules. Work order scheduling prioritizes the schedule based on the order's delivery date and processing time. It tracks all orders as they progress through the system. Work order scheduling is typically not a real-time nor near real-time function. Rather, it is utilized on a shift basis. The function interacts with the traffic coordination and status monitoring functions.

The work order scheduling function varies among system configurations. Various approaches, including analytical methods, heuristic methods, mixed integer programming, and goal oriented programming, have been utilized with mixed success [44]. The work order scheduling function can be further complicated when the availability of tooling, and high cost of jigs and fixtures is considered. Some manufacturers only carry limited quantities of these items. In this instance, a part can not be scheduled until the jig or fixture it needs is available. In addition, some systems do not have workcells with the capacity to store all the tools necessary, for all processes needed, for completion of workpieces scheduled in them. In these instances, work order scheduling must also schedule the delivery of tools to workcells.

The facility control's "work-to" list contains product types, order quantities and required delivery dates. Schedules are prepared on a shift basis, and transferred to the traffic coordination function in the order that they should enter the system. Prioritization allows workcell controllers to resolve conflicts in scheduling that may arise. Unrealistic or unattainable schedules are avoided through interaction with the status monitoring function, which determines if any workcells are down, and when they are expected to be back on-line.

### 4.2.5 STATUS REPORTING

The status reporting function generates and tracks system performance data, and reports it directly to the facility controller or the operator. Like the workorder scheduling function, status reporting need not be real-time. Reports are generated according to a schedule, such as at the end of every shift, or upon request from facility control or the operator via the operator interface. System performance data is utilized by facility control as an aid in altering or updating "work-to" lists when necessary. This information will also be used as trend data, which will be reported to management. Direct interaction with the status monitoring function and the workorder scheduling functions are necessary when determining system performance data.

The type of data which this function tracks and reports include quality control data on different aspects of production, the number of orders completed, the completion time of particular orders, workcell utilization, and average downtime of different cells. Most statistics reported by this function are generated at the system control level. Some of the statistics which relate more specifically to workcell activities, such as quality control, are generated at the workcell level. This data is product specific, as opposed to workcell specific, which describes the previously mentioned system statistics.

In keeping system statistics, this function interacts directly with the status monitoring function. In the hypothetical system, each status message received from different workcells has associated with it a time stamp. This time stamp is what enables the status reporting function to track time based statistics on cell activity. Examples include the workcell utilization and average downtime of workcells. Workcell utilization is calculated by tracking the amount of time that the work cells are busy. This is easily accomplished by noting the time stamp when each of the cells status is reported as busy, and again, when it changes to idle or down. The total busy time for each cell is tracked, and when the utilization is requested, this is divided by the total time the system has been operative. Average downtime is tracked in much the same manner, except, it is downtime which is tracked rather than busy time.

### 4.2.6 OPERATOR INTERFACE

The operator interface function provides a link between the human operator and the other system control functions. Situations can arise that dictate that a human operator interact with and take control from the automated system control. The operator interface function is a real time function with multiple capabilities. These capabilities include:

1. Pausing or halting the system,
2. Configuring system parameters,
3. Requesting statistical data,
4. Displaying system status,
5. Restarting the system,
6. Monitoring communications,
7. Editing of control software, and
8. Direct communication with workcells.

In most cases, the operator interface works via the keyboard and monitor of the computer running the systems control software. However, this is not always the case. The operator interface may be a separate processor linked either serially, or via a network node. The operator interface may also be an interface terminal/monitor, such as those available from some of the larger vendors.

The major problem which exists with this function is recognizing and processing requests from the operator interface without interrupting the control of the system. Either a flag to system control from the operator interface or a poll of the operator interface from systems control must be provided to periodically search for requests.

Several instances exist where the operator may want to interface with the system. The system may need to be shut down for periodic maintenance or the addition of machine tools. Also, the exact location of a specific workorder may be required for management or a customer. Lastly, reconfiguration of a part of the system may be necessary, if a problem has occurred from which the system alone can not recover.

The operator interface usually displays the different tasks or modes in a menu format. Requests for interaction are made through these menus, which lead to more detailed menus in different categories.

### 4.2.7 SYSTEM MAINTENANCE

The system maintenance function can be viewed as a software/computer system reliability function. It is designed to periodically check the operational system and prevent catastrophic system failure. System maintenance contends with system control problems, rather than the elements within the system control. It parallels the control enforcement function and the watchdog timer portion of the status monitoring function; however, it applies to the system control hardware/software, not the system itself.

Several potential problems exist with this function. First, in performing system backup, adequate peripheral storage must exist to produce three copies of the control code and data. Old backup copies should not be erased until a new backup has been verified to be successful. Second, available software for the checking of timing and responses from system control functions will require extensive development time and cost. Lastly, interfacing to hardware diagnostic software presents its own class of problems.

The function consists of two elements, the preventive/backup feature, and the system status checking feature. The backup feature is designed to prevent loss of the control software and associated data in the event of catastrophic failure of the computer and its memory. Periodically, the system software is copied to peripheral locations. Peripheral locations include other parts of a hard drive, floppy drives or onto another machines hard drive through the communication network. Multiple copies are maintained on different media, to preclude dependency upon one version of the control software and associated data. This assures a running copy, with recent valid data of the system software, in the event of a major systems failure.

The second feature, the system status checking feature, periodically assesses the operational and functional capability of the hardware running the system software. Such hardware diagnostics are run at specified intervals, such as the beginning of each shift. Observation of each function's timely response to a set of test data is required. In addition, synchronization of workcell and system clocks is tested to assess overall system timing.

### 4.3 OTHER FUNCTIONS

A number of other functions exist which may facilitate systems control. Although desirable, these functions are not necessary to control basic automated manufacturing systems. Some of these functions include:

1. Tool Management
2. Remote Distribution of NC data
3. Model Resources, and
4. Generate Requirements.

Each of these functions will be discussed in the context of their capabilities. This list is not all inclusive; it is not the intention of this research to identify all possible system control functions. This recognizes that there are other functions that are desirable, but not essential, to systems control.

#### 4.3.1 TOOL MANAGEMENT

The tool management function tracks the availability and total processing time on different tools in the system. This function ensures that the tools necessary for different processes are available at the proper workcells at the proper time. As was mentioned in the previous section, instances exist in some systems when all the tools necessary for the different processes are not kept in the tool racks at the necessary machines. In these situations, it is essential to know the necessary tools for the different processes. This function also tracks the location of all the tools in the system, whether in different workcells or tool storage. Once a workcell requires a tool it does not possess, the tool management function ensures that the tool arrives before the process is ready to begin.

Another task associated with this function is the tracking of how much time each tool has been used, in order to monitor tool wear. This provides a confidence factor in assuring that operations will be completed before tools break or affect the quality of the part. Tracking is required at the workcell level, where the tools are being changed during different processes. Since tools may be passed from workcell to workcell, the processing time associated with each tool is best monitored at the system control level for overall tracking. Other ways of tracking and predicting tool wear exist that may negate the necessity of this function. Most basic systems, to date, are small enough that they can store the required tooling in the machines tool racks for all the processes necessary. Therefore, this function is not considered to be a major function.

## 4.3.2 REMOTE DISTRIBUTION OF NC DATA

The remote distribution of NC data function supplies the proper robot and NC code for the different workorders to be performed at each cell. Several different ways exist in which this data can be distributed to the appropriate equipment. One method involves maintaining a common database or library containing all the different NC and robot programs. The workcells would either request the programs directly, or send a request to the control level maintaining this database. Another method consists of all the appropriate programs residing in each of the workcell controllers' memory. For these reasons, and because the NC code is generated off-line or at the facility control level, this function is not included as a major system control function.

### 4.3.3 MODEL RESOURCES

The model resources function provides operators the ability to simulate different levels of activity, in order to determine the capabilities and capacities of the system. Some systems utilize this functional capability prior to scheduling the activities of the system. It not only provides insight into the viability of different scheduling schemes, it also helps in determining the resources required for particular production levels.

This function is more easily done off-line. There are a multitude of simulation packages on the market that are more appropriate for this capability, as opposed to trying to create it within a system control program. Also, there is a duplication of effort when one considers that most Materials Resouce Planning (MRP) systems can provide capacity planning, consumption planning, scheduling and other forecasting functions [24]. MRP is a facility level function, so the use of this function is best left at the facility level.

### 4.3.4 GENERATE REQUIREMENTS

The essence of generate requirements function is the translation of item requirements from facility control to processing operations in different workcells in the routing. This function works very closely with the established routings and process plans from facility control. It is paramount that the function know the different processing capabilities at the different workcells. This provides a set of processing operations for all the workcells in the routing of a particular workorder.

This function is viewed as an intermediary form of computer aided process planning (CAPP). CAPP is the automated generation of a process plan. This function assumes that a process plan already exists. When an item requirement is produced, this function identifies the process plan and links it with a routing plan. For this reason, the function is usually found at the facility control level. Another reason that the generate requirement function is not considered a major system control function is that workcells can be preprogrammed with the different processing plans for different workpieces. In this manner, only a processing code needs to be downloaded to the workcell as the workpiece arrives.

# 5.0 EMULATOR DEVELOPMENT

The functions, listed in the previous chapter, comprise the capabilities found in most automated manufacturing system control structures. The development and integration of these functional capabilities into a workable control program requires a significant effort on the part of software developers. To aid in this development, especially in the test and verification stages, an emulation capability to mimic the controlled elements on the factory floor is needed. The development of the form and structure of this emulation capability is the essence of this chapter. These emulators must have the capacity to respond to the previously discussed control functions.

## 5.1 EMULATOR STRUCTURE

The structure of emulation capabilities used in the past has varied, from state tables used by NIST [7], to Petri nets used by researchers in both Europe and the U.S. [6] [16] [27]. One problem of both state tables and Petri nets is that they require an understanding of their structure in order to develop the emulation capability. Therefore, developers must be extremely well versed in the theoretical aspects of computer science. The state table and Petri net approach require large amounts of time when programming these emulators. A major shortcoming of the Petri net approach is the models intermix of process plans, control, and cell operation [25]. This intermix causes difficulties when attempting to utilize actual control code for the control of the models. Also, software reusability is not taken into account; much duplication of effort exists when developing the models.

One advantage both the state table and Petri net approaches, has been their efficient use of limited computing power. Now, with the relative low cost of fast, high power computer hardware, this is not as great a concem, as it was in 1982, when NIST first developed emulation capabilities. Today, an interactive, multi tasking, or object oriented approach is conceivable in a manner which was not possible ten years ago.

A quicker, more user friendly, and cost effective emulator structure is available with today's computing power. The structure, developed as part of this research, divides the emulators into different generic classes, according to the type of cells which they are emulating. These are easily configured to mimic the appropriate characteristics required for the system. Two appropriate classes of emulators for the system control application are identified. The first, workcell emulators, are sub-divided into machining, assembly, and inspection workcells. The second, material handling emulators, are subdivided into AGVs and conveyors.

## 5.2 EMULATOR OPERATIONS

The development of generic emulators began by decomposing the operations of each generic class into the operations performed. Table 5.1 and 5.2 identify the operations which are performed by a machining workcell and an AGV material handling system emulator, respectively. The operations performed, by the other subgroups in each class, are similar enough in nature to preclude the necessity of identifying them in detail. The identification of these operations facilitates the development and characterization of generic emulator functions.

## 5.3 GENERIC EMULATOR FUNCTIONS

With the development of generic functional capabilities, it is possible to easily configure an emulator to respond in a realistic manner to any of the major system control functions, or any other system control functions which a particular application may utilize. Table 5-3 lists the generic emulator functions.

### 5.3.1 READ FUNCTION

The read function interacts with the outside environment of the emulator. It also interacts with other functions internal to the emulator. Upon reading a message, the read function acknowledges the receipt of the message, and passes it to the data analysis function. In the event of an interruption of reading the message, the read function instructs the write function to write a not acknowledged (NAK) message back to system control. After a message has been read and passed to the data analysis function, the message is erased from the communication medium.

### 5.3.2 WRITE FUNCTION

The write function interacts with the other internal functions of the emulator to provide messages back to system control. When the write function sends a message back to systems control, a time stamp is written along with the message. Assuming that the clocks on both system control and the emulator are synchronized, systems control and/or the operator can analyze the time taken by the emulator to process the message. The format of the messages both read and written must be determined during the initial configuration of the emulator.

### 5.3.3 DATA ANALYSIS FUNCTION

The data analysis function decomposes and interprets the messages received from systems control or the operator interface, and determines the appropriate action required by the emulator. Some messages require no further action than providing a direct response back to system control. However, the majority of the messages require the data analysis function to interact with the other emulator functions.

The data analysis function must know which functions perform which actions, and what data the different functions need to perform their respective functions. Data is passed to the appropriate functions, in the form of parameters. The data analysis function also determines if a message received by a cell was intended for that cell. If a wrong message was received, data analysis directs the appropriate response to system control.

### 5.3.4 PROCESS DELAY FUNCTION

The process delay function generates time delays representing processing and transportation times for workorders. Delays are also simulated to represent the time required for AGVs to reach a requesting workcell. Process delays are dependent upon part type and operation. Transportation delays are dependent upon the distance between the cells in the routing. All delays include the time necessary for loading and unloading.

TABLE 5.1 Machining Workcell Operations

1. Read message
2. Acknowledge message
3. Periodically simulate correct message not recieved
   - according to some distribution
4. Analyze message
   - if processing not required simply pass on data
5. Determine processing time
   - look up table for part type and process number
6. Change workcell status
7. Write workcell status
8. Simulate processing
9. Write work completed
   - part number and finished code
10. Check queue
11. Track % time workcell is busy, idle, and down
12. Periodically generate error messages
    - according to some distribution
13. Determine repair time
    - different distributions for different error types
14. Respond to watchdog timer
    - write a simple acknowledgement
15. Track parts in queue
    - maintain number in queue and in which order they should be removed
16. In case of breakdown pass along repair time
17. Wait for signal from system control to begin repair
18. Respond to requests from system control for statistical data
    - % downtime, idle time , busy
19. Write messages to screen as well as mailbox
    - operator interface
20. Respond to keyboard requests for information
    - operator interface
21. Record time messages are read and sent
    - allows you to check the real-time characteristics

TABLE 5.2 (AGV) Material Handling System Operations

1. Read message
2. Write ACK to message
3. Periodically simulate NAK to message
4. Erase message
5. Analyze message
    a. determine part type
    b. determine pick-up point
    c. determine delivery point
6. Decrement number of AGVs available
7. Determine which AGV to send for pick up
8. Determine time to reach pick up point
9. Simulate pick up
10. Write Part retrieved
11. Determine delivery time
12. Simulate delivery time
13. Write part delivered
14. Increment number of AGVs available
15. Track AGV locations
16. Periodically simulate AGV breakdown
17. Track availability of AGVs
18. If multiple requests exist determine order of service
19. Respond to watchdog timer
20. Track parts awaiting pick up
21. Record time messages are read and sent

TABLE 5.3 Generic Emulator Functions

1. Read
2. Write
3. Data Analysis
4. Process Delay/Simulation
5. Data Retrieval
6. Data Generation
7. Data Tracking
8. Statistical Computation
9. Status Maintenance
10. Numerical Computation
11. Operator Interface

### 5.3.5 DATA RETRIEVAL FUNCTION

The next function is the data retrieval function. It retrieves data from internal or external storage. Repair times, processing times, and transportation times are examples of data required by the emulators. This data must be retrieved at the appropriate time, either from data arrays, linked lists or a data base.

### 5.3.6 DATA GENERATION FUNCTION

The data generation function creates several types of data. The first example of this is, it generates pass/fail data for parts passing through inspection workcells. Secondly, it generates error conditions which the emulator sends back to system control. Emulators generate different types of errors at varying frequencies, these are established upon the initial emulator configuration.

Errors that occur comprise two classes, communication errors and application errors. Communication errors deal with incorrect or incomplete data received from system control, and generate a not acknowledge (NAK) response. Application errors are associated with machine tools breaking, parts being dropped, robot shutdowns, etc. These are coded in such a way that the emulator can describe to system control, the type of error which has occurred. The data generation function must incorporate random number generators, and have the ability to generate numbers from distributions, such as exponential and normal. Error generation occurs according to an exponential distribution, defined by a mean of one divided by the failure rate [36].

### 5.3.7 DATA TRACKING FUNCTION

The data tracking function tracks non-stochastic data within the emulators. One example, is tracking the location of the AGVs in the material handling emulator. When a request arrives for an AGV at a particular workcell, this function provides parameters to the numerical computation function to determine the amount of time necessary for the AGV to reach the requesting workcell. The data tracking function also tracks the amount of time that different tools are being used in the workcells.

### 5.3.8 STATISTICAL COMPUTATION FUNCTION

The statistical computation function calculates and tracks statistical data, such as workcell busy and downtime, and workpiece production history. Status change information is received from the status monitoring function, and the statistical computation function updates this information. An integral part of this capacity is the capability of time stamping each command and status change.

### 5.3.9 STATUS MAINTENANCE FUNCTION

The status maintenance function monitors whether the workcell is idle, busy or down. The status of the workcell can be established from either internal or external stimuli. External stimuli exist if the systems control or operator interface direct the workcell to shutdown or reactivate the workcell when a repair has been completed. Internal stimuli exist when the workcell begins processing a workpiece and the cell status becomes busy. Upon completion of processing, if no parts are waiting in queue, the status of the workcell becomes idle. For the material handling class of emulators, the status maintenance function tracks the number of available AGVs.

Requests may come in from system control or the operator interface, concerning the status of the system. This function must have facilities to send the status of the emulator at that time to the requesting entity.

### 5.3.10 NUMERICAL COMPUTATION

The numerical computation function analyzes alternative conditions, when numerical parameters are known about each condition. This function does not actually determine the best alternative; it supplies another function with the results of its calculations, so that an informed decision may be made. An example of this is the assignment of AGVs to workcell requests. Data tracking supplies the numerical computation function with the location of all available AGVs. Utilizing transport times, supplied by the data retrieval function, the time for all available AGVs to reach the requesting workcell can be computed, by subtracting out the load and unload times.

### 5.3.11 OPERATOR INTERFACE FUNCTION

The operator interface function provides interaction to configure certain parameters of the emulator. It responds to direct information requests from the operator. The operator interface function accounts for both read capabilities from the keyboard and display capabilities on the monitor. The operator, through this function, may halt or pause emulator operation, or test the system controls reaction to independent condition in an open loop fashion.

## 5.4 SPECIFIC EMULATOR TASKS

The classification of generic emulator functions provides a basis for developing emulation capabilities to mimic real world scenarios on the factory floor. These scenarios can be broken down into specific tasks which emulators must have the ability to perform. Table 5.4 lists these tasks.

### 5.4.1 COMMUNICATIONS

The first task is a communication task. Both the workcell and the material handeling system emulators must communicate with the system controller. The manner in which communications occur vary from system to system, however, most systems utilize some form of messaging scheme. Messages can be passed through files, via shared memory, or across other advanced communication channels, such as sockets.

Communications capabilities include reading messages, acknowledging messages and writing messages. Messages can also vary in form. The simplest are numerical or character strings which are passed between sender and receiver. A more advanced form of message takes advantage of passing complex data structures which can contain numerous, varying forms of data.

TABLE 5.4 Emulation Tasks

1.) COMMUNICATIONS

2.) MESSAGE DECIPHERING

3.) STATUS MAINTENANCE

4.) PROCESS DELAY

5.) QUEUE MAINTENACE

6.) ERROR GENERATION

7.) ERROR RECOVERY

8.) STATISTICAL COMPUTATION

9.) OPERATOR INTERFACE

### 5.4.2 MESSAGE DECIPHERING

The next major task is message deciphering. It provides the ability to extract information from the messages. This task is highly dependent upon the form of message used. String messages need to be decomposed or compared to known messages to extract the information being passed. When utilizing structures data can be directly extracted from the different structure elements. The simulation of incorrect messages must be incorporated into the emulator, because messages are sometimes received that are not able to be deciphered, due to the addition or loss of data during transmission. The possibility also exists that messages can be sent to the wrong emulator.

### 5.4.3 STATUS MAINTENANCE

Status maintenance is another task. As work flows into and out of workcells, status changes occur. The status of the workcell is either idle, busy processing a part, or down, due to a malfunction. Status must be maintained so that inquires, from system control or the operator, can be responded to in a timely manner. Status maintenance also facilitates keeping statistics on workcell activity.

### 5.4.4 PROCESS DELAY

The process delay task simulates the processing and transportation times involved in producing a part. Processing time is dependent upon the part and operation type. Since an emulator represents all the elements in a workcell, this task takes into account loading time, processing time, and unloading time. Transportation time is dependent upon the parts present locaticn and the next location in the routing. All the pertanant data for determining delay times is passed in the message representing the part.

### 5.4.5 QUEUE MAINTENANCE

The next task is queue maintenance. This task involves tracking and mimicing the queueing process at workcells. Parts often arrive at a cell while its status is busy, queueing allows the part to be unloaded while the cell is still processing another part. This frees workpiece unload stations, which normally only have a much smaller capacity. Also, material handeling, when it is an AGV, is free to perform other tasks. The queue maintenance task also includes, determining which workpieces to remove from the queue, when multiple workorders have accumulated at a busy workcell. Different removal schemes include first-in first-out (FIFO), last-in first-out (LIFO) and priority schemes.

### 5.4.6 ERROR GENERATION

Another major task is error generation. During processing or transporting a workpiece, a number of unforeseen events occur, that would disrupt normal operation. In order to maintain their real world appearence, emulators must mimic the occurence of these unforeseen events. Error generation notifies system control of the time and nature of the disruption, so that appropriate action may be taken. This task simulates all these conditions and actions.

### 5.4.7 ERROR RECOVERY

An error recovery task simulates the repair of a workcell or material handeling device after a breakdown occurs. In a real world system, the operator is notified and plays a varying part in error recovery depending upon its severity. In order to simulate this process, this task accesses and delays appropriate repair times, dependent upon the type of error that occured. This task awaits a signal from system control to initiate the repair process, this simulates the availability of repairmen. The status of the workcell is returned to busy after the appropriate repair delay, and processing continues on the part which was being processed.

### 5.4.8 STATISTICAL COMPUTATION

The statistical computation task has the capability to compute, track and maintain statistical information, concerning the operartion of the cell. This task works hand-in-hand with the status maintenace task to provide system control and the operator with statistics concerning cell operation. Emulators have an internal clock which allow accurate time based statistics to be maintained. This task involves calculating and maintaining means, standard deviations and trends. Examples range from cell utilization to product tolerance variances.

### 5.4.9 OPERATOR INTERFACE

The operator interface task allows direct interaction by the operator. It provides an ability to configure certain parameters and processes information requests from the operator. The emulator can be used to test system control reaction to a specific condition; in this situation the operator configures the emulator to generate that condition. In addition, the operator interface task accounts for read capabilities from the keyboard and displays of information to the monitor.

# 6.0 PROBLEMS AND ISSUES

The spectrum of problems encountered in developing automated manufacturing systems is broad. From one extreme are the mundane problems of connecting the different physical elements in the system together. At the other extreme are cultural problems that many companies encounter when attempting to increase the level of automation within their facilities. Somewhere in the middle are problems of designing and developing the required software to control and integrate these automated manufacturing systems. These software problems are one of the major roadblocks in implementing automated manufacturing systems [25]. This chapter identifies and characterizes these software issues, as they relate to the use of emulation as a problem solving tool.

## 6.1 CENTRAL ISSUES

The problems encountered by software developers, when attempting to automate the system control of manufacturing systems, are integration and control of all the different elements in the system. Some problems appear early in the conceptual development stage, while others do not surface until late in the implementation portion of the project. The central issues are complexity, hierarchy, uncertainty, capacity, and communication.

### 6.1.1 COMPLEXITY

The complexity of the system control problem may be the source of all problems encountered. Complexity problems surface early in the conception stage, and continue through implementation. As was identified in Chapter 4, a number of different functions are involved in controlling an automated manufacturing system. These functions must work together to provide the best overall system performance possible. In complex systems the interaction of the functions can become muddled, especially when the functions are not clearly defined. Clearly defined functions implemented in a modular fashion provide a level of order to the complexity issue.

Many complexity problems arise as a result of poor planning at the initial stages of automated system development. A well defined plan, documenting the system functions and goals, is imperative to providing system designers a basis for establishing clearly defined flexible control schemes.

### 6.1.2 OPTIMIZATION

Optimizing system performance is an other issue designers must contend with. The best overall system performance is difficult to define because so many elements exist in a manufacturing system. It is nearly impossible to simultaneously optimize all the different parameters that define system performance. If developers took inputs from different departments in the corporation concerning which parameters should be optimized, an overwhelming list would result. System developers tend to sub-optimize different portions of the system, according to management's views of the most important performance measures.

Once management has identified the desired performance measures, the developer's problems are far from solved. Usually through trial and error, the best combination of algorithms and implementation methods of the different functions must be found, to optimize the system. Often, the best algorithms during conception do not perform adequately once coded, due to other limitations of the system, such as the hardware or communication constraints.

### 6.1.3 HIERARCHY

The hierarchical architecture of most automated manufacturing systems is another issue which presents problems closely related to those of complexity. Two major concerns are how decisions are made at the system control level, and what effect the operations of the elements will have on the other levels of the hierarchy. These problems do not become evident until the system control software has been coded and linked with the other levels in the hierarchy.

One example of such a problem is that data requirements at the system control level may not be accounted for at the cell level. Further data requirements of the cell controller may be detrimental to the efficient operation of the cell. Another example is a scheduling and tooling problem. The manner in which work orders are scheduled into the system is determined at the system control level. However, the schedule may have a direct, yet hidden, effect upon workcell control that will ultimately alter system control. Scheduling work orders of different product mix may require excessive amount of tool changes within the workcell. The schedule may have appeared acceptable at the system control level, but if it requires a large amount of time for tooling and setup changes, the performance of the system may degraded. The scheduling problem now becomes choosing when these tooling changes should occur [21].

### 6.1.4 UNCERTAINTY

Uncertainty relates to unpredictable random disturbances which may occur in the system. Examples include breaking down of machine tools and AGVs, shutting down workcells due to breaking a safety barrier, and failing to have required material reach its end destination.

The system control developer must identify all the different random error conditions or uncertainties which may occur in the system. Then how these uncertainties affect system control must be established. Lastly, how the system will recover from all the different uncertainties is determined.

The identification of random disturbances within a system is the most difficult task. The system developer has to rely upon poor documentation of disturbances, that have occurred in the manual system. The best source for this information is often intuition and the experience of others. It is unrealistic to assume that all possible random disturbances will be accounted for during the initial stages of development. Therefore, this process of dealing with uncertainties will plague the system developer throughout the design cycle, and on into implementation.

Once the uncertainties have been identified, they can be accounted for during the design of the system control software. Designing the system control software to be affected by such disturbances, in the least possible amount, is referred to as desensitization of the control system. In most cases, these disturbances will always have some effect on a subset of the functions within system control. The identification of these functions, is paramount to the last task.

Devising error recovery schemes for all the possible random disturbances is the last task. Formulating error recovery schemes involves identifying the problem, notifying the operator, altering necessary control functions, and possibly, running error recovery programs for simple disturbances. Once the disturbance has been resolved, system control must take the appropriate action to bring the system back to a steady state.

### 6.1.5 CAPACITY

All equipment within the system has a fixed capacity. System control must not execute schedules which exceed this capacity. At the same time, workcells and elements beneath system control must keep system control informed of their capacity. While they may have a fixed capacity, the equipment's capacity may change due to unforeseen interruptions. Development of a scheduling method to account for these unforeseen interruptions is a problem faced by system control developers. The capacity constraint issues can not be fully accounted for, until the system control software is linked to the elements beneath it.

Another related problem is system control's handling of facility control's alterations in the system requirements. Alterations to schedules already developed may be necessary, or if changes are excessive, system control reports back to facility control that the desired work cannot be achieved. System capacity cannot be overloaded, because this may have a detrimental effect upon throughput, which is, in most cases, one of the system performance parameters of major concern.

### 6.1.6 COMMUNICATIONS

The issue of communication deals with both internal and external communication problems. Internal communications involves how the functional elements within system control communicate appropriate data to each other. This problem has been well researched and documented in the literature by others, and is beyond the scope of this research.

External communications involves the problems associated with system command interactions with the environment. This applies to other levels in the hierarchy and the operator. Most automated manufacturing systems communicate over a network. Network communications problems range, from speed and protocols, to message translation and reliability. System control developers must contend with all these problems, and determine how they affect the operation of the system. Also, the interaction between the system controller and the operator is considered. The format and the means of portraying system data to the operator must be established and tested. Some of this information can be obtained by speaking with operators; however, the developer will not know until the system is operational, if the data is relevant and portrayed in an adequate manner.

## 6.2 EMULATION AND THE SYSTEM CONTROL DESIGN CYCLE

As has been discussed earlier, not all the problems which are encountered by system control developers, lend themselves to the use of emulation as a problem solving tool. The best way to categorize these problems into a subset where emulation can be used to help reach a solution, is to first relate the use of emulation to the system control design cycle. Figure 6-1 shows the system control design cycle [29].

### 6.2.1 CONCEPTION AND DESIGN

Conception is the first stage of the design cycle. During this stage, ideas for an automated manufacturing system are developed and cultivated into the specifications for the future system. Conception occurs just prior to bringing management "on-board". During the design stage, specific strategies are chosen to meet the system specification. The initial game plan is developed, and all the affected departments discuss and concur upon it. During these initial stages, the form of the final system is not always known. In fact, this is part of the second stage, design; establishing the system according to the goals of the company. Emulation is not the optimum tool for assisting the designers during this portion of the design cycle, since the underlying premise of emulation requires existing control code to drive the emulated physical system. At this stage system configuration and control software are still in the infancy of their development, designers are still playing "what-if" games to reach a consensus on the structure of the system. For these purposes, simulation is an excellent tool for system control designers [29].

### 6.2.2 FABRICATION

The third stage of the design cycle is fabrication. Emulation would provide a useful tool to the system designers in this stage. Designers develop the strategies and system architectures into software code to control the different elements of the system. This is the critical stage, where software needs to be debugged, tested and verified, before it is fully

CLASSES OF CONTROL PROBLEMS

1.) TIMING and SEQUENCING

2.) COMMUNICATIONS

3.) ERROR RECOVERY

4.) RESPONSE DATA

FIGURE 6.1 System Control Design Cycle

operational. After the initial syntax error debugging, the designer develops a test scheme to verify that all the functional elements of the code are operational and robust. Emulation provides a tool, or a platform, which will mimic the operations and communications of the different elements of the system. This tool allows the system designer to fully test the control logic for the entire system, without the actual hardware of the system. This is particularly useful for two reasons. If the hardware is not yet available, the system control design is not delayed. More importantly, if the hardware does exist and is being used manually, production does not have to stop while the software is being tested and debugged.

### 6.2.3 INSTALLATION

The fourth phase in the design cycle is installation, where control logic errors are normally encountered. This is typically one of the longer phases. Emulation would greatly reduce the time spent in this phase, since the control logic errors can be debugged prior to installation. Emulation is used during installation to verify that the physical installation matches the design. It is used to compare the "as specified design" with the actual physical system. Performance measures of the emulated system should match those of the physical system, less discrepancies from random disturbances. If these two do not correspond, then, either the installed system needs to be corrected or the emulation model needs to be changed, to reflect the physical installation [29].

### 6.2.4 OPERATION

The last of the design phases is the systems operation. By this stage, all the bugs have been eliminated from the system, and the implementation of the control software has been tested and verified. Since the emulation model was refined to correspond to the physical system in the installation phase, it can be used as a diagnostic tool during the operation phase. By running the emulation in real time, parallel with the physical system, each system can be continuously compared for discrepancies. Tolerances can be set for variation due to random disturbances. Outside of these variations, any discrepancy signifies an error condition in the operational system [29]. In addition, if there are error conditions, which are continuously causing production disturbances, these can be reconstructed on the emulation model. This allows the testing of control software upgrades, without shutting down the production.

## 6.3 EMULATION CLASSES OF PROBLEMS

Emulation is best suited, as a tool or platform, for system testing during the latter stages of the design cycle. In particular, the greatest savings will result from its use as a test-bed during the fabrication phase of the design. The types of problems which emulation is best suited for can be grouped into classes. The classes of problems which lend themselves to emulation are listed in Table 6.1.

### 6.3.1 TIMING AND SEQUENCING

Timing and sequencing take into account all potential errors associated with moving workorders through the system. This includes requesting and sending AGVs to the appropriate workcells at the appropriate times, tracking multiple workorders through the system, and handling and accounting for work in process (WIP). It also accounts for errors associated with how, and when, workorders are loaded into the system.

### 6.3.2 COMMUNICATION

Associated with communications problems are issues relating to system command interactions with the environment. This applies to other levels in the hierarchy, as well as to the operator. Since most automated manufacturing systems communicate over a network, some of the problems encountered will be related to network communications. These problems range from adequate transmission speed and protocol discrepancies, to message reliability.

The interactions between the system controller and the operator are not only functional problems, but also relevancy problems. The format and the type of data being portrayed to the operator must be adequate enough to monitor and troubleshoot the system efficiently. Some of this information can be gathered by speaking with operators during the development phase; however, the developer will not know until the system is operational, if the data is relevant and portrayed in an adequate manner.

The use of emulation allows the developer to test and verify all communication issues prior to installing the system control software on the factory floor. Altering the software after installation is a complex and risky task, as small changes may proliferate problems in other areas.

### 6.3.3 ERROR RECOVERY

Once error recovery schemes have been coded, they are a risky and costly portion of the software to test. It is not desirable to actually create the vast majority of the errors, since

TABLE 6.1 Classes of Control Problems

doing so may involve breaking or destroying equipment. Emulation is an effective tool for simulating the error conditions expected on the factory floor. These routines can be debugged, tested, and verified, without risk to an operating system.

### 6.3.4 RESPONSE DATA

Problems associated with response data are closely associated with communications. However, analysis of the messages received by system control are beyond the scope of communication problems. System command must recognize messages, then determine their meaning, and generate appropriate responses back to the workcells. These tasks can be effectively tested using emulators to represent the workcells generating the messages. Emulators can be structured so that they only recognize certain commands, and display the actions taken in response to those commands. This allows the developer to see the action, generated by the commands, sent to the different cells.

## 6.4 EMULATION MODES OF OPERATION

The manner in which emulation is used can take different forms. These forms, or modes of operation are dependent upon the class of problem and where in the system design cycle the developer is using emulation. The two modes of operation are partial and full. In the partial mode emulation is used to test specific, known conditions. In the full mode emulation is used test the operation of the whole system and to help uncover unforeseen problems.

### 6.4.1 PARTIAL MODE

In the partial mode of operation, the emulator is controlled via the operator interface. This mode is utilized to check the operation of system control, with respect to specific conditions, including reaction to workcell error messages, response to work being completed, and response to communications errors. The partial mode's intent is to verify individual functions of system control, prior to testing it as an integrated system. When it is run as an operational system, specific error responses will not cloud integration/timing problems that result when the entire system is coordinated.

### 6.4.2 FULL MODE

In the full mode emulators for the various cells operate autonomously, responding to system control without operator interference. This verifies the ability of system control to operate the entire system. Simulated work orders are processed through the system, and encounter the same conditions that can be expected on the factory floor. Error conditions and processing times, including transportation and loading times, are emulated providing system control with multiple responses at varying times. Developers will be able to recognize many unforeseen problems as the system is run in the full emulated mode.

## 7.0 SOFTWARE DEMONSTRATION

The final portion of this research was the development of a software demonstration program. The purpose of this demonstration was twofold. First, it provided experience in developing control and emulation software. Secondly, it showed that the concept of emulation, as it applies to automated manufacturing control systems, is a viable tool in debugging control software.

## 7.1 HARDWARE DESCRIPTION

The emulation and control software were developed on IBM RT workstations. These workstations operate under the AIX operating system. AIX is a Unix like operating system based on System V Unix. The RT workstations were connected via a Token Ring network. Five workstations were used, one ran the system control software, while the other four ran emulators representing three separate machining workcells and a material handling system. Figure 7.1 illustrates the demonstrations hardware configuration.

## 7.2 SOFTWARE DESCRIPTION

All of the code for the demonstration was developed in the general purpose programming language, C. There are many advantages to using C. It is compact, yet powerful. It allows for low level development, such as addressing of memory. It is easily portable compared to other languages. It encourages modular design and can be extended. Also, one of the major reasons for its use in this research was that the Unix and AIX operating systems are written

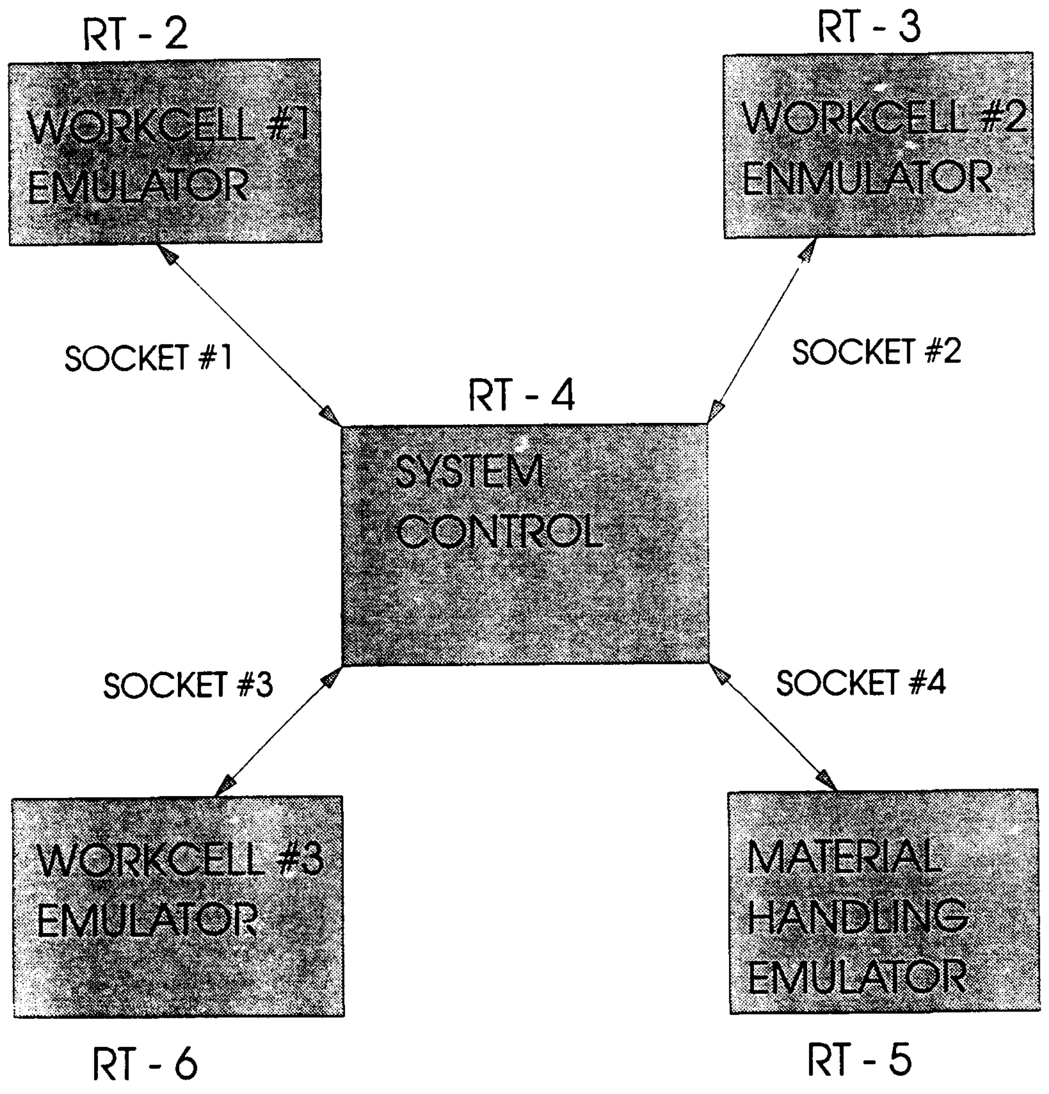

**FIGURE 7.1 Demonstration Hardware Configuration**

in C. By writing the demonstration in C, AIX system calls were able to be utilized. These included features for communications, multi-tasking, and signals.

### 7.2.1 MULTITASKING FEATURES

Both Unix and AIX support multitasking environments. Multitasking allows multiple processes to be executing, what appears to the user to be simultaneously, on the same machine. In reality, most multitasking environments are achieved via time slicing. Time slicing is a concept of dividing CPU time between multiple processes. Each process runs a predetermined amount of time. The process has access to the CPU during this time slice. This multitasking feature was utilized in the software demonstration for related processes, particularly in the material handling emulator.

Two of the main multitasking system calls utilized were fork and exec. Fork allows a process to make a copy of itself, resulting in two processes executing in the same environment. When a fork system call is executed, two process identifiers are returned. One process identifier, that returned to the parent process, has a value greater than one. This is the process identifier of the child process. The other process identifier is zero. This process identifier is returned to the child process. By returning a process identifier of zero to the child process, a test can be made to differentiate between the child and parent process.

The exec system call is utilized in conjunction with the fork system call. Exec allows the child process to discard all the parent code proceeding it, and execute a new program provided as an argument. The new program must be in executable form, and is laid over the old code in the child process. The new child process retains all process IDs, time left on alarms (discussed later with signals), and directory information.

### 7.2.2 COMMUNICATION FEATURES

A number of AIX communication features were utilized in this demonstration. Namely these were pipes, sockets, and select. Pipes and sockets are means of interprocess communications, while select is a polling feature which determines when communications are enabled on pipes or sockets. Figure 7.2 illustrates the communication links used in the demonstration.

Pipes are one way communication channels, between processes running on the same host. A processes is an instance of a program that is being executed by the operating system [55]. Since Unix and AIX allow multiple processes to be executing simultaneously. One method of communicating between these processes is via pipes. Pipes are initiated by one process, usually the parent process. The connection is made by passing the child process the file descriptor of the other end of the pipe. Figure 7.3 illustrates communications with pipes. Pipes were utilized in the software demonstration to communicate between AGV processes and the material handling emulator.

Sockets are another communication feature utilized in the software demonstration. Unlike pipes, sockets allow interprocess communications between processes running on different machines. Sockets were the means of communication between the system control software and the emulators on their respective workstations. Sockets are established, based upon a client-server model. The server creates a socket, and then allows the clients to connect to it. Once a client has connected to a socket, the server end of the communication channel is given a new, unique file descriptor. This allows other clients to connect to the server. Sockets also differ from pipes in the sense that they allow bi-directional communications.

The select system call is a means of identifying when a file descriptor is ready for reading or writing, particularly reading. In Unix, select can be used with either sockets or pipes. Unfortunately, AIX only supports the select system call for interprocess communications between processes residing on the same host. For this reason select was used by the material handling emulator to identify when AGVs wanted to communicate with the material handling control. Select works by first assigning a mask bit to each file descriptor of concern. When it is called, select can either wait a predetermined amount of time for a message to appear on the file descriptors masked, or it can be configured to return immediately whether a file descriptor is ready for reading or not. The masked file descriptors are then tested to see if a flag was set, indicating data is present for reading.

### 7.2.3 SIGNAL FEATURES

Unix and AIX also provide a signal feature that was utilized in the software demonstration. The signal system call allows actions to be predefined when different signals occur. In particular the software demonstration utilized an alarm signal to simulate different error conditions. An alarm can be set to occur at a fixed time after processing begins. Using the signal system call, this alarm can cause an interrupt to any processing and execute another section of code. Alarms simulated AGV breakdowns, tools breaking in a workcell, and outside disruption to a cell.

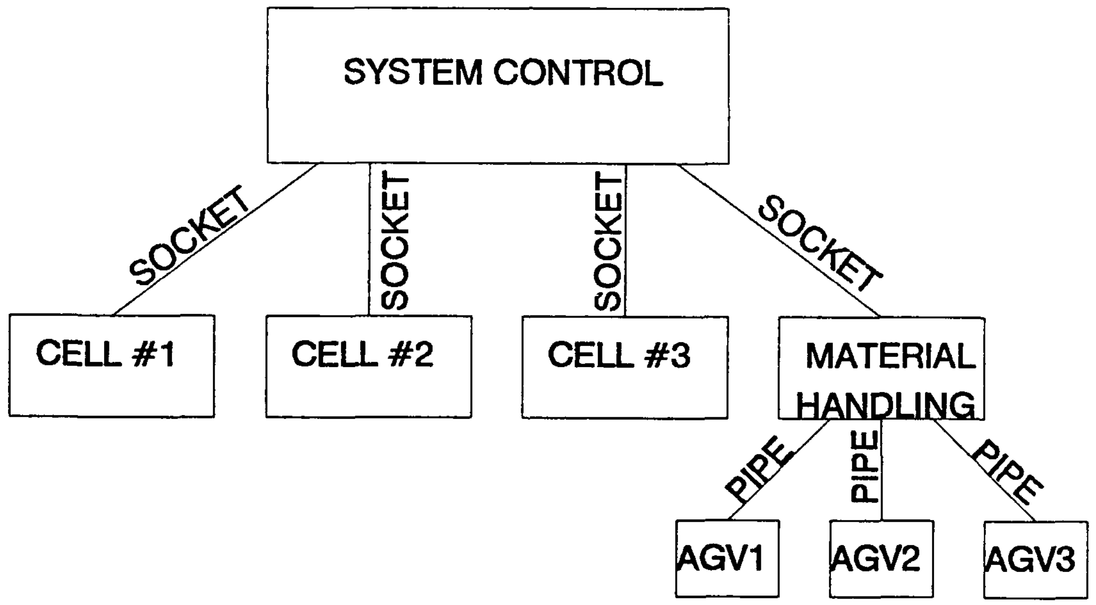

Figure 7.2 Communication Links

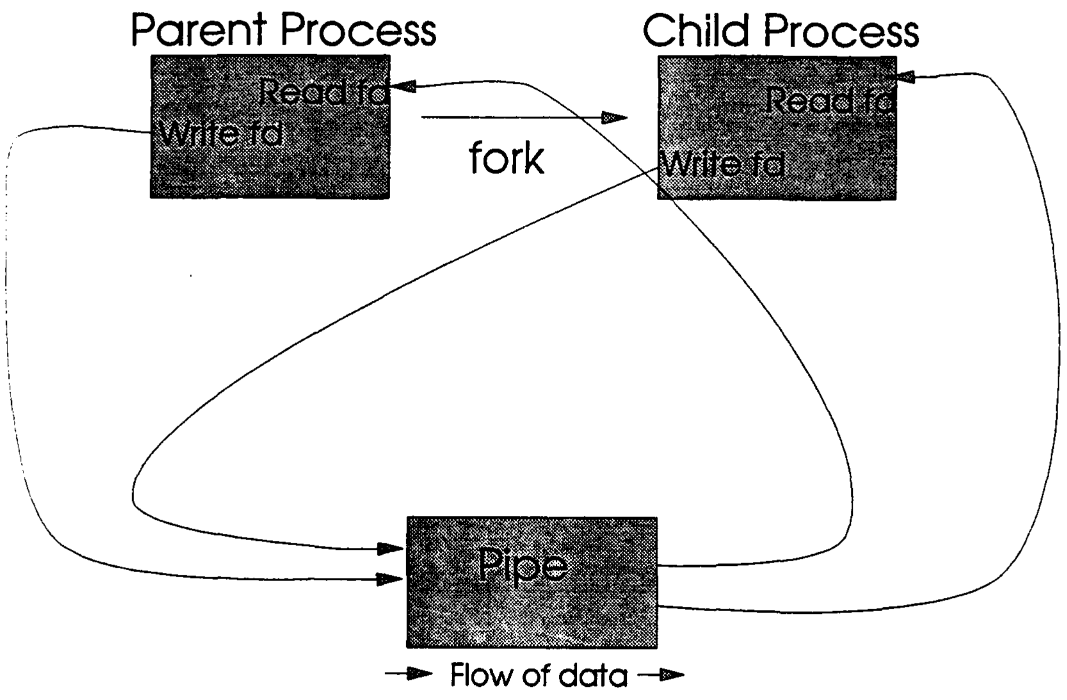

**Figure 7.3 Pipe Concept**

## 7.3 SPECIFIC ELEMENTS

The software demonstration is comprised of three different elements: system control, material handling emulator, and workcell emulators. The three workcell emulators are virtually alike, the only difference being that workcell number three can process workcell number one or number two parts. However this is accounted for in system control not the emulator. Each of the elements is built up in a modular fashion in an attempt to make them as generic as possible. The features of each element are described below. Each element utilizes a common header file INET.H, which can be found in the appendice.

### 7.3.1 SYSTEM CONTROL

A mock system control program was developed to represent a typical program to be tested with the emulators. The system control program was comprised of three programs which need to be compiled and linked together. They are system control, system communications, and system scheduling. These programs are found in the appendices listed under SYSCTR.C, SYSCOM.C AND SYSCED.C respectively.

The socket communication feature used by system control is unique so it was written as a separate process. SYSCOM.C establishes a server socket and waits for the predetermined four emulators (clients) to link to it. The file descriptors for the four emulators are located in an integer array defined as cell[5]. This was considered an initialization process and can be used by any system control program, utilizing Unix or AIX system calls, since the results are file descriptors which the read and write functions take as arguments. One requirement is that the system control communication process, SYSCOM.C, must be executed prior to the emulators, due to the client-server relationship of sockets. It is also important to note that workstation address for the host running the system control software must be identified in INET.H. These addresses can be found on the network server, in the HOST.H file.

The other two programs SYSCTR.C and SYSCED.C are not generic, but represent some of the system control functions discussed in Chapter 4. The system control program, SYSCTR.C, receives its "work-to" list from the keyboard. This represents facility control, which has been identified as the upper level of the control hierarchy, normally supplying this information. It is quite possible for facility control to have been represented by an emulator, however one of the limiting factors in the demonstration was the number of operational workstations connected to the token ring network. Therefore,the operator supplied the required amounts of each product type. The scheduling program was very simple, but was written as a separate program for modularity purposes.

### 7.3.2 MATERIAL HANDLING EMULATOR

This program was the most complex of all the elements of the demonstration. It too, was written as three separate programs. These programs can not be considered generic, since AIX system calls and communication features were utilized which may not be present in all facilities. However, for the purposes of this demonstration and within the confines of Unix operating systems two of programs, the workcell communication program and the process delay program are generic. These are listed in the appendices as WCCOM.C and DELAY.C respectively.

These two programs were used by all four emulators in the same form. WCCOM.C is the client equivalent of socket communication feature discussed in the previous section. It links itself to the server socket and returns a file descriptor identifying the client end of the socket. DELAY.C simulates the process and transportation delays necessary for the emulators. DELAY.C takes as an argument the number of seconds to delay, and returns any time remaining in the case that an alarm occurred. The other program is the material handling emulator, AGVEM.C, it emulates the control of three AGVs to transport workpieces between the three cells. Code associated with this program can also be found in the appendices. The AGV emulator is comprised of a parent process which initializes communications and receives messages from system control. It creates a child process which receives the message read by the parent and controls three AGVs represented by grandchildren processes. The child is also responsible for sending messages back to system control. This structure was utilized due to the AIX limitation on only allowing the select command to work with pipes. Figure 7.4 illustrates this structure and the associated communication links and file descriptors.

### 7.3.3 WORKCELL EMULATORS

The three workcell emulators are coded in the same manner the only difference being their workcell identification numbers. The workcell emulators also utilized the previously mentioned programs WCCOM.C and DELAY.C. The code for the three workcell emulators can be found in the appendices under WC1EM.C WC2EM.C and WC3EM.C respectively. The workcell emulators receive messages from system control with appropriate data to determine how long of a process delay to emulate prior to responding back to system control. The emulators continue execution until an appropriate shutdown message is received from system control.

The workcell emulators in addition to the material handling emulator do not utilize all the generic functions or tasks identified in Chapter 5. However for the purposes of the demonstration they provide an adequate illustration of error generation and process delay. The emulators access information from messages passed to them by system command in the form of a structure. The elements of this structure are listed in Table 7.1. Upon completion of processing, or in the event of an error, this same structure is passed back to system control. Using structures for message passing was a significant advancement over previous control schemes used in the manufacturing labs at Virginia Tech.

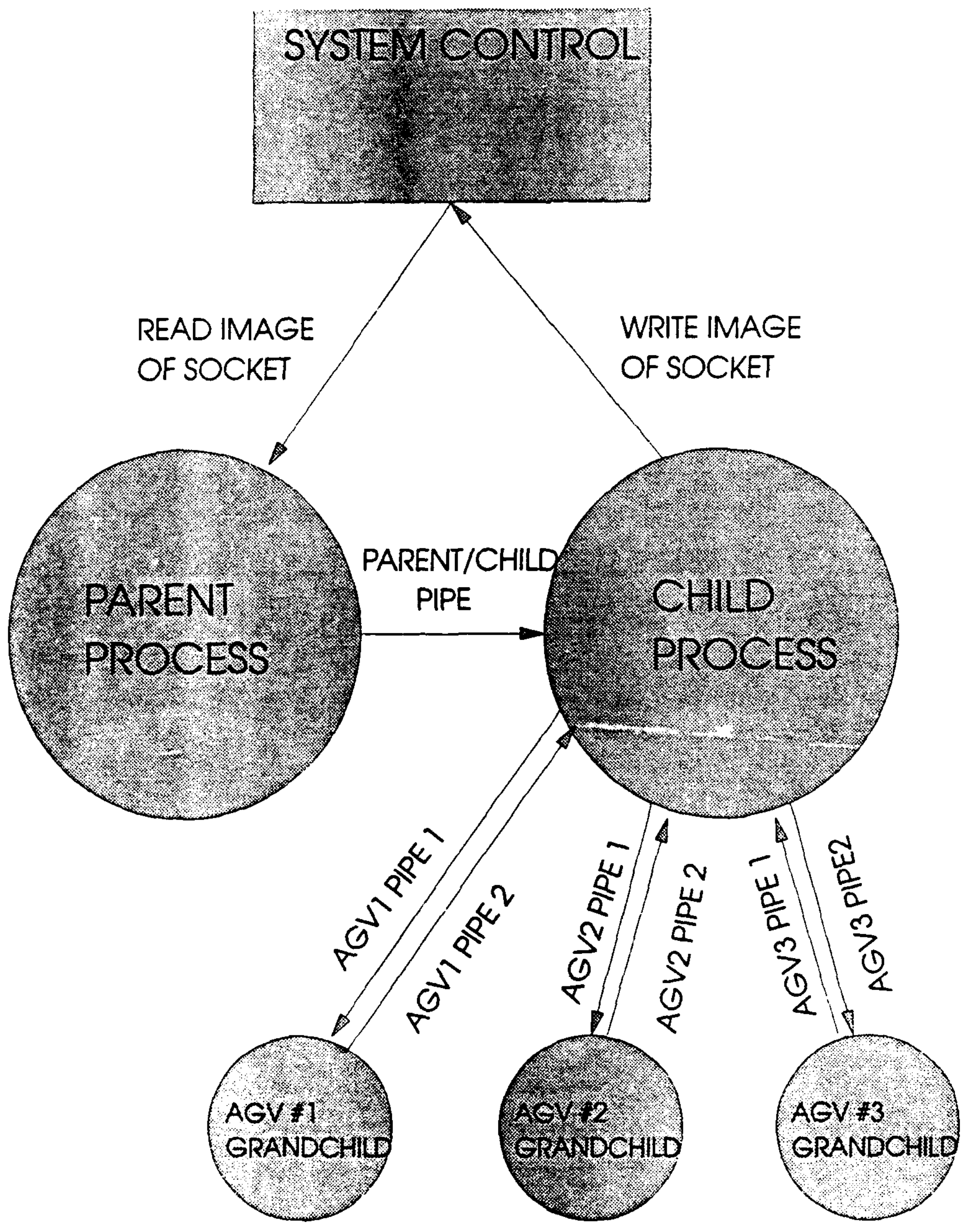

**FIGURE 7.4 Material Handling Emulator Structure**

TABLE 7.1 Message Structure

```
STRUCT COMMAND {
        int cell_id;
        int part_num;
        int op_num;
        int prior_num;
        int error_num;
        int part_type;
        int from;
        int to;
        int ack;
        int agv_num;
        int agv_cod;
}
```

## 8.0 CONCLUSION

The results and reccommendations that follow, are based on a survey of the literature, knowledge gained from the development of the framework, and experience in coding the software demonstration.

## 8.1 RESULTS

The results of this research support the application of emulation to high level control systems. Emulation can provide a significant advancement in the method used for testing and debugging system control software for automated manufacturing systems. This research identified a standard set of generic functions which can provide emulation capabilities for any foreseeable system control application. These lay the foundation for a user friendly approach to emulation, uncomplicated by data restrictions, language requirements, or theoretical background.

The knowledge gained during the course of this research suggests that future system control applications would benefit greatly from the use of Unix operating systems. These operating systems are consistent with the direction of advanced computer applications. The functional framework identified in this research is applicable to any operating system. However, implementation would be easist in the Unix environment.

A hierarchical control model was used as the basis for this research, due to its wide acceptance in the industry. This does not preclude the use of emulation for more distributed control schemes. On the contrary, distributed control schemes may benefit more from this technology, as they are less widely used and potential problems are not as clearly defined. Utilizing emulation techniques prior to implementation could greatly reduce skepticism and further the development of this and other advanced control schemes.

## 8.2 RECCOMMENDATIONS

Reccommendations resulting from this research point to the development of generic emulator generators. The functions outlined herein provide the basis for such a tool. An emulation generator should be a user transparent tool, allowing a user to build an emulation capability for his/her application with a minimal amount of background in emulation. System parameters should be queried from the user in a simple menu format. These parameters should supply the arguements to the generic functions and result in an emulation which accurately depicts the real-world scenario.

A further development would include expanding the domain of emulation to provide intelligent debugging capabilities. By incorporating a knowledge base into the emulation capability, control problems might be flagged and reccommendations provided. Current applications of emulation, still requires a knowlegable control engineer to identify what is causing the system to behave in possibly undesireable manners. Introducing a knowledge base would be a step toward reducing this dependency.

## REFERENCES

1. Bakker, H., "DFMS: A New Control Structure for FMS," Computers in Industry, pp. 1-9, Vol. 10, 1988.

2. Bey, I., "Major Projects on Control Systems for Discrete Parts Manufacturing in the Federal Republic of Germany," Proceedings of the IFAC International Symposium, pp. 331-338, 1977

3. Biemans, F.P. and Vissers, C.A., "A Systems Theoretic View of Computer Integrated Manufacturing," International Journal of Production Research, pp. 947-966, Vol. 29, No. 5, 1991.

4. Bell, R., Roberts, E.A., Shines, N., Newman, S.T., and Khanolkar, R., "A System for the Design and Evaluation of Highly Automated Batch Manufacturing Systems," Proceedings of the UK Research in Advanced Manufacturing Conference, pp. 85-90, December 1986.

5. Berry, G.L., "The Role of Shop Floor Information in Manufacturing Control," Proceedings of the Fourth IFAC/IFIP Symposium, Maryland, pp.119-124, 1982.

6. Bigou, J.M., Courvoisier, M., Demmou, H., Desclaux C., Pascal J.C., and Valette, R.J., "A Methodology of Specification and Implementation of Distributed Discrete Control Systems," IEEE Transactions of Industrial Electronics, pp. 417-421, Vol. IE-34, No. 4, November 1987.

7. Bloom, H.M., Furlani, C.M., Barbera, A.J., "Emulation as a Design Tool in the Development of Real-Time Control Systems," Proceedings of the 1984 Winter Simulation Conference, pp. 627-636.

8. Buzacott, J.A. and Shanthikumar, J.G., "Models for Understanding Flexible Manufacturing Systems," AIIE Transactions, Vol. 12, No. 4, pp. 339-349, Dec 1980.

9. Chang, C.H., "A New Perspective on Realization of Computer Integrated Manufacturing," Manufacturing Review, Vol. 2, No. 2, pp. 82-90, 1989.

10. Charles Stark Draper Laboratory, Flexible Manufacturing Handbook, Noyes Publications, 1984.

11. Clark, G.M. and Withers, D.H., "Architecture for an Integrated Simulation/CIM System," Proceedings of the 1989 Winter Simulation Conference, pp. 942-948.

12. Cloud, K., "A Framework for using Concurrent Processors to Emulate a Distributed Multi-Process System," Proceedings of the Annual Pittsburgh Conference, Vol. 21, Pt 3, pp. 1099-1103, 1990.

13. Co, H.C. and Chen, S.K., "An Automation Laboratory for MBA Education in Advanced Manufacturing Management," Computers & Education, Vol.13, No. 3, pp. 255-263, 1989.

14. Costa, A. and Garetti, M., "Design of a Control System for a Flexible Manufacturing Cell," Journal of Manufacturing Systems, Vol. 4, No. 1, pp. 65-84, 1985.

15. Courvoisier, M., Bigou, J.M., Valette, R., Desclaux, C., and Benzakour K., "The S.E.CO.I.A. Project," Proceedings of the 6th European Conference on Electrotechnics - EUROCON 84, pp. 1-4, September 1984.

16. Courvoisier, M., Valette, R., Pascal, J.C., Barbolho, D., Baudin, Y. and Benzakour, K., "Distributed Emulation of Flexible Manufacturing Systems," Proceedings - IECON '87, pp. 957-964, November 1987.

17. Davis, W.J. and Jones, A.T., "A Functional Approach to Designing Architectures for CIM," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 19, No. 2, pp.164-174, March/April 1989.

18. DeYoung, M. and Tal, J., "Using Personal Computers and PLCs in Simulation Systems," Proceedings of the 1990 ISA Conference, pp 59-61, 1990.

19. Duffie, N.A., Chitturi, R. and Mou, J.I., "Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities," Journal of Manufacturing Systems, Vol. 7, No. 4, pp. 315-327, 1988.

20. Erickson, C., Vandenberge, A. and Miles, T., "Simulation, Animation and Shop Floor Control," Proceedings of the 1987 Winter Simulation Conference, pp. 649-652.

21. Gershwin, S.B., Hilderant, R,R., Suri, R. and Mitter, S.K., " A Control Perspective on Recent Trends in Manufacturing Systems," IEEE Control Systems Magazine, Vol. 6, No. 2, pp. 3-14, April 1986.

22. Godio, C. and Vignale, A., "Plant Emulators for Control Software Test," Proceedings of 3rd International Conference on Simulation in Manufacturing, pp. 137-147, November 1987.

23. Groover, M.P., Automation, Production Systems, and Computer Integrated Manufacturing, Prentice-Hall, Inc., 1987.

24. Gupta, R. and Kaplan, H., "Resolving the Planning and Control Dilemma in Modern Manufacturing," Computers and I.E., Vol. 11, No. 1-4, pp. 261-265, 1986.

25. Hadj-Alouana, N.B., Chaar, J.K. and Naylor, A.W., "The Design and Implementation of the Control and Integration Software of a Flexible Manufacturing System," Proceedings of the First International Conference on Systems Integration, pp. 494-502, 1990.

26. Harbison, S.P., and Steele, G.L. Jr., C: A Reference Manual, Prentice Hall, 1991.

27. Harhalakis, G., Lin, C.P., Hillion, H. and Moy, K.Y., "Development of a Factory Level CIM Model," Journal of Manufacturing Systems, pp. 116-128, Vol. 9, No. 2, 1990.

28. Harmonosky C. M. and Barrick D. C., "Simulation in a CIM Environment: Structure for Analysis and Real-time Control," Proceedings of the 1988 Winter Simulation Conference, pp. 704-711.

29. Hitchens, M.W. and Ryan, T.K., "Direct Connect Emulation and the Project Life Cycle," Proceedings of the 1989 Winter Simulation Conference, pp. 843-847.

30. Hutchinson, G.K., "The Control of Flexible Manufacturing Systems: Required Information and Algorithm Structures," Proceedings of the IFAC International Symposium, pp. 285-292, 1977.

31. Ito, Y., "Conceptualizing the Future Factory System," Manufacturing Review, pp. 252-258, Vol. 1, No. 4, Dec 1988.

32. Jaikumar, R., "Postindustrial Manufacturing," Harvard Business Review, pp. 69-76, Nov-Dec 1986.

33. Johnson, T.L., Milligan, S.D., Fortmann, T.E., Bloom, H.M., McLean, C.R. and Furlani, C., "Emulation/Simulation of a Modular Hierarchical Feedback System," Proceedings of the 21st IEEE Conference on Decision & Control, pp. 360-361, December 1982.

34. Jones, A.T. and McLean, C.R., "A Proprosed Hierarchical Control Model for Automated Manufacturing Systems," Journal of Manufacturing Systems, Vol. 5, No. 1, pp. 15-25, 1986.

35. Kahn, F.U., "Development of a C-Based Simulation Toolkit Supporting Discrete, Continues, and Combined Simulation," Masters Thesis, Va Tech, 1991.

36. Kimemia, J. and Gershwin, S.B. "An Algorithm for the Computer Control of a Flexible Manufacturing System," IIE Transactions,Vol. 15, No. 4, pp. 353-362, 1983.

37. Kochan, S.G. and Wood, P.H., eds., UNIX Networking, Hayden Books, 1990.

38. Kochlar, A.K., "Advances in Computer Software Packages for Manufacturing Control," Proceedings of the Fourth IFAC/IFIP Symposium, pp. 133-140, 1982

39. Larin, D.J., "Cell Control: What We Have, What We'll Need," Manufacturing Engineering, pp. 41-48, Jan 1989.

40. Meredith, J.R., "Automating the Factory: Theory and Practice," International Journal of Production Research, Vol. 25, No. 10, October 1987

41. Mikes, S., UNIX for MS-DOS Programmers, Addison-Wesley, 1989.

42. Nof, S.Y., Whinston, A.B. and Bullers, W.I., "Control and Decision Support in Automatic Manufacturing Systems," AIIE Transactions, Vol. 12, No. 2, pp. 156-167, Jun 1980.

43. Odajima, T. and Toril, T., "Functional Modeling of Cell Controller in Computer Integrated Manufacturing Systems," Proceedings of the IEEE/CHMT '91 IEMT Symposium, pp. 105-109, 1991.

44. Ogilvie, J.W.L., Advanced C Struct Programming: Data Structure Design & Implementation in C, John Wiley & Sons, Inc., 1990.

45. O'Grady, P.J., Controlling Automated Manufacturing Systems, Kogan Page Ltd., 1986.

46. Quinn, E.B., "A Simulation Based System for Automatic Development and Testing of AGV Control Software," Proceedings of the 3rd International Conference on AGV Systems, pp. 219-227, 1985.

47. Ramchandran, J., "Postindustrial Manufacturing," Harvard Business Review, pp. 69-76, November-December 1986.

48. Ricci, L.P., "Application of Dynamic Simulation for Control System Development," Proceedings of the 11th Annual Advanced Control Conference, pp. 81-84, 1985

49. Rochkind, M.J., Advanced Unix Programming, Prentice-Hall, Inc. 1985.

50. Scott, H.A., Davis, R.P., Wysk, R.A. and Nunnally, C.E., "Hierarchical Control Model for Automated Manufacturing Systems," Computers & Industrial Engineering, Vol. 7, No. 3, pp. 241-255, 1983

51. Schutz, W., "A Test Strategy for the Distributed Real-Time System MARS," IEEE COMPEURO 1990, pp. 20-27, 1990.

52. Shin, K.G., Throne, R.D. and Muthuswamy, Y.K., "Communication and Control in an Integrated Manufacturing System," Proceedings of the First Annual Workshop on Space Operations Automation and Robotics (SOAR 87), pp. 405-411, 1987.

53. Siggard, K. and Alting, L., "Methodology for Test and Validation of Shop Floor Control Systems," Factory Automation and Information Management, pp. 952-960.

54. Sohlenius, G., Hjelm, S. and Landsell, G., "FMS - Research and Industrial Development in Coordination," Robotics and Computer Integrated Manufacturing, Vol. 4, No. 1/2, pp. 271-276, 1988.

55. Stevens, W.R., Unix Network Programming, Prentice-Hall, 1990.

56. Teicholz, E., "Computer Integrated Manufacturing," Datamation, Vol. 30, No. 3, pp. 169-174, March 1984.

57. Williams, T.J., "The Integrated Approach to Industrial Conrtol," IFAC 10th Triennial World Congress, pp. 187-198, 1987.

58. Zhang, Y.L., "An Integrated Intelligent Shop Control System," Masters Thesis, Department of Industrial Engineering and Operations Research, Virgrinia Polytechnic Institute and State University, 1989.

## APPENDIX

## SOFTWARE CODE

```
/*                                  INET.H                         */

#define BSD_INCLUDES 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/ioctl.h>
#include <errno.h>


#define SERV_UDP_PORT   6000
#define SERV_TCP_PORT   6000
#define SERV_HOST_ADDR  "128.100.0.4"
#define IDLE 0
#define BUSY 1

int cell[5];    /* array of file descriptors associated with workcells */
int AGV[4];

typedef struct init{            /* structure for cellid checking */
        int cellid;
};

typedef struct command{  /*  struct for cmds between control and emulators */
        int cell_id;
        int part_num;
        int op_num;
        int prior_num;
        int error_num;
        int part_type;
        int from;
        int to;
        int ack;
        char er_msg;
        int agv_num;
     int agv_st;
        int agv_cod;  /* 1=arrived at requesting stat.   2= arrived at destination stat */
};

  typedef struct sellist {
        long fds_bits[2];
 };

□
```

```
/*                    SYSCTR.C                    */


#include "inet.h"
#define OK 1
struct command temp, syscom;
struct timeval timeout;
int ldnum, agvidl, tot;
int part[4][7] = {
        {0,0,0,0,0,0,0},
        {0,1,2,1,3,2,0},
        {0,2,1,3,1,2,0},
        {0,1,3,2,1,2,0}
};

int on, off;



main()
{
        int qnty_a,qnty_b,qnty_c, order[50], sch_siz, num;
        init_com();
        ldnum = 1;
        agvidl =3;
        on = 1;
        off =0;

        if (ioctl(cell[1], FIONBIO, (char *) &on) < 0)
                perror("ioctl FIONBIO error C1");
        if (ioctl(cell[2], FIONBIO, (char *) &on) < 0)
                perror("ioctl FIONBIO error C2");
        if (ioctl(cell[3], FIONBIO, (char *) &on) < 0)
                perror("ioctl FIONBIO error C3");


        if (cell[1] != 0 && cell[2] != 0 && cell[3] != 0 && cell[4] != 0) {
          printf("The system can manufacture three part types a,b, and c.\n");
          printf("Enter the amounts of each you would like to produce.\n");
          printf("Type a:");
          scanf("%d",&qnty_a);
          printf("Type b:");
          scanf("%d",&qnty_b);
          printf("Type c:");
          scanf("%d",&qnty_c);
        }
```

```
      printf("\nPROCESSING INITIATED\n");
         sched(qnty_a,qnty_b,qnty_c,order);
         sch_siz = qnty_a + qnty_b + qnty_c;
         tot = sch_siz;


         load(3,order);
      printf("\nMONITORING ALL CONTROLLERS\n");

         while ( tot  > 0 ) {

          monitor(order,sch_siz);
             }

close(cell[1]);
close(cell[2]);
close(cell[3]);
close(cell[4]);




}


load(num, ord)      /* function to put new parts into the system */
int num,  ord[];
{
int x;



         for (x=0; x<num; x++)  {
                 syscom.part_num = ldnum;
                 syscom.part_type = ord[ldnum];
                 syscom.from = part [syscom.part_type] [0];
                 syscom.to = part [syscom.part_type] [1];
                 syscom.op_num = 1;
                 syscom.prior_num =0;
                 syscom.error_num =0;
                 syscom.ack =0;
                 syscom.agv_num =0;
                 syscom.agv_cod =0;
                 syscom.cell_id = 4;
                 if (write(cell[4], &syscom, sizeof(syscom)) , 0)
                         perror("server: error sending load info");
                 ldnum ++;
                 agvidl --;
         }

         return;
```

```
}



errfunct(x)
int x;
{
        printf("ERROR: %s\n",temp.er_msg);
        return;
}


monitor(order1,sizo)
int order1[];
int sizo;
{
int monflag, nfound, x;
        monflag =0;


         if (ioctl(cell[4], FIONBIO, (char *) &on) < 0)
                perror("ioctl FIONBIO error");

        while ( monflag == 0) {  /*  monitor until cell message comes in */

          errno = 0;

                if (read(cell[4], &temp, sizeof(temp)) ,0)
                        perror("server: error reading AGV controller");


                if (errno != EWOULDBLOCK){
printf("MESSAGE RECEIVED FROM MATERIAL HANDLING CONTROLLER\n");
                  if (temp.error_num > 0){
                        errfunct(temp.error_num);
               }

                  else  {

printf("AGV%d at location %d with part %d\n\n",temp.agv_num, temp.to, temp.part_num);
                         temp.agv_cod =0;
                         temp.agv_num =0;
                 temp.ack =0;
                         agvidl ++;

                         if (temp.to == 0) {
                           tot --;
   printf("\n                      WORKORDER %d COMPLETED\n\n",temp.part_num);

                           if (tot == 0){
```

```
                        temp.error_num = 999;

                       for(x=1;x<=4;x++) {

                         if(write (cell[x], &temp, sizeof(temp)) < 0)
                            perror("serv: error writing to cell emulator");
                          }

                        printf("FINISHED PROCESSING\n\n");
                        printf("CELL SHUTDOWN PROCEDURE INITIATED\n");
                        agvidl = 0;
                                 return;
                        }
                               }

                            else {
                                    temp.cell_id = temp.to;
                            if (write(cell[temp.cell_id], &temp, sizeof(temp)) ,0)
                                    perror("serv: error writng cell emulator");
                                    }
                     }
                 }

        for (x=1; x<=3; x++) {

   ermo = 0;
            if (read(cell[x], &temp, sizeof(temp)) , 0)
                  perror("serv: error reading cell request");

            if (ermo != EWOULDBLOCK) {

 printf("MESSAGE RECEIVED FROM CELL %d\n",x);

         if (temp.error_num >0)
         errfunct(temp.error_num);

         else {
printf("Part %d Operation %d completed\n\n",temp.part_num,temp.op_num);
                    monflag = 1;
                       agvidl --;
                 temp.ack = 0;
                       temp.op_num ++;
                       temp.from = temp.to;
                       temp.to = part[temp.part_type][temp.op_num];
                       temp.cell_id = 4;
                    if(write(cell[4], &temp, sizeof(temp)), 0)
                          perror("serv:error sending cmd to AGV contr");
                       return;
                 }
            }
        }
        if(agvidl > 0 && ldnum <= sizo)
          load(1,order1);

          } /* close monflag while loop */
}

□
```

```
/*                              SYSCOM.C                              */

#include "inet.h"
#define MAXLINE 512

int init_com()
{
        struct init *sys;
        int sockfd, newsockfd[4],  clilen, childpid,rval,i,cl,inbuf;
        struct sockaddr_in  cli_addr, serv_addr;
        char buf[MAXLINE +1],inreq[7];
        struct init temp;
        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                printf("server: can't open stream socket");




        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family      = AF_INET;
        serv_addr.sin_addr.s_addr  = htonl(INADDR_ANY);
        serv_addr.sin_port        = htons(SERV_TCP_PORT);

        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                printf("server: can't bind local address");




        listen(sockfd, 4);


        cl =0;
        strcpy(inreq,"initcl");
     printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
     printf("INITIALIZING COMMUNICATIONS\n");

        while( cl < 4) {


                clilen = sizeof(cli_addr);
                i = accept(sockfd, (struct sockaddr *) &cli_addr,&clilen);
                newsockfd[cl] = i;



                if (newsockfd[cl] < 0)
                        perror("server: accept error");

                if (write(newsockfd[cl], inreq, sizeof(inreq)) , 0)
```

```
                    perror("server: write error");

                if ( read(newsockfd[cl], &temp, sizeof(temp)) , 0)
                    perror("error reading stream message\n");

                cell[temp.cellid] = newsockfd[cl];


                printf("CELL %d INITIALIZED\n",temp.cellid);
                cl++;
            }
        printf("ALL WORKCELL COMMUNICATIONS INITIALIZED\n");
            return;
}

□
```

```
/*                              SYSCED.C                        */


#include "inet.h"



sched(a, b, c, ord)
int a, b, c;
int ord[];
{
        int  pos = 1;

        while ( a>0 || b>0 || c>0) {
                if (a != 0) {
                        ord[pos] = 1;
                        pos ++;
                        a --;

                if (b != 0) {
                        ord[pos] = 2;
                        pos ++;
                        b --;
                }
                if (c != 0) {
                        ord[pos] = 3;
                        pos ++;
                        c --;
                }
        }

        return;
}

□
```

```
/*                              WC1EM.C                              */


#include "inet.h"
#define INI 1

struct command c1com;
int ptime [4][7] = {
        {0,0,0,0,0,0,0},
        {0,13,9,20,15,10,0},
        {0,6,21,15,17,9,0},
        {0,9,13,24,12,5,0}
};


main()
{

        int celfd, stat, del, errflg;
        long rem,tmp;

        celfd = wc_init(INI);  /* initialize comm socket, return file desc. */
        stat = 0;
      alarm(145);

        while(stat < 100) {

                if (read(celfd, &c1com, sizeof(c1com)) , 0)
                        perror("WC1: error reading command");

                if (c1com.error_num == 999){
                        stat = 1000;
                        continue;
                }

                else if (c1com.cell_id != INI) {
                    c1com.error_num = 1;
                    strcpy(c1com.er_msg,"WC1: Command had wrong cell number");
                    if (write(celfd, &c1com, sizeof(c1com)) , 0)
                          perror("WC1: error writing on stream socket");
                    continue;
                }
                else
                {
              del = ptime[c1com.part_type][c1com.op_num];

              if (c1com.ack == 1){
                 del = 25;
                 printf("Repair Cycle Began\n");
                 proc_del(del);
```

```
                        printf("Repair Cycle Completed\n");
                        c1com.error_num = 100;
                        if (write(celfd, &c1com, sizeof(c1com)) < 0)
                           perror("WC1: error writing on stream socket\n");
                        continue;
                        }

                        if (errflg != 1)
                        printf("PART %d   OPERATION %d   Processing time = %d\n",
                                c1com.part_num,c1com.op_num,del);
                        else {
                printf("PART %d OPERATION %d Resuming, Processing time = %d\n",
                           c1com.part_num,c1com.op_num,tmp);
                           del = tmp;
                           errflg = 0;
                           }




                              printf("Processing Began\n");
                              rem = proc_del(del);

                       if (rem > 0){
                          c1com.error_num = 3;
                          printf("Processing Stopped, time remaining = %ld\n\n",rem);
                          strcpy(c1com.er_msg,"WC1: Processing error, Work suspended");
                          errflg = 1;
                          tmp = rem;
                          }
                       else printf("Processing Finished\n");

                              if (write(celfd, &c1com, sizeof(c1com)) <0)
                                   perror("WC1: error writing on stream socket");

                       }
                  }

             printf("CELL 1 SHUTDOWN\n");
             }

             □
```

```
/*                              WC2EM.C                         */


#include "inet.h"
#define INI 2

struct command c2com;
int ptime [4][7] = {
        {0,0,0,0,0,0,0},
        {0,13,9,20,15,10,0},
        {0,6,21,15,17,9,0},
        {0,9,13,24,12,5,0}
};


main()
{

        int celfd, stat, del;
        long rem;

        celfd = wc_init(INI);  /* initialize comm socket, return file desc. */
        stat = 0;

        while(stat < 100) {

                if (read(celfd, &c2com, sizeof(c2com)) , 0)
                        perror("WC2: error reading command");

                if (c2com.error_num == 999){
                        stat = 1000;
                        continue;
                }

                else if (c2com.cell_id != INI) {
                        c2com.error_num = 1;
                strcpy(c2com.er_msg,"WC2: Command had wrong cell number");
                        if (write(celfd, &c2com, sizeof(c2com)) , 0)
                                perror("WC2: error writing on stream socket");
                        continue;
                }
                else
                {
              del = ptime[c2com.part_type][c2com.op_num];

 printf("PART %d   OPERATION %d   Processing time = %d\n",
                      c2com.part_num,c2com.op_num,del);
                /* alarm(8); */


                printf("Processing Began\n");
                rem = proc_del(del);
            printf("Processing Stopped, time remaining = %ld\n\n",rem);

                if (write(celfd, &c2com, sizeof(c2com)) <0)
                        perror("WC2: error writing on stream socket");

            }
        }
printf("CELL 2 SHUTDOWN\n");

}
```

```
/*                              WC3EM.C                           */


#include "inet.h"
#define INI 3

struct command c3com;
int ptime [4][7] = {
        {0,0,0,0,0,0,0},
        {0,13,9,20,15,10,0},
        {0,6,21,15,17,9,0},
        {0,9,13,24,12,5,0}
};


main()
{

        int celfd, stat, del;
        long rem;

        celfd = wc_init(INI);  /* initialize comm socket, return file desc. */
        stat = 0;

        while(stat < 100) {

                if (read(celfd, &c3com, sizeof(c3com)) , 0)
                        perror("WC3: error reading command");

                if (c3com.error_num == 999){
                        stat = 1000;
                        continue;
                }

                else if (c3com.cell_id != INI) {
                        c3com.error_num = 1;
                        strcpy(c3com.er_msg,"WC3: Command had wrong cell number");
                        if (write(celfd, &c3com, sizeof(c3com)) , 0)
                                perror("WC3: error writing on stream socket");
                        continue;
                }
                else
                {
               del = ptime[c3com.part_type][c3com.op_num];
 printf("PART %d OPERATION %d  Processing time = %d\n",
                         c3com.part_num,c3com.op_num,del);
```

```
                    /* alarm(8);  */


                    printf("Processing began\n");
                    rem = proc_del(del);
                printf("Processing stopped, time remaining = %ld\n\n",rem);

                    if (write(celfd, &c3com, sizeof(c3com)) <0)
                        perror("WC3: error writing on stream socket");

                }
        }

printf("CELL 3 SHUTDOWN\n");

}

□
```

```
/*                              AGVEM.C                         */


#include "inet.h"
#include <signal.h>
#define INI 4
#define READY 1


struct command c4com, tmp, tmp1, tmp2, tmp3, agv1, agv2, agv3;
fd_set mask;
int tran[4][4] = {
                {0,11,17,11},
                {10,0,16,15},
                {16,15,0,13},
                {10,14,12,0}
            };



main()
{

        int celfd, stat, dely, childpid, agv1id, agv2id, agv3id;
        int pipe1[2], agv1p1[2], agv1p2[2], agv2p1[2], agv2p2[2];
    int agv3p1[2], agv3p2[2];
        int agv1_loc =0, agv2_loc = 0, agv3_loc = 0;
        int maxfd, agvid[4], nfound, x;
        long rem;
        AGV[1] = 0;
        AGV[2] = 0;
    AGV[3] = 0;

        celfd = wc_init(INI);  /* initialize comm socket, return file desc. */
        stat = 0;


        if (pipe(pipe1) < 0)
                perror("can't create pipe1");

        if ( (childpid = fork()) < 0)
                perror("can't create fork");
```

```
if  (childpid > 0) {           /* PARENT */
    close(pipe1[0]);

    while (stat < 100) {

            if (read(celfd, &c4com, sizeof(c4com)) < 0)
         perror("WC1: error reading command");


                  if (c4com.error_num == 999){
                            stat = 1000;
         printf("AGV CONTROLLER SHUTDOWN\n");
              write(pipe1[1], &c4com, sizeof(c4com));
              exit(0);
              kill(childpid,SIGKILL);
                  }

           if ( write(pipe1[1], &c4com, sizeof(c4com)) < 0)
                  perror(" error writing to child pipe");
         }
  close(pipe1[1]);
  exit(0);
}








  if (childpid == 0) {            /*  OLDEST CHILD  */
         close(pipe1[1]);

     if (pipe(agv1p1) < 0 || pipe(agv1p2) < 0)
            perror("error creating agv1 pipes");

     if ( (agv1id = fork()) < 0)            /* CREATE AGV1 */
            perror("can't create agv1 fork");


         if ( agv1id > 0) {         /* OLDEST CHILD PARENT FOR AGV1 */


     close(agv1p1[0]);
     close(agv1p2[1]);
    if ( pipe(agv2p1) < 0 || pipe(agv2p2) < 0)
          perror("error creating agv2 pipes");

    if ((agv2id = fork()) < 0)       /* CREATE AGV2 */
       perror("can't create agv2 fork");


    if (agv2id > 0) {    /* OLDEST PARENT FOR AGV2  */

close(agv2p1[0]);
close(agv2p2[1]);

      if (pipe(agv3p1) < 0 || pipe(agv3p2) < 0)
            perror("error creating agv3 pipes");


      if ((agv3id = fork()) < 0)     /*  CREATE AGV3 */
            perror("can't create agv3 fork");


      if (agv3id > 0) {    /* OLDEST PARENT FOR AGV3  */

   close(agv3p1[0]);
   close(agv3p2[1]);

             agvid[1] = agv1p1[1];
             agvid[2] = agv2p1[1];
             agvid[3] = agv3p1[1];

             if ( agv1p2[0] > agv2p2[0])
                maxfd = agv1p2[0];
             else{
                 maxfd = agv2p2[0];
                   }

             if ( agv3p2[0] > maxfd)
                maxfd = agv3p2[0];

             if ( pipe1[0] > maxfd)
                maxfd = pipe1[0];

          maxfd ++;

              FD_ZERO (&mask);


for (;;){
 FD_SET(agv1p2[0], &mask);
 FD_SET(agv2p2[0], &mask);
 FD_SET(agv3p2[0], &mask);
 FD_SET(pipe1[0], &mask);
```

```
                    nfound = select(maxfd, &mask, NULL, NULL, NULL);

            if(FD_ISSET(pipe1[0], &mask) && (AGV[1]==0 || AGV[2]==0 || AGV[3]==0)) {
              if (read(pipe1[0], &tmp, sizeof(tmp)) < 0)
                perror("child: error reading parent");

             if( tmp.error_num == 999){
               kill(agv1id,SIGKILL);
               kill(agv2id,SIGKILL);
               kill(agv3id,SIGKILL);
               exit(0);
               }
printf("\nMESSAGE RECEIVED FROM SYSTEM CONTROL\n");
printf("Part %d to be delivered from %d to %d for operation %d\n",
    tmp.part_num,tmp.from,tmp.to,tmp.op_num);



                  if (tmp.cell_id  != 4 ){
                     tmp.error_num = 1;
                     if (write(celfd, &tmp, sizeof(tmp)) < 0)
                          perror("error writing on socket");
                        continue;
                        }



                   if (AGV[1] == IDLE || tmp.agv_num == 1){

              if(AGV[1] == IDLE){
                printf("Part number %d sent on AGV 1\n\n",tmp.part_num);
                }

              AGV[1] = BUSY;
                        if(write(agvid[1], &tmp, sizeof(tmp)) < 0)
                          perror("error writing on pipe to agv1");
                        }


                    else if (AGV[2] == IDLE || tmp.agv_num == 2){

                 if(AGV[2] == IDLE) {
```

```
    printf("part number %d sent on AGV 2\n",tmp.part_num);
    }

  AGV[2] = BUSY;
           if(write(agvid[2], &tmp, sizeof(tmp)) < 0)
                perror("error writing on pipe to agv2");
           }


  else if (AGV[3] == IDLE || tmp.agv_num ==3){

     if(AGV[3] == IDLE) {
       printf("part number %d sent on AGV 3\n",tmp.part_num);
       }

AGV[3] = BUSY;
         if(write(agvid[3], &tmp, sizeof(tmp)) < 0)
  perror("error writing on pipe to agv3");
         }

   } /* close if FD_ISSET loop */




if (FD_ISSET(agv3p2[0], &mask)) {
  if(read(agv3p2[0], &tmp3, sizeof(tmp3)) < 0)
    perror("error reading agv pipe");


  if (tmp3.agv_st == IDLE) {
    AGV[3] = 0;
    }

   if (tmp3.agv_cod == 2 || tmp3.error_num > 0){
     if(tmp3.agv_cod == 2)
       printf("AGV3 at delivery point\n");

     if(write(celfd, &tmp3, sizeof(tmp3))<0)
       perror("error writing on socket");
     }

     else {
         printf("AGV3 at pickup point\n");
         tmp3.ack = 1;
         if(write(agvid[3], &tmp3, sizeof(tmp3)) < 0)
           perror ("error writing on pipe to agv3\n");
         }
    }


          if (FD_ISSET(agv1p2[0], &mask)) {

                if(read(agv1p2[0], &tmp1, sizeof(tmp1))<0)
               perror("error reading agv pipe");


    if (tmp1.agv_st == IDLE){
       AGV[1] = IDLE;
       }

    if (tmp1.agv_cod == 2 || tmp1.error_num > 0){
       if(tmp1.agv_cod == 2)
         printf("AGV1 at delivery point\n");

                if(write(celfd, &tmp1, sizeof(tmp1))<0)
                perror("error writing on socket");

    }

    else {
         printf("AGV1 at pickup point\n");
         tmp1.ack = 1;
         if(write(agvid[1], &tmp1, sizeof(tmp1)) < 0)
           perror("error writing on pipe to agv1");
         }
              }


  if (FD_ISSET(agv2p2[0], &mask)) {

    if (read(agv2p2[0], &tmp2, sizeof(tmp2)) <0)
       perror("error reading agv pipe");


    if (tmp2.agv_st == IDLE){
       AGV[2] = IDLE;
       }

    if (tmp2.agv_cod == 2 || tmp2.error_num > 0){
       if(tmp2.agv_cod  == 2)
         printf("AGV2 at delivery point\n");

       if (write(celfd, &tmp2, sizeof(tmp2))<0)
          perror("error writing on socket");
       }

       else {
```

```
                        printf("AGV2 at pickup point\n");
                        tmp2.ack = 1;
                        if(write(agvid[2], &tmp2, sizeof(tmp2)) < 0)
                          perror("error writing on pipe to agv2");
                        }
                   }



              }      /* close infinite loop */

                        }  /* close agv3 parent loop */

     } /*  close agv2 parent loop */

   }   /* close agv1 parent loop */




              if (agv1id == 0) {              /*  AGV1 grandchild  */
                      close (agv1p1[1]);
                      close (agv1p2[0]);


while(1) {

                      if (read(agv1p1[0], &agv1, sizeof(agv1)) <0)
                              perror("AGV1: error reading pipe");


                      if (agv1.ack == 0){
                         agv1.agv_num = 1;
                         agv1.agv_st = 1;
                         dely = tran [agv1_loc][agv1.from];
                         proc_del(dely);
                         agv1_loc = agv1.from;
                         agv1.agv_cod = 1;
                         if (write(agv1p2[1], &agv1, sizeof(agv1)) <0)
                              perror("AGV1: error writing on pipe");
                         }

                      else if(agv1.ack == 1){
                           dely = tran[agv1_loc][agv1.to];
                           proc_del(dely);
                           agv1_loc = agv1.to;
```

```
                    agv1.agv_cod = 2;
             agv1.agv_st = 0;
                    if (write(agv1p2[1], &agv1, sizeof(agv1)) <0)
                        perror("AGV1: error writing on pipe");
                    }
     }

     } /* close of AGV1 grandchild */




           if (agv2id == 0) {           /* AGV2 grandchild */
                 close (agv2p1[1]);
                 close (agv2p2[0]);



while (1) {
                 if (read(agv2p1[0], &agv2, sizeof(agv2)) <0)
                   perror("AGV2: error reading pipe");

                 if (agv2.ack == 0){
                   agv2.agv_num = 2;
                   agv2.agv_st = BUSY;
                   dely = tran [agv2_loc][agv2.from];
                   proc_del(dely);
                   agv2_loc = agv2.from;
                   agv2.agv_cod = 1;

                   if (write(agv2p2[1], &agv2, sizeof(agv2)) <0)
                       perror("AGV2: error writing on pipe");
                 }

               else if(agv2.ack == 1){
                    dely = tran[agv2_loc][agv2.to];
                    proc_del(dely);
                    agv2_loc = agv2.to;
                    agv2.agv_cod = 2;
             agv2.agv_st = 0;
                    if (write(agv2p2[1], &agv2, sizeof(agv2)) <0)
                        perror("AGV2: error writing on pipe");
                 }
  }
           } /* close of AGV2 grandchild */
```

```
                    if (agv3id == 0) {              /*  AGV3 grandchild  */
                      close (agv3p1[1]);
                      close (agv3p2[0]);


while (1) {

                      if (read(agv3p1[0], &agv3, sizeof(agv3)) <0)
                           perror("AGV3: error reading pipe");

                      if (agv3.ack == 0){
                        agv3.agv_num = 3;
                        agv3.agv_st = BUSY;
                        dely = tran [agv3_loc][agv3.from];
                        proc_del(dely);
                        agv3_loc = agv3.from;
                        agv3.agv_cod = 1;

                        if (write(agv3p2[1], &agv3, sizeof(agv3)) <0)
                             perror("AGV3: error writing on pipe");


                       }
                    else if (agv3.ack == 1){
                           dely = tran[agv3_loc][agv3.to];
                           proc_del(dely);
                           agv3_loc = agv3.to;
                           agv3.agv_cod = 2;
                   agv3.agv_st = 0;
                           if (write(agv3p2[1], &agv3, sizeof(agv3)) <0)
                              perror("AGV3: error writing on pipe");
                       }
   }
                      } /* close of AGV3 grandchild */


        } /* close of oldest child loop  */


} /*  close main  */

□
```

```
/*                      WCCOM.C                       */


#include "inet.h"
#define MAXLINE 512
#define DATA "Test String"

int wc_init(id)
int id;
{
        struct init temp;
        struct init *cel1;
        int sockfd;
        struct sockaddr_in   serv_addr;

        char buf[7];
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family       = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
        serv_addr.sin_port         = htons(SERV_TCP_PORT);
        temp.cellid = id;
        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                printf("client: can't open stream socket");



        if (connect(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) <0)
                printf("client: can't connect to server");


        cel1 = &temp;

        if (read(sockfd, buf, sizeof(buf)) ,0)
                perror("WC1: read error on stream socket");

        if ( strcmp(buf,"initcl") == 0)
        {
                printf("CELL %d INITIALIZED\n",id);


                if (write(sockfd, &temp, sizeof(temp) )  ,0)
                        perror("write error on stream socket");

        }

        return(sockfd);
}


□
```

```
/*                              DELAY.C                         */


#include<stdio.h>
#include<sys/errno.h>
#include<fcntl.h>
#include<sys/time.h>
#include<signal.h>
int don = 1;


nullfcn()
{
        don = 0;
        return(don);
}




proc_del(n)
int n;
{
        long start,delay,tnow,left,time();
        signal(SIGALRM,nullfcn);
        start = time((long *) 0);
        delay = start + n;
        tnow = time((long *) 0);
        while (tnow < delay && don != 0)
        {
                tnow = time((long *) 0);
        }
        don = 1;
        left = delay - tnow;
        return(left);
}


del(n)
int n;
{
        long start,delay,tnow,time();
        start = time((long *) 0);
        delay = start + n;
        tnow = time((long *) 0);
        while ( tnow < delay)
        {
                tnow = time ((long *) 0);
        }
        return;
```

## VITA

The author received his Bachelors of Science in Electrical Engineering from Clemson University in December 1985. He was subsequently commissioned in the United States Air Force as a Contracting/Manufacturing Officer. He served four years at Aeronautical Systems Division, Wright-Patterson Air Force Base, as both a contracts manager and most recently as a Manufacturing Engineer. The writing of this thesis completed a Master of Science in Industrial and Systems Engineering, (Manufacturing/Automation) started in August 1990. His next assignment will be with the Defense Logistics Agency, Defense Contracting Administrative Service, Regional Office at Boston, Massachusetts.

[illegible]