An Object-Oriented Simulation-Based Method for Emulation Development for Testing Shop Control Software

by

Malay A. Dalal

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Industrial and Systems Engineering

APPROVED:

M. P. Deisenroth, Ph.D., Chair

R. B. Badinelli, Ph.D.

G. Ioannou, Ph.D.

C. P. Koelling, Ph.D.

S. C. Sarin, Ph.D.

March 1, 1996
Blacksburg, Virginia

Keywords: Object-oriented Simulation, Emulation, Software Testing, CIM, Manufacturing Automation

An Object-Oriented Simulation-Based Method for Emulation Development for Testing Shop Control Software

by

Malay A. Dalal

M. P. Deisenroth, Ph.D., Chairman

Industrial and Systems Engineering

(ABSTRACT)

An emulator is a computer program that mimics the behavior of a production facility as seen by the control program. Emulation has been used as a tool for dynamic, off-line testing of control software for automated manufacturing systems. However, research efforts in emulation have focused mainly on controllers for equipment, AGVS, and workstations. This research focuses on emulation for testing shop control software.

Though conceptually simple, emulator development efforts tend to be ad hoc in nature and lack a strong conceptual framework. Currently, the effort involved in developing an emulation model may outweigh the potential benefits. The approach used in this research centers around adapting a detailed simulation model, i.e., used for testing control *strategies*, for emulation, i.e., control *software* testing. This approach promotes software reuse and thus limits the emulation development task.

Due to the limitations of conventional simulation languages and modeling techniques, a simulation model is not readily adapted for emulation. The main problems lie in turning off the control logic in a simulation and interfacing the model to the actual controller. An object-oriented modeling methodology was developed for systematically transitioning from simulation to emulation. Basically, the method calls for encapsulating manufacturing control logic into controller objects and modeling the system from the perspective of exchange of messages among controllers. The developed method also promotes the rapid development of a driver for verification and validation of the emulation model.

A prototype system was developed to demonstrate the feasibility of the emulation development method. The MODSIM II language for object-oriented simulation was used to implement the object classes.

The second issue addressed is the ability to use emulation to test shop control system in faster-than-real-time (FRT) mode. Currently it is necessary to test the system in real-time, which makes it impractical to observe extended operation of the shop. The mixed-mode emulation method, which switches time-advance between real-time and next-event modes, was developed. Issues in implementing and using the mixed-mode and the delay-scaling technique for FRT emulation were discussed. Experimental results showed that mixed-mode emulation had the potential to reduce run-times by more than 50% over real-time emulation.

This dissertation is dedicated to my father Ashvin,

a man just slightly ahead of his times.

ACKNOWLEDGEMENTS

This has been a group effort; there are so many to thank ....

First, I thank God for allowing me this accomplishment.

Down on Earth, I start with thanking my advisor Dr. Mike Deisenroth for having taken me under his wing. "Dr. D" has been the epitome of the word *mentor*, having given me his support--moral, professional, and financial--above and beyond the call of duty. It has been easy to work under someone whose deep knowledge, varied talents and moral character you respect. The biggest motivation a Ph.D. candidate can have is the knowledge that his advisor is on his side. Dr. D, never underestimate the power of positive reinforcement--even ***I*** completed the Ph.D.!

I have been fortunate to have on my committee talented researchers who have been helpful in their suggestions and objective in their criticisms. For that, I'd like to thank Dr. Ralph Badinelli, Dr. Pat Koelling, Dr. George Ioannou and Dr. Subash Sarin.

I share this accomplishment with my dear and ever-loving wife Nandita (WE did it!). It doesn't seem fair that I alone will have the degree. But until the University accepts my proposal to co-grant the Ph.D. to the student *and* spouse, these words acknowledge her unreasonable amount of support and patience that made it all possible. I want her to know that with her beside me, there is little I will not be able to do.

I will remain eternally indebted to my family in Bombay--parents Kishori and Ashvin, sisters Nimmi and Urmi--who made many personal sacrifices so that I could see this day. I acknowledge the inspiration I drew from my father--an ordinary man with extra-ordinary accomplishments--and thank him for always providing opportunites. I am thankful to my other family ("in-laws")--parents Bharati & Rashmikant, and Naimesh & Trupti, Anjana & Kedar Parikh--for their understanding and whole-hearted support.

Over the many years in the Ph.D. program I have been blessed with the friendship and support of many friends: Andre "Kommander" Ramos, Rafael "Chinky" Marshall, Rosendo Molina, Mark Eaglesham, Janis Terpenny, Joni Quessenberry, Lovedia Cole, Namita & Sanjay Arora, and Bharati & Bharat Patel. Knowing that you all were always rooting for me, made it so much easier to keep going and going and .....

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Testing of control systems, at all levels of automated manufacturing systems, is an extremely complex and time consuming task. It is considered to be one of the major bottlenecks in the implementation of Computer Integrated Manufacturing (CIM) systems. Common software test methods, such as "black- box" and "white-box" testing schemes, are not adequate for testing control software due to the dynamic nature of the manufacturing system [Kockerbeck & Schlictherle, 1990] and because of the possible distribution of control across multiple computers [Ben Hadj-Alouane et al., 1990]. Conventional software debugging methods only permit limited, static testing of portions (some modules) of the control software. Since the software interacts with external (software and hardware) systems, the functionality (i.e., dynamic behavior) of the control software is usually tested only after all equipment and systems are installed and running. Thus, a significant part of the control software testing process is performed after the initial development and testing, and away from the controlled, "office" environment. The problems of software testing are exacerbated by the interdependencies of the subsystems and inconsistent definition of the interfaces. Consequently, implementation time is, more often than not, considerably longer than expected. Godio and Vignale [1987] further observe that "the drawbacks caused by the uncontrolled environment are accompanied by undeserved blame: software is delivered last and puts into evidence all the delays and inefficiencies which have been adding up during the whole project life."

Emulation technology can be used to overcome these difficulties by shifting the software testing activity out of the critical path of the project. Webster's New Collegiate Dictionary defines emulation as "an exact imitation." The function of the emulation model is to *imitate the dynamic responses* of the system to the commands from the actual system controller. An emulator acts as a substitute for the plant components, from the perspective of the controller, permitting much of the software testing to be performed independently of the installation activities and without the use of physical equipment. In the past, emulation has been commonly used to aid in the testing of

process control systems. Recent research has demonstrated the feasibility and benefits of using emulation to test software for control of discrete-part manufacturing systems.

Simulation has become an important tool for systems analysis and decision-making when new manufacturing systems are conceived or existing ones are altered. It can be used in a number of ways. One use of a simulation might be to model a proposed factory layout to study space utilization and material movement. Alternatively, simulation can be used to test the effect of various sequencing priorities on queue buildup and equipment utilization. In general, simulation is used to *estimate the performance* of a system under specific conditions and operational policies. Two of the major benefits of simulation analysis are the ability to observe the behavior of the system without experimenting with the actual equipment, and the possibility of exercising the dynamic behavior of the model with respect to time and predicting future conditions. There are many similarities between the simulation of a manufacturing system and the emulation of that system when testing the control software. However, in general, simulation and emulation have been considered separate tasks, and a different computer program is developed for each.

This research addresses the issues of:

- improving the task of testing manufacturing control software with emulation;
- integrating simulation and emulation model development; and,
- using the emerging object-oriented simulation (OOS) technology to improve emulation development.

1.1 Emulation

A simple and concise definition of an emulator is as follows: an emulator is a computer program that mimics the behavior of a production facility as seen by the control software [Godio & Vignale, 1987]. The emulation program is configured to communicate using the same protocol as the system it replaces. Consequently, the control system is unaware of whether it controls the actual plant/equipment or the emulator. This is illustrated in Figure 1. At any level of manufacturing systems control, the controller can

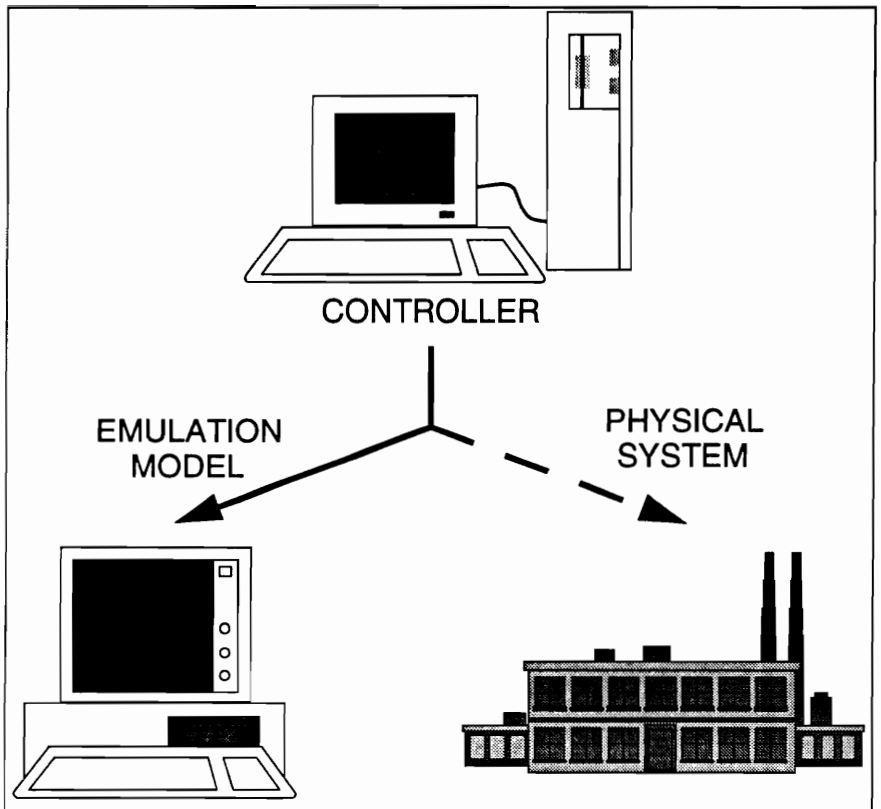


Figure 1: Concept of Emulation for Testing Control Software

be envisioned as being connected to the physical system or to an emulation model. Commands and control signals emanating from the controller result in responses from either the actual, physical system or the emulation model. To the controller, these responses are the same. It assumes it is in control of the physical system at all times.

Depending on the method we use for generating the emulated signals (commands and responses), an emulation may be classified as hardware or software. Co and Chen [1989] presented an interesting application of a hardware emulation system. The emulation consisted of a model train set which travels along a track and trips switches to indicate its position on the track. This emulation was a representation of a system of Automated Guided Vehicles (AGVs) used for material movement within a manufacturing facility. The AGV system controller, whose logic was to be tested, was wired to the model train set. Command and control signals from the controller resulted in the actions of the model train, which in turn produced response signals that were returned to the controller as if they had originated from the actual AGV system. In contrast, a software emulation uses logic contained in the computer program to generate signals in response to commands. Such a program would be responsible for interpreting controller commands (e.g., move to a workstation), simulating the time delay associated with the movement of the AGV from one track segment to the next, as well as generating a signal in the same format as that of actual AGV. The latter type of emulation is of interest in this research, and the term emulation will be used to mean software emulation in the remainder of the document.

1.1.1 Emulation Versus Simulation

The key difference between emulation and simulation lies in the manner in which the behavioral model of the system and the control logic which drives the model are represented. In a simulation model these two elements not only reside on the same computer, but are also intertwined with each other. In an emulation model the two elements are physically separated since the *control commands originate from the controller which performs the control function in the actual production system.* Thus, information about which operation to execute next comes dynamically from the control system and not from a pre-prepared list of tasks or an algorithm within the simulator.

While simulation and emulation both aid in testing control, simulation is used to test alternative control *strategies*, whereas emulation is used in testing actual control *software*. The control logic used in the simulation is a model (i.e., an abstraction) of the actual controller software. When the control logic is implemented in the actual controller--by manual re-coding, possibly in another programming language--it may give rise to inconsistencies between the intended strategy and its software implementation. This problem may become more pronounced at the level of shop control systems, especially when attempting to represent the complex logic of knowledge-based controllers. As noted in [Narayanan et al., to appear] "network modeling abstractions (commonly used in simulation) do not facilitate the representation of decision-making on system wide objectives since scheduling and control decisions are distributed through the model."

1.1.2 Emulation For Hierarchical Control

Manufacturing systems are, most often, structured into hierarchical control models as shown in Figure 2. While the number of levels in such control models may range from as few as three to as many as six or more, a four level model is commonly used. At the lowest level is equipment control. The tasks of the controlled equipment include machining, measurement, and transportation. Robots, CNC machines and the like are organized into groups and placed under a supervisory cell controller. The function of the cell controller is to schedule and control the elements of the cell in order to meet the goals set at the shop level. The shop controller is concerned with the overall coordination and control of many cells which comprise a major section of the manufacturing system. The facility-level controller is the interface between the business functions and the manufacturing system. It provides shop controllers with aggregate production plans.

Emulation technology can be used in testing the control logic associated with any node (or level) within the hierarchy. Consider again the four-level control model shown in Figure 2. At the lowest level is *DNC emulation.* Functionally, the machine controller receives commands from the cell controller and generates control signals that are transmitted to the machine actuators. The machine sensors, which detect changes in the physical equipment resulting from controller commands, serve as inputs to the

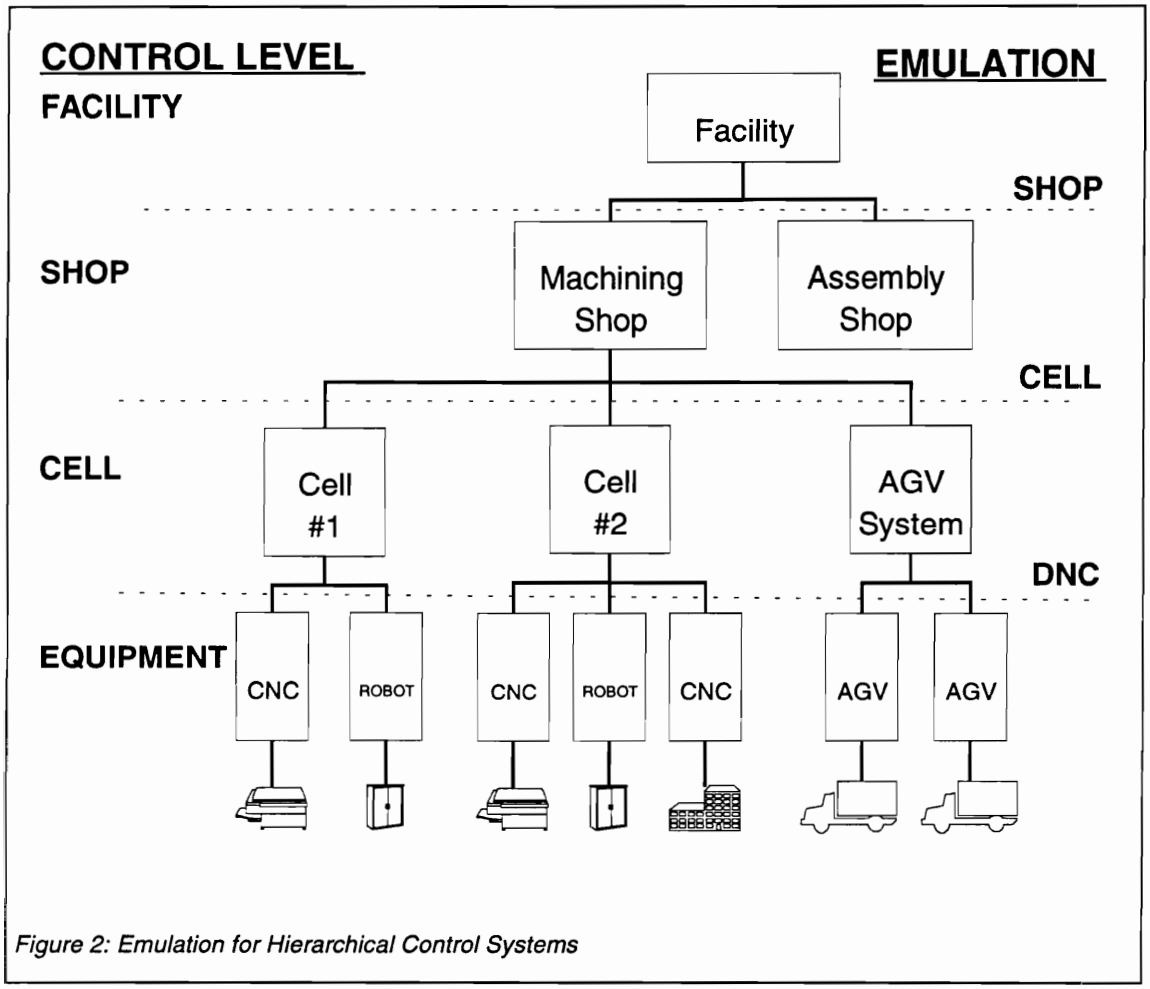


Figure 2: Emulation for Hierarchical Control Systems

machine controller. Hence, an emulator at this level must model both the system above the machine controller (i.e., cell controller) as well as the system it controls (i.e., the equipment).

In a similar manner, higher levels within the hierarchy can also make use of emulation technology for testing control logic. At the cell level, the emulation task involves replicating the interface between the shop controller and cell controller. In addition, response signals from equipment controllers are emulated based upon simulated operation of the equipment. The emulator for testing shop control software must generate inputs, such as work orders, on behalf of the factory-wide material and process planning computer systems, which the emulator replaces. Also, the emulator should mimic the status messages and decision requests which result from simulating the activities (material movement, job completion, etc.) in the various cells under the control of the shop controller.

1.1.3 Benefits of Emulation

The key benefit of emulation is the ability to uncover some of the control software problems during in-house integrated testing prior to systems installation and the actual start-up of computer controlled manufacturing systems [Voller & Webster, 1991]. Thus, it reduces the need for connecting the control system to the process for obtaining the validation of control. Other possible benefits of using emulation are as follows:

- Emulation also aids in the integration of subsystems from different vendors. Each vendor uses the emulation model of the other interacting subsystems, resulting in a smoother physical integration on the plant floor. In fact, the first (commonly known) cell emulation effort was carried out to overcome the problem of interfacing various control systems during the development of the Automated Manufacturing Research Facility (AMRF) of National Institute of Standards and Technology (NIST) [Bloom et al., 1984].

- Erroneous control sequences can potentially cause damage to the process. Since emulation is based on dynamic simulation, it is possible to uncover timing and other response related software problems before actual systems implementation.

- It is possible to emulate various fault situations and error messages from production equipment and/or subsystems. This makes it easy to systematically test the fault handling capability of the control program.

- An additional advantage of dynamic simulation is that it produces pseudo-random sequences which are dynamic but repeatable. To test corrections, the application software can be re-run against a known repeatable scenario.

- When the emulation model is run in parallel with a fully operational system, it can play the role of a fault monitoring/diagnosis/isolation system [Hitchens & Ryan, 1989]. When the model activity shows a significant departure from that of the operational system, an error situation can be detected. The fault can then be easily diagnosed and identified for maintenance personnel.

- During the upgrading of an existing control system, the new software can be conveniently tested off-line with minimal disruption of production activity. This results in a saving of time and money, as exemplified at Texas Instruments [Bradshaw, 1987].

- Operators may be trained to handle emergency situations, in a safe environment and prior to system installation, since the emulation perfectly represents the dynamics of the actual system [Siggard & Alting, 1991; Waye, 1988].

- A final benefit is the ability to evaluate the performance of the computer hardware (e.g., memory usage, disk performance, load on the computer network) and determine the effect of communication delays on system performance [Voller & Webster, 1991].

1.2 Simulation In Manufacturing

Simulation is a widely used tool in modeling manufacturing systems. It is used to support decision-making prior to committing valuable resources to a project. It is commonly used to test the control policies used in the operation of a manufacturing system since the system performance is highly influenced by these policies. The type and number of physical components used in the system, not only depend on, but also

affect the policies which control their operation. Simulation is invaluable in designing a system: an iterative process of specification and evaluation, in the quest for one which achieves maximum results for the least cost.

Recently, a broader definition of simulation modeling has been used to extend the use of simulation beyond the system design phase and into the design implementation phase. Hitchens and Ryan [1989] have proposed the use of progressively detailed models in the following five phases of a CIM project (Figure 3):

1. The Conception Phase: During this phase simulation is used to sell a project idea to management. The model contains few details and generates limited statistics.
2. The Design Phase: The designer's task is to create a plan of the automated manufacturing system which achieves maximum results for the least cost. Simulation provides the system designer a computer model of the system in which operating conditions can be changed and the results evaluated.
3. The Fabrication Phase: A modified simulation model, which emulates the signals emitted by the system components, is connected to the computer containing the control software. The controllers use this emulation model, as a replacement for the physical system, in order to test the control logic.
4. The Installation Phase: A simulation model representing the "as specified" version of the system, provides the means to verify the operation of the "as built" system.
5. The Operation Phase: During the operation of the system, the simulation model serves a number of purposes. When the model runs in parallel with the system, it may be used for monitoring the system behavior. The model can also be used for real-time evaluation of alternative control policies without disturbing the actual production process.

The concept of using simulation in the various phases of the CIM project life-cycle is a good one and possesses tremendous potential for the future of CIM systems. However, there are many practical issues to be resolved. A major issue is the life-cycle of the simulation model itself, i.e., the ability to have the model evolve along with the CIM project.

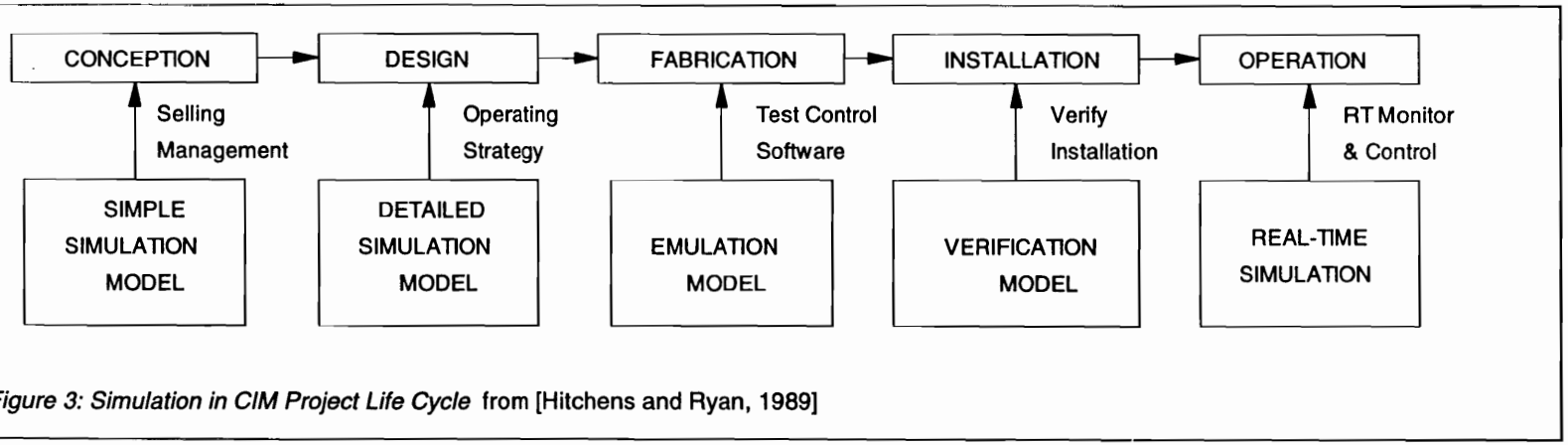


Figure 3: Simulation in CIM Project Life Cycle from [Hitchens and Ryan, 1989]

In conventional simulation, the model is built with a particular problem or design question in mind. Such models usually embed the control aspects within the description of the system. This makes it difficult to modify either the description of the physical system or the control policy, and consequently limits the use of the particular simulation model. In its traditional form, the simulation model, although very useful, is not very amenable to reuse. Hence, as a result of interleaving of model and control elements, it is difficult to extend the use of a simulation model beyond the second phase of the CIM project life-cycle. Besides the limitations imposed by modeling methodology, current simulation languages make it difficult to incorporate complex decision procedures and all of the important constraints. Consequently, simulation models tend to overestimate the capabilities of the system [Davis et al., 1993]. The emerging object-oriented simulation paradigm has the promise to resolve both the above issues.

1.3 Object-Oriented Simulation

1.3.1 A New Paradigm

All simulation languages provide the user with a set of pre-defined objects. For example, a network-based queuing language views a system as having entities that travel through a network of queues being served by resources. In this case the set of object classes, i.e., "language," include queues, activities, resources, etc. In addition there are objects for statistics gathering, statistical distributions, etc. These set of objects are provided as a means for simplifying the task of simulation mechanics--an improvement over using general purpose programming languages for simulation.

Two problems arise form such traditional simulation languages. The first is that they force one into thinking in one particular style, in the sense that the user has to think of the system and problem in terms of available tools/objects provided by the language. It is not always easy to translate the system into these objects, and at times may even be impossible (Roberts, et al. [1988] suggest trying to model a tennis match as a network model.)

The second problem is that there is no provision for adding to the list of available object types, or even modifying the behavior of existing objects. For a manufacturing

application it is desirable to have objects such as machines, parts, robots, part routings, etc. For other applications, other objects may be needed. The list of objects would be endless. Even if a machine object was provided by the simulation language, there is no way to ensure that it could cover the entire, or even a substantial, realm of machine types. Furthermore, as the complexity of manufacturing systems (and their components) increases, so does the need to accurately model their behavior.

Rather than attempting to provide all possible objects, perhaps the simulation language should have its own facility for defining objects. This is in addition to pre-defined objects that are furnished with the language, so that the user should not have to depend solely on pre-defined objects. "The goal of object-based simulation and object-oriented simulation (OOS) is to allow the user to extend and customize the language by creating new objects and building them on a consistent platform of concepts" [Roberts & Heim, 1988]. The reader is referred to [Roberts & Heim, 1988; Bischak & Roberts, 1991] for an introduction and general discussion of OOS in manufacturing.

1.3.2 Benefits

The benefits from the design philosophy associated with an object-oriented system are visible if one considers the long term opportunities and the increasing complexity of the systems being modeled. In the short term, traditional simulation techniques and languages prove to be fairly adequate for the modeling of less complex systems. The benefits of OOS go beyond the ability to create more detailed, "natural" (due to one-to-one mapping) models. Models developed using OOS techniques tend to be reusable and extensible, i.e., being able to use the model for other than its original purpose, with minimal modifications. In the context of this research, OOS holds the promise to allow using a simulation model for emulation. Other relevant benefits are listed below:

- Not only is it possible to reuse "simulation mechanics" code, but also the code for complex objects. The same complex object can be reused in a different model, with the user having to pay less attention to implementing the inner workings.
- It is difficult to incorporate the behavior of intelligent agents, such as a human decision maker, into the model with traditional simulation. In object-oriented

modeling (OOM), the rules of object behavior are included within the definition of the object itself. This makes it much easier to model intelligent agents using AI techniques and then incorporate this intelligence into the functionality of the object.

- Most physical objects have a corresponding object in the simulation model. As a result, OOS models often have a natural pictorial representation and are easily animated. This also allows the use of icons to develop models [Thomasama & Ulgen, 1988].

- There is a natural division among the simulation components when viewed as objects. An object or a set of objects could have their own processor so that the total processing power available to the simulation (which is very CPU intensive) is greatly increased. It is easy to conceptualize distributing objects across processors connected by a network.

- From a research environment perspective, an object-oriented simulation imparts continuity to a research effort [Glassey & Adiga, 1989]. Understanding the code created by another programmer requires considerable effort. If clean interfaces are maintained, so that knowledge of internal details is not necessary, then successive efforts to expand the code will require much less re-understanding and/or modification. Such a "black-box" approach is characteristic of encapsulation in object-orientation (OO). Also, a research simulation model should contain no more detail than necessary for the particular purpose. OOS permits selective use of characteristics of objects used in the model.

1.4 Problem Statement

From an earlier section it is clear that the benefits of emulation are many and extremely significant. Then why, one may wonder, is emulation still an emerging technology? This point is evident from the small number of industrial applications, especially compared to the popularity and use of conventional simulation.

Perhaps the biggest roadblock is the amount of effort required to develop an emulation. Since each production environment differs from the previous one, the emulation development effort currently tends to be repeated for each project. Even for a

system of modest complexity, the emulation task demands time and manpower in an amount significant enough to substantially reduce the benefits of the effort. One difficulty stems primarily from lack of standardization: ranging from controller interfaces (more pronounced at the level of DNC emulation), to controller design/schemes (more evident at shop control level). Another limitation is the absence of emulation programming methodology and software to aid in the rapid development of emulators.

The absence of a formalized approach for emulator development and use, not only makes the task time-consuming, but also has the potential of introducing additional sources of error and reducing the reliability of the resulting emulator. Developing an emulator only makes sense if its development time is substantially lesser than that for the control software, and if the emulator is reliable enough so that there is a reasonable assurance that the control software is the only source of any errors detected during testing.

Past research efforts in emulation have focused mainly on equipment and subsystem emulation and the concept can be beneficially extended to higher control levels. However, there are very few published research efforts in emulation for the purpose of testing shop control software. In view of the increasing trend towards automated and/or knowledge-based shop controllers, there is a need for a convenient means of testing these systems. The decision-making process at the shop level involves the use of information from many sources (e.g., databases, knowledge-bases, communication messages) in a variety of formats. The ability of the shop controller to make the right (i.e., pre-planned) control decisions depends on the availability of accurate and timely data from those sources, and the proper software representation of the decision-making logic. The use of hardware emulation (comprised of switches, timers, lights, etc.) is not even a viable alternative at this level of control; primarily due to the format of the information that is exchanged, as well as the time characteristics.

1.5 Research Objectives

Emulation is one of the most promising ways of reducing the time and cost associated with developing control software for CIM systems. Yet, most current emulation efforts are ad hoc in nature and lack a strong conceptual framework in their

implementation. Additionally, little attention has been paid to using emulation at the shop control level to ensure proper implementation of complex control logic. Emulation development time is a significant problem. The goal of this research is to simplify and expedite the process of developing and using a software emulator in testing and debugging shop control software. This research addresses the goal through two specific objectives.

The first objective is to develop a methodology for emulator development for the purpose of reducing the effort for future emulation projects. The methodology developed in this research utilizes the concept of a dual-purpose modeling structure, based upon object-oriented techniques, for emulation and simulation model development. The methodology permits emulation models to be built by extending/building upon simulation models which are based on this framework. The emulator development task is thus limited to adding emulation-specific functionality or enhancing some details.

Another, secondary, objective of this research is to improve the process of testing with software emulation. In its basic form the emulation merely provides a substitute for the physical system. The onus of performing tests and detecting errors in the control software are still upon the manufacturing engineer. This research concentrates on one function that will help in the testing process: the ability to observe several hours of shop operation in a fraction of the time. Currently, this is difficult since the time evolution in the shop controller and emulation are independent of each other. This research examined alternative mechanisms for emulator operation in faster-than-real-time mode.

The long-term thrust of this research is to move towards an environment for emulator development, and off-line testing using emulation. This environment is analogous to those currently available for simulation development. The emulation development environment is envisioned to contain function/class libraries for a bottom-up emulation model development. Capabilities for testing, such as model initialization to a specific control scenario, and injection of fault situations, would also be included.

1.6 Scope of Research

The primary objective of the research is to develop a structure--based on object-oriented modeling (OOM)--better suited for transitioning between simulation and emulation, rather than to develop a better OOS. Hence, no attempt is made to design a new OOS framework; instead existing research in this area is used and adapted to fit the specific requirements of this research. In particular, Simulation Support Classes are not developed, and among Manufacturing Support Classes the emphasis is on objects potentially affected by the transition from simulation to emulation, e.g., cell and shop controllers.

The proposed methodology for emulation development is intended to be independent of a particular object-oriented programming language and of a particular hierarchical control scheme. Using this methodology, it would be possible to use the software for both simulation and emulation; switching back and forth between the two modes of operation with minimal configuration changes or recompilation. The viability of this concept is demonstrated on a prototype. However, the development of a generic emulator (configurable for any application) is outside the scope of this research.

The research focuses on the on-line control functions of the shop controller. The broad functions that fall into this category include dispatching, coordination, and monitoring. Schedule generation/regeneration which is typically done off-line is not considered.

This research makes a distinction between testing control strategy and control software. Hence, the research is concerned only with testing the software implementation of a chosen strategy, and does not attempt to develop an improved control strategy. Collecting statistics on system performance is of interest only to the extent they help in testing, evaluating, or verifying the control software, i.e., implementation of a control strategy.

The feasibility of a mechanism which permits faster-than-real-time operation of the emulation model is investigated as the secondary objective of the research. This research is in the context of a discrete event system and at the level of shop control.

Naturally, the developed method exploits characteristics within these limits, and may not be generally applicable for all systems and/or levels of control.

1.7 Summary

This research addresses the development and use of emulation models for testing shop control software. Chapter Two surveys published accounts of developments in several related research areas. In Chapter Three a method for emulation development is presented along with an object-oriented modeling framework to support the method. Chapter Four describes a proof-of-concept implementation in which an emulation was developed using the framework presented in Chapter Three. Chapter Five discusses several issues pertaining to using an emulation, including emulation validation and verification. In addition, it describes the results of experimental efforts to use the developed emulation in faster-than-real-time mode. Finally, Chapter Six summarizes the research accomplishments and makes concrete recommendations for future research in this area.

2. LITERATURE REVIEW

2.1 Introduction

The survey of literature relevant to this research is presented in four sections. The first provides an overview of control systems in discrete event manufacturing; modeling and development of such systems is emphasized. The second section compares the use of simulation in non-conventional roles and the nature of the models in these applications. In the third section, a survey of research efforts and applications involving the use of emulation in testing different classes of control systems is provided. The final section discusses relevant developments in the rapidly growing field of object-oriented modeling and simulation of manufacturing systems.

2.2 Control Systems

2.2.1 Distributed Control

Initial attempts at automated manufacturing control employed a centralized control system, with all the computing power and decision making capabilities residing on a single, centralized machine. Reliability problems, caused by dependence on a single computer, and the availability of relatively inexpensive microcomputers gave rise to distributed control schemes. Such schemes are characterized by the location of computer power for decision making closer to the site of the problem, and cooperation (albeit, to varying degrees) among the computing nodes to achieve overall control. Two distinct control architectures have emerged: hierarchical and heterarchical, with the former being the currently more popular one.

Hierarchically controlled systems are constructed using the levels of control (delegation of authority) concept. These systems are comprised of a number of control modules arranged in a pyramid structure. Control systems at higher levels are concerned with longer planning horizons than those at the lower levels. Higher level, broader, goals are decomposed into more specific ones for lower levels. Control systems at each level make decisions based on commands from level above and the feedback from the level below. While the basic philosophy of distribution of decision-

making remains the same, the number of levels and the specific responsibilities of each level vary across different hierarchical architectures.

Proponents of the heterarchical control architecture, most notably Hatvany [1985] and Duffie and Piper [1987], tend to describe hierarchical control relationships as master-slave. This is a fair assessment in comparison to a heterarchical system which is made up of multiple intelligent entities, none of which exhibits direct control of any others. Instead, the entities cooperate to meet overall system objectives. This scheme is based on the belief that supervisory decision making should be located at the point of information gathering rather than in a central location [Duffie & Piper, 1987] Increased flexibility and improved fault tolerance are cited as their primary advantages of heterarchical control over hierarchical [Duffie et al., 1988]. Although the heterarchical scheme may be more robust, the control software to support the decision making is more complex since a minimal amount of assumptions about other control modules are allowed. Ongoing research efforts in scheduling and control of heterarchical systems can be found in [Duffie & Prabhu, 1994].

The hierarchical control model developed by the National Institute for Standards and Technology (NIST) is perhaps the most well known [Jones & McLean, 1985]. The NIST hierarchy (Figure 4) is made up of five levels organized in a strict hierarchy: peer-to-peer communication is not allowed, and each controller has direct contact with only the immediate higher or lower level controller.

1. At the top is the *facility* level, which includes process planning, production management (including long term schedules), and information management (including links to financial and other administrative functions).
2. Below this is the *shop* level, which manages the coordination of resources and jobs on the shop floor. The processes involved at this level include the grouping of jobs into part batches using a group technology classification scheme. The concept of a virtual manufacturing cell is introduced at this stage. These virtual manufacturing cells comprise machines which are grouped together in a dynamic fashion, i.e., the configuration and number of virtual manufacturing cells varies with time. Besides

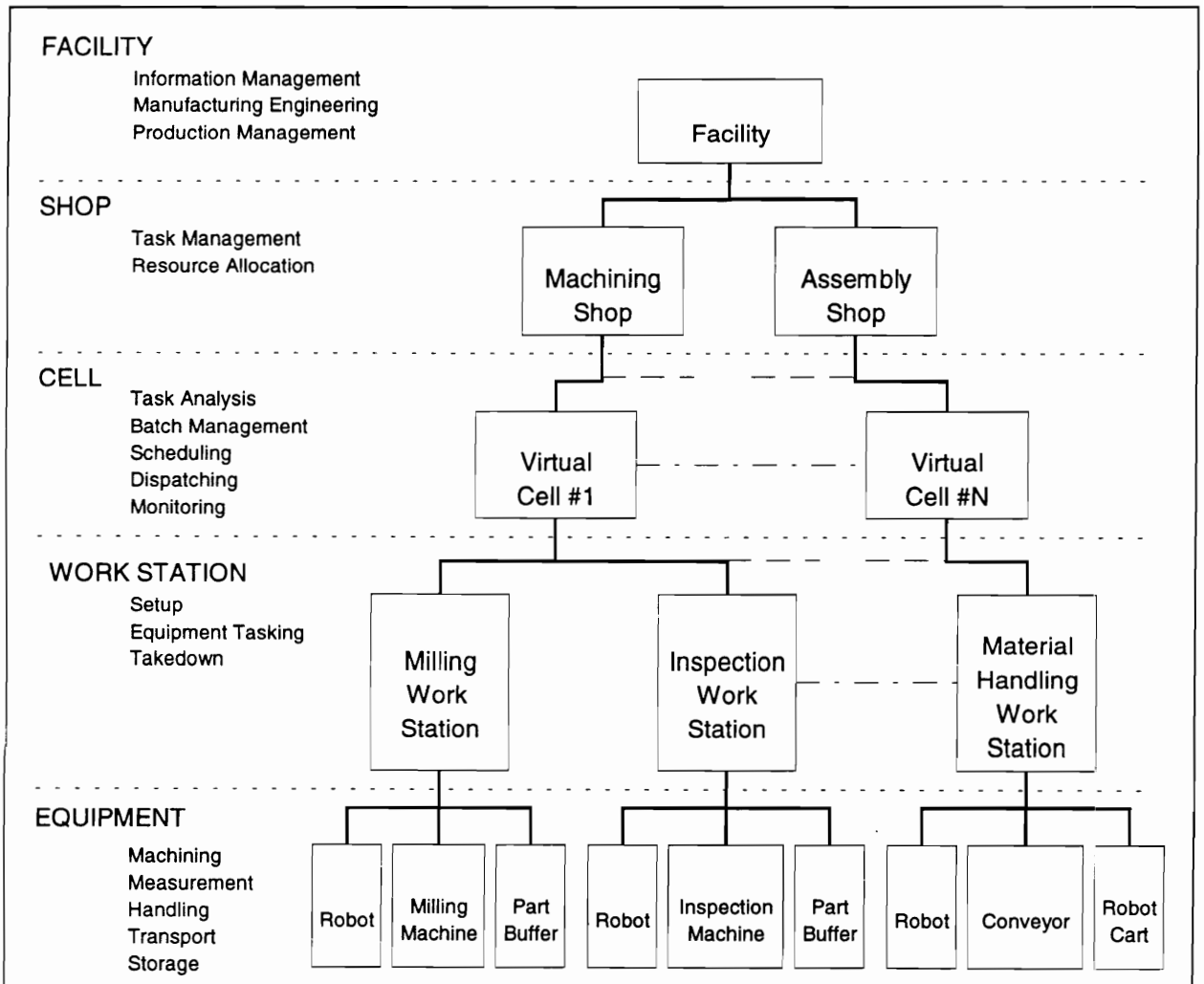


Figure 4: NIST Hierarchical Control Model from [Jones &McLean, 1985]

job groups and virtual manufacturing cell configuration, the tasks at this shop level include allocation tooling, jigs/fixtures and materials to specific workstation/job combinations. These activities at the shop level are re-evaluated on the basis of feedback from the cell level and on changes in requirements from the facility level.

3. Below the shop level is the *cell* level, where the cell controls system schedules the jobs. These jobs have already been divided into groups, with the jobs allocated to each cell being somewhat similar. Also involved in scheduling and controlling the jobs is the scheduling of material handling and tooling within the cell.

4. The level below is the *workstation* level, which consists of coordinating the activities of the AMRF workstation which is taken typically to consist of a robot, a machine tool, a material storage buffer and a control computer. The workstation controller then arranges the sequencing of operations in order to complete the jobs allocated to the cell control system.

5. The lowest level of the planning and control hierarchy is the *equipment* level, which consists of the controller for individual resources such as machine tools, robots or material handers.

The Advanced Factory Management System (AFMS) (discussed in [O'Grady, 1986]) is a hierarchical control system architecture for production control developed by Computer Manufacturing International, Inc. (CAM-I). The hierarchy consists of four levels (Figure 5):

1. The factory control system is the top level and is concerned with top level factory management functions. It considers such aspects as determining end-item requirements, product structure definitions (process planning), and individual shop capacities and capabilities.

2. The job shop level is directly below the factory level. It takes commands from the factory level in order to determine commands for the work center levels. Included in this are the taking of end-item production and exploding this into processing operations. Having done this, the shop order events are scheduled.

	MODEL RESOURCES	MAINTAIN PLANNING INFORMATION	GENERATE REQUIREMENTS	DETERMINE TIMING	PLAN RESOURCES	INITIATE EVENTS	MONITOR STATUS	PREDICT EVENTS	EVALUATE PERFORMANCES
FACTORY LEVEL	determine shop capacity and capability	update product structure definition	establish end item production	schedule end item production events	configure shops and plan production for shops/suppliers	set end item production requirements and requests	monitor actual and predicted completion of items	predict completion of products	report product manufacturing performance
JOB SHOP LEVEL	determine work center capacity and capability	update process routing information	explode item requirements into processing operations (create shop orders)	schedule shop order events	configure work centers and plan operations for work centers suppliers	release shop orders (order processing)	monitor actual and predicted completion of operations	predict completion of parts	report manufacturing performance
WORK CENTER LEVEL	determine resource capacity and capability	update process description	explode operations into detail tasks	schedule task events	configure units and plan tasks for resources	dispatch work assignments to resources	monitor actual and predicted completion of task	predict completion of operations	report process operations performance
UNIT/ RESOURCE LEVEL	verify capacity to perform	update control information	translate tasks into subtasks	schedule subtask events	allocate resources to subtasks	perform process (subtasks)	monitor actual and predicted completion of subtasks	predict completion of tasks	report task performance

Figure 5: Advanced Factory Management System architecture from [O'Grady, 1986]

3. The work center takes commands from the job shop level and generates detailed task requirements. The task events are then scheduled and commands for these tasks are passed to the next level--unit/resource level.

4. At the unit/resource level tasks from the work center level are broken into subtasks and these subtasks are carried out.

O'Grady [1986] observes that, "The number of intermediate levels of the hierarchy between factory and the machine levels is two (jobshop and workcenter) for the AFMS and three (shop, cell and workstation) for the AMRF. The extra level is partly caused by the requirement for the virtual cell reconfiguration; if it is assumed that this is done relatively infrequently, then we could reduce the number of levels in the AMRF hierarchy to two. In any case the number of levels is somewhat artificial, what is more important is what gets achieved at each layer."

Figure 6 [Joshi & Smith, 1992] shows the characteristics of controllers under the assumption of an AMRF hierarchy. These characteristics may be considered typical but within the context of the discussion in the previous paragraph. If the AFMS hierarchy is considered, the computational requirements would be lower at the cell level since many functions would shift to the shop level. In any case, the response time at the shop level may be as high as a couple of minutes.

2.2.2 Modeling of Control

Petri nets are useful tools for the modeling and analysis of discrete event dynamic systems. Recently, they have been used in the modeling of manufacturing systems which display the following characteristics: concurrency or parallelism; asynchronous operations; deadlock; conflict; and, event driven [Desrochers, 1990]. These types of systems have been difficult to accurately model with differential equations and queuing theory. A static analysis of Petri nets, using mathematical foundations, reveals many qualitative and quantitative aspects of the modeled system. The operation of the Petri net can be simulated for a dynamic analysis of the control system [Duggan & Browne, 1988b; Chen & Wongladkown, 1991]. Petri net models can also be used to implement real-time control systems for automated manufacturing

	Equipment	Workstation	Cell
Hardware	Lathe, Mill, T-10, Bridgeport Series I, IBM 7545 Robot	Robot tended machine center, Cartrac Material Handling Systems	Variable Mission System, Several Integrated Workstations
Controller	Mark Century 2000, Accuramatic 9000, Custom-single-board system	Allen-Bradley PLC-4, IBM-PC/AT, etc.	VAX 11/750, SUN Workstation, etc.
Controller Type	Single-board processors, machine tool controller, Servo-Controller, etc.	PLC, PC, Minicomputer	PC, Microcomputer, Super-Minicomputer
Language	Assembler, Part programming, Robot programming, etc.	C, Ladder logic, Pascal, other sententual languages	C, LISP, FORTRAN, other high level languages
Memory/Size Requirements	8Kb - 128Kb RAM plus custom ROM, EPROM, etc.	256Kb - 1Mb RAM 1Mb - 80Mb hard drive	512Kb - 4Mb RAM 10Mb - 1Gb hard drive
Response time	$< 10^{-3}$ sec	< 1 sec	< 20 sec
Machines/ Interconnects	1-1 connect	1-many 1-{1,8] machine tools 1-{1,50] material handling	1-many 1-{1,8] Workstations

Figure 6: Characteristics of Controllers from [Joshi & Smith, 1992]

systems. Like a programmable logic controller (PLC), they can sequence and coordinate the subsystem activities [Valette et al., 1983].

Thus, "Petri nets constitute a formalism that allows one to use the same model for analysis and real control of the system" [Villarroel & Muro-Medrano, 1994]. Consequently, Petri net modeling is used as the common tool in the SECOIA project [Bijou et al., 1987; Courvoisier et al., 1984, 1987]. The Specification, Emulation, and Conception of Integrated Automation (SECOIA) project is an attempt provide an integrated methodology to specify, test, and implement discrete distributed control in a FMS. A Meta Language for Control has been defined to describe several types of Petri nets (e.g., binary, colored, timed, etc.) for the various levels (local machine, cell coordination) of control.

However, as noted in [Yim & Barta, 1994], researchers have commonly recognized that high level control functions, such as scheduling and vehicle dispatching systems, are difficult to model with Petri nets. Thus, Petri nets are more appropriate for modeling of lower level--workstation and cell--control tasks involving the coordination of various pieces of equipment.

Petri nets are an improvement over using finite state machines (FSM) which have been used to model control of discrete event systems, e.g., AMRF [Bloom et al., 1984]. The major drawback of FSM is that they require explicit representation of all states of the system components. This results in complexity and loss of flexibility since the entire machine has to be modified when adding or removing a component [Desrochers, 1990; Lin et al., 1994]. Network flow diagrams tend to be cluttered when using Petri nets to model complex systems [Cecil et al., 1992].

The object-oriented (OO) approach is gaining popularity in modeling manufacturing control. Briefly, the OO world view consists of looking at the world as a set of interacting objects. An object is an encapsulation of data (or attributes) and procedures (or methods) that operate upon the data. An object has state, behavior, and identity [Booch, 1994]. Objects interact by exchanging messages, and an object uses its methods to respond to a message.

In the context of manufacturing, this paradigm has been used to model control situations as a sequence of messages that are exchanged between objects representing manufacturing entities, e.g., equipment, cell controllers, databases. Message flow diagrams (MFD) [Booch, 1994; Adiga & Glassey, 1991] help in modeling *system level* interactions among objects. State transition models focus on *individual* object states, and are used to complement MFD. Since each object stores its own state and "knows" when and how to transition between states, control information is no longer managed by a single (centralized) state table. This approach avoids the exponential growth of combinations expected with state tables, resulting in greater flexibility in accommodating changes [Lin et al., 1994; Smith & Joshi, 1992].

The OO approach to modeling control is the basis for recent efforts in developing generic control software [Ben Hadj-Alouane et al., 1990; Fabian & Lennartson, 1992; Smith & Joshi, 1992; Chanchien et al., 1995]. Basically, the approach involves use of internal and external objects [Fabian & Lennartson, 1992] (or software/hardware components [Ben Hadj-Alouane et al., 1990]); internal objects are software representations of external objects, i.e., equipment. Generic control software is developed using internal objects which communicate with each other using a generic language. Internal objects then translate generic commands into device specific commands for the external objects.

2.2.3 Development of Control Software

Krogh et al. [1987] have developed a system for automatic generation of computer programs for the real-time control of devices in a workcell. The first step is for the manufacturing engineer to describe the elements and structure of the state-transition models using a rule-based specification language. In the next step a Petri net model is extracted from this specification, stored in a system specification database, and is analyzed using Petri net verification methods. Once the process and control logic have been correctly specified, the specification in the database is automatically encoded into a control program in C or other similar language.

Smith [1992] has developed a method for reducing the development effort for shop floor control systems. According to him, control software consists of a generic part

and an implementation specific part. The generic part can be reused in other similar controllers, while a portion of the specific part can be automatically generated. This is done by first creating a formal model, of the executable portion of the shop floor controller, using the message-based part state graph (MPSG) approach. A translator then converts the MPSG model into a controller program.

Recently, simulation models have been used to develop control programs. McHaney [1988] identifies four methods for transferring logic from a simulation model to an actual system controller: Philosophic Transfer, Pseudocode Transfer, Database Transfer, and Actual Code Transfer. The four methods vary in the amount of communication required (between simulation developers and controller developers), initial planning and design, and duplication of effort. Often, the above four methods are not independent of each other and may all be used sequentially in a large project.

The first two methods are common during the initial project phases: under the Philosophic Transfer scenario only the key ideas and assumptions are presented in the form of a written report, while with the Pseudocode Transfer method the ideas are further detailed and documented using programming pseudocode. The Database Transfer method will most commonly be used where many similar systems are being designed using a generic simulation and a generic controller. As an example the author describes, in the context of AGVS, a table of candidate pick-up points, and corresponding intermediate points, for selecting the next destination for each drop-off point. The table containing data validated during simulation may directly be used in the actual controller if the structure and format of the database have been agreed upon by both (simulation and controller development) teams a priori. The major benefit of the Actual Code Transfer is the reduction in the duplication effort since the logic--coded in a general language such as C or FORTRAN--is developed only once and used in the simulation as well as actual controller. An added benefit is that simulation results become more credible since the logic used is not simply a model of the actual control logic, and has been decided upon and validated early on in the project. The obvious limitation is the need for a more concerted initial effort requiring the simulation and control system programmers to each have a clear understanding of the others' program

structure. Additionally the method assumes that the ability to clearly delineate the control logic exists in the simulation language.

Due to the initial simplicity, the first two methods, and to a somewhat lesser extent the third method, have been commonly used in practice. The following paragraphs discuss recent research efforts in transferring actual control code from the simulation model into the physical controller.

AutoSimulations, Inc., had planned a software module that would permit compilation of AGV control software directly from the AutoMod simulation model [Quinn, 1985]. Their approach exploits the similarity of the (patented) internal structure of AutoMod and actual AGV control systems, and is based on simply translating the tables and process subroutines in the simulator into software that can be executed on a controller. Naturally, a customized simulation and compiler is required for each supplier and controller combination.

Bilberg and Alting [1990] have developed and tested SIMCON--a prototype system for developing control programs. Their approach involves using the main parts of a detailed simulation program as the kernel for the actual control program (in contrast to recoding the simulation control logic in C or other language). The parts that can be reused in the control mode are the production rules--the "brain" of the simulation program--which are the basis for decision making during production. SIMCON can be used for real time control by enhancing the capabilities of the ProModel simulation program with two modules: the signal manager and the communication protocol manager. The signal manager transmits commands to the shop floor based to the status of the simulation, and converts status messages to changes in the simulation. The protocol manager translates generic commands such as "start processing" into controller specific byte instructions.

Research efforts are underway at the University of Illinois to demonstrate the feasibility of using a simulation model to control an FMS [Davis et al., 1993, 1994]. The simulation model is developed using a new simulation approach--Hierarchical Object-Oriented Programmable Logic System (HOOPLS) (described in detail in the section on OOS). Essentially the approach consists of using object-oriented principles to model, in

detail, the elements of the FMS--parts, equipment, controllers, etc.--and explicitly modeling the interaction of the controllers in the form of control message exchanges. All controlling influence in the FMS is contained within controller objects which model the physical controllers in the real system. The encapsulation of the controller logic into individual controller objects, and the message relay enable the logic to be directly ported over to the actual control computers.

2.2.4 Knowledge-Based/Intelligent Control

After significant progress in automated control at the equipment, workstation, and cell level, recent research efforts have been directed towards the automation of supervisory control. In the absence of such controllers, human supervisors were responsible for much of the decision making associated with converting production plans into the actual efficient production of parts, and reacting to disturbances on the shop floor, i.e., reactive scheduling. One of the common approaches to intelligent control is the dynamic selection of dispatching rules based upon the particular performance objectives (e.g., tardiness, machine utilization) to be "optimized" and current shop conditions. Examples of intelligent dispatching include the Dynamic Sequencing Rule module in the ReDS system [Hadavi et al., 1990] and the EXPERT dispatcher [Chandra & Talavage, 1991].

Some controllers also include the ability to react to dynamic situations, such as rush jobs, material shortage, machine breakdown, etc., and attempt to minimize the length and effects of the disturbances. The resulting control decisions, which may go beyond merely re-sequencing jobs at machine queues, include splitting up a batch, delaying release of a job to the shop, or altering the processing plan for a job. Such controllers are also known as knowledge-based controllers (KBC) since they rely on the knowledge-bases developed from the experience of human supervisors. Some recent papers on the development of KBCs include [Sarin & Salgame, 1990; Manivannan & Banks, 1992]. The following paragraph elaborates on the last reference as an example of a typical KBC.

At the heart of the knowledge-based on-line simulation (KBOLS) architecture [Manivannan & Banks, 1992], for supervisory control in a CIM environment, is the

knowledge-based controller (KBC). The KBC consists of both static and dynamic knowledge bases, inference engines, and a learning module. The KBC controls the activities on the shop floor and interacts with an on-line simulation (OLS) to evaluate alternative control decisions. Thus, the KBC contains rules for shop floor control (e.g., how to react to a machine breakdown) as well as for simulation purposes (e.g., how frequently to conduct resimulation). The KBC consists of two inference engines: the causal reasoner to diagnose a fault based on causal relationships; and, the forward chain inferencer for arriving at a control decision. A learning module stores the knowledge from each new fault situation, thereby reducing the need for resimulations. The KBC is implemented using a blackboard architecture allowing several knowledge sources to cooperate in reaching a control decision. One of the knowledge sources is the human supervisor who may ratify or change the alternative control decisions reached by the other (computerized) knowledge sources.

The KBOLS architecture highlights two significant issues, resulting from the variety of knowledge and information sources, in testing KBCs. The first is that static testing is not adequate; dynamic real-time response characteristics of such controllers are critical. The second issue is that the potential of a discrepancy between the control strategy and its implementation is likely to be increased; this underscores the need to validate the resulting control decision.

2.3 Simulation

A discussion of the traditional use of simulation in the design of manufacturing systems and the emerging role of simulation during the fabrication and operation of the system was provided in Section 1.2. In this new role, the model interacts with the hardware in the manufacturing system. A simulation model may be connected to a physical controller for testing control software, or to run the model in parallel with the shop for detecting exceptions. Similarly, a simulation is interfaced to a physical system for remote graphical display (monitoring) of shop status, or to initialize the model to current shop status for look-ahead simulation. The nature of the simulation model used in these applications, and the issues of interfacing are discussed in this section.

2.3.1 Real-time Monitoring, Scheduling and Control

Ericson et al. [1987] and Thompson [1994] discuss some general issues, while Harmonosky [1990] provides a very good discussion of the implementation issues of using simulation in unconventional roles such as real-time scheduling, control, and monitoring. In these roles there are two basic modes of operation for the simulation model. The first is the monitoring mode, wherein the simulation is linked to the physical system and inputs from the system allow the model to run in parallel with it. In the second mode (the decision making mode) the future impact of alternative control decisions are examined.

Harmonosky [1990] points out that interfacing the simulation and system (either for remote graphical display or to provide the current shop status as the starting point during look-ahead simulation) gives rise to data transfer and processing issues. It is likely that the base language (e.g., C, FORTRAN) of the simulation language (e.g., SLAM, SIMAN) may differ from the base language of the plant control software, thus requiring a translation. Additionally, information may be shared between physical devices in the form of simple flags or by highly structured messages such as 'data telegrams' [Kockerbeck & Schlictherle, 1990]. Mapping system data to model data will thus require a particular variable to be set to a specific value based on the flag, or stripping the necessary data element/s from the structured data. A third issue is that of determining the source of data, especially in a distributed system consisting of many databases instead of a central one.

A potential problem in real-time monitoring resulting from a high frequency of shop-floor events exists [Erickson et al., 1987; Shires, 1988]. For each event (breakdown, job completion) the computer running the simulation model must receive the data, possibly translate it, change internal variables, and update the graphical display and/or animation. If the time to process events is large and many events occur in a small period of time, the real-time monitoring capabilities will be diminished. Thompson [1994] suggests considering alternatives such as transferring messages individually or in bulk for the entire system, and transmitting messages at fixed intervals or asynchronously in order to alleviate the problems.

Manivannan & Banks [1991] develop a framework known as real-time knowledge-based simulation (RTKBS) for the real-time control of a manufacturing cell using a dynamic knowledge-base and a simulation model. The framework provides an integrated environment for the controller to evaluate various control policies using simulation. A key element of this framework is the event/time synchronization (ETS) module for the synchronization of the various events and their times of occurrence in the real system and the model. The synchronization becomes necessary since the actual time of an event, e.g., job completion, may differ from the estimated completion time in the model, and because of time sharing within a computer during simulation. (This method is an alternative to continuously obtaining data from the floor, which is impractical in a discrete event system.) The module uses three temporal rules to adjust the times of past, present, and future events on the simulation's event calendar. The module also ensures that data collection only occurs for those events being considered in the model. The framework also includes a dynamic knowledge base which stores the results from simulations performed by the supervisor to examine control problems. This feature reduces the need for re-simulations by first examining the knowledge-base for previously stored similar problems.

The simulation model used in the two modes of operation have different requirements: in the monitoring mode only a model of physical elements of the shop is needed since no decision making is involved; while, in the decision making mode the simulation model should contain the decision logic to drive the model. Thus, as Ericson et al. [1987] note, a model structure is needed which allows us to switch between the two modes of operation. Harmonosky and Barrick [1988] are of the opinion that a model which focuses on the communication logic between the controllers is more appropriate than one with the traditional perspective of parts flowing through the shop. Another model issue pertains to the ability to keep track of shop events while the model is in the decision making mode [Harmonosky, 1990]. When the model returns to the monitoring mode it needs to catch up with the current status of the shop.

A critical review of recent literature on simulation in real-time decision-making (i.e., scheduling/dispatching) can be found in [Rogers & Gordon, 1993]. In using such systems several factors or parameters have to be decided, including frequency of

rescheduling and the degree of reaction to shop-floor events [Manivannan & Banks, 1992; Rogers & Gordon, 1993]. Decisions about other factors, such the length of the look-ahead horizon and number of simulation runs, are tradeoffs between the confidence in the resulting statistics and the ability/need to arrive at the control decision in real-time [Harmonosky, 1990]. Similarly, deterministic simulations of the shop are typically used in the interest of speed [Marcus, 1990]. However, a more realistic decision may be reached by including at least some random events [Harmonosky, 1990; Rogers & Gordon, 1993]. Other considerations, such as criteria for schedule evaluation and storage of results, follow the look-ahead simulation runs [Rogers & Gordon, 1993].

The final portion of this section describes a real-time shop floor monitoring system which has been successfully implemented at Asea Brown Boweri [Shires, 1988; Bastos & Shires, 1987]. In addition to the real factory, the factory control system controls a simulator for real-time decision support. As a pre-requisite, the simulator is indistinguishable from the actual factory, to the control system. The control system, AZR, is responsible for the operational control of the shop floor. It receives a job priority list from the production planning system on a daily or weekly basis. Events on the shop floor, such as breakdowns, insufficient material, inefficiencies, etc., require changes to the original plan. The simulation software is used to test the effect of any modifications to the plan.

The simulator, called OCS, is a visual interactive system which permits the state of the factory elements to be continuously displayed; and allows the user to interrupt the simulation, alter views, display statistics, etc. OCS is actually an emulator since it represents only the physical system while the control task is external to it (by the AZR control system). Each component of the system--AZR controller, monitor, and OCS simulator--has an individual database. A copy of the relevant parts of the main database are made each time the monitor or simulator are started, so as to not affect the main database and the on-going operations in the factory. A copy (of the relevant parts) of the control system containing the control logic installed on the simulator workstation, perform the control task when using the simulator for look-ahead decision making.

2.3.2 Separation of Physical and Control System

Krogh et al. [1987] adopt a new approach to Petri net models of manufacturing systems and its elements by separating the process model and control model into separate graphs; previously proposed models have represented the control logic and the process states in a single graph. The new approach makes it easier to distinguish the function of the control computer from the state transitions inherent to the process, thus facilitating control program evaluation (and generation). Such a graph structure is also suited for transitioning to emulation since the net representing the control logic can be conveniently detached, allowing the actual PLC to make the control decisions; the process models merely maintain the state of the individual devices.

A similar approach has also been used in [Yim & Barta, 1994] for the design of an FMS. Hardware components of the FMS are modeled by Petri net objects, while control functions are separately modeled and integrated into the Petri net model to resolve conflicts in Petri net execution. This corresponds to the control functions of the cell controller.

Duggan and Browne [1988a] have developed a simulation tool to validate different Production Activity Control (PAC) configurations prior to their implementation on the shop floor. The simulation architecture differs from conventional simulation models in that it explicitly separates the control and physical layers of the shop floor. With reference to the five building blocks of the PAC architecture, the Scheduler, Dispatcher, and Monitor represent the control modules, while the Producer and Mover comprise the physical representation of the shop floor. Thus, scheduling strategies may be tested by modifying only the relevant control modules without requiring a change in the physical model of the shop floor. In the simulation architecture, the physical layer is defined in the Shop Floor Emulator (SFE). The SFE simulates all of the events that occur on the shop floor, including machine breakdown, workstation usage and WIP. The SFE models the operation of the physical layer using Petri nets [Duggan & Browne, 1988b].

Barnichon et al. [1990] develop a method for simplifying the testing of different scheduling and control policies by separating the physical system from the control

system in a simulation model of a manufacturing system. Extended Petri nets are used to represent the physical system, while C or FORTRAN routines describe the control. Both the above systems are described in the "knowledge model" (for static analysis) by separate Petri nets. Communication between the nets occur by way of messages (the mechanism for which is incorporated into the nets). The authors have developed a mapping from the Petri net model of the physical system to SIMAN statements, thus automating the generation of the dynamic "action model." At each decision point in the SIMAN model a message is sent to the control system. This message is handled by the coordination level Petri net which determines if the control level needs to be invoked. The coordination level calls the control level with an EVENT statement. The actual disposition of the job entities is handled at the control level by using the policies which are coded in FORTRAN or C subroutines. A change in control policy usually requires only a modification of the subroutine without requiring a change in the SIMAN model (representing the topology of the FMS).

VirtualWorks is a system for building and operating a virtual factory [Onosato & Iwata, 1992]. This system employs a three-dimensional modeling system for management of spatial data, and an object-oriented programming language to deal with the technical data attached to the geometric models. In VirtualWorks there are three important classes that facilitate class definitions of machines and devices: model_object, power_driven_object and programmable_object. An object instanced from the power_driven_object subclass has a variable indicating its internal state and a set of methods for executing unit actions. Upon receiving a pulse from the clock object the methods are executed, based on the current state, resulting in a transition to the next state. Each machine class of the programmable_object class contains an interpreter to translate programs written in the machine-specific programming language into methods defined in the machine class. Actual control programs stored in the object instances are interpreted and executed step by step according to the system clock. VirtualWorks only provides the executable models of the shop floor level and includes no decision making mechanism. Thus, the description of the virtual factory is at the physical level rather than the functional.

Researchers at Arizona State University have worked on the separation of model logic control structures from the simulation model while using OOP [Ogle, 1994; Ogle et al., 1991]. They identify control in manufacturing as falling into two categories: System Simulation Control that provides the monitoring and control not present in the actual system but necessary for computer simulation, and User Logic Control (ULC) captured in objects that are constituents of the user's conceptual model. ULC represents the decision making and physical control processes present in the modeler's view of the system. The research introduces a partitioned conceptual model of a Model-Experiment-Control structure that separates ULC from the definition of (physical) model elements and also the experiments under which the elements are evaluated. Each model element is viewed as an expansion box with the capability of being expanded by plugging in additional objects. An Experimental Slot facilitates expansion for an experiment run by the user, while a Model Logic Control slot contains the logic to manipulate the model element as part of a larger system.

Other research in the separation of physical and control system using OOM techniques include the OSU-CIM system [Bhuskute et al., 1992] and BLOCS/M object library [Glassey & Adiga, 1989]. The OSU-CIM approach calls for the splitting up of the abstraction of a real world entity in a model into three objects: physical, control, and information. Thus a workstation object may be composed of a processor (physical) object and a queueController (control) object. Similarly, in the BLOCS/M framework there is a strict separation between those objects that represent physical states, those that contain data or information, and those that implement decision heuristics. Resource objects (e.g., Workstation, Lot) do not contain decision-making capabilities, even implicitly. Decisions regarding the next lot to process or next workstation to visit are sent as messages from the Decision Support objects (e.g., Manager, LotDecisionInfo). These frameworks are discussed in detail in the section on OOS.

2.4 Emulation/Control System Testing

2.4.1 Continuous/Process Control Systems

The use of emulation to test continuous/process control software appears to be a fairly common concept which has been present for decades [Pobanz & Hernandez,

1989]. This is possibly because of the formalization of the control theory and the representation of processes in the form of equations. Reports of process control emulations abound. For example, [Pobanz & Hernandez, 1989] provides examples of testing control for air supply/exhaust system, surge protection system for a gas compressor, and an oil refinery, and [Cosic, 1992] discusses testing of a digital servo controller for laser-beam targeting. Though continuous and discrete event control are inherently different, three applications are discussed here because of their relevance to this research.

Rihar [1994] describes the development of a software simulator used in the development of control algorithms for the pulp cooking process (in paper manufacture). The simulator itself emulates some operator control actions, the behavior of typical pulp cooking process variables, and important signals from the process equipment. The simulation can be performed in real time or in two different accelerated modes--the feature of relevance to this research. Acceleration--used for testing specific algorithms which have long-term effects--is realized by two different techniques:

- by proportionally decreasing the duration of all batch phases, and
- by increasing the simulation step frequency.

For the first technique, all real phase durations must be replaced by the modified (reduced) values at initialization. However, the shortening factor should be selected carefully by experimentation so as to not distort the real situation, and the possible effect on other application software that the algorithm interacts with must be examined. The second technique is based on increasing the simulation step frequency, i.e., reducing the cycle/step time. At the end of each cycle, simulation activities such as sending simulated values to the application, logging process variables to disk, displaying reports on screen, etc., are performed by the emulation. Acceleration is limited by the LAN and disk activity constraints, and may result in data loss if the threshold is crossed. To avoid data loss, a special algorithm is implemented which forces the prolongation of the simulation step until all data transfer is completed. The author reports reducing the overall testing time from 20 to 5 days by using carefully selected acceleration values.

SIMSMART is a dynamic simulator for design, operator training and automated control of processes (primarily in the pulp and paper industry) [Waye, 1988]. It integrates techniques for discrete event emulation (for PLCs), continuous emulation (for distributed process control systems), and dynamic process simulation (for pump, control valve, piping, and tank models). SIMSMART provides the ability to perform real-time as well as faster-than-real time simulation, although no mention is made of faster-than-real time emulation.

Payne and Mills [1992] have presented a concept called the Virtual Factory (VF). The concept envisions a virtual environment in which closed loop process control software systems and scenarios can be tested without the use of actual manufacturing equipment. The concept calls for replacing all hardware components with software simulation models capable of simulating all hardware components and manufacturing processes. This approach reduces the issues associated with the design, development, test and analysis of a controller. In the VF each simulation model is a separate executing client simulation process which communicates with its neighboring simulation clients through standard communication protocols. The task of reconfiguration is simplified to 'popping' modules in and out since simulations can be executed irrespective of other client simulations. Eventually, the software controller can be moved from the virtual factory to the shop floor without modifications. The ease of reconfigurability and the modularity of VF are important from the perspective of this research and are discussed in the following paragraph.

The VF is composed of three vital components: the distributed process manager (DMAN), a set of communication libraries, and a simulation executive. The VF operates in a distributed environment (networked workstations) which allows component simulations to execute concurrently on the processors linked by the network. The DMAN intelligently selects the appropriate processor for each simulation and provides seamless communication between simulation models. This is made possible by a suite of communication libraries, built on top of the standard socket communication, which allow models and DMAN to connect and send information without making low-level socket system calls. The calls at bottom level--of a three level hierarchy--provide the interface between the virtual factory and actual communication medium, which may be a

network or a RS-232 connection. The middle level is used only by DMAN to set up communications between simulation clients within the VF. The top level provides hardware and process simulation clients with simple message passing calls such as ConnectNet(), SendMsg(), ShutDownNet(), etc. All simulation clients in the VF are developed using a template called the simulation executive. It provides a standard state transition approach to simulation state modification based on changes in simulation variables and time. The template keeps track of time and state, and relieves the simulation developer of the responsibility. Instead it allows the developer to focus on the algorithmic process to perform. Thus the simulation executive permits simulations to be developed quickly and easily. The distributed architecture greatly improves the speed of executing the simulation. In addition, individual simulations or hardware components can interact with any other system without affecting the execution or processing of other models. VF applications run on Sun SPARC workstations and use the X Window System for graphical output of simulation information. Several commercial software packages, such as ABAQUS for FEM analysis, PARTRAN for graphical display and analysis, and MatrixX for process simulation modeling, have been integrated with the virtual factory

2.4.2 Hardware Emulation

In hardware emulation/simulation switches are used to simulate system inputs and lights or meters simulate system outputs. While it represents a safer and less expensive alternative to using actual production equipment, hardware emulation suffers from the drawbacks of limited reconfigurability and reproducibility, as well as reliance on human observation of system outputs [Roberts et al., 1991].

A physical emulator has been constructed at the Manufacturing Systems Laboratory at the University of Illinois [Davis et al., 1993]. The main objective of the emulator is to support research in the design and control of FMS using object-oriented simulation. The emulated FMS has four machining centers, each comprised of a machine Spindle, and a fixturing center which are served by the cell MHS. The machining centers as well as the fixturing center have their own dedicated material handling system, controlled by a dedicated PLC. Neither the machining or fixturing

process is explicitly modeled. The machine center controller references a real-time clock (in lieu of the machine/spindle controller) to update the status of the center, and issues control messages to the MHS PLC. Hence the emulation is performed at the machine/equipment (not machine center) level. The objective of emulating the processing is to permit extended experimentation without the consumption of materials. The cell MHS controller supervises a dedicated PLC which is the AGV controller. The AGV controller is responsible for powering up track segments, resolving track segment contention, determining the location of each AGV, etc. Signals from the movement of AGVs are emulated by a model electric train. Thus, the system adopts two approaches to equipment level emulation: first, software emulation for the machining and fixturing processes, and second, hardware emulation for the AGVs.

In a similar laboratory environment [Co & Chen, 1989], hardware emulation is used to study distributed shop control. While real PLCs and a PC based supervisory controller are used in the FMS, signals from machining centers and material handling equipment are emulated. Manual toggle switches and timers of two PLCs are used to simulate the cycle times of workstations. A model train is used to emulate signals from an AGV which moves between the workcenters. The authors note that limited reconfigurability results from hardware emulation.

2.4.3 Machine Emulation/PLC Testing

An alternative to hardware display panels commonly used for PLC program experimentation is the use of software simulators. Such tools permit the simulation of the execution of the logic system when presented with a variety of logic inputs. The physical PLC is not needed; the scanning operation of the PLC is simulated on the PC. Inputs can be provided through the keyboard and graphical interface, or can be read through the PC port. One such simulator--SIMLOG--has been developed [Ashfal & Balagamwala, 1990] using PROLOG, a language commonly used for programming using predicate logic. The SIMLOG approach is limited in that it only simulates the operation of the PLC, while the model of the manufacturing system is implicit in the ladder. In contrast, the research in software simulation at Virginia Tech [Galgocy, 1986;

Mecker, 1989], explicitly model the system and simulate its operation in addition to the scanning operation of the PLC.

A simulation model has been developed by Galgocy [1986] for use in PLC ladder diagram testing. The model is capable of detecting initial ladder errors, displaying state changes in the ladder, and logging an operations history. The operation of the FMS and PLC are both simulated on a computer using FORTRAN and MACRO assembler languages. The next event list in the simulation consists of two types of events: system, e.g., breaking of a photo beam indicating part arrival, and ladder, i.e., completion of a timing period and the rescheduling of another scan. The concept of ladder events is fundamental to this integrated software simulation (of FMS and PLC operation) approach, and necessitates additional explanation. A time-out completion is scheduled as an (ladder) event when a timer is activated, and the appropriate event is taken off the event list if the timer is disabled. A rescheduling of a scan becomes necessary each time a scan results in a change in state of some element. If a scan does not change an element, subsequent scans are skipped over until the scheduled (simulated) time of the next event. This event will (by definition) change the state of the system and hence a new scan is necessary. While this software simulation approach provided the many benefits of off-line testing of PLC ladders, it placed the burden of developing the simulation program on the user.

Mecker [1989] extended on the preceding work and developed a generalized System Description Language (SDL) for handling the simulations of the physical systems and the corresponding ladder-system interactions via the usage of specially designed constructs. The ladder logic program is external to and separate from the description of the system in SDL format, allowing alternate control logic programs to be tested on the same manufacturing system model. The SDL--based on the SIMSCRIPT language--incorporates the capability of processing the ladder logic and integrating its effects with the system behavior.

Sue-Tang et al. [1987] discuss the development of a system which permits use of computers to simulate factory floor information used by the PLC in its control functions. The hardware component of this system is a custom made I/O rack emulator

card. The PLC is unable to distinguish between the signals from the emulator card and actual signals from the factory floor. Each card emulates up to 16 I/O racks. (A similar concept of using emulator cards to test PLCs is also generally discussed in [Erickson et al., 1987]). The simulation software was under development at the time of publishing the paper. The authors envision a hybrid ladder logic language that features, in addition to the conventional graphical blocks, special user defined blocks to model delays, and typical processes. The dynamics of the special blocks would be programmed using an existing simulation language, C-SIM, which supports discrete event and continuous simulation.

One of the objectives of the SECOIA project (described previously) [Bijou et al., 1987; Courvoisier et al., 1984, 1987] was to provide a validation environment for the FMS control programs. The machines and the material transport equipment are emulated on independent processors connected to their respective controllers (which are implemented on physically separate processors). Emulated machines are modeled by means of timed Petri nets. Actuators are associated with transitions and each time an action is performed by the corresponding PLC, one and only one transition has to be fired in the emulator. The set of enabled transitions in the emulator corresponds to the set of authorized actions on the machines at that time; contradictory actions and be detected and exception handling procedures can be called. The autonomous evolution of the machine are emulated by a discrete event calendar driven by a real time clock. The emulators representing material transport equipment are connected by serial links to the concerned machine controllers in the cell.

Researchers at the Technical University of Denmark have developed a system to test cell control programs using software emulation [Siggard & Alting, 1991]. The emulator mimics the communication interfaces between the cell controller and controllers for production equipment such as CNC machines and robots. The emulator software is made up of three modules: Emulator Creation Module for defining the DNC communication protocols; Emulator Execution Module for actually emulating the protocols defined in the ECM; and Log Book Analysis Module for converting the communication history from a compressed format to a textual report format. With the ECM, DNC protocol can be enhanced to include fault and other spontaneous messages.

These interactive functions can be called up in the EEM for simulating different fault situations. It is possible to survey all communications between the controller and emulator during the emulation, or later by analyzing the log file which records the communication byte for byte.

Krogh et al. [1987] have adopted FACTOR--a commercial, on-line factory scheduling simulation package--for real-time process simulation (i.e., emulation). Their objective is to permit testing of control programs responsible for coordinating activities of equipment such as robots and lathes in a work-cell. A special timing process interacts with the simulation event calendar and data structures to impart real-time capabilities to the 'conventional' simulation software. A C program handles the physical I/O for actuator/sensor signals. In addition, the timing process synchronizes the real-time (computer) and simulation clock by introducing physical delays in the simulation. An advantage of using a simulation package is the ready availability of standard features for model building and analysis. The authors do not discuss the ease of, or even the ability to, transition from a conventional simulation to real-time simulation. However, it is clear that knowledge of internal, perhaps proprietary, data structures is necessary.

2.4.4 AGV Systems

Quinn [1985] describes the use of AutoMod--a high level simulation language based on GPSS--to develop an emulator for AGV Systems. The simulator outputs a detailed trace file of events in the system, which is used by the graphics system to animate the movements and update the status in the display. The emulator searches the simulation trace file for events that would cause the physical system to send a signal or message to the physical controller. For such events the emulator sends the appropriate response to the controller. In return, the emulator translates commands from the controller into future event logic and places them on the event list of the simulator. Thus, the controllers commands affect the "movement" of the AGVs in the simulation model. It is necessary to customize the emulator for the controller architecture and logic of each vendor's systems. The author suggests that since AutoMod contains an internal system of logic representation similar to that in control application software for several material-handling systems, the generation of actual control software is a fairly straightforward task.

Gaskins and Tanchoco [1989] have developed AGVSim2: a discrete event simulation system for testing free-ranging AGV supervisory controller software. The supervisory controller, which is responsible for the routing and dispatching of AGVs, is directly connected to the simulation (i.e., emulation) model. AGVSim2 generates events from the operation of an AGV system in a manufacturing shop, and emulates responses to controller commands. The emulator is flexible in accomodating more than one vehicle type and different system configurations. AGVSim2 contains a module for the graphic animation of the vehicle movements. Information exchange between AGVSim2 and the controller occurs by way of data files. A discrete event approach to simulation is used; no mention is made of real-time simulation. Consequently, it appears that the two systems interact in a synchronous manner (i.e., the simulation clock stops when a decision request is made to the controller), while the actual interaction between the real system and the controller is asynchronous. Controller response time is explicitly taken into account when scheduling the event for the response of the controller, and allows for simulation of events in the shop during the delay associated with the controller response. However, this approach assumes a fixed response time for all decisions, and that an estimate of this time is available.

Miles [1989] discusses some of the important practical concerns of linking a discrete event simulation system to a material handling controller. One issue involves the time advance method in a discrete event simulation and emulation. In conventional discrete event simulation, the simulation clock advances immediately to the next event time. In contrast, during emulation the simulation time must progress smoothly. He provides a mechanism, which uses a reference astronomical time based on the computer system clock, to achieve a smooth simulation clock. The other issue is the interfacing of the hardware and simulation software.

UNI/SIM [Schurholz & Noche, 1987] is a test environment for AGV control software developed at the Fraunhofer Institute, Germany. In the control structure being examined, information is communicated between the controller and the vehicle or workcenter through a well specified message structure, referred to as a control telegram. The simulator (i.e., emulator), which interfaces directly with the AGV controller, is also able to interpret these telegrams. The simulator recreates the

exchange of telegrams, simulates the vehicle's and machine's reaction to the control commands as well as to disturbances in the system. The simulation model analyses the messages it receives and queues appropriate simulation events. In turn, the simulation events generate triggers to transmit status and requirement reports to the controller. A trigger is converted into an appropriate data telegram and transmitted to the controller. The control software and the simulator are both written in C and implemented on separate computers under the UNIX system. The simulator includes an animation system to visualize the movements on the shop floor, and generates a trace file to help debug extended operation of the shop. The authors estimate a 1:3 ratio for efforts of simulator (emulator) development to that of control software. They found the definition of interfaces between the simulator and control system to be an issue of concern.

A later paper [Kockerbeck & Schlictherle, 1990] proposes the integration of UNI/SIM into the UNI/XXX product line for integrated control software testing and maintenance. The other modules include: UNI/EDIT--a graphical editor to describe the system topology; UNI/SCRIPT which generates a language-independent description of the system; UNI/COMP--a pre-compiler which creates modules in the target language such as C; UNI/LEARN which generates interference scenarios for operator training; UNI/DIAG for remote diagnosis and maintenance by the system vendor; and, UNI/PA the performance analyzer for locating bottlenecks.

2.4.5 Shop Control

Researchers at NIST have developed a Hierarchical Control System Emulator (HCSE) [Johnson, T. L. et al., 1982; Bloom et al., 1984] to aid in the design and testing of control systems for the AMRF testbed. This is probably the earliest account of an emulator for discrete or discrete/continuous systems. The emulation tool allows concurrent execution of modules emulating both physical and decision processes. Each module in the control architecture of the AMRF is represented as a finite state machine residing across one or more processors. The control system is specified in state tables which identify all inputs, outputs, states, and state transitions of each subsystem. The controller modules communicate with each other in a synchronous manner by writing

messages in a database designated as "common memory." Each area in the common memory can be written to by only one module but may be read by all modules.

The emulation follows the structure of the AMRF control system. The state-table design of each control module is translated into a standard format based on a computing structure designed to execute state-transition tables. Thus, the emulator is programmed by specifying for each module, conditions--values of internal variables and variables in shared memory; and actions--consisting of code fragments to execute when the condition evaluates to true. (This programming model of condition/action pairs is often naturally suited for systems and machines which have stimulus/response types of controls [Kuhn, 1989].)

The HCSE was initially written in the PARAXIS language to run under the DEC VAX OS but, has since been ported to C [Kuhn, 1989]. The HCSE has been used in the development of the AMRF by permitting hardware to be incrementally added to the system. The modularity of emulator design ensures that each controller module is distinctly identifiable, and this permits functional interchangability with the actual hardware. All levels (workstation, cell, etc.) of the facility are emulated with state tables except the equipment level, which is simulated with simple timing functions. The time scaling function permits the emulator to be used to simulate extended operation of the shop in a fraction of the actual time, or when emulation speed is equal to the actual clock speed, the emulator can function as a mock-up of the actual shop-floor control software. In addition, the emulator has the capability to represent communication and computational delays that would be present in an actual distributed control system connected by a network. Changes of variables in common memory are recorded in a logging file and a snapshot of common memory can be recorded any time. These two features aid in debugging the control software.

A Petri net-based approach to testing FMS control software has been demonstrated in [Muller & Neumann, 1992]. The system considers three levels of hierarchical control: workshop (using PLCs), FMS, and MRP. The FMS controller is responsible for coordinating the computer processes, communicating with the system engineer, and for calculating the optimal schedule. It communicates with the workshop

controllers in real time conditions. The emulator software consists of two parts--each representing executable computer processes: the Emulation Machine Module (EMM), which is a Petri net simulation modeling the behavior of a workshop with machines and jobs; and the Emulation Communication Manager (ECM), which is a communication process for managing the exchange of messages between the FMS controller and emulator. In addition, the ECM provides the real time synchronization.

Johnson et al. [1992] describe a prototype simulation-based system for testing shop-floor control software. A MONITROL/UX-based shop-floor control system is integrated with a SIMAN simulation of a printed circuit board assembly line. The simulation is actually a hybrid software/hardware model. Part of the assembly line is modeled with SIMAN statements while Fishertechnik scaled physical models are used for the conveyors, and inputs from a real bar-code reader are sent to the control system. (The reason for such a hybrid system is neither explained nor is intuitively obvious.) Functionally, the SIMAN model is interfaced with the controller by using SIMAN Event blocks. When an entity arrives at an Event block, a user written subroutine is called which writes information to the serial port of the computer where it is captured and processed by the control system. Synchronization of the simulation clock with the real time is achieved through calls to user-event routines. Though the authors argue in favor of using a simulation model for the design of the system, testing of the control software, and then for on-line decision support, they provide no description of the simulation structure needed to accomplish these objectives.

Ennulat [1992] has developed a framework for a generic emulation for testing system (i.e., shop level) control software. He identifies generic/common system/shop control functions that may be tested by using an emulation model. A demonstration prototype of cell emulation was implemented essentially as distributed simulation. All cell controllers, including the AGVS controller, are emulated on individual computers which were connected through a LAN to the shop controller.

Godio and Vignale [1987] have recommended the development of Emulation Generators (EG) to automate the development of emulators. Their approach centers around the use of predefined routines which can be selected as needed and assembled

into an emulator. This would greatly simply the emulator development task by reducing the effort required for each custom application. Additionally, the library of routines would force de-facto standardization. An EG would provide, in one mode of operation, the functionality to develop new routines, and in the other mode would permit these building blocks to be assembled into a composite emulator. They note that an emulator must be able to mimic the plant from the communication as well as application point of view. They suggest splitting the emulator into two partial ones, each of which addresses only one point of view, and later merging the partial emulators into a complete, detailed, and accurate behavioral representation of the plant. The initial effort at cell and machine control emulation--in an actual injection molding plant--had proved to be unsatisfactory because of non-conformance to emerging European communication standards, and due to insufficient reconfigurability.

2.4.6 Industrial Applications

From the preceding sections it is evident that most emulation applications are in research environments. While industrial applications are common in the continuous process industry, reports of successful use of emulation in discrete-event control systems have only recently begun to appear. One reason is the availability of a product called Automation Master (AM) from HEI Corporation. With AM it is possible to simulate, emulate, and monitor the operation of factories using PLC-based control systems. A case study [Bradshaw, 1987] describes the use of the HEI software to test warehouse control software at Texas Instruments. Most debugging and testing of the software for controlling conveyors, AS/RS, cranes, etc., was done off-line in an office environment. The control software upgrade was done without shutting down operations in only three weekends instead of the estimated six. Harnischfeger Engineers Inc. now routinely make use of emulation modeling (with Automation Master) to test controls for integrated material handling systems [Weil, 1993]. They promise customers 30% startup cost savings in addition to safety benefits. The typical testing functions include load movement and traffic control, load tracking and data transfer, fault definition and diagnostics. However, both Automation Master and AutoMod (described earlier) deal with material handling subsystems such as conveyors, AGVs, palletizers, and carousels.

Supervisory control software for a fully automated material handling system at a new Inland Steel plant was tested using emulation [Voller & Webster, 1991]. Automation Associates Inc. report developing a simulation model of the material flow, and then adapting it for real-time simulation of the individual equipment controllers. They describe the many benefits from using emulation for off-line testing; however, they do not discuss the simulation software used or the issues in adapting the model for emulation.

2.4.7 Real-Time Software

There is copious literature in an area of software engineering called software testing. Over the years, software testing has become an academic discipline as represented by text-books on the subject, e.g., [Beizer, 1990]. Several testing and debugging techniques are described in these books but are not discussed here since it is not the focus of this research. The testing of one class of software--software for real-time systems or real-time embedded systems--is briefly discussed here for completeness.

Quirk [1985] identifies the following characteristics of real-time software: 1) sequencing and timing of the inputs are determined by the real world and not by the programmer; 2) demands on the system may occur in parallel rather than in sequence; 3) the system must meet deadlines in order to satisfy real physical time constraints; mere functional correctness in not sufficient; and, 4) such systems have long mission times. Shop control software displays these characteristics, and additionally it is categorized as a *soft* real-time system, as opposed to a *hard* real-time system in which response time is absolutely critical [Schutz, 1993]. The aspect of testing real-time software that is most relevant to this research is evident from this statement by Hennell, et al. [1987]: "One .... feature of real-time testing is that an environment simulator or emulator .... is almost universally used as a program development and testing aid." Much can be learnt from testing of other real-time systems and applied to testing shop control software. However, most of the literature in real-time systems deals with testing the software *using* rather than on *developing* the environment simulator, i.e., emulator, which is the emphasis of this research.

2.5 OOS/M in Manufacturing

Discrete manufacturing systems have much to gain from object-orientation. A view voiced by Santamarina et al. [1991], and shared by many, is that, "event-driven systems can be better coded with object-oriented languages." Similar agreement is found in the observation by Shewchuk and Chang [1991] that, "the event model (for simulation) is both the easiest to implement in the object-oriented fashion and probably has the most to benefit from this approach." Recent trends in the use of OO methodologies in control, modeling, and simulation of manufacturing systems are presented in this section.

2.5.1 OO Languages

Booch [1994] provides a description of several languages commonly used in OOP. He classifies languages as object-based if they directly support data abstraction and classes, e.g., Ada, Smalltalk-74. An object-oriented language is one that is object based, but also provides support for inheritance and polymorphism, e.g., C++, Smalltalk-80, Ada 9x, CLOS.

Doyle [1990] notes that the advantages of OOP apply most particularly to discrete event simulation, but, these advantages may come at the expense of execution time overhead. He investigates the tradeoff between the benefits and performance of seven languages commonly used in simulation--some object-oriented and some procedural. His experimental results reveal a great disparity between two of the more common OO languages: C++ and Smalltalk. These results echo (and provide quantified support for) the slower performance experienced by Smalltalk users [Drolet et al., 1991; Guo et al., 1990] as well as the relative absence of such problems by C++ users [Joines et al., 1992]. Booch [1994] explains that the speed of C++ (a compiled language) is inherited from its procedural ancestor C, while the mechanism for implementing inheritance and late binding contribute to performance degradation in Smalltalk (an interpreted language).

In another comparison between OO languages for simulation [Thomasama & Madsen, 1990], the authors observe that while the difference in the execution efficiency

of C++ and Smalltalk exists, the performance of Smalltalk systems has shown marked improvement over the years--in fact some even produce compiled code.

An object-oriented language of particular interest is MODSIM II [Belanger, 1990], which supports object-oriented general purpose programming, but differs from other such languages in that it has a built-in simulation world-view. Additional constructs, such as 'ASK' and 'TELL' methods and 'WAIT' statements, allow users to write process based simulations. The built-in object-oriented constructs of MODSIM II include multiple inheritance, dynamic binding, polymorphism, encapsulation, data abstraction, and information hiding. Integrated dynamic graphics capabilities allow real-time display of charts as well as animation of icons. MODSIM II's portability is due to its compiler which emits C code. One possible limitation of MODSIM II may be execution efficiency since its performance was even slower than Smalltalk/V 286 in the comparative study discussed above [Thomasama & Madsen, 1990].

2.5.2 OOS Toolkits and Languages

OOS toolkits provide a library of base classes and simulation specific classes. They could easily be used as a starting point for an object-oriented simulation for manufacturing. Toolkits differ from languages in that they do not provide a comprehensive high-level syntax for writing simulations. Many OOS toolkits and languages have procedural ancestry (i.e., started out as C-based simulation toolkits) are not pure OOS packages.

DISC++ [Blair & Selvaraj, 1989] is a library of routines written in C and C++ which supports the programming of simulation models under both the event scheduling and process interaction world-views. DISC++ is a superset of DISC which consists entirely of C functions. DISC++ consists of two libraries of C functions and C++ classes. The first library consists of basic tools to perform the mechanics of discrete event simulation and, user environment objects for windows, plots, etc.. It contains all the elements to support the event scheduling world-view. The second library consists of a set of base classes in C++ for further development of object classes. These classes support two approaches to the process interaction world-view: network (e.g., SLAM) and process/resource (e.g., Simscript II.5).

CSIM [Schwetman, 1990] is a simulation package which allows programmers to write C and/or C++ programs which are process oriented simulation models. The C++ library is a recent extension to the original C based package.

SimPack [Fishwick, 1992] is a set of C and C++ tools--libraries and executable programs--supporting the creation of executable simulation models. SimPack supports several simulation algorithms--discrete, continuous and combined--but it also supports a variety of model types such as Petri net models, queuing models, etc. It also allows programming at the process level since it compiles process code into event-oriented code. Despite the flexibility available in the form of model types, SimPack does not explicitly support object-oriented programming. The package was originally coded in C and parts of it are being ported over to C++. As a result, it does not yet provide a complete class library.

Sim++ [Lomov & Baezner, 1990] is a C++ package of object types and routines specially designed for writing object-oriented parallel simulations that execute on multiprocessors. The simulation supports discrete event simulation and has a process interaction world-view. The program is transparently scaleable and hence can be executed sequentially on a single processor or in parallel on any number of processors without requiring modifications to the program text. As a commercially available software, Sim++ is unique in its support for parallel simulation, this is a definite advantage for highly complex systems. Sim++ provides standard simulation libraries for random number generation, data collection, and linked list manipulation.

YANSL (Yet ANother Simulation Language) [Joines et al., 1992] is an object-oriented simulation language which provides the ability to create discrete event simulations based on network queuing model using C++. It is important to note that YANSL's main difference from (and, from the perspective of this research, only improvement over) simulation languages, e.g. SLAM or SIMAN, is that it provides the ability to extend the language, i.e., to craft new types of nodes, to satisfy additional modeling requirements. However, the limitations of the other languages still persist in that one is restricted to thinking in terms of entities flowing through a network. It must

thus be kept in mind that Object-Oriented Simulation is more than just a simulation created from an Object-Oriented Language.

The real benefit of using OOS, from the manufacturing systems engineer's perspective, arises from modeling power of application and domain specific classes, i.e., simulation and manufacturing, respectively. Hence, the biggest challenge is the definition of these object classes. As Glassey and Adiga [1989] observe, "We have to deal with the apparently opposing requirements of genericity and immediate utility. While it is true that all future requirements and applications cannot be anticipated, it is also necessary that they must be planned for." In order to accomplish this, a strong conceptual framework, such as those described in the following section, is needed.

2.5.3 OOS Frameworks

A comparative analysis of six persistent research efforts in OOS frameworks is presented in [Narayanan et al., 1994]. The discussion is very comprehensive, and also includes summaries of other research in OO modeling and control. Four frameworks (including ROOCH/HOOPLS which is not analyzed in [Narayanan et al., 1994]) are discussed in this section.

Among the earliest research efforts on a framework for OOS in manufacturing is the BLOCS/M library [Glassey & Adiga, 1989; Adiga & Glassey, 1991]. The framework supports a bottom-up approach to simulation modeling, since the library comprises of objects which can be assembled into a simulation model. The strict separation of control and physical entities, discussed previously, is a consequence of a design philosophy of "one object, one function." The library was motivated primarily by software development concerns, and there is no formal framework for representing manufacturing control.

The OSU-SIM project [Mize et al., 1992; Bhuskute et al., 1992] is motivated by the needs for model reusability and a common unifying framework for modeling techniques such as Petri nets, queuing theory, etc. OOM/S is seen to provide a "natural" modeling environment. OSU-SIM share many characteristics of BLOCS/M: library of classes as building blocks, strict separation of control, physical, and

information elements, and the absence of an internal framework for control representation.

An object-oriented, control-based simulation methodology has been developed by the researchers at University of Illinois [Davis et al., 1994]. The methodology is comprised of the Recursive Object-Oriented Coordination Hierarchy (ROOCH) for modeling the scheduling and control in a FMS, and the Hierarchical Object-Oriented Programmable Logic Simulator (HOOPLS). The basic element of ROOCH is the coordinated object (CO). Each CO contains processing resources which process entities, i.e., jobs and supporting resources (e.g., tools, processing information). The CO is under the control of a supervisor, which may, in turn, be a CO under the control of its supervisor, giving rise to a recursive hierarchy for modeling the levels of control in a FMS. Control over an entity is transferred from one CO to another by using the concept of input/output ports and queues. An entity is under the control of the subordinate CO form the time it enters the Input Queue until it is placed in the Output Port. The corresponding Input Port and Output Queue are under the control of the supervisory CO. Thus a consistent chain of command for control of an entity is created. The ROOCH allows for the explicit consideration of the flow of supporting resources and the controller-interactions which result in "jobs flowing" through a FMS.

A HOOPLS-based FMS model consists of four primary frames: model frame, control frame, processing plan frame, and experimental frame. The control frame contains details resulting from explicitly modeling controller interactions: definition of all control messages, and the resulting state transitions and response message. A special feature of HOOPLS is that in lieu of the traditional event calendar, it employs a message relay. (A similar concept of a message calendar is used in [Mujtaba, 1992].) The message relay stores control messages that are passed among the controllers, and delivers them to the designated recipients at appropriate times. The operation of the message relay mimics the communication network (LAN) linking the controllers in the actual FMS.

OOSIM is an object-oriented architecture for modeling and simulation of manufacturing systems being developed at Georgia Tech [Govindraj et al., 1990;

Narayanan et al., to appear; Govindraj et al., 1993; Narayanan et al., 1992a; Sreekanth et al., 1993; Narayanan et al., 1992b; Bodner et al., 1993]. In addition to direct-image objects for representing entities and their interactions, it provides abstractions for representing manufacturing decision making and control. Four fundamental abstractions are used in OOSIM to represent a manufacturing system: material, location, controllers, and process plans. Locations--which have the capability to either process, store, or move material--are organized into controlled domains (e.g., shop level and cell level). Each controlled domain has a controller. Shared locations belong to more than one controlled domain are used as interfaces for transferring material between domains. (This is similar to the ROOCH concept of input/output queues and ports, though not as rigorous.) Controllers uses abstractions called controller scripts to specify control logic. Scripts are similar to software recipes input to computerized controllers, and are used to explicitly represent decisions such as routing, dispatching, and induction. Thus, a formal structure exists for representing and encapsulating control logic in the simulation.

2.5.4 Class Hierarchy

Shewchuk and Chang [1991] have identified three libraries of classes needed for OOS: base classes, Simulation Support (SS) object classes, and Manufacturing Systems Simulation (MSS) object classes. If we ignore the syntactical differences, we see that this classification of classes is typical of all planned manufacturing OOS projects. The base classes are general classes and, as such, are not unique to simulation applications, e.g., array--a class to operate upon lists. The second set of classes provide support for typical simulation activities such as random number generation, event calendar manipulation (derived from the class array), statistics tracking, etc. These two class libraries are commonly provided in toolkits and languages for OOS.

Differences among OOS projects become obvious in the MSS classes. Although classes such as part, job, machine, etc. must be present in all, the manner in which the functionality is distributed over these classes depends on the perspective of the designer (and in some cases, the set of problems to be examined by the simulations).

For example, a hierarchy composed of active entity classes, e.g., Supplier, Factory, Carrier, and Customer, and passive entity classes, e.g., Orders, Parts, and Product Structures, indicate the modeler's focus on information flows in the manufacturing system [Mujtaba, 1992]. The remainder of this section, compares and contrasts MSS class hierarchies of some OOS applications and frameworks. The emphasis is on the manner in which decision making, i.e., User Control Logic (borrowing the terminology from [Ogle et al., 1991]) is represented in the model.

Researchers at the University of Laval [Montreuil et al., 1995] (also discussed in [Narayanan et al., 1994]) classify manufacturing entities into one of two groups: intelligent entities called agents and non-intelligent entities called objects (Figure 7a). Objects include classes for entities such as Order, Vehicle, and Cell commonly found in the manufacturing system. The focus of this modeling is on mapping real-world decision making entities onto agents. Thus, agent entities encapsulate the control logic of computers as well as of human supervisors. There are agents for scheduling, transportation, material management, etc. In addition to specifying agents, the relationships between agents--which may interact with each other--are also defined.

In addition to the classes to represent jobs, fixtures, process plans, orders, etc., the HOOPLS architecture [Davis et al., 1994] defines a Controller class to explicitly model the many automated controllers typically present in a FMS (Figure 7b). The abstract class controller defines the generic behavior that all controllers possess, including message handling functionality which permits each controller to process and queue incoming messages. Controller subclasses include TransportNode, e.g., ASRS controller, UnitProcessNode, and CoordinateNode, e.g., Cell and Workstation controllers. A separate Message class defines the various messages exchanged among the controllers, and contains fields for the address of the sender and receiver, type and content of the message, and the delay before relaying the message. The objects of the Controller class, along with Message objects, mimic the interaction of the automated controllers which communicate by messages passed over a network.

In the BLOCS/M library [Glassey & Adiga, 1989], resource allocation decision making is contained within an object called Manager (Figure 7c). A dInfo (decision

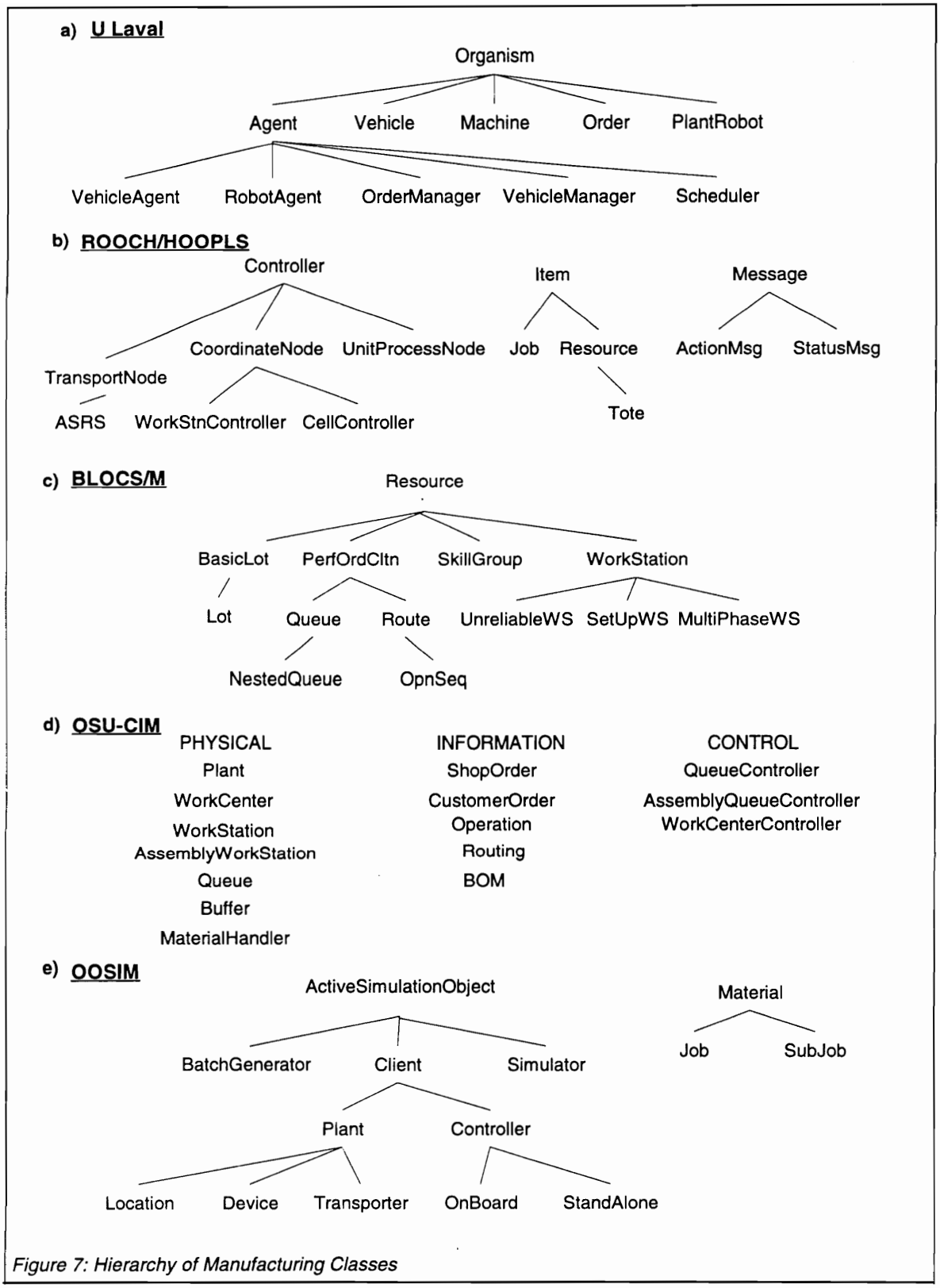


Figure 7: Hierarchy of Manufacturing Classes

information) object accompanies each resource object and contains information (e.g., priority index computation) pertaining to decision making about that resource. Manager objects as well as local Queue objects make decisions based on this information. Thus, while a strict separation between physical and control elements is maintained, there are no abstractions, other than Manager, for modeling control.

While a class hierarchy is not explicitly defined for the OSU-CIM project [Mize et al., 1992], they do provide a description of some classes that would be used (Figure 7d). There are three major groups of objects (based on the modeling approach to separate real-world entities into their components): physical, information, and controller. The controller objects contain the decision making logic. However, there is no internal architecture for control.

In OOSIM [Narayanan et al., to appear] classes are first distinguished into those containing "passive" objects (e.g., Material) and "active objects" (e.g., Device). ActiveSimulationObject is the parent class for all active-object classes; only objects deriving from this class can be placed on the event calendar list. In Figure 7e, a clear separation of physical (plant) entities from control entities in the main class tree can be seen. In keeping with this separation, a Device class is created from an aggregation of locations (processing and storage), and additionally contains an instance of the class OnBoardController (which is a subclass of the class Controller). Controllers such as shop and cell controllers, which are not part of a device and can exist by themselves, are instanced from the subclass StandAloneController. Controller objects contain instances of ControllerScripts for decision-making, and also provide capabilities for communication with other objects.

2.6 Summary

The foregoing discussion of relevant literature lead to the following conclusions:

- Research in automated control systems has emphasized system design, and to a lesser extent, system development. Control software debugging and testing has been relatively neglected.

- Emulation has commonly been used for testing continuous process control systems, while for discrete event systems its use is limited mostly to testing PLC logic, and controllers for AGVS and other subsystems. The trend towards complex, intelligent shop controllers, accentuates the need for means of verifying and validating their implementation.

- While simulation is being used as a tool in various phases of a CIM project, there is no adequate structure for manufacturing simulation which permits a model built for one phase to be reused in another. In particular, simulation models cannot be adapted for emulation, and vice-versa.

- Many object-oriented simulations tend to be simulations implemented in object-oriented languages; the primary benefits are from the software engineering perspective. Recently, the focus has begun to shift towards OOM of manufacturing system entities. There has been some work on OOM of manufacturing control, and on encapsulation of decision-making in OOS. However, there are no published accounts of OOM for emulation.

3. EMULATION MODEL DEVELOPMENT

3.1 Introduction

In this research the process of testing and debugging control software for shop-level controllers using the software emulation approach is viewed as two distinct phases: emulator development; and emulator application. Currently, the first phase involves a significant amount of effort as most emulator development projects start anew due to the absence of a standardized procedure or tools. The second--emulator application--phase is commenced following the development of the emulation model. During this phase the emulator is employed, as a substitute for the physical system, to verify the functionality of and aid in the development of the shop control software.

The stated research objectives of reducing the emulator development effort and moving towards a generalized framework are addressed in this chapter. A methodology for emulator development is presented that is based on exploiting similarities between simulation and emulation to achieve software reuse by way of a dual-purpose modeling structure. A modeling framework that supports the methodology is also described.

The development of a proof-of-concept implementation is described in the subsequent chapter. Issues pertaining to the validation of the emulation and its use in detecting errors in the control software are discussed in Chapter Five.

3.2 Emulator Development Methodology

3.2.1 Overview of Methodology

As previously discussed, a method for developing an emulation--one that is general enough to be used by others--is lacking. The emulation development methodology presented in this research is quite powerful though simple. In its essence, the method for emulation development is to create the emulation by building upon a simulation model. This is a departure from the common practice of building an emulation from the ground up. The methodology involves four main steps as shown in Figure 8. The dashed lines emphasize the fact that the model in each preceding step is not lost in the transition and is always available for use. The first step in the

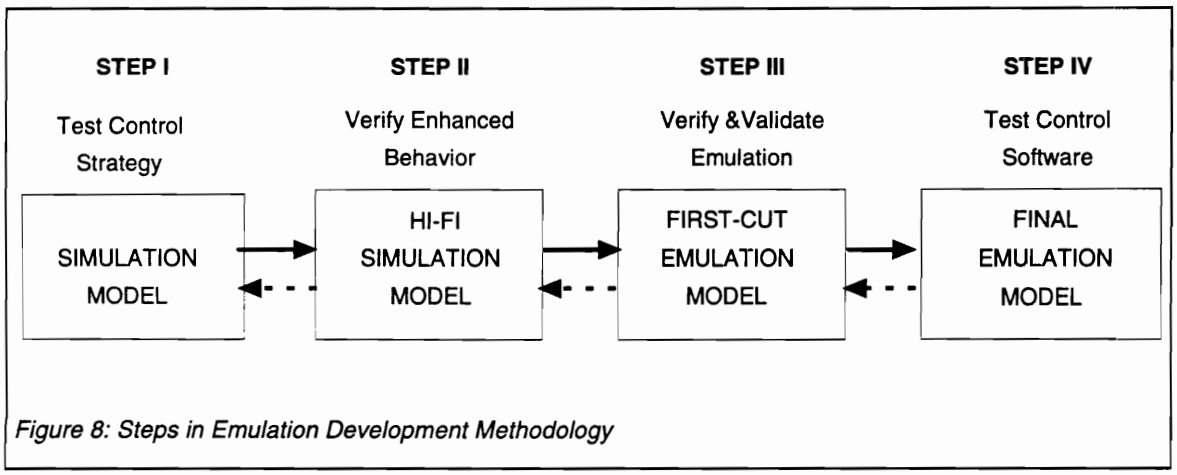


Figure 8: Steps in Emulation Development Methodology

methodology is to implement a simulation model of the system for which the shop control software is to be tested. Shop control strategies are evaluated with this model. In the next step, elements of the model are detailed to make it more suitable from the perspective of emulation. In the third step, a first-cut emulation is crafted by removing the controlling influence (of the shop-level controller), adding a mechanism for real-time advance, and adding capabilities to communicate with an external controller. Many aspects of the emulation can be tested (verified and validated) with this model. In the fourth step, the final emulation model is produced.

The benefits of this approach to emulation development--building upon the simulation--are threefold. First, a replication of modeling effort is avoided, resulting in considerable time savings when the emulation is first developed, as well as later, when a new model is needed following system redesign. Thus the model can evolve with the system. Second, the reliability of the emulation increases because much of the model code has been previously verified and validated during the simulation study. Finally, it becomes possible to use the software for simulation or emulation; switching back and forth between the two modes of operation with minimal effort.

However, conventional modeling methods and simulation languages are a hindrance to this method of emulation development. To overcome these limitations and lend feasibility to the above development method, an object-oriented modeling framework is formulated in this research. The framework emphasizes the object-oriented (OO) paradigm in which the elements of the manufacturing system and simulation-support mechanisms are represented as interacting objects. Under the OO framework (which in itself is not the solution) a modeling structure that is suitable for the simulation *and* emulation tasks is specified. An essential requirement of the framework is that the system be modeled from the perspective of interactions (i.e., communications) among controllers. Guidelines are provided for modeling the important abstractions in the simulation and emulation models.

In the four step method described above, the first step is arguably the most critical one, since it lays the foundation for future expansion. The second and third steps are included to allow an incremental transition from simulation to emulation, and

are good practices for building a reliable and valid emulation. Only the issues of appropriately structuring a simulation model (step one) and then building an emulation (step four) upon it are discussed in this chapter. The issues of validating and testing the emulation model, and the role of the two intermediate models in this task, are temporarily de-emphasized, but described later in Chapter Five.

3.2.2 Issues in Generating an Emulation from a Simulation

Strategies for shop-floor control are commonly evaluated by using a simulation model prior to their selection and implementation. The simulation model used for this purpose is fairly detailed and incorporates most entities, activities, and constraints that affect the shop performance. The approach to building an emulation model by adapting/extending an appropriately structured simulation model of the factory is intuitively appealing because of the similarity in the basic focus of both models: to imitate the behavior of the shop floor. Thus, despite the inherent difference in objectives--strategy evaluation versus software testing--both models have shop control as their central theme. The method is feasible since the simulation model used for control strategy testing has a level of abstraction that is compatible with, though not necessarily identical to, the needs of an emulation model.

Here, the word "compatible" is used to imply "amenable to adaptation" since there is an overlap in requirements for both models. Not all manufacturing system simulation models satisfy this criteria. It is apparent that a model for examining sequence control of equipment in a cell has too much detail and yet does not include the other shop sub-systems. At the other extreme, a model for estimating operating costs of a multi-shop facility has a fundamentally different goal and too high a level of abstraction.

Generating an emulation model from a discrete event simulation model and then connecting it to an actual control system to is conceptually simple [Schmidt et al., 1989; Schurholz & Noche, 1987]. All that is needed is a means for the controller commands to affect the sequence of events in the model (i.e., emulation), and, in turn, for the decision requests and status messages (as a result of simulated events) to be fed back to the

controller. However, in practice, a number of issues have to be resolved and the problem lies in addressing them in a manner that reduces the programming effort.

1. Internal control: During emulation, all of the controlling influence, corresponding to the controller being tested, resides outside of the emulation model. Thus, the internal control needs to be turned off (i.e., remove or disable the code) in the simulation, so that the behavior of the shop model is strictly a result of the logic in the physical controller [Schmidt et al., 1989]. In conventional simulation models, this is often not a straightforward task since the control definition is intermingled with the definition of the physical system and distributed throughout the model. In other cases, especially with commercial simulators, certain control aspects may be implicit in the simulation language and may not be accessed or modified by the user.

2. Data exchange/communication: An interpreter would have to be present at the emulation model--controller interface to preside over the exchange of data between the two components [Schmidt et al., 1989]. Apart from language and format conversions, the interpreter's responsibilities include: 1) mapping signals generated from the model onto syntactically and semantically correct messages to the controller; and 2) translating controller commands into a form that affects events in the model. The problem here is to accomplish this task in a consistent and organized manner.

3. Time-advance mechanism: In a conventional (discrete-event) simulation model, time-advance for all the elements in the model is controlled by a central entity. Time, in a discrete-event model, advances faster than real (wall-clock) time. In contrast, the controller typically operates according to the on-board real-time clock. The shop model and physical controller must evolve at the same rate, and hence a real-time simulation is used during emulation. For simulators that do not have this capability--and many do not--a routine would have to be written to impart real-time capability to the discrete-event simulator, e.g., as in [Miles, 1989].

With the above three essential issues resolved, the model is ready to be connected to the control computer. In addition, the following modifications may also be necessary.

4. Degree of model detail: When used in testing control software, one may typically like to include additional detail in the model than when testing control strategies. The increased detail may be in the form of additional constraints, e.g., computational delays; refined representation of activities, etc. Hence, the simulation model should be extensible, and often models developed from conventional simulation languages are not [Mize et al., 1992].

   A simulation model and an emulation model also differ in the emphasis on statistics. A simulation model is primarily used to generate data to statistically analyze the system. An emulation model provides a stimulus-response mechanism for the control software so that it may be dynamically tested. Data generated from the emulation model is useful only for verifying the control software. Comparisons are typically made on a few key statistics by subjectively eyeballing the data; statistical analysis is not very meaningful. Hence, it is desirable to reduce or disable collection of some data in the emulation model.

5. Testing and debugging aids: The emulation model may need to be enhanced to include aids to assist in testing and debugging the control software. This includes mechanisms to detect improper control commands, and to introduce specific fault situations.

   One way to utilize an emulation would be in the passive role of an event generator to stimulate the control software. In this case, detection of erroneous operation of the controller would be done manually, possibly by examining traces and communication logs. Alternatively, the emulation could play an active role in error detection. To accomplish this the model requires to be enhanced with error trapping abilities. Error trapping may be as simple as parameter range checking, e.g., a negative number in the job number field, or consistency checking, e.g., a command to process a job even though the cell is in a "breakdown" state.

Typically a simulation experiment involves considerations of run length (including terminating versus non-terminating simulations), number of replications, statistics clearing to reduce start up bias, initial loading of system, selection of seeds for random number streams, report printing, etc. Emulation experiments are different; some of the above considerations are irrelevant in emulation testing, e.g., replications, start-up bias reduction, and steady state analysis. Emulation runs are typically of shorter length (in terms of simulation-clock time) but consume a lot of real-time. The testing process involves many short runs which focus on testing a portion of the code, and fewer extended runs for the software as a whole. For emulation experiments, an advanced form of system pre-loading is needed; one that is not limited to merely inserting entities in queues. It is referred to as scenario initialization in this research. A scenario could be defined as a combination of a desired system state and a set of activities specified in absolute or relative time and in a particular order. Scenario initialization allows the control software developer to introduce specific "fault" situations during the emulation experiment and observe the control software's ability to correctly handle it. Thus, while the concept of an experimental frame exists in emulation models, it is quite different from the form used in simulation.

3.3 Modeling Framework

The previous section outlined a method for emulator development. The method is based on the observation that similarities between simulation and emulation can be exploited to achieve software reuse by way of a dual-purpose modeling structure. Conventional simulation software and modeling methods are a hindrance to putting this emulator development method into practice. In this section a modeling framework which makes the method feasible is described. The framework provides guidelines for the design of key simulation model elements to enable a major portion of the simulation code to be reused in the emulation model. As a result, the task of emulator development is reduced to only making emulation-specific enhancements to the simulation model.

3.3.1 Considerations for Dual-purpose Modeling

A dual-purpose modeling structure, i.e., one that is suitable for simulation as well as emulation, must consider similarities as well as differences between the two types of models. The previous section identified some additions that must be made in order to create an emulation from a simulation model. It should be clear that emulation is not a strict superset of simulation. Rather, as seen in Figure 9, the functions of the two models only partially overlap. Thus, it must be recognized that some "things" needed in a simulation model are either useless or a hindrance in an emulation model, and that other "things" are specific to emulation models only.

The appropriate modeling structure should take advantage of the similarities between simulation and emulation and be flexible enough to accommodate the differences. The following characteristics are all desirable: modularity, extensibility, suitability for simulation and emulation, and reusability.

•Modularity is important for simplifying the simulation programmer's task; it limits the effect of a change in one part of the code on the rest of the model and promotes the reuse of pre-packaged pieces of code (libraries).

The argument for separating model from experiment[1] can be extended to separating all simulation infrastructure elements from system elements, as well as separating the controlling influence in the system from the physical entities that are controlled. The benefits of such modularity apply not only to simulation of different control strategies as in [Ogle, 1994] but also to emulation model development. Changes can be made to only those elements that vary from a simulation to an emulation model.

•Extensibility is the ability to extend a program to handle cases not included in the original program with minimal recompilation. Extensibility is *desirable* in a simulation

[1] Partitioning functionality to promote flexibility in a simulation model began with the theoretical work on separation of the model from the experiment [Zeigler, 1985]. A practical application of the concept emerged in the SIMAN simulation language [Pegden, 1982]: "The system model defines the static and dynamic characteristics of the system. In comparison, the experimental frame defines the experimental conditions under which the model is run to generate specific output data. For a given model, there can be many experimental frames resulting in may sets of output data. By separating the model structure and the experimental frames into two distinct elements, different simulations can be run by changing only the experimental frame. The system model remains the same."

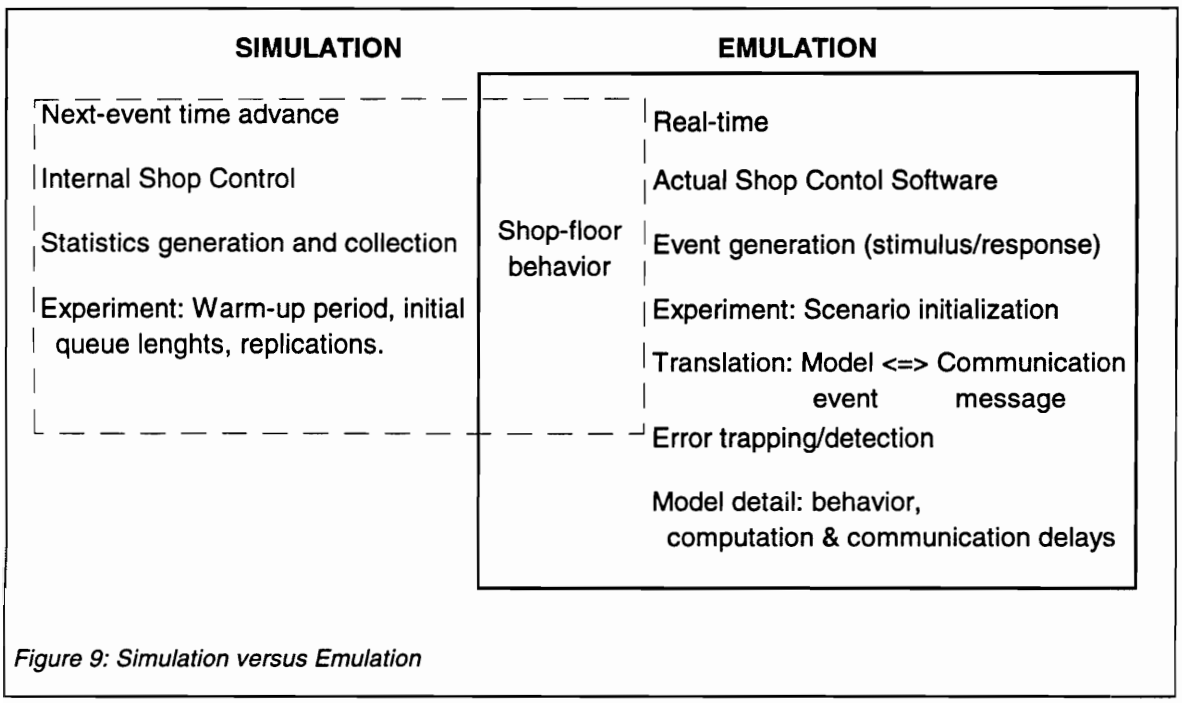


Figure 9: Simulation versus Emulation

model to accommodate changes which, though not always predictable, are nonetheless imminent during the system's lifetime [Bhuskute et al., 1992]. In this situation it is a *fundamental requirement* since the emulation development method mandates extending the simulation to form an emulation model. Thus, model extensibility is necessary since change is guaranteed in order for the model to be useful from one phase of the CIM project to the next (Figure 3).

•Suitability of the modeling structure for the dual purpose mandates that in simplifying the emulator development task, the simulation development task should not be substantially complicated or increased. It is not adequate to merely shift the development effort out of the emulation phase and into the simulation phase; the overall effort should be reduced. Also, each model should be valid in its own right.

•Reusability of the model is an important characteristic since a one time transition from simulation to emulation is not adequate. This is because the manufacturing system goes through the iterative process of design and implementation over its lifetime. Since it is desirable to transition between the models in either direction, it is important to be able to selectively include and exclude features for the particular model.

3.3.1.1 Limitations of Network-flow Simulation Languages

Going by the above mentioned list of desirable characteristics, the commonly used network-flow modeling approach proves to be inappropriate for many reasons. While blocks representing the behavior of queues, resources, delays, etc. provide a modular approach to model development, they are "essentially hard wired to each other in the code" [Ogle, 1994]. Changes to the model, especially those involving user control logic, are thus very difficult to implement. Additionally, complex control logic is very difficult to model using the network modeling constructs [Davis et al., 1993; Narayanan et al., to appear].

Although these constructs have been used to develop simulation as well as emulation models [Johnson et al., 1992; Voller & Webster, 1991], it results in one-time solutions without links between the two models. There is no reusability of code from

one model to another; changes must be made separately to the simulation and emulation models to keep them updated.

Extensibility is *not* a characteristic of conventional simulation modeling languages [Bhuskute et al., 1992], partly due to the procedural language upon which the simulation language is based. In such procedural languages, it is common to use "switch" statements to handle the selection of behavior based on object type. However, when a new object type is incorporated, it requires the addition of code in each relevant switch statement. This not only tends to be error prone but also requires recompilation of, perhaps large, program modules. Thus, procedural programming languages (and practices) inhibit extensibility.

A simulation model emphasizing the flow of jobs through the system is not ideally suited for emulation. The job-flow modeling perspective assumes that "the job knows what to do next" [Shewchuk & Chang, 1991]. This view directs attention away from the true source of decision making, i.e., the controllers. The decision making logic is dispersed through the model and cannot be readily separated. In using such a model for emulation, it is necessary to identify locations in the model code where a signal must be sent out to the actual control system; this is not an intuitive or straightforward task.

A better alternative--controller-interaction modeling perspective [Harmonosky and Barrick, 1988]--is described in the next section[2]. But, a network-flow language (NFL), such as SIMAN and SLAM, is less suited for the controller-interaction modeling perspective for two main reasons: the underlying procedural language (e.g., FORTRAN), and the entity flow view that is enforced. The modeling constructs (the "objects") in NFLs are designed to support the job-flow perspective of "queue, seize, and free resources", and it is not possible to either modify the behavior of these built-in objects or add to new object types. Entities representing communications message can be made to flow through the model to represent the controller-interaction modeling perspective. However, a communications message does not seize any resources

[2] The arguments in following paragraphs neccesitate use of concepts which are introduced to the reader only latter in the document.

(except, perhaps at a much lower level of abstraction, such as in modeling of the communications network itself).

The implementation of the controller-interaction modeling perspective, as described in [Harmonosky and Barrick, 1988], requires modeling of two entity types flowing through the system: physical parts and requests for service. In contrast, nothing is required to flow through the system in OOS, as seen in the proof-of-concept implementation in Chapter Four. Also, the synchronization of the flow of both these entities does not appear to be an intuitive task. Further, the authors note that, "A part is delayed in different queues for communications delays and for processing or handling delays, without actually seizing resources ..." As a result, statistics on equipment utilization were required to be collected manually, thus, losing the benefits of the built-in resource construct in the NFL.

In NFLs, entities representing requests for service have a set of attributes (as do all entities) which typically store numerical data representing the message. This data is interpreted in the control logic code using some form of a "switch" statement, which as explained earlier, inhibits extensibility. A message object in OOS may not only hold any data type, but more importantly, also encapsulates methods for using, communicating, or modifying the data contained in the message. Substantial benefit can be derived from using inheritance for various message sub-classes [Macro, et al., 1994], and a new sub-class may be added without changing the code that reacts to the message content. The model sections representing the control computer, i.e., control logic, need to be written separately, in the underlying procedural language (although not explicitly stated in [Harmonosky and Barrick, 1988] this is a fair conclusion which is supported by a similar implementation in [Barnichon, et al., 1990]), thus, once again losing the benefits of OOP.

In terms of usefulness in dual-purpose modeling, and specifically the controller-interaction perspective, perhaps the biggest drawback of the network-flow languages is the lack of support for data-hiding. Information in the simulation model, e.g., number of jobs in the input queue of a machine, is globally visible and may be used for decision making in any part of the model. While this does not usually present a problem in

conventional uses of simulation models, it can have serious consequences when the decision making component is removed from the model, as for emulation. Use of data which is "out of bounds" potentially affects the validity of the emulation, as described later in Section 5.1.1.

These problems, taken together with the previously described limitations of network-flow languages, make it clear that though the controller-interaction perspective can be implemented--for simulation alone--in network-flow languages, a model for the *dual* role of simulation and emulation, based on this perspective, is better implemented with OOM.

The foregoing discussion indicates that conventional simulation software and traditional modeling techniques are not ideally suited for dual-purpose modeling (with an eye towards emulator development). The modeling framework which supports the emulation methodology developed in this research, and presented in the following section, recommends using OOM techniques to develop a modular, reusable, and extensible model structure. These qualities help make the model more amenable to the dual functions of simulation and emulation.

3.3.2 Development of a Dual-purpose Modeling Structure

The OO approach (to programming, modeling, and simulation) holds a lot of promise for overcoming the above limitations and is used in this research. OO has been the focus of much research attention especially for issues of controlling manufacturing systems. Recently, researchers have documented that high-fidelity modeling is more easily and intuitively done with OO methods [Davis et al., 1993; Narayanan et al., to appear; Bhuskute et al., 1992].

OOP promotes a clear separation between the object interface and the implementation of the object methods. As long as the interface to an object is constant, its implementation may be modified without affecting the other objects that interact with it. This feature directly promotes modularity. Encapsulation, or data-hiding, is an integral part of the object-oriented paradigm, and can be used to limit the visibility of data in the model sections. In OOP, extensibility can be obtained through inheritance and polymorphism. Using inheritance, a new subclass can be derived by merely

specifying the differences from the superclass. Polymorphism permits redefinition of a method in the subclass, and thus override the behavior specified in the parent class. A single, identical message sent by a client to objects of both the classes will result in two different behaviors. This eliminates the need to define a new message in each client when a new case (i.e., object type) is added.

With the OOP inheritance mechanism, there is not just a one-time path from simulation to emulation. Capabilities may be added to objects for emulation testing without affecting the simulation model. But, if changes are made to the simulation model in a matter that affects the emulation, e.g., the number of buffer spaces between two processes, the emulation is automatically updated.

Since an emulation model is defined as one which "mimics the manufacturing system from the perspective of the controller" [Godio & Vignale, 1987], a simulation model based on modeling the interactions among controllers would prove to be more appropriate. The explicit representation of the control system within the model of the manufacturing system readily enables the exclusion of the control logic from the model as required during emulation. Focusing on the communication between controllers is much more suitable in addition to being amenable for both simulation and emulation. It is contended in this research that the controller-interaction modeling perspective is inherently suited to the OOM environment (though originally proposed and implemented in a network-flow language--SIMAN). There is a conceptual similarity between objects exchanging messages and control computers communicating by messages over a network.

This research recommends adopting the following three principles while developing a simulation model which is used as the basis for the emulation model:

1. Use OOM techniques to separate the functionality in a simulation model. The major groups of entities are those that: 1) control and support simulation; 2) represent the physical elements in the manufacturing system; and 3) perform decision-making to control the operation of entities in the manufacturing system.

2. Adopt a controller-interaction perspective for simulation modeling. An alternative to the traditional view of events being a consequence of a job flowing through the system is that of events being associated with the communication among controllers; the flow of jobs is a result of the controller interactions.

3. Design the class structure keeping in mind the dual nature of the model. While the simulation model contains only software objects (models of physical objects) the emulation system additionally contains at least one physical object, namely, the shop controller. The class structure must have provisions to allow interfacing with the physical controller. Also, there should be provisions to accommodate the differences between simulation and emulation experiments.

The OOM approach, identified in the first principle, achieves a separation of concerns and results in flexibility. Elements of the model can be easily identified and individually modified. Also, all manufacturing system control decision making is brought together within one set of entities. The most critical of these entities for this research is the shop controller object. The logic associated with it is isolated from the significant amount of decision making within the model that is not affected by the transition between simulation and emulation.

The use of the second principle directs the modeler's focus to provide a clear definition of the interface to the various controllers and a set of messages that are exchanged among them. This brings us closer to mimicking the real-world communication/interaction, and thus conceptually simplifies interfacing the emulation model with the actual shop controller.

The third principle acknowledges that the use of OOP/OOM/OOS in itself is not a panacea for all problems. Object-oriented (OO) methodologies are tools whose appropriate use must be planned for in order to maximize the potential benefits. While developing the class hierarchy, significant attention should be paid to how the design affects the transition between simulation and emulation. For example, in an appropriately defined class structure, an actual controller could be substituted for the controller object with minimal changes. The interaction with the actual controller would be transparent to the other objects in the simulation.

3.3.2.1 Overview of Framework Implementation

The following paragraphs provide an overview of an OOM method for systematically integrating an actual shop controller with the simulation/emulation model. The system is modeled as a collection of interacting controllers which drive the activities in the system. The controllers and the other elements of the system are represented as objects which encapsulate the behavior of their physical counterparts. Additionally there are simulation infrastructure objects, which provide the mechanics of the simulation (event calendar, time advance mechanism, etc.)

In the simulation mode, manufacturing system objects interact with each other either directly (by invoking the object method), or through simulation infrastructure objects (by sending event messages). The simulation model focuses on the interaction between the SimShopCon object and other controller objects (Figure 10). SimShopCon is an abstraction of the logic in the actual shop controller, i.e., all simulation logic which is associated with the shop-level controller is collected in SimShopCon. During emulation, the SimShopCon object is replaced with VirShopCon--a virtual shop controller object. The VirShopCon object lacks the decision making abilities of the SimShopCon object. It merely forwards messages from objects in the emulated system to the actual shop controller after performing the necessary translation. Additionally, the VirShopCon receives messages transmitted by the actual shop controller and conveys the messages to the appropriate controller objects in the emulated system. However, since VirShopCon presents a similar object interface as SimShopCon, this switch occurs transparently to other objects and without mandating their modification.

The developed methodology addresses the previously identified implementation issues as follows:

1. Internal control: By replacing the SimShopCon object with VirShopCon, which is devoid of decision making capabilities, internal control is removed from the simulation program. The only shop-level control influence exercised over the emulated system originates from the actual shop controller. This addresses issue number one.

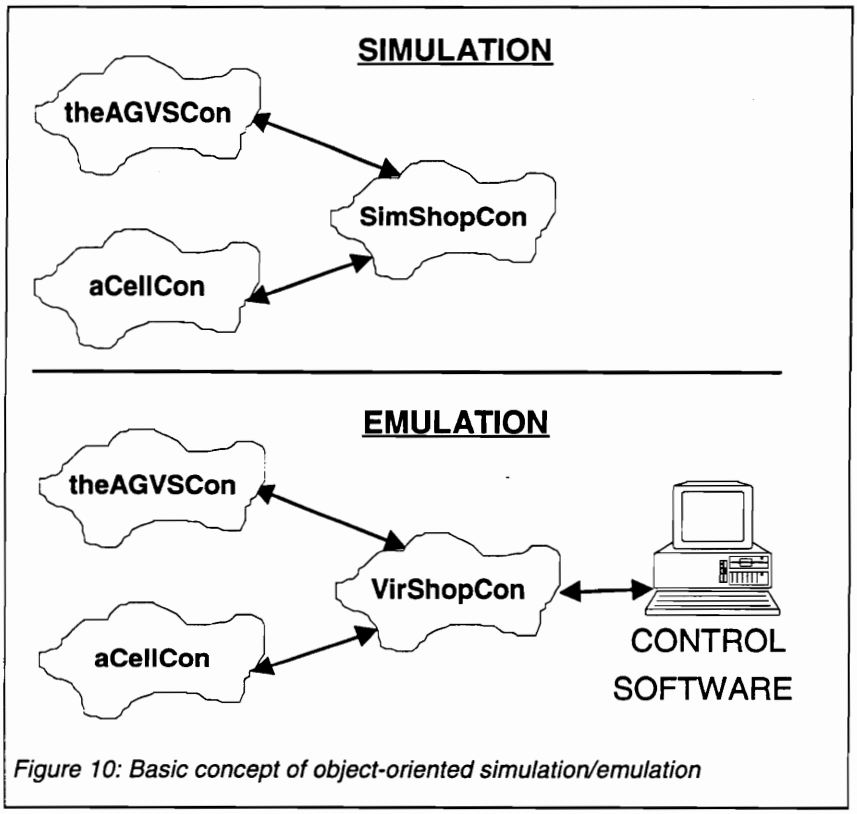


Figure 10: Basic concept of object-oriented simulation/emulation

2. Data exchange and communication: All communication between the actual shop controller and the emulated system is channeled through VirShopCon. This provides a centralized location for the bi-directional interpreter services. As an object in the simulation program, the VirShopCon object has the ability to interact with the simulation mechanism. It is responsible for interpreting commands from the actual shop controller and introducing the appropriate events (i.e., messages to other objects, in OOS) in the emulation. Similarly, messages from other objects are interpreted and communicated to the actual shop controller using the same protocol as corresponding actual controller (cell, factory, or human). Thus issue number two is resolved in a systematic manner.

3. Time-advance mechanism: A new class is derived by inheriting from the simulation infrastructure object which provides the time advance capability. In the new class the method for time advance is overridden and replaced with one which synchronizes the passage of simulation time with the time on the temporal clock. Objects in the model are oblivious to and unaffected by this change.

4. Degree of model detail: If required, any additional model detail is conveniently added by using the OO inheritance mechanism, i.e., deriving new manufacturing system objects from existing ones and enhancing their methods or creating additional ones. Modularity and encapsulation/data-hiding, which are characteristic of OOP, help pinpoint specific objects and methods that need to be altered and limit the effect of the change to the remainder of the software.

5. Testing and debugging aids: Most enhancements in this category can be made under the OOM framework since manufacturing system objects are not strongly coupled to simulation infrastructure objects. The differences between simulation and emulation experimentation can be accommodated in distinct implementations of experiment objects for a each model.

3.3.2.2 Guidelines for OOM of System Elements for Simulation

To make the above emulation development methodology possible, special attention must be paid to the design of the objects in the system. In this section, guidelines for modeling abstractions that have a major influence on the simulation and

emulation model are described. These include: 1) jobs and orders, 2) controllers, 3) human supervisors, 4) data/knowledge-bases, 5) cells/workstations, 6) statistics tracking, 7) system.

1) Jobs and Orders: For the controller-interaction modeling perspective, jobs and orders are modeled as passive objects, i.e., they do not trigger simulation events directly. Events in the system result from the messages exchanged between controllers; as a consequence of these events, the state of the jobs and orders is altered until they are "complete." Jobs and orders are treated more as information entities and less as physical entities. This is a departure from the traditional modeling view. Software objects that represent jobs and orders do not have any logic methods. These objects only have methods for updating their state and for performing tasks associated with the software, e.g., duplicating, deleting, equality testing, etc. Routing information is not attached to them but instead is part of the knowledge of a controller.

2) Controllers: Since the software of the shop controller is being tested with the emulation model, the modeling of the controller is central to the framework. In this research, a controller is seen as performing two major functions: decision making and communicating. The controller uses its internal decision making logic to arrive at a decision based on available data and the knowledge about the system. This decision is then conveyed to other entities such as controllers, equipment, human supervisor, which are within its control domain and others which are outside, e.g., supervisory controller. Additionally, external entities provide feedback and other input (including decision requests) causing the controller to use its decision making logic. Hence the communications function provides a means for the controller to exercise its influence over external entities and vice versa.

It is common for research relating to OOM/S of manufacturing control systems to ignore the representation of the communication function in the model of controllers. For example, in [Ogle, 1994] and [Mize et al., 1992] the controller class is used primarily as a container for logically encapsulating the control logic into modular objects. The main objective is to separate the description of the physical system elements from control logic in the simulation model.

In such models, controller objects, and other objects, interact by message passing. While conceptually similar, this is not the same as exchange of messages in a communications system. In OOP, message passing implies invocation of a method of another object, but the message is not broadcast [Entsminger, 1990]. In the context of controllers communicating over a network, a message is a concept in the vocabulary of the problem domain, at a higher level of abstraction[3]. Communication messages that are passed in a CIM system fall into two main categories:

- between control computers and equipment controllers,
- among control computers.

The second category of message passing is of interest to us since it is associated with higher levels of control (e.g., cell, shop, factory).

In support of OOS efforts that do not consider the communications aspect of system behavior, it may be argued that such consideration is not critical in all simulation studies, e.g., evaluation of alternate control strategies. But, often, the communications function is not only a time-elapsing operation but also dictates a part of controller's decision making logic, and thus cannot be ignored. Hence, it is inappropriate for some high-fidelity simulation models, and for all emulation models, to consider *only* the controller's decision making capabilities. The viewpoint that, in order to accurately mimic the CIM system an emulator must mimic the plant from a communications standpoint too, is asserted by both [Godio & Vignale, 1987; Clark & Withers, 1989].

The controller models used in OOSIM [Narayanan et al., to appear; Govindraj et al., 1993] and HOOPLS [Davis et al., 1994] identify communication as one of the primary capabilities of controllers in a manufacturing system. However, they do not explicitly address design issues related to use of the controller object in both simulation and emulation. A major consideration is that while inclusion of communications delays, failures, and other aspects of communications is a fundamental requirement in an emulation model, it is not so in *all* simulation models. An appropriately designed

[3] This explanation is paraphrased from an example of a train control system in [Booch, 1994, pg. 455].

controller object should have provisions for allowing the communications function to be represented, with varying degrees of detail, as appropriate for the particular model. The key to the design for the dual purpose is to understand the interactions of the shop-level controller, i.e., controller being tested, with other entities in the system. A shop-level controller interacts possibly with other controllers (superior, peer, and subordinate), a human supervisor, and a central database.

To support the modeling needs of both simulation and emulation, a conceptual model of a controller was developed in this research. According to this model, a *controller* is composed of three interacting components: *data*, *logic*, and *communication* (see Figure 11). *Data* and *logic* are considered private to the *controller* and are thus not visible to other *controller*s. They are, however, visible to the *communication* component, and the *communication* component is in turn visible to other *controller*s. The *communication* component encapsulates the *data* and *logic* and thus acts as a sort of middleman. *Controller*s interact by the exchange of *messages*. *Messages* are generated by and acted upon by the methods of the *logic* component. However, transmission of the *message* from the sending *controller* to the receiving *controller* involves only each other's *communication* methods. That is to say that a *controller* does not directly invoke another *controller*'s decision making method. Similarly, since the *data* component is private to a *controller*, it cannot be directly accessed. Data exchange between *controller*s should only be done by the sending and receiving of *messages*. In general, any interaction between the controller and an entity which is physically outside the control computer must result in a communication message. The chief motivation behind the notion that *data* and *logic* are private to the *controller* object, is to enforce the "controller-interaction" modeling perspective and assist in its implementation.

This model of controllers and their interaction is primarily targeted towards automated controllers, but may be used for other entities as well. These entities, which may be categorized as DecInfo objects (for 'Decision and Information'), include human supervisors and databases, and are discussed separately.

To increase the modularity and flexibility of the implementation, the *logic* and *communication* components should be packaged in separate classes. The class which

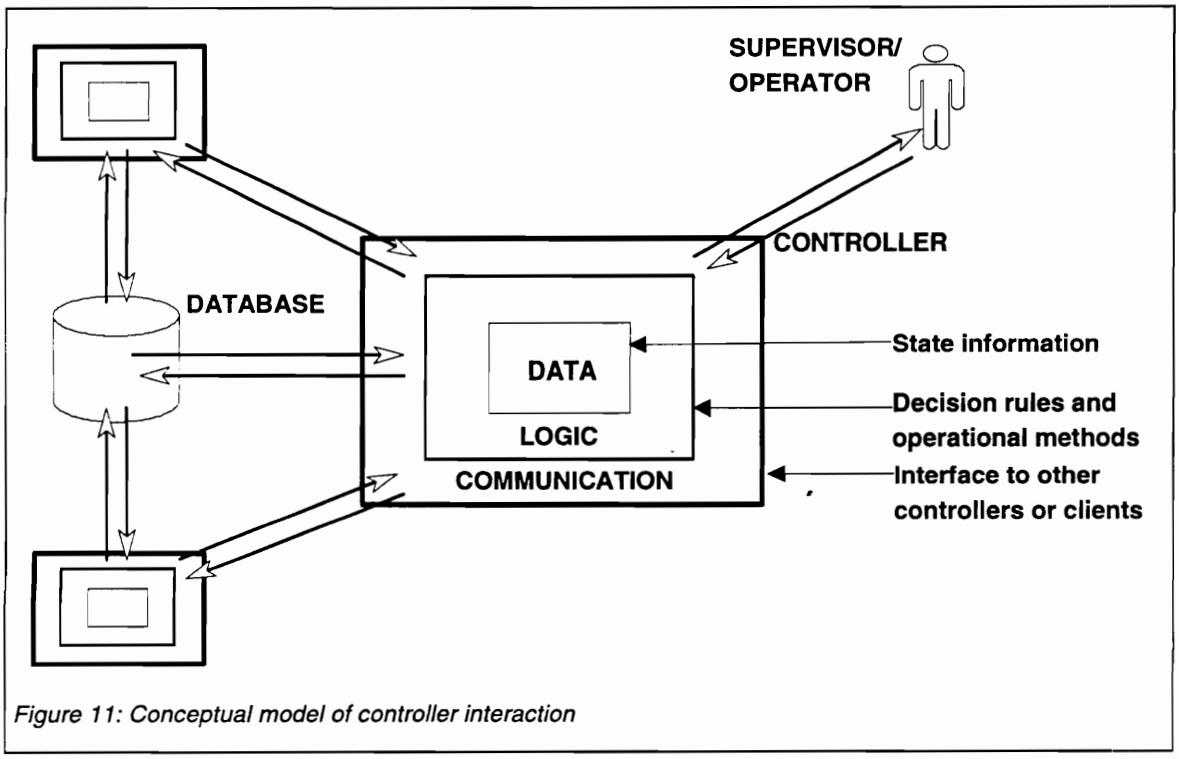


Figure 11: Conceptual model of controller interaction

encapsulates the communications is termed the InterfaceObj class, while the fields and methods for decision making are packaged into the LogicObj class. The complete functionality of the controller class results from the union of these two classes. One way of implementing this union is by making the controller class to be a container for an instance of the logic class and of the interface class. Thus, a controller object *has_a* LogicObj and a InterfaceObj, and the LogicObj *uses* the services of InterfaceObj to communicate a decision to other objects.

The flexibility alluded to earlier derives from the ability to use combinations of different logic and interface objects in a controller object. During initial simulation, a simInterfaceObj is used. This object merely passes on the message to the receiver assuming an error free, instant transmission. Later, in a hi-fidelity simulation model, the same LogicObj is used, however the communications function is modeled by a subclass of the InterfaceObj class. In this the default transmission method is overridden and enhanced with communication delays and network reliability considerations. Finally, for emulation, the controller object uses an instance of emInterfaceObj. This object has sendComm and getComm methods which know the protocol to communicate with the actual controller. However, the controller object used in the emulation does not posses decision making capabilities. This can be represented by a emLogicObj which has no methods for shop control. Hence the interface object provides a systematic means of incorporating increasingly complex communications details into controller objects.

At the very least the interface object should provide the ability to convey a message to and receive a message from other DecInfo objects. In its most basic form the message contains a reference to a specific logic method of the receiver. When the message object is told to execute, it invokes the logic method on behalf of the sender (another DecInfo object). Thus, the interface object merely represents an indirect way of invoking a DecInfo objects' decision making logic. This basic level of representing communication is appropriate for situations where it is cumbersome to consider communication details, as in a simulation model for testing control strategies. The inclusion of this additional step in the object interaction process (message passing) is a small price to pay for the extensibility and flexibility it affords in the future.

3) Human supervisors: Most manufacturing systems are not fully automated and require some form of human intervention or supervision. Even the most sophisticated knowledge-based control architectures allow for a human to examine and override the automated controllers' decision [Manivannan & Banks, 1992]. It is thus important to include the human controller in the simulation/emulation model. During emulation testing, it is desirable to control the variable behavior of all entities so that specific scenarios may be tested repeatedly in a comparable manner. It is fair to say that human behavior is not highly repeatable especially in terms of reaction time. Hence, putting the human interaction/behavior into the emulation model has the advantage of reducing a source of uncontrolled variability. Initial testing, which focuses on the production control logic component of the shop control software, may be done in this fashion. Later, when the complete software including the user interface, is being tested the interaction with the human controller is no longer emulated but is replaced by an actual human.

The conceptual model of the controller described above is flexible enough to represent a human supervisor/controller too. Even though the human-computer interaction occurs through a different medium (typically, monitor and keyboard) than the computer-computer interaction (communications network), the interaction can be modeled as an exchange of messages. The human is modeled as a DecInfo object, and messages are exchanged between the *communications* module (i.e., InterfaceObj) of the human supervisor object and the controller object.

4) Databases and Knowledge-bases: Data used by a controller may be: a part of the control program; owned by another controller; separate but residing in a database program on the same or another computer; or, in a common database in another computer.

The decision on whether to model external databases and knowledge-bases as separate entities or simply label the data as private, should be influenced by the controller software testing process. If during the testing process, the database will always be treated as an integral component of the shop controller, then it need not be represented as a separate model entity. In this case, the data may be labeled private

regardless of where it resides. On the other hand if there is a possibility that a portion of the testing is done with an emulated database (due to reliability or repeatability considerations) but later replaced by an actual database program, the database should be modeled as a separate entity. The object should be treated as a DecInfo object, the *communication* component encapsulating the data or knowledge retrieval logic as well as the data. If the SimShopCon object requires the use of this data, it should be obtained by the exchange of message objects.

5) Cells: Cells are associated with a cell-level controller which dictates the operation of the cell elements. They have a field (and a method for setting it) which holds a reference to an instance of a cell controller object. Thus it is much easier to change the specific instance of the controller since is not physically bound to the cell.

6) Statistics tracking and reporting: It is important to make a distinction between tracking and reporting. In OOS, tracking of statistical data ceases to be a system-wide activity; where appropriate, individual objects are given the responsibility of collecting statistics on themselves. Objects representing system elements such as resources and queues, that are common targets of data for statistical analysis, should have the ability built into them at the class level. Statistics reporting, on the other hand, is a central activity under the control of the Experiment Object (EO). (More details on the concept of an EO can be found in [Ogle, 1994]) Variables and object fields which are to be tracked should be registered with the EO. The EO has the responsibility of enabling the statistics collection of only those variables and object fields which are of interest in the particular simulation study. Hence, statistics are collected and reported only on those objects that receive a message such as `SetTracking(TRUE)`. This separation of responsibilities is beneficial for simulation as well as emulation.

7) System: The system object is the top level object in the simulation or emulation model. Its task is to assemble a system, from a set of system element objects described above, for a specific modeling purpose. This arrangement increases the likelihood that only one module is modified when the "players" change. Relationships, such as *has* and *uses*, between specific instances of objects can be explicitly set by this object. For example, the system object invokes the

`setMaster(shopConObj)` message to inform instances of the cell controller class that `shopConObj` is the object instance to whom they send status information for a particular model. By making the system object responsible for setting relationships, the binding of relationships between objects at the object definition level is discouraged [Ogle, 1994]. This flexibility is desired for transitioning between simulation and emulation without requiring excessive changes to system element objects.

In summary, the foregoing guidelines for the object-oriented modeling of key simulation model elements are as follows: Jobs and orders are modeled as passive, information entities. This is because controllers, and not jobs, are the primary active entities in the model. Controller objects are designed so that all data and logic contained in them may be accessed by other controller or manufacturing system entities only through its communication module, i.e., the methods of the communications module represent the interface to other objects and conceal the implementation of the decision making logic. Also, a controller object is implemented as an union of a logic object and a interface object. Entities belonging to the class referred to as DecInfo objects, e.g., human supervisors, databases and knowledge-bases, may be modeled in a similar manner in order to facilitate seamless interaction with their physical counterparts during emulation. The differing emphasis on statistics in simulation and emulation is accommodated by separating the statistics collection task from the reporting task. Individual (manufacturing) objects have methods to collect statistics upon themselves, but do so only when directed by a system-level experiment object which does the statistics reporting. Links/coupling between objects are not defined at the object definition level, instead instances of the objects needed in a particular model are created by a system object which also establishes the relationships between them.

3.3.2.3 Emulation-specific Additions/Enhancements

This section describes how a simulation model, which is based on the modeling guidelines presented in the previous section, can be extended to create a basic emulation model. A basic emulation model is one that can be connected to the shop controller and can react to its commands. A brief discussion of enhancements for testing and debugging is also provided.

3.3.2.3.1 Time advance

The time advance mechanism has to be modified for running the emulation. In OOS, time advance is the responsibility of an object, which may be referred to as SimTimeMgr. For emulation, an EmTimeMgr is created by inheriting from SimTimeMgr object and providing a new implementation for the time advance method. In the new implementation, passage of simulation clock time is synchronized with that of wall-clock time. The reference to an instance of EmTimeMgr is captured in ModelTimeMgr--the same variable that references a SimTimeMgr instance in the simulation. Objects that have to interact with ModelTimeMgr are not affected by this change since the set of messages (i.e., method definitions) remain unchanged. Additional methods which provide control over the real-time advance mechanism may be added to the EmTimeMgr. This includes turning on and off this feature; specifying a time scaling factor (for "slow-motion" or "time-lapse"); or, selecting a particular means of accomplishing real-time advance. Of course, the clients of the ModelTimeMgr will have to be modified to send these new messages and take advantage of the additional capabilities.

3.3.2.3.2 Detailing behavior

If the modeler wishes to increase the level of detail for the emulation model, the OO inheritance mechanism is the primary means for doing so. New (manufacturing system) objects may be derived from those used in the simulation model. Behaviors in the derived object may be modified by overriding inherited methods without affecting the clients of that object. This is made possible by the OOP feature of polymorphism which allows multiple objects to have methods of the same name but possibly different implementations.

Relatively minor modifications which involve extending the behavior, are accomplished by including a call to the inherited method in the code for the new method and adding the code for the additional behavior. For example, the computational delay associated with a decision making method of a controller object may be included in the emulated object in this manner. More extensive modifications may be made by completely re-coding the overridden method in the derived object. This includes eliminating unwanted behavior by providing a NULL implementation, e.g., when

simulated communication delays are replaced by actual delays in the emulation. However, one should be aware that re-coding breaks the inheritance link so that future changes to the parent method may need to be re-implemented in the derived object.

New behavior may be introduced by adding new methods to the derived class. For example, in an emulation the cell controllers may request the shop controller to download a part-program for a particular job, whereas this activity was not important enough to be modeled in the simulation.

Another means of increasing the detail in emulation model is by adding new objects to incorporate additional constraints into the model. In the simulation model one may assume that movement of jobs between cells occurs without problems. For emulation, it may be desirable to represent the automated guided vehicles as objects in the model, including interference (zone control considerations) and breakdowns.

When adding detail to the emulation, the focus of the modeler should be on how the modifications affect the interaction, i.e., communication, with the shop controller object. If the change involves the addition of new logic methods in a cell controller, it is more likely that the communications with the shop controller are affected. On the other hand, detailing an existing method is less likely to alter the interaction. However, it is very possible that even such a change results in the need to communicate a new type of fault situation to the shop controller.

The most important realization is that the original objects, used in the simulation, remain unchanged. Hence, at any time during the emulation development, the simulation model may be rerun to reproduce the original results. At the same time, changes to the simulation model, e.g., increasing the number of resources, or the default queue priority, are automatically carried over into the emulation model.

It should be noted here that sometimes it is desirable for the detail added for emulation to be brought back into the simulation. This can be accomplished under the OO paradigm, but without the help of an automated mechanism, such as inheritance. It must be done by manually moving the code associated with the detailing out of the emulation and into the parent class used in the simulation. The derived class, used in

the emulation then inherits this capability. This action of identifying basic elements of similar objects and pulling them to a higher level in the class hierarchy is referred to as abstraction and is an integral part of the object-oriented design process.

3.3.2.3.3 Emulation Interface Object

The purpose of the emulation interface object is to interface an emulation model with a physical controller This object is primarily responsible for translating information from the emulation model into a format acceptable to the actual shop controller, and vice versa. To accomplish this it needs to have the ability to communicate with the proper protocol, and perform the mapping between internal and external messages. In the interest of flexibility, the communications task should be separate from the mapping task. The actual communications protocol should be encapsulated within "generic" object methods such as GetComm and SendComm; clients of the object or other parts of the object implementation do not directly call the lower level communications functions. It is important that an appropriate data structure be used to hold the message mapping information. In particular, the use of "case" statements must be avoided since it impedes extensibility. The mapping may even be delegated to the different message classes and thus distribute implementation of the dictionary (although a distributed dictionary might be more difficult to maintain).

3.3.2.3.4 Virtual Shop Controller

The virtual shop controller object primarily uses the capabilities imparted to it by the emulation interface object. Some mechanism is needed to ensure that decision making capabilities (that may have been inherited) are either not present or are disabled from being used. The method for achieving this depends on the specific implementation of controller object. For example, if the object has a field for holding a reference to a logic object, then the field may contain a NULL pointer or a pointer to an object with virtual methods, i.e., no implementation.

3.3.2.3.5 Statistics Collection

In the emulation model there may be a need to track fewer statistics, primarily to reduce the computational overhead, or other statistics (e.g., message count) than in the simulation. By creating a new experiment object for the emulation model, it should be

possible to turn off the statistics collection on all variables that are not helpful in verifying the functionality of the control software.

3.3.2.3.6 Scenario Initialization/Fault Introduction

Adding the scenario initialization and fault introduction capabilities to the model requires an extension of the experimental frame concept discussed earlier. A scenario could be implemented in an object that is used by or contained in the Experiment Object. Alternately, it may even be a method of the Experiment Object, since a scenario conceptually fits well into the process-oriented simulation view. A scenario can be modeled as a process which direct objects to perform actions, at specific times or at occurrence of certain events, by scheduling delivery of messages. This is not to be confused with the random generation of events in simulation. Scenario initialization is not limited to failures, and could include any specific situation, e.g., arrival of two rush orders at the same time. In general, it is some specific perturbation to the regular operation of the shop and for that particular model run. It may be possible to formalize the description and representation of scenarios by the concept of scripts used in OOSIM [Narayanan et al., to appear]. The emulation serves as an event generator and stimulates the shop controller.

3.3.2.3.7 Error Trapping

Error trapping may be implemented at a local level or a system/global level. At the local level, an object is only concerned with the validity of its own state and/or the state transition command. The implementation of this ability is relatively simple and may consist of extending one or more methods (by inheritance) or adding a new method to perform the check.

Error trapping at the global level involves tracking/monitoring the status of shop floor elements on a system-wide basis and flagging an invalid *system* state before or after the system enters it. (Note that the system may be in an invalid state even if the elements are all in valid states.) For example, the material handing controller is asked to deliver a part to a work cell despite its input buffer being full. It is evident that implementing global level error trapping is more complicated.

In keeping with the OO framework, the error trapping task can be assigned to a Debug Manager (DM) object. The DM should have methods for periodically (at fixed times or predetermined events) retrieving and/or receiving states of system elements. For this purpose it needs to have both global visibility, and unrestricted access to the fields of all objects. Global visibility can be accomplished by making it a component of the System object (in a similar manner as the Time Manager object, for example). Accessing object fields through methods is cumbersome but should be preferred over making the fields public. The C++ concept of a FRIEND class and an intermediate level of export control--PROTECTED--is ideal for this situation. The DM also needs some type of rule base to perform the parameter or consistency checks. The rule base may be implemented internally within its methods or externally in an expert system. Initially the rule base may contain a few rules which cover anticipated problems based on knowledge of the system, and later be expanded as the testing and debugging progresses.

3.4 Summary

The material presented in this chapter addresses a research void, namely the lack of a method for emulation development for testing shop control software. The method presented in this research is unique in that entails that the emulation be developed by a stepwise transition from a simulation model; in the past, the tasks of simulation and emulation model development have been viewed as being separate and non-overlapping. The developed method avoids a replication of effort and limits the emulation development task to making emulation specific enhancements to a simulation model.

The dual-purpose modeling structure that supports the above method, represents a research contribution since it eliminates the dichotomy that exists between simulation and emulation model development. Such a structure requires essentially that the simulation be object-oriented and be based on modeling the interaction among the controllers in the system. All logic associated with a controller is encapsulated into a object, so that the controlling influence can be easily removed for the emulation model. In addition, since all control influence is channeled through the methods of an interface

object, controller objects become oblivious to whether control commands originate from another controller object in the model or an actual controller. This greatly simplifies, and lends structure to, the process of interfacing a simulation model with an actual controller. Further, OOM of simulation support and infrastructure elements ensures that emulation specific enhancements can be made easily. This includes detailing the behavior of system elements as well as incorporating scenario initialization and fault detection capabilities. Thus, the conceptual OOM framework enables the stepwise transition from simulation to emulation and helps overcome the limitation that current emulation efforts tend to be ad hoc in nature.

The following chapter decries a proof-of-concept implementation for the modeling framework presented in this chapter. The issues of emulation validation and application are discussed in Chapter Five.

4. PROOF-OF-CONCEPT IMPLEMENTATION

4.1 Introduction

The concepts developed in the previous chapter are implemented in a prototype system to demonstrate their viability. In particular, the prototype demonstrates the feasibility of transition between simulation and emulation, and simplification of this transition by the use of OOM methods. The simulation and emulation models are implemented in the context of a prototype shop controller for an actual system.

4.2 Description of the System

The Flexible Machining and Assembly System (FMAS) was used as the basis of the system modeled in the prototype (Figure 12). The FMAS is housed in the Robotics and Automation Laboratory in 161 Whittemore Hall. The FMAS has three major components: a machining workcell (MWC), an assembly workcell (AWC), and a material handling system (MHC). The machining cell has one IBM 7545 robot for loading and unloading parts from two DYNA CNC milling machines. In the assembly cell, a larger IBM 7547 robot performs kit-building and machined parts assembly. A Shuttleworth conveyor system for inter-cell material handling, and an Automated Storage and Retrieval System (ASRS) comprise the material handling system.

The control system for the FMAS is implemented in a three layer hierarchy: System/Shop, Cell, Equipment. Each of the major components described above is under the control of a cell-level controller implemented on a PC. The two cell controllers and the material handling controller are supervised by a shop or system controller which is implemented on a fourth PC. A strict control hierarchy is maintained by not permitting any direct communication between cell-level controllers. Thus a move request from the assembly cell controller is routed to the material handling controller via the shop controller. At the highest level of control (Factory control in the NIST reference model) is a human supervisor who enters orders into the shop controller (SC) and may occasionally override the normal sequence of operations.

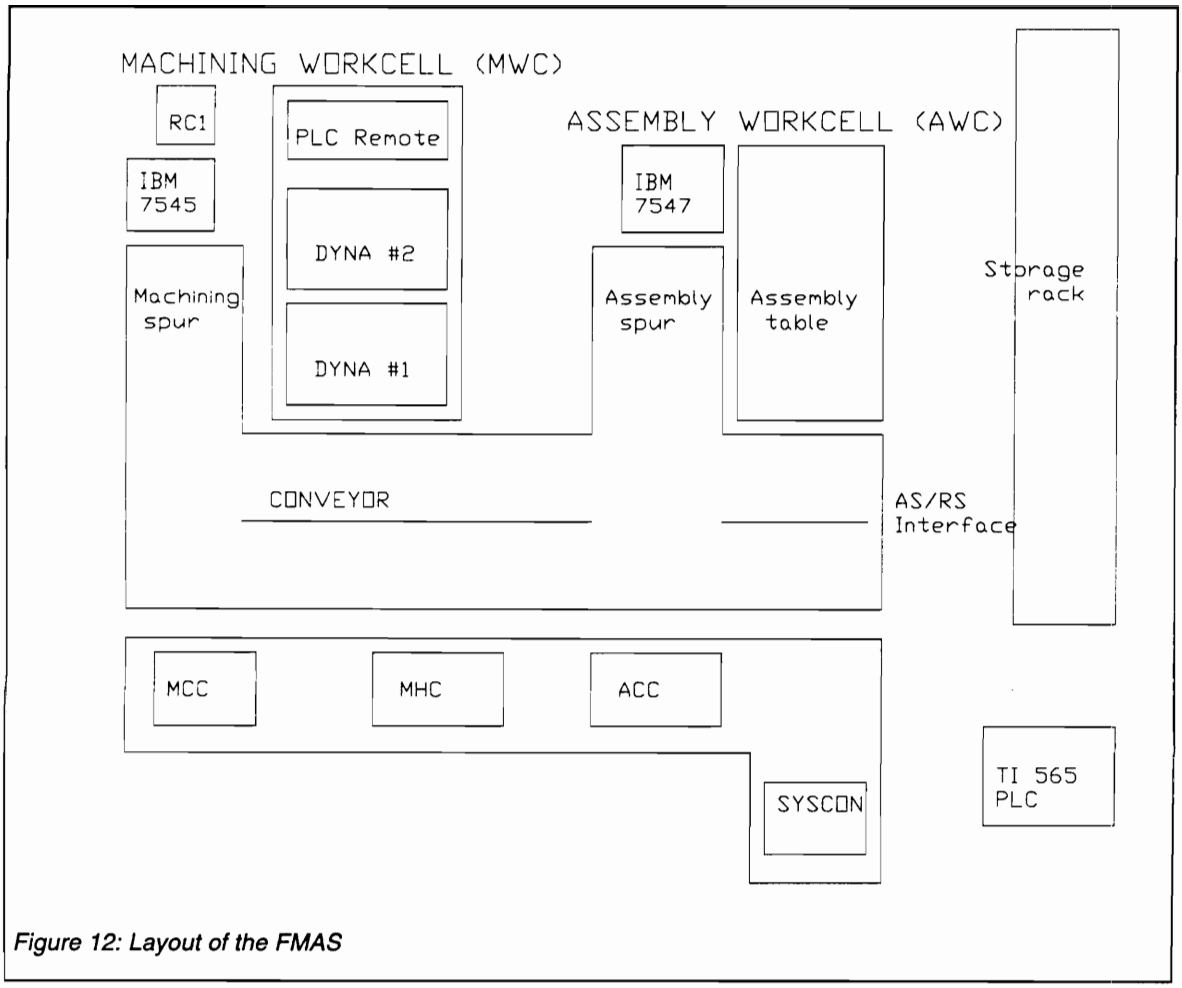


Figure 12: Layout of the FMAS

The communication between the three cell controllers and the SC is based on file transfer using the mailbox mechanism. There are two files associated with each cell controller and they reside on the hard-drive of the communications network server. The SC writes commands for the cell controllers in the xxxin.dat files and cell controllers write their responses in the xxxout.dat files. Each cell controller periodically examines the corresponding input file while the SC monitors all three output files for any messages.

4.2.1 Operation of the FMAS

The FMAS is capable of producing a family of products and is currently set up to make two product types: a robot and a milling machine. These products, which are models of the equipment used in the FMAS, are made of parts machined from wax blocks and held together by pins.

A job for either of the products follows the following set of activities:

1. Kit Building: The appropriate type and quantity of raw material (wax blocks) is loaded from the feeders onto an empty pallet at the assembly cell.

2. Machining: At the machining cell, raw material on the pallet is machined on the CNC mills.

3. Assembly: The assembly cell robot attaches the links to the bodies with one or more pins.

Other activities in the system include:

1. Storage and Retrieval: Empty pallets are unloaded from the ASRS onto the conveyor, and pallets containing the kited parts or assembled product are loaded back into the ASRS. The assembled product is manually removed off the stored pallet and packaged.

2. Replenishment: Feeders in the assembly cell are periodically replenished by a human operator. In this demonstration prototype, this activity is not modeled.

For this demonstration, the shop control strategy described in the following paragraph is used. The SC exercises three major forms of control: job release, job movement, and job prioritization.

Each order received by the SC is broken down into jobs, and the jobs are then released to the shop floor immediately (on resource availability). The first major control exercised by the SC is of the CONWIP type--work in process (WIP) is maintained below a specified maximum level. In addition, to WIP level constraint, pallet availability determines release into the shop floor. (The maximum WIP level is determined by runs of the simulation model.)

Once a job is released, it is routed through the activities described above. However, since the machining operation is very long and the each job must visit the assembly cell twice, kits are not always sent to the MWC directly. Instead, if the MWC is busy, the kit is temporarily returned to storage, thus freeing up the AWC to build the next kit (or assemble the job currently at the MWC). This represents the second major control exercised by the SC. The SC's third major decision involves prioritizing jobs contending for the same resource. Partly done jobs take priority over new jobs. Hence, an empty pallet is not unloaded from the ASRS if a pallet containing a kit is present in storage. Similarly, an empty pallet is sent to the AWC for kit building only if a pallet containing machined parts is not present at the MWC.

A simulation model was developed to ensure that the strategy met acceptance criteria. Later, the shop control software which implements this strategy was developed and tested using an emulation model created for the purpose.

4.3 Implementation Environment

The prototype system consists of two parts: a simulation/emulation model of the shop; and a shop controller. Each of these parts is implemented as a separate program which runs on a separate processor connected by a LAN. This arrangement prevents computational power interference and it represents the real-world system with greater fidelity. Specifically, the entire simulation model runs on the workstation named RT4,

while during the emulation phase, the shop control software runs on RT3 and communicates across a network with the emulation model of the FMAS on RT4.

4.3.1 Hardware

Two IBM RS6000 workstations, running the AIX operating system (the IBM version of UNIX), were used as the hardware platform. Each of these workstations have a 20 MHz RISC CPU, 32 Mb of RAM, and collectively more than 2 Gb of hard drive space, thus affording adequate computing power for running the simulation.

4.3.2 Software

The modeling framework described in the previous chapter is meant to be independent of the choice of software platform (as long as it supports object-orientation). However, the specific features provided by the language will strongly influence the actual implementation details.

Initially, three object-oriented simulation software platforms were considered: SimPack [Fishwick, 1992], C++SIM [Little & McCue, 1994], and MODSIM II [Belanger, 1990]. The first two offer the power and speed of the C++ language, are available as freeware[4], and provide source-code for libraries of base classes and classes for supporting simulation mechanics. In comparison to SimPack, C++SIM has a stronger object-orientation, is more extensively documented, and provides support for concurrency and asynchronous object interaction.

The third candidate was the MODSIM II language--a product of CACI Products Company. MODSIM II is an object-oriented simulation language which supports process-based discrete simulations. Additionally, it provides built-in general and simulation support classes and functions, an environment for simulation programming and debugging, and capabilities for graphical displays and animation. MODSIM II also provides an interface to C (support for C++ is available in MODSIM III) for writing custom routines.

---

[4] SimPack is available via anonymous ftp from ftp.cis.ufl.edu and C++SIM from arjuna.ncl.ac.uk. Authors for both programs also maintain internet home pages, and their respective URLs are: http://www.cis.ufl.edu/~fishwick, and http://ulgham.ncl.ac.uk/C++SIM/homepage.html.

From an ongoing research perspective, SimPack and C++SIM afford the maximum flexibility in molding the final product to meet specific research needs. Unlike MODSIM II, the two C++ tool-kits come with no strings attached (licensing fees), but also necessitate reinventing the wheel to some extent, and do not have any of the "bells and whistles" found in the more fully developed CACI product. The maturity of the software, by virtue of its commercial availability since seven years, and the availability of copious documentation and technical support were also major advantages of MODSIM II. In the end, CACI's waiver of the licensing fee for use in academic research, sealed the decision in favor of MODSIM II.

4.3.2.1 Brief Introduction to MODSIM II

As in all OOL, an object in MODSIM II is essentially an encapsulation of data and code. The data describes the object's current status and is stored in the fields of an object. The code describes what the object does and is provided in the methods[5] of the object. An object is programmed in two parts: its definition/declaration, and the implementation. An object type declaration simply specifies the data fields and methods of an object and serves as the interface to that object. The actual code for the methods is supplied in the object implementation block and is not visible to other objects.

MODSIM II supports and promotes modular program development through the provision of library modules. Each library module typically contains declarations for a set of related objects and procedures and the executable code which constitutes the procedures and methods. A library module consists of two parts, each of which is stored in its own file and complied separately. One is the Definition module and the other is the Implementation module.

In MODSIM II, an object field may be of any type such as `INTEGER`, `ARRAY`, or even `OBJECT`. Object methods may either be `ASK` methods or `TELL` methods. `ASK` methods execute instantaneously with respect to simulation time and are similar to function calls in languages such as C. When an `ASK` method is invoked, the caller

[5] In C++ terminology, member functions.

pauses and control passes to the invoked ASK method. When the invoked method completes, the caller resumes. No simulation time can pass in an ASK method.

In contrast, TELL methods are asynchronous. When the TELL method is invoked, it is simply scheduled for execution, and the caller immediately continues execution without pausing for the TELL method to execute. The TELL method then starts execution under control of the built-in simulation timing routine at the correct simulation time. Simulation time can elapse in TELL method.

Process oriented simulation is supported directly by built-in language constructs--primarily WAIT statements. The WAIT statement is used to make simulated time pass. Two forms of the WAIT statement let methods synchronize themselves.

By default, the data fields and methods of an object instance are visible to all other parts of a program. Fields may be read by using an ASK statement but may be modified only by the object itself. Methods may be invoked by using the appropriate statement, ASK, TELL, or WAIT FOR depending on method type and behavior desired. The visibility of fields and methods, i.e., ability to read or invoke, outside the object instance may be limited by declaring them to be PRIVATE. (In comparison, C++ provides three levels of export control: PUBLIC, PRIVATE, and PROTECTED.)

Multiple inheritance is supported in MODSIM II. With inheritance, new object types can be defined in terms of existing object types. It is possible to OVERRIDE the behavior of the inherited method, if its behavior is no longer appropriate for the newly defined object. In the new method a more elaborate behavior can be defined.

MODSIM II provides libraries of object types to support simulation modeling, e.g., ResourceObj and RandomObj, and other general object types, such as QueueObj and StreamObj. ResourceObj and grouping objects such as StatQueueObj have a built-in ability to gather statistics. A unique feature of MODSIM II is the monitored variable which has detailed statistics gathered every time its value is modified. Statistics collection on object fields and program variables is thus greatly simplified.

4.4 Design and Implementation of Classes for the FMAS

4.4.1 Class controllerObj

The class `controllerObj` is a base abstract class, i.e., there are no instances of this class. It provides the methods and fields that all controllers must have. For example, a controller must have the ability to initialize and shutdown itself and everything in its domain. These abilities are provided by the `TELL` methods `initialize` and `shutdown` (Table 1). Since it is expected that the specific implementation will be provided within the subclass/derived class, these are virtual methods. The two methods are labeled `PRIVATE`, as are all internal decision making methods, so that they cannot be invoked from outside the controller objects. This design conforms to the conceptual model of the controller described in Section 3.3.2.2 which is meant to enforce the controller-interaction modeling perspective.

Three classes inherit from `controllerObj`: `factConObj`, `shopConObj`, `cellConObj` (Table 1). One instance of each of these classes, or a derived class, is used in a simulation or emulation model of the FMAS (Figure 13).

4.4.2 Classes for factory controllers

The `factConObj` class serves as the base class for factory-level controller objects. In the FMAS, control at this level is performed by a human supervisor (and not a computer) who is included in the emulation model. As described in the modeling guidelines earlier, the physical aspect of the supervisor is not important from the perspective of interaction with other controller objects, since all entities belonging to the category of DecInfo objects are modeled in the same manner. The controlling behavior of the human supervisor is captured in the sub-class `supervisorObj` (Table 2).

As with all other higher level controllers, `supervisorObj` uses `inform` and `receive` methods inherited from `simISObj`, to communicate with other controller objects. The class `supervisorObj` has two non-`PRIVATE` methods: `generateOrders` and `setSlave`. `generateOrders` simulates order arrivals to the factory. It is first invoked from outside the object (by `systemObj`) to schedule the first arrival, and the method puts itself on the activity list for future arrivals. The `ASK` method

Table 1: Definition of controller object classes

```
TYPE
  controllerObj = OBJECT
    PRIVATE
    id : INTEGER;
    status  : cellStatusType;
    TELL METHOD initialize;
    TELL METHOD shutdown;
    ASK METHOD setStatus(IN newStatus: cellStatusType);
  END OBJECT; {controllerObj}



TYPE
  factConObj = OBJECT(controllerObj)
  END OBJECT; {factConObj}



TYPE
  shopConObj = OBJECT(controllerObj)
    acc : assCellConObj;
    mcc : macCellConObj;
    mhc : mhcCellConObj;
    master : interfaceObj;
    ASK METHOD setMaster(IN boss:interfaceObj);
    ASK METHOD ObjInit;
  END OBJECT; {shopConObj}



TYPE
  cellConObj = OBJECT (controllerObj)
  END OBJECT; {cellConObj}
```

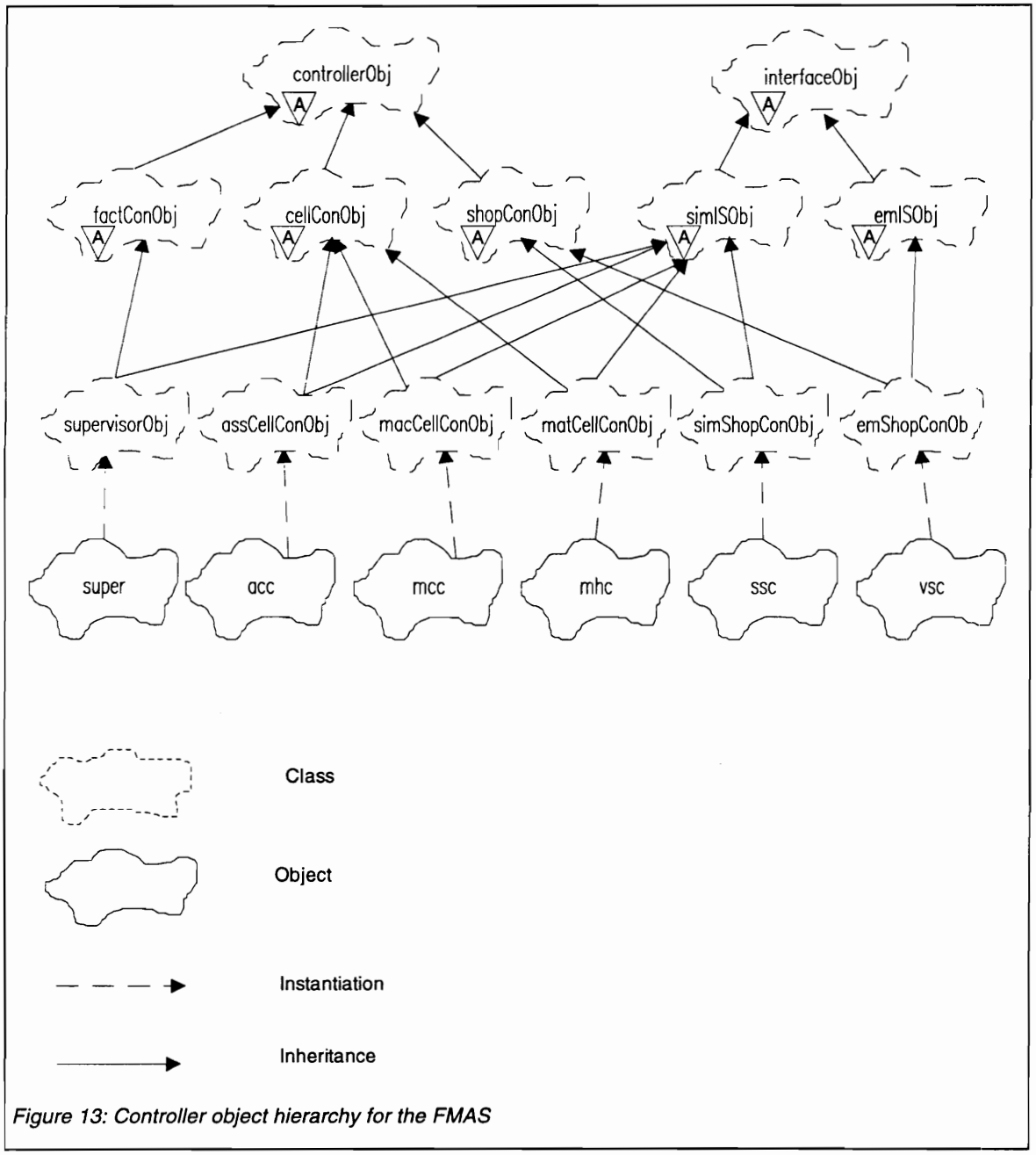


Figure 13: Controller object hierarchy for the FMAS

Table 2: Definition of supervisor object class

```
TYPE
  supervisorObj = OBJECT(factConObj,simISObj)
    stream1,
    stream2,
    stream3,
    stream4 : RandomObj;
    maxOrders,
    doneOrders :  INTEGER;
    slave:interfaceObj; {of the shop controller}
    ASK METHOD setSlave(IN scon:interfaceObj);
    TELL METHOD generateOrders(IN timeBetOrd:REAL);
    ASK METHOD generateFields;
    OVERRIDE
    TELL METHOD initialize;
    TELL METHOD receive(IN msg:messageObj);
    ASK METHOD ObjInit;
  END OBJECT; {supervisorObj}
```

`setSlave` is used to establish a coupling with a particular instance of a shop controller object. The reference to the subordinate (shop-level) controller is captured in the variable `slave`, which is of type `interfaceObj` (rather than `controllerObj` or its sub-classes) to allow visibility of only the methods and fields inherited from `interfaceObj`. Thus, with reference to the controller model described in Section 3.3.2.2, the *logic* module is hidden and all interaction between controller objects is channeled through the *communications* module.

4.4.3 Classes for shop controllers

Since two main types of shop controller objects are needed for our situation--one for use in the simulation model and the other for the emulation model--a class is designed for each (Table 3). While both classes inherit from class `shopConObj`, `simShopConObj` inherits "communications" methods from the `simISObj`, and `virtualShopConObj` from `emISObj` (Figure 13). The `shopConObj` class provides both sub-classes the `master` field for holding the reference to an instance of the factory-level controller object. The `master` field, as well as the `slave` field, is implemented as reference variable which hold the reference to an object that is instanced elsewhere (somewhat similar to the C concept of a pointer). The field is assigned a value using the `setMaster` method. This, in conjunction with the `setSlave` method of the `factConObj` described earlier, allows relationships between instances of controller objects to be established at compile time. Thus, the `slave` field of an instance of `supervisorObj` may equivalently hold a reference to an instance of either `simShopConObj` or `virtualShopConObj`. In other words, since the object instances are not connected at the design phase, the controller classes may be implemented independently of each other. This is consistent with the modeling guidelines described earlier which recommend that the coupling between objects be provided by the top-level system object.

For the sake of simplicity in this demonstration prototype, a similar, formal link between the `shopConObj` and its domain--cell controller objects--is not established. Instead, a reference variable for each of the objects is included in the definition of `shopConObj` and the objects are instanced in the `ObjInit` method. Thus, the relationship is implicit and somewhat hard-wired into the design. This is acceptable for

Table 3: Definition of shop-level controller object classes

```
TYPE
  simShopConObj = OBJECT(shopConObj, simISObj);
    acIOBuf,
    mcIOBuf,
    stInBuf : ResourceObj;
    PRIVATE
    TELL METHOD processOrder(IN pri, numC, numR:INTEGER);
    TELL METHOD make(IN job:jobObj; IN order:orderObj);
    TELL METHOD store(IN job:jobObj);
    OVERRIDE
    TELL METHOD initialize;
    TELL METHOD receive(IN msg:messageObj);
    ASK METHOD ObjInit;
  END OBJECT; {simShopConObj}



TYPE
  virtualShopConObj = OBJECT(shopConObj, emISObj)
    TELL METHOD operate;
    OVERRIDE
    ASK METHOD mapIP;
    ASK METHOD mapOP;
    TELL METHOD inform(IN msg:messageObj);
    TELL METHOD receive(IN msg:messageObj);
    ASK METHOD ObjInit;
  END OBJECT; {virtualShopConObj}
```

our example since only the shop controller object is changed between the simulation and emulation models. To accommodate the (more likely) situation that even the cell controller objects are somehow enhanced for emulation, `shopConObj` and `cellConObj` classes should have a similar mechanism for setting the "master-slave" relationship.

In `simShopConObj`, three methods are added which contain the shop-level control logic. The method `processOrder` is invoked by the object instance when it is informed of the arrival of a new order by its master--the factory-level controller object. For each order, this method creates one instance of `orderObj`, and as many instances of `jobObj` as jobs contained in the order. The shop controller's responsibility of coordinating the activities of the shop floor is modeled as a process of guiding one job through a sequence of activities. This process is captured primarily in the `make` method and supplemented by `store`. The basic flow of the job through the system is specified by `make`, while `store` is a sub-process dealing with the alternate sequence of activities. A new `make` method is invoked for each instance of a job object. Since both `make` and `store` are `TELL` methods (i.e., capable of elapsing simulation time), the details of the interaction of the jobs is implicitly handled by the process-oriented simulation mechanism of MODSIM II. Instances of message objects are created by `make` and `store` and passed as a parameter to its `inform` method.

For emulation, an instance of `virtualShopConObj` replaces `simShopConObj` as the shop controller object in the model. No decision making methods, i.e., `processOrder`, `make` and `store`, exist in `virtualShopConObj`. The only method it has, in addition to those inherited from `emISObj`, is `operate`, which is a `TELL` method. The implementation of the `operate` method is very simple: it calls the `readIP` method to read in any commands from the shop controller, creates a message object to hold the contents of the commands, and invokes its own `inform` method to send the message to the appropriate controller object in the emulation. The method then `WAIT`s 15 seconds before re-checking for commands.

4.4.4 Classes for cell controllers

The classes encapsulating the behavior of the three cell controllers are derived from the `cellConObj` class and `simISObj` class. The `cellConObj` class is an abstract class, and although it presently does not provide any common fields or methods to its subclasses, it may be used for this purpose in the future. The `simISObj` class provides methods for representing the communication with other controller objects in the simulation, and is described in detail later. The three subclasses--`assCellConObj`, `macCellConObj`, `matCellConObj`--are all implemented in a similar manner (Table 4). Each subclass has one or more objects of type `RandomObj` to generate random samples from statistical distributions for activities in the cell. In addition, there are fields of type `ResourceObj` to model the resources, e.g., machines, which comprise the cell. Finally, each cell controller subclass has a set of decision/action methods for governing the resources. In all subclasses, the `ObjInit` method creates new instances of the resource objects and `ASK`s them to `Create` the appropriate number of resources.

Declaring the resources as a field of the cell controller objects tends to blur the demarcation between the cell controller and the physical cell. Such a design, while being adequate for the purpose of this demonstration prototype, is not consistent with the modeling guidelines. A better implementation, with a clearer separation between the controller and the cell it controls, can be accomplished by creating a separate cell object which owns the resource objects, e.g., a `ListObj` holding `ResourceObj`s. A reference to this cell object may then be captured in a field of the cell controller object. The advantage of such an implementation is the flexibility to change the logic (controller object) implementation without affecting the physical (cell object) implementation, and vice-versa.

As with all other controller objects, the decision making/action methods are `PRIVATE` to assist in the controller-interaction modeling perspective. Clients of the cell controller (only the shop controller, in this case) may interact with it (i.e., issue a command to perform an action) only by means of the methods of the `simISObj` which, in turn, actually invoke the decision/action methods. As a result of this arrangement, the

Table 4: Definitions for cell-level controller object classes

```
TYPE
  assCellConObj = OBJECT(cellConObj,simISObj)
    stream5,              {kiting time}
    stream6 : RandomObj; {assembly time}
    assRobot : ResourceObj;
    PRIVATE
    TELL METHOD assemble(IN productType:INTEGER);
    TELL METHOD buildKit(IN kitType:INTEGER);
    TELL METHOD loadFeeder(IN partType:INTEGER);
    OVERRIDE
    ASK METHOD ObjInit;
    TELL METHOD receive(IN msg:messageObj);
  END OBJECT; {assCellConObj}



TYPE
  macCellConObj = OBJECT(cellConObj,simISObj)
    stream7 : RandomObj; {machining time}
    macRobot : ResourceObj;
    PRIVATE
    TELL METHOD machine(IN productType:INTEGER);
    OVERRIDE
    ASK METHOD ObjInit;
    TELL METHOD receive(IN msg:messageObj);
END OBJECT; {macCellConObj}



TYPE
  mhcCellConObj = OBJECT(cellConObj,simISObj)
    asrs   : ResourceObj;
    stOutBuf  : ResourceObj;
    cncPallet : ResourceObj;
    robotPallet : ResourceObj;
    linkPallet : ResourceObj;
    PRIVATE
    TELL METHOD loadPallet(IN palletType,jobId:INTEGER);
    TELL METHOD unloadPallet(IN palletType,jobId:INTEGER);
    TELL METHOD sendPallet(IN from, to:INTEGER);
    OVERRIDE
    ASK METHOD ObjInit;
    TELL METHOD receive(IN msg:messageObj);
  END OBJECT; {mhcCellConObj}
```

receive method, inherited from simISObj, is uniquely implemented inside the three cell controller classes.

4.4.5 Class interfaceObj

According to the modeling framework described in Chapter Three, all interaction between controller objects should be done through the methods of the interface object. In this implementation, interfaceObj, and its subclasses, provide controller objects the ability to communicate with each other. This class has no instances; it belongs to a style of classes referred to as *mixins*[6].

The base class--interfaceObj--has two TELL methods: inform and receive, both of which take a message object as a parameter (Table 5). For each message to be transmitted, a decision making method of a controller creates a new instance of messageObj, and then invokes the inform method. Since TELL methods are asynchronous in simulation time, all messages are implicitly queued up and delivered in the FIFO sequence. A default implementation of the inform method is provided at the base class level and is retained by the subclasses. The inform method invokes the receive method of the controller object identified in the receiver field of the message, with the message object as the parameter. It is the receive method's responsibility to interpret the incoming message and invoke the appropriate logic method of the controller object, passing it the necessary parameters from the message. Since its implementation is specific to each controller object, no implementation is provided for the receive method at the base class level (in C++ terminology, it is pure virtual.)

The inform and receive methods are designed to be very versatile and can be used to model many aspects of communication with great fidelity. By default, the sender's inform method calls the receiver's receive method with a WAIT FOR statement. With this arrangement it is possible to simulate message transmission problems by INTERRUPTing the receive method, and consequently the inform method, according to a

[6] A mixin is defined as a class that embodies a single, focused behavior and is used to augment the behavior of some other class via inheritance [Booch, 1994].

Table 5: Definition of interface object classes

```
TYPE
interfaceObj = OBJECT
  TELL METHOD inform(IN msg:messageObj);
  TELL METHOD receive(IN msg:messageObj);
  ASK METHOD ObjInit;
  END OBJECT;



TYPE
  simISObj = OBJECT(interfaceObj)
  END OBJECT;



TYPE
  emISObj = OBJECT(interfaceObj)
    command : STRING;
    param1,
    param2,
    param3 : INTEGER;
    param4 : REAL;
    inFNames,
    outFNames : ARRAY INTEGER OF STRING;
    inFile,
    outFile : StreamObj;
    PRIVATE
    ASK METHOD readIP(IN inFNames:STRING;
                      IN mark:INTEGER):INTEGER;
    ASK METHOD writeOP(IN outFNames, com:STRING;
                       IN p1, p2, p3:INTEGER);
    ASK METHOD mapIP;
    ASK METHOD mapOP;
    OVERRIDE
    ASK METHOD ObjInit;
  END OBJECT;
```

random variable in the receive method. In case of a "successful" transmission, the receive method calls an action method with a WAIT FOR statement. This allows an appropriate response message, e.g., "machining done" or "machine breakdown," to be returned depending on whether the action method is INTERRUPTed. Delays for message transmission can be easily modeled by including a WAIT statement before invoking the receive method.

Also, its present default behavior does not make assumptions about synchronous or asynchronous communication and thus does not constrain the modeler. By contrast, the OOSIM controller model assumes that all communication is asynchronous [Narayanan et al., to appear]. In real-time control, especially, there are circumstances which mandate synchronous communication. The control logic blocks in the client until the server either acknowledges receipt of the message (i.e., successful transmission) or responds with a status message. The desired modeling flexibility is provided by making inform and receive TELL methods, but invoking the server's (i.e., receiving controller) receive method using a WAIT FOR statement for synchronous communication, or a TELL statement for asynchronous communication. If the inform method is invoked with a WAIT FOR statement, e.g., WAIT FOR SELF TO inform(msg), the logic method blocks until inform returns. If a TELL statement is used, e.g., TELL SELF TO inform(msg), the message is scheduled for transmission and the next statement is executed.

The methods of emISObj impart the ability to communicate with a physical object outside of the (emulation) model. The methods readIP and writeOP encapsulate the code which represents the actual communication protocol for the physical object. In addition, mapIP and mapOP provide translation between actual communication messages and messages to/from objects in the emulation. All four methods are PRIVATE and are invoked by the inform and receive methods (inherited from interfaceObj). Although all interaction with the emulation model of the FMAS is done through the virtualShopConObj, it is possible for other objects to interact with physical entities by inheriting from emISObj (or its subclass) and providing the details of the message translation.

4.4.6 Class messageObj

Objects of this class are used to convey information between controllers. The class includes fields to hold the information contained in the message and methods to set the values of the fields (Table 6). The `ObjInit` method provides each new message instance with a unique message identifier (field `msgId`) and a time-stamp indicating the (simulation) time of creation. The server (receiving controller) can use the `sender` field to return an acknowledgment or response to the client controller. Note that the `receiver` and `sender` fields are of the `interfaceObj` type instead of `controllerObj`. As a result, the server is restricted to interacting with the client by the methods of `interfaceObj` (i.e., `inform` and `receive`) since the decision/action methods of the client are not visible to it. This is consistent with, and enforces, the concept of separation of *communication* and *logic* components as described in the conceptual model of a controller in Section 3.3.2.2.

As presently implemented, this class is little more than simple records of data. However, its implementation as a class, rather than a structure, allows for future development of specialized categories of messages. A message belonging to a category, such as action or status, could have specific behavior built into it. This extension is possible without modifying the definition of the interface class and its subclasses.

4.4.7 Class jobObj and orderObj

The objects belonging to both of these classes are passive simulation entities in accordance with the previously described modeling guidelines for jobs and orders. Consequently, these objects do not invoke methods of other objects, such as machines. Instead, they are implemented as information objects which record static data, e.g., ID number, number of parts in an order, and dynamic data, e.g., status, priority (Table 7). The fields containing the dynamic data are modified by the shop controller object, which is an active entity (along with other controller objects), as events occur on the factory floor.

The ID number for `orderObj`s is set by itself in the `setFields` method using a counter which is a class variable (i.e., only one copy of the variable exists, regardless of

Table 6: Definition of message object class

```
TYPE
  messageObj = OBJECT
    msgId : INTEGER;
    msgName : STRING;
    sender :   interfaceObj;    {NOT controllerObj!}
    receiver : interfaceObj;    {NOT controllerObj!}
    jobId : INTEGER;
    jobType : INTEGER;
    priority : INTEGER;
    numCNC : INTEGER;
    numRobot : INTEGER;
    moveFrom : INTEGER;
    moveTo : INTEGER;
    status : INTEGER;
    timeIn : REAL;
    timeOut : REAL;
    ASK METHOD setname(IN newval:STRING);
    ASK METHOD setrcver(IN newval:interfaceObj);
    ASK METHOD setsnder(IN newval:interfaceObj);
    ASK METHOD setjtype(IN newval:INTEGER);
    ASK METHOD setjid(IN newval:INTEGER);
    ASK METHOD setpri(IN newval:INTEGER);
    ASK METHOD setnumC(IN newval:INTEGER);
    ASK METHOD setnumR(IN newval:INTEGER);
    ASK METHOD setstatus(IN newval:INTEGER);
    ASK METHOD ObjInit;
  END OBJECT; {messageObj}
```

Table 7: Definition of job and order object classes

```
TYPE
  jobObj = OBJECT
    jobId      : INTEGER;
    type       : INTEGER;
    priority   : INTEGER;
    status     : INTEGER;
    parent     : INTEGER;
    ASK METHOD setFields(IN id, typ, pri, par:INTEGER);
    ASK METHOD updateStatus(IN newStatus:INTEGER);
  END OBJECT; {jobObj}



TYPE
  orderObj = OBJECT
    id : INTEGER;
    priority : INTEGER;
    numCNC : INTEGER;
    numRobot  : INTEGER;
    doneCNC : INTEGER;
    doneRobot : INTEGER;
    status : INTEGER;
    timeIn : REAL;
    timeOut : REAL;
    ASK METHOD updateStatus(IN newStatus:INTEGER;
                            IN job:jobObj) :INTEGER;
    ASK METHOD giveStatus : INTEGER;
    ASK METHOD setFields(IN pri, numC, numR:INTEGER);
  END OBJECT; {orderObj}
```

number of object instances.) In contrast, jobObjs are assigned an ID number by an external object (the shop controller object) which is based on the ID of the order to which they belong and the sequence in which they are introduced into the shop.

Note that although the arrival of orders is generated by the factory-level controller, orderObjs and jobObjs are instanced, maintained, and DISPOSEd by the shop controller object.

4.4.8 Class systemObj

An object of type systemObj corresponds to the system object described in the modeling guidelines, and is used primarily to provide a coupling among the various objects that participate in a particular simulation or emulation model. The systemObj is an aggregation of manufacturing domain objects, simulation infrastructure objects, simulation support objects. The definitions of all these objects are IMPORTED into the definition module of the system object (Table 8). Descendants of systemObj inherit the definition of the method create, but provide the implementation themselves. In this method new instances are created, and messages are sent to objects asking them to perform some action now, e.g., initialize themselves, or to schedule activities for some future simulation time, e.g., print report. A particular scenario may be set up by scheduling specific events in advance. The system object and the report object, described later, isolate the experiment from the model.

The create method creates a new instance of type supervisorObj and shopCon--an object that plays the role of shop controller--and establishes a master-slave relationship between the two by invoking their respective methods. In simSysObj the shopCon variable is defined to be of type simShopConObj so that an object with decision making capabilities drives the simulation model. For emulation, the shopCon variable is defined in emSysObj to be of type virtualShopConObj. Note that in order to have this type flexibility in the child classes, shopCon is not defined as a field in systemObj since in MODSIM a field type definition cannot be overridden in a derived class.

The system object contains a simMgr field--which is a reference variable of type rtSimControlObj--whose job is to provide a simulation-time advance mechanism.

Table 8: Definition modules of system object classes

```
DEFINITION MODULE systemMod;
FROM sscMod     IMPORT simShopConObj; {simulation}
FROM superMod   IMPORT supervisorObj;
FROM timeMod    IMPORT rtSimControlObj;
FROM simrepMod IMPORT simReportObj;

VAR
  system   : simSysObj;
TYPE
  sysObj = OBJECT
    super:supervisorObj;
    TELL METHOD create;
  END OBJECT; {sysObj}

TYPE
  simSysObj = OBJECT(sysObj);
    shopCon : simShopConObj;{for simulation}
    timeMgr : rtSimControlObj;
    finalReport : simReportObj;
    OVERRIDE
    TELL METHOD create;
  END OBJECT; {sysObj}

END MODULE.    {DsystemMod}



DEFINITION MODULE emsysMod;
FROM vscMod     IMPORT virtualShopConObj;{emulation}
FROM superMod   IMPORT supervisorObj;
FROM timeMod    IMPORT rtSimControlObj;
FROM emrepMod   IMPORT emReportObj;
FROM systemMod IMPORT sysObj;

VAR
  system   : emSysObj;
TYPE
  emSysObj = OBJECT(sysObj);
    shopCon : virtualShopConObj;{for emulation}
    timeMgr : rtSimControlObj;
    finalReport : emReportObj;
    OVERRIDE
    ASK METHOD create;
  END OBJECT; {emsysObj}
```

The SetTimeAdvance method is called with a FALSE parameter in simSysObj, for the default next-event time advance method, and with a TRUE parameter in emSysObj to enable the real-time advance mechanism.

4.4.9 Class reportObj

The primary responsibility of an object of type reportObj is statistics reporting. References to objects, variables, etc. on whom statistics have to be reported are imported into the Implementation module. Since statistics collection is the responsibility of individual objects--as per the modeling framework--a report object only sends a message to them to turn on, or off, statistics collection. With this design, statistics collection may be turned off during emulation without making changes to the manufacturing system objects used in the simulation. In MODSIM, resource objects and group objects have statistical collection capability built into them. Statistics on object fields or program variables are collected easily in MODSIM by associating them with statistical monitoring objects provided by the language. It is important to note that variables and fields are part of the manufacturing objects, but monitors are part of the reportObj.

Since the importance of statistics is different, simReportObj for simulation and emReportObj for emulation, have distinct implementations. A virtual TELL method--print--to print a report of the collected statistics is declared by the base class but the implementation is provided in the sub-classes. All statistical objects have a Reset method which can be invoked from resetStats method of simReportObj. The concept of clearing statistics after a warm-up period is not relevant in emulation and thus resetStats method is not provided in emReportObj.

4.4.10 Real-time Simulation Control Object

MODSIM includes a simulation control object class which provides mechanisms for fine-tuning the execution activities within a simulation. This object class is called SimControlObj and is defined in module SimMod. It provides capabilities for time advance notification, event tie-breaking, and tracing simulation activities. There are two ASK methods for each of these capabilities: one for providing a specific implementation

and the other for enabling (and disabling) its invocation, e.g., TimeAdvance and SetTimeAdvance. Ordinarily, the user is not required to instance and use an object from this class, but must do so if any of the default behaviors are required to be overridden.

In this implementation, an emulation control object is developed primarily to provide a real-time advance mechanism for emulation. The rtSimControlObj class (Table 9) derives from SimControlObj, and is defined and implemented in DtimeMod.mod and ItimeMod.mod respectively. In rtSimControlObj the inherited TimeAdvance and ChooseNext methods are overridden and re-implemented. In addition, it includes a TELL method, realWAIT, and two ASK methods, setTimeScale and setStepSize. The method realWAIT is responsible for elapsing temporal time so that the temporal clock stays synchronized with the simulation clock. The setTimeScale method allows the ratio between temporal time and simulation time to be varied by assigning a value to the MODSIM global variable Timescale. The default value of 1.0 indicates that one second of real time will elapse for each unit of simulation time, while a value closer to 0.0 is time-lapse (almost next-event), and greater than 1.0 is slow-motion[7]. This method enables the Timescale to be changed at anytime during a simulation run.

If SetTimeAdvance has been called with the TRUE parameter, the TimeAdvance method is invoked when the simulation time is about to be advanced. Any desired work may be performed from this method including scheduling more activities. The new implementation of TimeAdvance determines the scheduling of the TELL method realWAIT as seen in the flow diagram in Figure 14. Basically, the method breaks the interval between the current simulation time and the time of the next scheduled activity into smaller intervals based on the value of the field stepSize. A realWAIT method is scheduled for execution either one stepSize time units later or at the time of the next scheduled activity, whichever is earlier. In the latter case, a tie exists between realWAIT and another activity. The new implementation of

[7] The assumption is that delays in the simulation model are specified in seconds. If minutes are used as time units, then real-time advance is obtained by assigning a value of 60.0 to the variable Timescale.

Table 9: Definition module of real-time advance classes

```
DEFINITION MODULE timeMod;
FROM SimMod IMPORT SimControlObj, ActivityGroup;

  VAR
    simStartSecs : REAL;{val of ClockRealSecs when sim started}
    startFT,
    endFT,
    totalFT : REAL; {FT => Fast Time, mixed mode time adv}
    idRealWAIT : ACTID;
    trace : BOOLEAN; {flag to control printing of trace
                      thru methods}

  TYPE
    rtSimControlObj = OBJECT(SimControlObj)
      TELL METHOD realWAIT;{pass time on temporal clock}
      ASK METHOD setTimeScale(IN scale:REAL);
      ASK METHOD setTrace(IN flag : BOOLEAN);
      ASK METHOD elapsedTime() : REAL; {real seconds elapsed
                                        since sim start}
      OVERRIDE
      ASK METHOD ChooseNext(IN group:ActivityGroup) : ACTID;
      ASK METHOD TimeAdvance(IN newTime:REAL) : REAL;
END OBJECT;  {rtSimControlObj}



    frtSimControlObj = OBJECT(rtSimControlObj)
      OVERRIDE
      ASK METHOD SetTimeAdvance(IN flag:BOOLEAN); {calculates
                                                   total FT}
      ASK METHOD elapsedTime() : REAL; {adds Fast Time}
    END OBJECT;  {frtSimControlObj}

END {DEFINITION} MODULE. {timeMod}
```

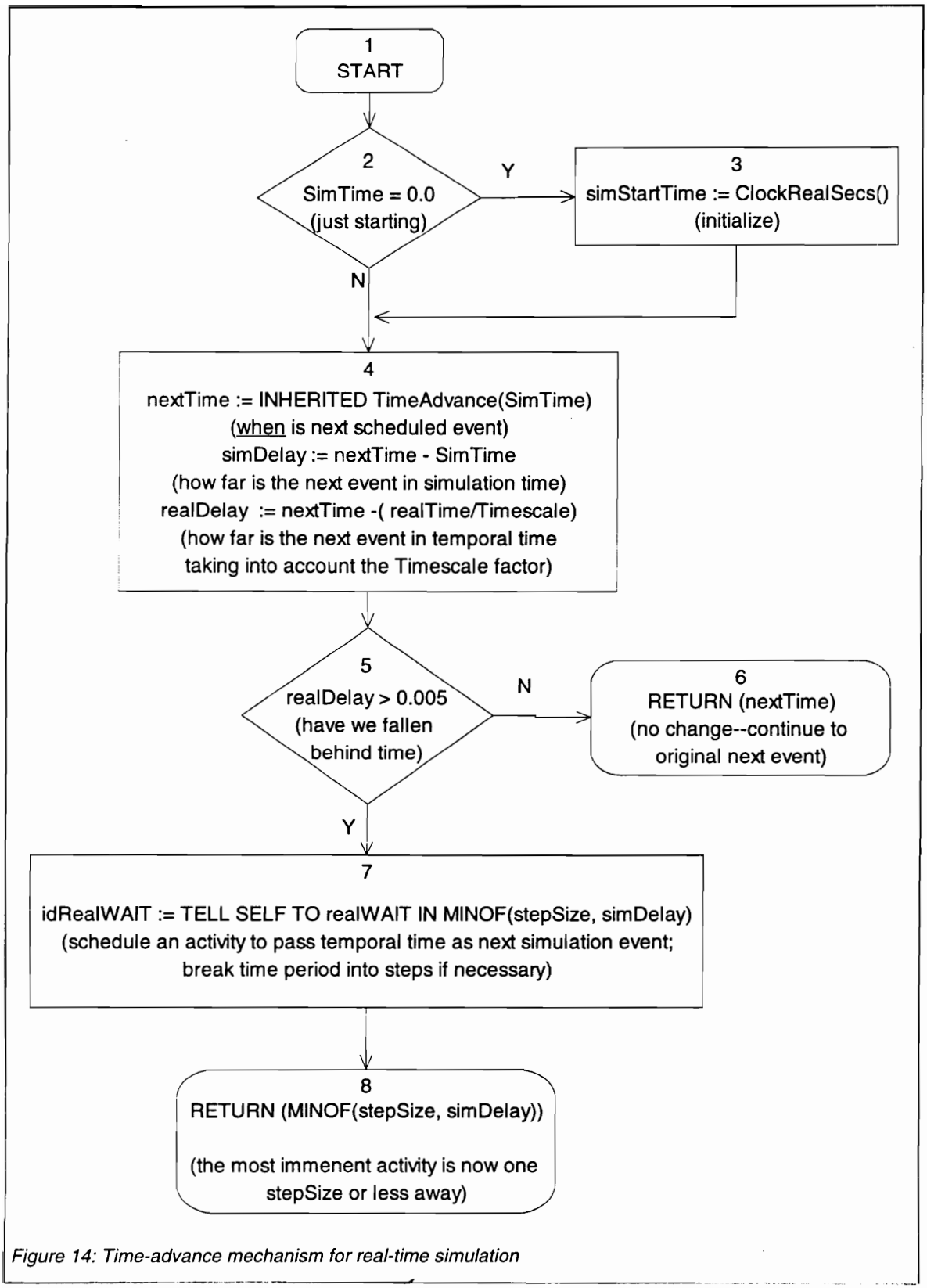


Figure 14: Time-advance mechanism for real-time simulation

`ChooseNext` supplied to the `rtSimControlObj` ensures that `realWAIT` is always activated first.

4.5 Simulation Model Operation

This section describes the use of the model in answering the particular question: "For the given system, what is the appropriate maximum WIP level in order to reduce mean job time-in-system?" Thus, the objective of the simulation was to examine the effectiveness of alternate production control policies, while the objective of the exercise was to show that the simulation was indeed a valid one. The emphasis being on demonstrating its appropriateness for the purpose, i.e., control strategy testing, rather than on the fidelity of representing the real system. Later, much of the simulation model code would be reused in the emulation model.

The classes described in the previous section were used to create a simulation model of the FMAS. Table 10 shows the values of various time and other parameters used in the model.

Verification and validation: The model was first run as a deterministic simulation, i.e., arrival rates, order sizes, processing times,. etc. were constant and not samples from a probability distribution. As a result it was easy to predict and verify job completion. Initially, the system was subject to handing just one order comprised of only one job. Later, the number of jobs was increased, leading to the normal situation of multiple orders simultaneously in the system. In all cases, activities in the system were traced by use of print statements at key places in the code. In addition, copious use of the MODSIM run-time debug facility was made to step through each line of code and observe the changes in program variables. It must be noted that the modularity afforded by MODSIM, as well as the object-oriented programming paradigm, substantially simplified the verification and validation process by permitting model complexity to be partitioned and increased incrementally.

Experimentation: Two models, differing only in the maximum WIP level allowed, were each run twice for a total of four runs. A different starting seed value was used for generating order arrivals for each run, but the same seeds were used for both models to

Table 10: Parameters for FMAS simulation

Order arrival rate	Exponential(15.0)
Jobs per order: type 1	UniformInt(0,3)
Jobs per order: type 2	UniformInt(1,3)
Jobs per order: total	Min: 1; Max: 6
Kit-building time	Triangular(0.75, 1.0, 1.25)
Machining time	Triangular(3.75, 4.0, 4.25)
Assembly time	Triangular(1.75, 2.0, 2.25)
Travel time between any two locations	1.0
Pallet loading/unloading time	0.5
Expected job flowtime Direct route: With intermediate storage:	 12 15

reduce variance [Law & Kelton, 1991]. Statistics were cleared after 20 minutes of shop operation in order to reduce start-up bias. The simulation was then run for an additional 240 minutes. Statistics were collected on mean job and order time-in-system, frequency and length of blocking of the machining workcell, and queue characteristics at the ASRS. These measures are representative of those that would be used to evaluate the performance of a control strategy. Also, they go to show how various types of statistics--on resources, groups, object fields, program variables--are collected.

Analysis of output: A visual analysis of the resultant data in Table 11 suggests that Model 2 produced better results than Model 1 for all the performance measures considered.[8] This conclusion should be substantiated by statistical tests of significance. Of course, the size of the data set is far too small to make a conclusive decision, but that is not the objective of this research. The exercise has satisfied its objective by demonstrating that the capabilities of the developed model are at par with those normally expected from simulation models used in studies of control strategy.

4.6 Emulation Model Development

The emulation model was crafted from most of the object classes used in the simulation model. As shown in Table 12, the classes specific to the emulation were `rtSimControlObj`, `emSysObj`, `emRepObj`, `virtualShopConObj` and `emISObj`. The real-time advance capability provided by `rtSimControlObj` had been independently verified earlier. `emSysObj` and `emRepObj` were conceptually similar to their simulation model counterparts, and modifications needed to reflect the different nature of emulation experiments was a very straightforward task. The behavioral model of the FMAS shop-floor, contained in the cell-level objects, was reused without modification from the simulation. `virtualShopConObj` represents the only manufacturing domain object that required coding for the emulation. Consequently, the verification and validation effort focused on only the methods of this object and its parent object class `emISObj`. In particular, the mapping between the actual controller communication and messages to/from controller objects in the emulation was verified.

---

[8] As a matter of fact, the results are consistent with the theory of CONWIP and of pull systems which suggest that lower WIP levels contribute to reduced waiting times, and hence flowtimes.

Table 11: Results from FMAS simulation runs

	Model 1: WIP = 5			Model 2: WIP = 4		
Performance Measures	**Run 1**	**Run 2**	**Mean of two runs**	**Run 1**	**Run 2**	**Mean of two runs**
Job time in system: mean	39.38	26.48	32.93	39.00	25.09	32.05
Job time in system: SD	17.48	12.34	14.91	14.35	10.69	12.52
Blocking of MWC: count	17	12	14.50	16	9	12.50
Blocking of MWC: mean duration	1.74	1.62	1.68	2.02	1.52	1.77
Number of jobs in storage: time-weighted average	1.52	0.61	1.07	1.00	0.4	0.70

Table 12: Components of simulation and emulation models

	Simulation:	***Emulation:***	
	Basic	***First-cut***	***Final***
Purpose	Control strategy testing	Validation of emulation	Control software testing
Cell controllers	assCellConObj macCellConObj matCellConObj	assCellConObj macCellConObj matCellConObj	assCellConObj macCellConObj matCellConObj
Report to:	simShopConObj	virtualShopConObj	virtualShopConObj
Shop control logic contained in:	simShopConObj	protoscon.mod (uses simShopConObj as kernel to mimic control software)	shopcon.mod (actual control software)
Model time manager	SimControlObj	rtSimControlObj	rtSimControlObj
Experiment manager (statistics collection)	simRepObj	emRepObj	emRepObj
System object	simSysObj	emSysObj	emSysObj

The difficult task of verification and validation of the emulation as a whole was aided by creating a rapid prototype of the shop control software. The prototype was created by using `simShopConObj` as the kernel. This was possible since the complete shop control logic in the simulation model was encapsulated in `simShopConObj`, and because the exchange of communications messages were explicitly modeled in the controller objects. Some changes were needed in the `inform` and `receive` methods to reproduce the syntax of the actual shop controller. The program was written in MODSIM and run as a real-time simulation. The emulation was also run in real-time mode and, for the first time, as an independent program with all shop control logic external to it. The rapidly prototyped shop controller served as an event generator for the emulation (in a reversal of roles) making it possible to test the real-time behavior of the emulation as well as the message translation capabilities of the virtual shop controller object.

4.7 Shop Control Software

A shop control software was developed and its dynamic operation was tested against the emulation model described in the previous section. This shop control software is a prototype implementation and is representative of the one that will be developed for controlling the actual FMAS. From the perspective of this research, its main purpose is to demonstrate the validity and usefulness of the developed emulation program. The prototype shop controller software was coded in MODSIM II. (The final implementation is expected to be in the C language.)[9] In this case, MODSIM is used as a general purpose language and none of the simulation constructs are used. Hence, neither the quality or validity of the prototype implementation are compromised.

It is assumed that the development of the shop control software has reached a point where the control logic is the focus of the testing. The graphical user interface, for interaction with the human supervisor, which is also an important part of the final shop control software, is assumed to either have not been developed or is being tested

[9] Booch [1994 pg. 251] indicates that this practice is common in the software industry: "It is not unusual to see proofs of concept developed in one language (such as Smalltalk, for example) and the product development to proceed in another (such as C++)."

separately. In any case, the result is that the control software being tested has been slightly modified to aid in the testing of the control logic. This is consistent with the recommended practice of modular software development and verification.

The shop controller is event driven. Events include commands from the human supervisor or status responses from the cell-level controllers. These are referred to as control events. The other type of events are time events, in which the passage of a certain amount of time or a specific time value triggers a response from the controller. The only time events currently coded into the control software is the communications function of examining the output mailboxes (files) of cell controllers every 15 seconds.

The central routine of the shop control software is `operate` which is responsible for periodically checking for and interpreting messages from other controllers. This routine may call other routines such as `processOrder`, `removeJob`, and `releaseJob` to assist in responding to the messages. Note that due to the nature of the logic in `processOrder`, it can be tested without an emulation (and, in fact, separately from the shop control program as a whole.) The same is not true for the `releaseJob` routine. Its behavior is dependent on, and affects, other variables, data structures, and the time dependent behavior of the program as a whole. The testing and debugging of this routine is described in the following section.

Since the final shop control software will be written in C which does not have built-in support for concurrency, this feature of MODSIM is not exploited. As a result the design of the shop control software requires a mechanism for handling simultaneous messages from multiple sources. All incoming messages are captured in a new instance of `messageObj` which are added to a message queue. These messages are then removed from the queue and acted upon one at a time. The action may be to issue a new command in response or just update the data. In either case the `messageObj` is `DISPOSE`d. If some other conditions have to be satisfied before a new command can be issued, e.g., resource at the next destination has to be available, the message is removed from the main queue and added to one of the other queues. There are five such queues based on the present location of the job and its destination:

stAssQ, stMacQ, acStrQ, acMacQ, and mcAssQ[10]. There may be more than one queue associated with a particular destination, in which case they are polled in a predetermined priority order.

The program keeps track of WIP level, and the availability of the buffers (in its control domain) so that this information is readily available for use in decision making. The shop controller does not maintain a count of the number of pallets available[11]. This information is obtained from the emulated MHC when required (releasing jobs to the shop floor) via a synchronous communication; the normal flow of the program is interrupted until the appropriate response is received.

4.8 Testing the Shop Control Software

The prototype shop control software was built through an iterative process of development and testing. As the code for new control capabilities was added, it was tested using the emulation. The first version of the software incorporated only the most basic control logic for processing a single job through the system. The routing of the job was predetermined and not based on system status. The primary objective was to test the communication at the application level, i.e., interpreting and reacting to status messages, and issuing command messages.

With the testing of basic communications capabilities out of the way, increasingly complex decision making capabilities were added to the software. Sometimes the emulation was run repeatedly under a specified scenario, e.g., forcing the MWC to be "busy," thus causing all jobs to be sent to storage after assembly. A few scenarios were pre-planned but most were designed as specific bugs in the control software were identified. Some scenarios required changes, often temporary, to existing objects. These changes were mostly in the form of variables to track a specific value, or a method to modify a object field from outside the object (usually the system or report

[10] The QueueObjs actually hold the messages, but in effect the job specified in the message is waiting for service.

[11] In a practical sense it means that a pallet is not freed immediately after a completed job is returned to the ASRS. The operator signals the MHC that the part has been removed and the pallet is free for use.

object). Due to the modular nature of the emulation software, i.e., objects, most scenarios could be coded by changes to only a couple of modules.

Since all entities interacting with the shop controller, including the human supervisor, were represented in the emulation the results were reproducible over multiple runs. Much of the testing involved relatively short emulation runs. However, several runs were needed in order to detect a problem, pinpoint the source of the problem, or establish that the problem had been fixed. As a result, the wall-clock time requirements quickly added up to the point of impeding the testing process. Delay-scaling, i.e., reducing all delays in the model by a common factor, was used to mitigate the problem by reducing time requirements. During initial testing, the emphasis was on determining that the shop-control software was reacting correctly to system events and thus screen out major, obvious bugs in the control software. Consequently, a limited amount of deviation from normal (i.e., real-time) emulation results that was caused sometimes by delay-scaling was considered acceptable. (The issue of accelerated testing using emulation is discussed in greater detail in the following chapter.) Usually, a couple of regular (un-scaled) runs were performed after several scaled runs to confirm the observations.

The following examples describe specific instances of errors that were detected in the shop control software by use of the emulation model. Many of the errors could not have been detected without a dynamic (time-related) test environment, and others were more easily identified because of it. Indicators of possible errors include crashes or run-time error messages, extended periods of inactivity in the emulation, non-termination of finite length runs, and analysis of communication logs.

Synchronous communication: This capability of the shop control software worked without problems during the initial, shorter runs. In a longer run, all shop floor activity in the emulation model ceased after a while; the system had reached a state of deadlock. An examination of the communication log of the emulation and of the shop control software showed that all messages sent by one were being correctly received by the other. Also, since the responses written to the output files were consistent with the

commands, the problem did not lie either in the emulated system or the communications.

Closer examination revealed that although messages were being received correctly by the SC, they were not all being acted upon. The synchronous communication logic had ignored the fact that the communication file was deleted after being read. The messages that arrived while the program awaited a response to the status request were being lost. As a result, pallets were not being moved to the next destination. This problem was rectified by adding messages (other than `palletsFree`) received in the interim, to `inMsgList`.

Release Job: When the shop control software program crashed, the final message in the communication log was for the completion of job 102--a definite indicator of a problem since a similar message had been logged earlier. Job 102 had been re-released along with jobs from order 2. The source of the problem was `jobList`--an object of type `ListObj`--which held a collection of currently active jobs. All jobs from this list, instead of only those that were not already released, were being released since there was no way to distinguish one type from the other. It is important to note that the error resulted from the incorrect use of a bug-free object. This experience also showed that the implementation of the control logic could be made more robust by explicitly setting the status field of a job each time a command/decision was made that affected its state.

Load pallet: Part of the way into a run, the emulation model gave a run-time error "attempt to return more resources than allocated." The debug/traceback capability of MODSIM was used to identify that the error involved the resource `stOUTbuf`--the output buffer at the ASRS--and job number 201. The trace capability was used to step through the program statements from the time order number 2 arrived. When jobs 201-204 were released, the assembly cell was busy and so they were put into the `acStrQ`--a queue of jobs at the ASRS waiting to move to the assembly cell. The command to load the pallet onto the conveyor was inadvertently left out when jobs were retrieved from the `acStrQ`; jobs from the main message list were being handled correctly however. The

error came to light when the pallet for job 201 moved to the assembly cell and attempted to release the output buffer which it had never acquired.

4.9 Assertion of Proof-of-Concept

The implementation exercise described in this chapter serves to demonstrate the practicability of the developed emulation development method, and in that sense validates the conceptual foundations. The simulation and emulation models of the FMAS are a practical application of the conceptual modeling framework and guidelines (described in Chapter Three) to a specific manufacturing system. The implementation, in MODSIM II language, showed that is possible to develop an emulation model from a simulation model instead of the common practice of building the emulation from ground up.

Both developed models were valid in that they were appropriate for their intended use, i.e., they were capable of providing the services normally expected from such types of models. With the simulation model it was possible to gather data on a variety of measures of performance typically needed to evaluate the quality of a particular shop control strategy. Table 11 lists the data collected on job flowtime, occurrence of blocking, and queuing characteristics of the storage system. The emulation model was a very useful resource in the development and dynamic verification testing of the prototype shop controller software as described in Section 4.8. Its use made it possible to identify many errors in the software, including those involving temporal logic and communications logic, during the development stage instead of the installation stage.

The implementation also demonstrated that complex shop control logic can be packaged into a controller object, isolated from the physical definition of the system, and later, removed from the simulation for the purpose of creating an emulation model. In particular, a CONWIP-like shop-floor production control strategy was incorporated in the simulation model (and later implemented in the shop control software). Thus, the control logic of the modeled FMAS is not limited to simple queue priorities but is actually representative of that which may be expected in real shop control situations. Subsequent removal of the control logic from the emulation was accomplished with only

a few program statements. This demonstrates that the developed approach of building an emulation from a simulation is indeed suited for application at the shop control level (and in general, higher levels of control).

The exercise clearly showed that an emulation can be built by reusing objects from an appropriately structured simulation model. Code was taken en masse from the simulation for crafting the emulation; other than `rtSimControlObj`, `emISObj` and `virtualShopConObj`, no additional object classes were mandated. These limited changes greatly increased the confidence in the functional reliability of the resulting emulation. It was also shown that enhancements to any of the manufacturing domain objects, e.g., cells, jobs, and to simulation infrastructure objects, e.g., time manager object, can be done for the emulation model under the OO framework.

Although it is difficult to quantify the amount of code reuse achieved or development time saved, it is evident that these objectives were accomplished. The only major constraint imposed was of using the controller-interaction perspective (instead of the job flow). It must be acknowledged that this requires a greater amount of understanding of the control architecture and its implementation, and that this knowledge may not ordinarily be available to the simulation model developers.

4.10 Summary

This chapter described the use of the object-oriented modeling framework, described in Section 3.3, in implementing of a set of object classes for modeling the FMAS using the MODSIM II language. These classes were used to create a simulation model for testing (and fine tuning) a shop control strategy. Later, an emulation model was developed from new classes which were derived from those used for the simulation model. The code for the control logic in a prototype shop controller was tested and debugged using this emulation model.

Thus, it has been shown that a basic emulation can be developed starting from an appropriately structured simulation. Other than the shop controller, all elements may be reused in emulation mode with little or no modification. If these elements are used as is, they represent verified and validated pieces of code. Consequently, a lot of effort

is saved by not having to repeat the verification and validation process for the emulation model.

In its basic form, the emulation primarily provides an off-line testing environment, but only limited assistance in actually highlighting or detecting errors in the shop control software. The following chapter takes a closer look at the issues affecting the testing and debugging process.

5. EMULATION APPLICATION

The basic emulation developed in the first phase is not wholly adequate for use in testing the shop control software. The onus of discovering errors is still upon the engineer. Observing the simulation for extended time periods and tracing every path through the software, in the hope of detecting errors, is clearly impractical. Thus, the emulator should be enhanced with functions and tools to aid in the testing process. To this end, an emulation development and testing environment is envisioned. This environment would provide specific tools and functions to assist in testing and debugging similar to those found in integrated simulation environments. These include capabilities of capturing system "snapshots," tracing the sequence of events, animating system operation, dynamic displays of statistical data, etc. These general features will be useful even in an emulation environment. In addition, some specific features to aid control software testing include the following: the ability to initialize the model to a given state; generating random as well as specific fault situations; temporarily pausing the model without interfering with the operation; logging communications sequences (messages) to a file for subsequent analysis; faster-than-real-time emulation to enable accelerated testing; etc.

The second part of this chapter discusses issues that arise after an emulation model is built and is to be used for debugging the control software. But first, an important assumption made during emulation testing is discussed. The assumption is that any deviation from expected behavior of the control software can be attributed to errors in the software. For this assumption to be valid, the emulation needs to be verified and validated, and this is not a straightforward task.

5.1 Validation of an Emulation

An important part of the simulation model development process is the verification and validation of the model. These activities address the issue of whether the model and its results are "correct." Model verification refers to "ensuring that the computer

program of the computerized[12] model and its implementation are correct" [Sargent, 1994]. Model validation is defined to mean "substantiation that a computerized model, within its domain of applicability possess a satisfactory range of accuracy consistent with the intended application of the model" [Sargent, 1994]. Stated simply, the verification task consists of determining that the computer program executes as expected by the modeler, while validation is the process of determining that the simulation model is a useful or reasonable representation of the system.

Much research has been done on validation techniques for conventional simulation models. Sargent [1994] states that a combination of the following are commonly used: animation (operational graphics), comparison to other models, degenerate tests, event validity, extreme condition tests, face validity, fixed values, historical data validation, historical methods, internal validity, multistage validation, parameter variability, predictive validation, traces, and Turing tests. Some of these may be applied directly to emulation models, while others need to be interpreted in the context of emulation and modified prior to their application.

5.1.1 Emulation Validation: 3 Aspects

Three key aspects of an emulation model are: control, communication, computation. Emulation validation efforts should focus on ensuring that the model has no more capabilities, with respect to each of these aspects, than the physical system it replaces. This is fundamental to the definition of a valid emulation model.

In particular, all control/intelligence associated with the controller being tested (shop-level controller in this case) should be absent in the emulation. For example, if the shop controller is responsible for directing jobs to their next destination, no object in the emulation model should initiate this move under its own accord. A completed job must continue to indefinitely wait at its present location until a move command is received and acted upon by the material handling system.

[12] This is distinct from the *conceptual* model which is the "mathematical/logical/verbal representation of the problem entity for a particular study" [Sargent, 1994].

Another similar basic issue is that emulated (cell controller) objects should not: 1. provide the shop controller any information that the actual controllers are incapable of providing, e.g., the anticipated completion time of a machining activity; and 2. use information not available to the actual controllers, e.g., the status of another cell. The emulation validation task then becomes one of ensuring that all use and exchange of information is legitimate.

In addition to responding *correctly* to commands from the shop controller, the emulated objects should do so in a *timely* manner. An excessively fast or slow response may invalidate the emulation.

5.1.2 The Problem of Dynamic Validation

From a control software developer's perspective it is imperative that the emulation be free of errors (i.e., fully verified and validated) so that any abnormal results during testing can be directly attributed to a flaw in the control software. But, even simulation models cannot always be completely validated [Balci 1995]; the most one can hope for is an increased level of confidence in the model. This is not only true for emulation models, but also a unique dimension is added to the validation task.

Balci [1994] notes that for conventional simulation models, validation, verification, and testing (V,V&T) techniques belonging to the categories informal, static, and dynamic are more commonly and widely used. Informal techniques rely on human evaluation of the model and static techniques are concerned with the assessment of static model source code. Dynamic techniques require model execution and are intended for evaluating the model based on its dynamic behavior. The model output from runs is compared using statistical techniques to system output (if available) or output from other models [Sargent, 1994]. This is referred to as operational validation.

For emulation models, operational (dynamic) validation is complicated by the fact that an emulation does not evolve autonomously, i.e., it is dependent on an external driver, and the driver itself is being tested. Hence, the emulation validation task appears to be confounded by the inability to "run" the emulation, thus limiting the validation to informal and static techniques.

In some cases an alternative "driver" may be available, e.g., if the system under consideration is being upgraded, the old control software might be of limited utility in validating a portion of the emulation. Otherwise, operational validation cannot be performed for want of a suitable driver. Developing a driver for the sole purpose of validating the emulation model appears to be an impractical idea. This sentiment is expressed by Schutz [1993] in the context of real-time testing of a distributed computer system, "The problem of testing the (emulation) software itself remains to be addressed. Clearly, it is not practical to implement yet another simulator which simulates the environment of the (emulator); it would be identical to the application system itself."

Fortunately, the emulation development method presented in this research opens up additional avenues for the operational validation of an emulation model. A method for emulation validation, verification and testing has been included as an integral part of the overall development method. Its emergence is a natural outcome of the object-oriented modeling framework described in Chapter Three.

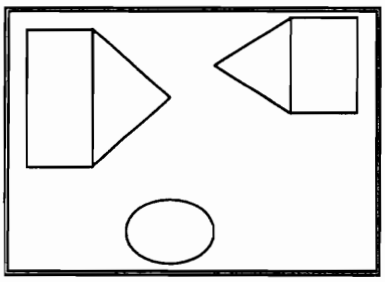
5.1.3 Solution: Hi-fi Simulation and First-cut Emulation

The solution developed in this research makes use of the simulation model to rapidly prototype an emulation driver (i.e., mimic the actual shop control software.) For this two additional models are introduced between the simulation and the emulation model. At first, the idea of building yet another model (in addition to the emulation) may appear to be paradoxical to the stated objective of reducing the emulation development effort. However, the model's usefulness in testing the control software becomes suspect if it is not adequately validated and verified. The additional steps are a meaningful increase in effort and are more affordable due to the savings resulting from using a simulation as the basis for the emulation model.

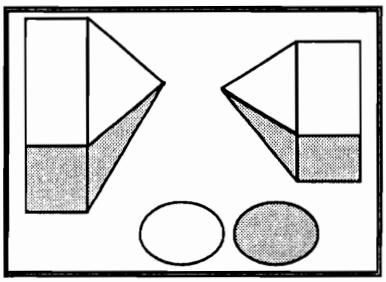
Figure 15 provides a graphical description of the steps in the method for emulation validation. In the figure, a rectangle is used to represent the *logic* module and the triangle the *communications* module of the controller object (with reference to the conceptual model of a controller described in Section 3.3.2.2). Other objects (i.e., besides controllers) in the model are represented by an oval. A clear shape is used to depict previously tested code and a filled shape represents new or untested code.

STEP I

1. SIMULATION

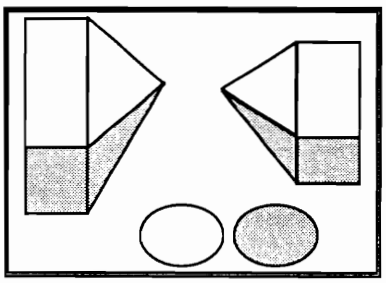


2. HI-FI SIMULATION

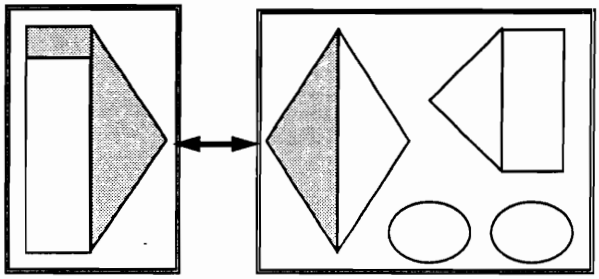


STEP II

2. HI-FI SIMULATION



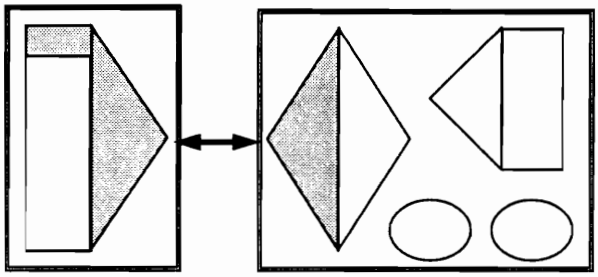
3. FIRST-CUT EMULATION



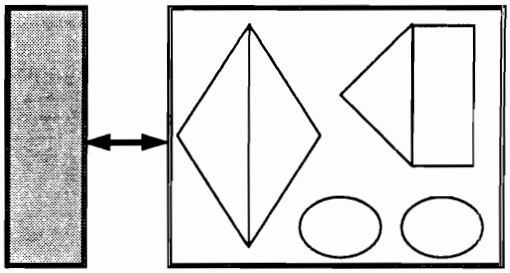
"PROTO"
CONTROLLER

STEP III

3. FIRST-CUT EMULATION



4. FINAL EMULATION



ACTUAL
CONTROL
SOFTWARE

Figure 15: Emulation Validation Method

The process starts with a simulation model used in control strategy testing which has been verified, validated and tested. The first step involves enhancing the model elements to represent the system with the greater degree of fidelity needed for the emulation model but not available in the simulation model for control strategy testing. The objects can be enhanced to be appropriate for use in emulation by incorporating, for example: 1. additional resources/constraints; 2. new behavior, e.g., respond to an "are you operational" message from the controller; and 3. refinements to some behavior, e.g., dis-aggregate a single delay into component activities. Correspondingly, the shop controller object is enhanced to respond to the expanded event set and constraints, such as communication delays, which are unlikely to have been included in the strategy testing model. In other words, the shop controller object is made to more closely mimic the actual shop controller in all respects. The simulation model formulated from these enhanced objects is termed the high fidelity (hi-fi, for short) simulation. With a hi-fi simulation, objects and sub-models to be used in the emulation can be more conveniently verified and validated because the testing is performed in a single, integrated software rather than as a distributed system. The advantage of the hi-fi simulation model is twofold: the first is the assistance provided in the testing of the sub-models used in the emulation; and the second is that statistical information generated from running the hi-fi simulation provides a better yardstick by which to compare the operation of the control software during its testing.

In the second step a distributed system is created and tested. A first-cut emulation model is created by removing the logic module of the shop control object, adding the (controller-to-model) message translation capability, and adding the real-time advance mechanism. The most important aspect of this step is that the shop controller object from the hi-fi simulation is used as a driver to aid in the verification and validation of the emulation model. The procedure involves using the shop controller object as the kernel in a separate program which mimics the shop control software. As a result of the software reuse, a driver for the emulation is created with little programming effort--mostly in providing communications capabilities. The driver program is run as a real-time simulation so that time events occur as in the actual control software. With this setup, substantial V,V&T of the emulation as a independent program can be performed.

Since no changes are made to the manufacturing system objects, the statistical data generated by this system will be very closely comparable to that from the hi-fi simulation. Knowledge of this characteristic allows the testing task to focus on the bi-directional translation between actual control messages and messages in the emulation model. Thus, many basic problems affecting the validity of the emulation can be identified with such a partitioned system.

The final step is to undo any temporary modifications, including "probes", used for verifying the first-cut emulation and incorporate any additional features to interface the emulation model with the actual control software. The emulation model resulting from the previous step has been largely verified, validated and tested. Consequently, the only new code is in the actual shop control software and is more likely to be the source of any problem observed during the testing process.

5.1.4 Simulation versus Emulation Validity

One of the advantages cited of the developed emulation development method is the increased validity and reduced validation effort. It is true that much of the validation of the emulation, especially in regards to the behavior of the system and its elements, is done when the simulation model is validated. For example, the simulation can be examined to ensure if the machining workcell continues to remain unavailable to other pallets as long as the pallet currently at the workstation is unable to move to its next destination. Validation of this behavior does not have to be repeated for the emulation model.

It is possible, albeit not highly probable, that elements of the simulation model (i.e., control strategy testing) have exactly the right level of detail and can thus be used directly in the emulation. In this case, all that is needed is a virtual shop controller object to perform the mapping between actual and simulation messages; validation is limited to the functionality of this object. It is more likely however that some enhancements to the objects are required. But, due to the limited changes to the validated simulation objects, only a few subjective (informal or static) validation tests may be needed before using the emulation in the software testing process.

A word of caution should be made here about interpreting the validity of the simulation and emulation to be the same thing. Law and Kelton [1991] note that "Models are not universally valid, but are designed for specific purposes." The advice can be restated in the context of this research as, "A valid simulation model does not necessarily imply a valid emulation model." Thus, when using a simulation model as the basis of an emulation model, care should be taken to ensure that any undesirable, residual effects of the simulation model do not invalidate the emulation.

Consider the case when communication (and/or computation delays) are added to the objects for use in emulation. The use of these objects in a simulation run of the hi-fi model should logically result in increased job flowtime when compared to the basic simulation model. If the results of the hi-fi simulation confirm the expectations, it is important to avoid falling into the trap of extrapolating the validity to the emulation. A valid hi-fi simulation is a necessary but not sufficient test of the validity of the emulation model, and is illustrated with an example in the following paragraph.

With reference to the FMAS emulation implementation described in the previous chapter, one way to incorporate communication delays is in the `inform` methods of each of the controller objects. A similar result can be achieved by making changes to only the `simShopConObj`. Delays for outgoing messages can be modeled by a `WAIT` in the `inform` method, and the transmission delays for all messages from cell controllers can be simulated by performing a `WAIT` in the `receive` method of `simShopConObj`. While the desired behavior will be achieved in the hi-fi simulation model with both approaches, the emulation model resulting from the first approach, however, will not be valid. This is because the communications delay is performed twice--once in the actual message transmission and again in the `inform` method. With the second approach, the simulated delay is removed along with the `simShopConObj` which is replaced by the `virtualShopConObj`.

5.1.5 Comparison of Results

It has been suggested that a problem arises since the sequence in which control requests are received in the emulation may differ from the actual system and this will affect the control decision [Clark & Withers, 1989]. The implementation experiences in

this research were in agreement with the above statement. However, this is not necessarily a problem and the validity of the emulation should not be immediately suspect if the results of a simulation and emulation are not identical. All stochasticity in a simulation is controlled (due to pseudo-random numbers) and is repeatable. Some of the stochasticity in the emulation system results from the actual controller and communications network, and is not contained inside the emulation model. It is for this reason that there may be some variability between runs even with the same emulation model.

Thus, it is not important that the emulation model produce an exactly identical outcome (in terms of aggregate values and sequence) when compared to the (hi-fi) simulation as long as the response is consistent with the situation. In other words, the results should fall with a "reasonable" range rather than hit a specific value. For this reason, if statistical methods are used for validating an emulation against simulation results, or comparing results from different emulation runs, confidence interval tests should be used rather than hypothesis testing [Law & Kelton, 1991].

5.2 Emulator Application

During the emulation application phase, the main task of locating and eliminating the sources of error in the software is performed. The emulator provides an off-line environment for testing various fault situations. In a sense, the problems associated with debugging the control software have just commenced.

5.2.1 Design for Testing

Verifying control software requires two aspects of the controller be tested: communications and control logic. Especially at the shop control level, the control software may be comprised of yet another component that needs to be verified: the user interface. It has been recommended that, in order to reduce the complexity, the testing task be split and each of the problems be dealt with independently at first and later in an integrated manner [Godio & Vignale, 1987].

Testing of control logic is the primary focus of this research, and the scope of testing shop control software has excluded testing the user interface and

communications. Thus, an assumption has been made that a clear separation between control logic and the other two components can be achieved. The ramifications of such an assumption for the testing process, and the effect on the emulation if this assumption cannot be made, need to be understood.

While it is a common practice to perform modular testing of software, it is important to keep in mind that the software being tested has been temporarily modified. On one hand, the modification helps partition the complexity, but, on the other hand, also has the potential to introduce a bug into the software. This bug may show up during, and confound, the initial testing process, or come to light only during the final integrated testing. Or worse, it may obfuscate some "real" errors in the software module being tested. In any case, this presents a strong argument for integrated testing even if portions of the control software have been individually tested.

In some cases, independent testing of two components may not be possible since one part is highly dependent on the other. For example, the synchronous communication capability directly affected the control logic of the FMAS shop controller, as described in Section 4.8. The complete control logic could not be tested without the communications capability in place.

In other cases, separating the components may be prevented by software design and/or implementation constraints. This is exemplified by the situation where the control software includes a software component which has a user interface built into it, but for which the source code is not available, e.g., a library module or other third party software. Another situation is when the code for the control logic is so heavily intertwined with the code for the user interface that the separation is best not done. Two possibilities exist for dealing with this situation. One option is to not include the human supervisor, who interacts with the controller through the interface, in the emulation. However, consistency and repeatability in the testing process is lost due the variability in human behavior. The second option is to write code, in the interface object of the emulated human supervisor, to mimic the sequence of keystrokes and/or mouse-clicks and pass them on to the control software. While this is certainly not an impossible task, it definitely increases the complexity of the emulation program.

The obvious conclusion from the preceding discussion is that "design for testability" should be a major emphasis right from the early stages of the control software development. One must also keep in mind that testing of control software components cannot completely replace integrated testing, and, by extension, emulation testing cannot eliminate testing with the full-scale hardware system.

5.2.2 Active Testing

The basic emulation, though useful during the initial development of the control software, is of limited use since it passively responds to commands from the controller. In this sense, the basic emulation is a passive test driver and primarily lends consistency to the testing process. The utility of the emulation is greatly enhanced if it can be used in an active role rather than passive one. By this it is meant that the emulation should drive the control software by generating rare events and special situations. The software developer can thus observe the controller not only under normal operation of the emulated shop but also under abnormal conditions. As a result of tests which push the limits of the control software, the software developer may decide to make the implementation even more robust.

In Chapter Three a discussion was provided on scenario initialization and fault introduction capabilities. Scenarios may emphasize dynamic production control situations, e.g., rush jobs, material shortage, machine breakdown, etc. In real time control, control logic is also affected by communications problems, e.g., incorrect or unexpected response, no response, time out, etc. Scenarios can also reproduce infrequently occurring, but common, shop operation situations, e.g., emergency and non-emergency shutdown and restart, initial/cold startup, recovery from temporary stoppage, etc.

5.2.3 Reduction of Testing Time

In this research, the term faster-than-real-time (FRT) emulation is used to refer to emulation runs which elapse less temporal time than conventional emulation. There appear to be two methods for reducing wall-clock time when using emulation for testing control software: delay-scaling, and mixed-mode emulation. The delay-scaling approach involves reducing all time delays associated with shop floor activities in the

model by a predetermined, common factor, say 50%, prior to beginning the emulation run. Thus, eight hours of shop operation may be observed in only four hours of wall-clock time. Determination of an appropriate scaling factor is critical, and must be estimated by trial and error [Rihar, 1994].

The second method for speeding up testing, mixed-mode emulation, is based on using dual modes of time-advance for the emulation (Figure 16). Time in a simulation model typically evolves faster than real-time when using the discrete event approach. Since changes in system state only occur at events, which are considered to be instantaneous in simulation time, the discrete event model steps from one event to the next. Large time periods of operation are thus simulated in a fraction of the wall-clock time; the time savings being dependent on the ability of the simulation computer to perform the processing of each event. However, in the emulation system, a real-time simulation is needed. This constraint is imposed by the communications and computational delays in the actual shop controller. If a strict next-event approach is used, the time in the emulation model may have marched on ahead by the time the shop control software responds to a signal from the emulation.

It may be possible to take advantage of the comparatively large inter-event time at the shop control level to avoid a strict real-time advance in the emulation. Even in the on-line control mode, several seconds or minutes elapse between events that require a controller response, which typically is of the order of several seconds. The emulation can run in real-time mode when the shop controller is computing a decision that potentially affects the events in the shop. Otherwise, the emulation runs in next-event mode when the controller is "idle". This dual mode of emulator operation could reduce testing time. Of course, the reduction factor depends on the 'utilization' of the shop controller. The key to this approach is the ability to identify the periods in which the emulator must run in real-time, and then be able to control the simulation clock advance mechanism.

5.2.3.1 Experiments in FRT Emulation

Experiments were conducted with the goal of gaining a better understanding of the issues in the FRT emulation testing of discrete event shop control systems. Many

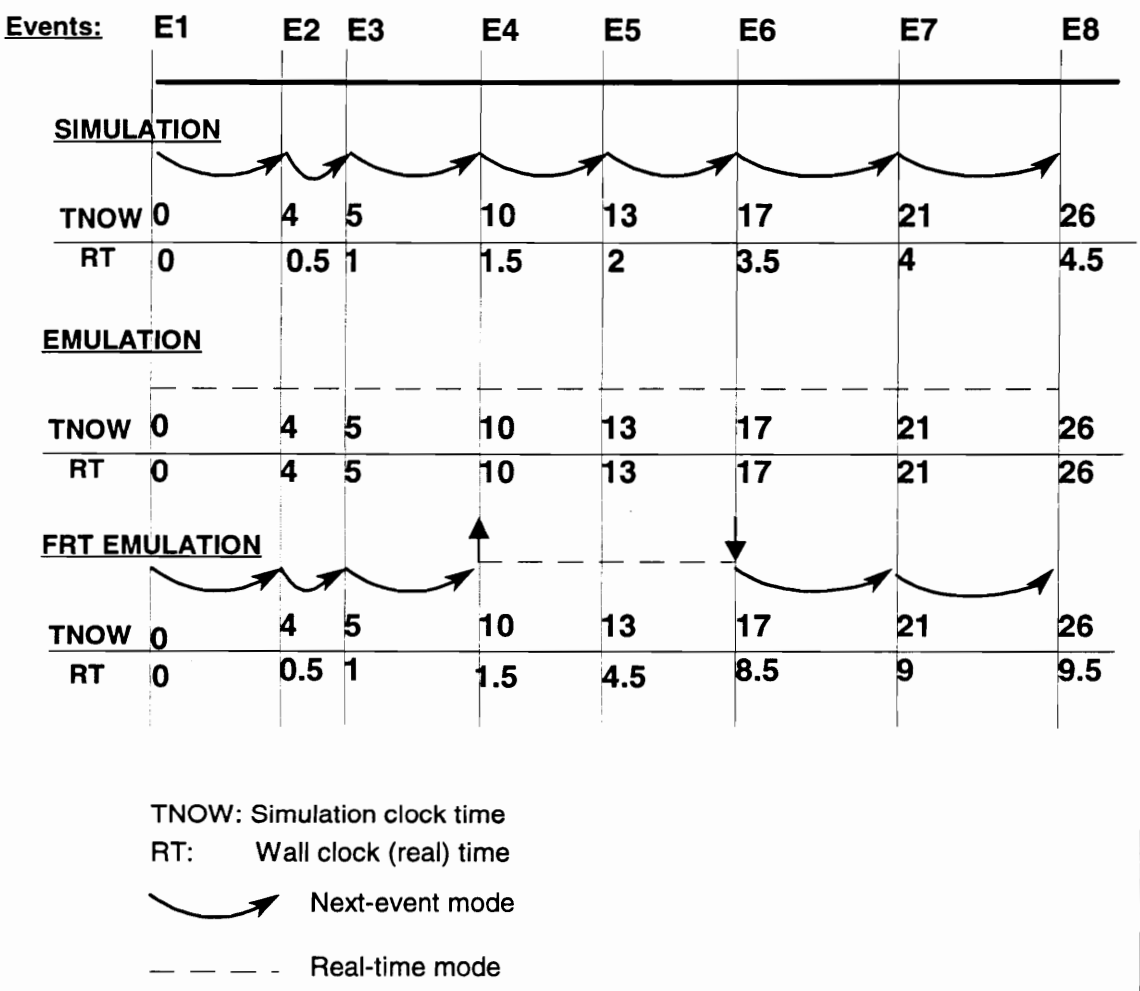


Figure 16: Faster-than-real-time emulation

runs were made under differing circumstances, but information from one particular scenario is reported and discussed here. For each FRT emulation technique, the emulation was run until three orders were processed through the system. A completely debugged control software was used to drive the emulation models. The time delays in the model were identical to those used in the simulation model (see Table 10) except for the time between order arrivals which was exponentially distributed with a mean of 20.0 minutes. The run time (wall-clock time) for each of the runs was benchmark against that of a conventional, i.e., real-time, emulation. Statistics on makespan, flowtime, and other performance measures were collected (Table 13) and compared. If they were found to differ, the trace of system activities were examined to understand the causes of the differences.

Delay scaling: An instance of an object of type `rtSimControlObj` was used to manage model time advance for accelerated emulation using the delay scaling technique. The object's `setTimeScale` method was used to vary the amount of wall-clock time elapsed for each unit of simulation time. This method was chosen over modifying the delay specification in the methods of each object. A 50% time scaling was applied to all delays in the emulation model. As a result, an assembly operation requiring two minutes of simulation time delay was completed in one minute of wall-clock time. Communications delays were also affected by the scaling. As a result, the communications file read operation in the emulation was performed every 7.5 seconds. However, the control software read the response and issued a command only every 15 seconds. Consequently, the run time was not 50% of the benchmark.

As seen in Table 13, this method tended to distort the statistical results, in addition to causing a substantial modification of sequence of activities in the FMAS. The main cause was the alteration in the message exchange pattern between the two components. Since messages from the emulation were being generated at a faster rate, they were being queued up for reading by the shop controller. Decisions were made by the shop controller while being unaware that other events had already happened in the model. The decisions were valid based on the status information available to the shop control software but invalid based on the true state of the shop.

Table 13: Results of faster-than-real-time emulation

	Emulation			
	Real-time	Delay-scaling 50%: only emulation	Delay-scaling 50%: emulation and control software	Mixed-mode
Run-time (wall-clock sec.)	6255	3510	3127.5	2055
Makespan (simulated min.)	104.25	117.0	104.25	105.25
Job time in system: mean	26.2	34.2	26.2	26.2
Blocking of MWC: count	3	3	3	3
MWC Robot: % utilization	38.77	34.54	38.77	38.4
time in use	40.41	40.41	40.41	40.41
AWC Robot: % utilization	29.16	25.99	29.16	28.89
time in use	30.40	30.40	30.40	30.40
Jobs in storage: count	8	8	8	8
Jobs in storage: time-weighted average	0.3165	0.6624	0.3165	0.3159

To overcome the problems caused by the above approach, a modified approach to delay scaling was adopted. In this, the emulation was run with 50% scaling and a similar reduction in communication time (i.e., 7.5 seconds between file reads) in the control software. This brought the file read-write operation (i.e., communications) in both programs in sync with each other. This reduction was possible because the minimum time required for reliable communications was only 3 seconds (found out by trial and error).

Under these settings, the run time requirements were reduced by the scaling factor (compared to the bench mark) and the system performance measures were identical. Also, the sequence of events in the emulation were unchanged from the conventional emulation.

FRT Emulation: A `frtSimControlObj` was developed to implement the hybrid time advance mechanism for FRT emulation. This object was derived from `rtSimControlObj` (described in Section 4.4.10) and thus inherited the real-time advance capabilities. The major difference between the two was the manner of computing the amount of elapsed wall-clock time. In `frtSimControlObj`, a correction factor is required in order to account for the periods when the real-time advance mechanism is turned off in the emulation model (i.e., when the next-event mechanism is used). This correction factor is computed by tracking the amount of simulation time passed in next-event mode, and is then added to the actual real time elapsed. The `SetTimeAdvance` method inherited from `SimControlObj` is overridden and extended to note the simulation clock time.

In addition to the time manager object, the shop controller object was required to be modified to implement FRT emulation. The new object--`fastVirtualShopConObj`--was derived from `virtualShopConObj` which is used in conventional emulation. The `inform` and `receive` methods of `fastVirtualShopConObj` augment the inherited capabilities to implement the mixed-mode time advance algorithm. The algorithm is as follows:

1. Start up and run the emulation in real-time mode.

2. If a command from the shop controller is received (read in), then tell the time manager object to change to next-event mode (if not already in that mode).

3. If a signal is sent (written out) to the shop controller, then tell the time manager object to change to real-time mode (if not already in that mode).

The rationale is that when a command, or more precisely a batch of commands, is read in by the emulation no more commands will be sent by the shop controller at least until after a signal is written out by the emulation. In other words, the next external event that potentially affects the activities in the emulation will occur only after an event in the emulation results in a message to the shop controller. Thus, until the occurrence of a write event, the emulation can safely execute next-event time advance, as in a simulation model.

This simple mechanism for opportunistic "collapsing" of emulation events works adequately in the presence of three major assumptions:

1. all entities, other than the shop controller, are contained in the emulation model;

2. there are no time events, other than reading in communications input, in the shop control software; and

3. computational delays are not considered.

With the dual modes of time-advance, the model had a run time of only 2055 seconds when compared to 6255 seconds with strict real-time advance. In all, 4260 seconds were saved by use of next-event time advance. The sequence of emulation events and the system performance measures were almost identical to the conventional emulation. However, a very slight discrepancy in numbers was observed in this and other model runs using FRT emulation. This was attributable to the fact that communications messages being picked up in different batches. However, the problem was not as pronounced as for the delay-scaling. Switching into real-time mode delays the occurrence of events in the model and thus reduces the number of messages queuing up in the shop controller's mailbox.

Further improvement is possible to this method. A more robust and general implementation calls for: knowing which emulated signals mandate switching to real-time mode; and, knowing which response to wait for to return to next-event mode. The simplicity of the implementation makes this technique attractive even though the resulting time savings may not be optimal/maximized. When testing time is of the order of several hours or few days, even a modest 10-20% savings in testing time is substantial.

Analysis: It was possible to *stimulate* the control software with all three techniques. The emulation models using the second method of delay scaling and the mixed-mode of time advance were valid from a control perspective because they reproduced the results from the conventional emulation. However, it is debatable if the model using the first method of delay scaling represents a valid emulation. While the emulated responses (i.e., behavior of the shop floor model) was consistent with the control commands received, Small amounts of deviations in statistical values or in sequence of events ought to normally be acceptable in exchange for the time-savings. However, when the order of events is disrupted, it is important to consider the possibility that the control software might be subjected to situations which may be atypical of, or even impossible in, the actual system.

It is fair to say that FRT emulation potentially compromises the validity from a communications perspective, since accelerated testing results in a higher messaging rate than in the conventional emulation. It is important to not overlook some potential implementation problems due to the capacity constraints of the communications LAN and control computer. Shortened activity times means increased communication rate with the controller, and hence, increased traffic on the LAN, as well as frequency of real-time decision making by the shop controller. While it is unlikely that FRT emulation will overwhelm the computer running the emulation (since it is already capable of running the discrete event simulation), it is very possible to overload the control computer.

FRT emulation also creates a problem since the accelerated time advance in the emulation results in lesser time being elapsed by the real-time clock used by the control software. This fouls up computation of time-dependent statistics in the shop control

software and leads to underestimation of these values. Consequently, control decisions (especially those that involve thresholding, e.g., IF robot has been idle for more than 10 min., THEN ...) based on these values will be affected. The job time-in- system data shown in Table 13 had to be manually computed since the control software underestimated the elapsed time. With the second delay scaling technique it was at least possible to compute the correct values by merely multiplying the results by the scaling factor. It is important to note that computation of statistics in the emulation is based on the simulation time, and is thus not affected by either delay-scaling or FRT emulation. One way to overcome this problem is to have the control software use a new function for time information, and have the function update its item based on the time in the simulation. Of course, this may not be a straightforward task.

Although the synchronization mechanism needed for the dual mode of emulation operation appears to be more complex than the delay scaling approach, it also promises to be a more robust method. The strength of mixed-mode emulation is in situations which have fairly large inter-event times. In the emulation of the FMAS, this was made apparent when the shop was idle between the completion of order number 1 (at SimTime 23.0) and the arrival of order number 2 (at SimTime 36.6). Almost 13% of run-time was saved by "collapsing" the two events into one.

5.3 Summary

A verified and validated emulation is critical to debugging control software since deviations from expected behavior can be attributed to flaws in the control software. The emulation verification and validation task is confounded by the inability to dynamically evaluate the emulation since the control software that drives it is under development. The dual-purpose modeling structure, for simulation and emulation, developed in this research provides a means to alleviate this problem.

It is recommended that the emulation be enhanced with features and an environment for emulation testing be developed to assist the software engineer in the testing process. The feature of faster-than-real-time emulation, to reduce the run time needed for studying extended shop operation, was examined. The mixed-mode emulation technique was found to reliably reduce emulation run times by more than

50%. It is ideally suited for shop-level control systems due to the relatively large inter-event times.

6. CONCLUSIONS AND RECOMMENDATIONS

6.1 Summary

There has been a growing interest in developing automated ("intelligent") shop control systems in both academic research and industrial applications. This trend has not been matched by research in testing and debugging these complex computer programs. This research addressed the issues of emulator development and its application in the testing of shop control software. Software emulation provides a means of dynamic, off-line testing of control software prior to the installation of the shop sub-systems. However, current emulation efforts tend to be ad hoc in nature and lack a strong conceptual framework in their development.

A methodology was presented in this research for reducing emulation development effort by use of a common structure for emulation and simulation models. This methodology is based on building an emulation model by adapting/extending an appropriately structured simulation model of the system. The methodology involves four main steps. The first step is to implement a simulation model of the system for which the shop control software is to be tested. This simulation model is used in the conventional task of evaluating shop control strategies. In the next step, a simulation model which incorporates additional manufacturing system details and constraints relevant in emulation testing is created by enhancing the model developed in the first step. This hi-fi simulation model is used primarily to verify the implementation of new model elements. In the third step, a first-cut emulation is crafted by removing the controlling influence (of the shop-level controller), adding a mechanism for real-time advance, and adding capabilities to communicate with an external controller. Many aspects of the emulation, especially those pertaining to the exchange of messages between the model and control software distribution, can be validated and tested with this model. In the fourth step, the final emulation model is produced and interfaced with the actual shop control software being tested.

Conventional simulation software and modeling methods are a hindrance to putting this emulator development method into practice. A framework was presented

which provides guidelines for the design of key simulation model elements to enable a major portion of the simulation code to be reused in the emulation model. The following modeling principles are presented as enablers of the emulation development method:

1. Use OOM techniques to separate the functionality in a simulation model. The major groups of entities are those that: 1) control and support simulation; 2) represent the physical elements in the manufacturing system; and 3) perform decision-making to control the operation of entities in the manufacturing system.

2. Adopt a controller-interaction perspective for simulation modeling instead of the traditional view of events being a consequence of a job flowing through the system.

3. Design the class structure keeping in mind the dual nature of the model.

The guidelines for the object-oriented modeling of key simulation model elements, based on the above principles, are summarized as follows: Jobs and orders are modeled as passive, information entities. This is because controllers, and not jobs, are the primary active entities in the model. Controller objects are designed so that all data and logic contained in them may be accessed by other controller or manufacturing system entities only through its communication module, i.e., the methods of the communications module represent the interface to other objects and conceal the implementation of the decision making logic. Also, a controller object is implemented as an union of a logic object and a interface object. Entities belonging to the class referred to as DecInfo objects, e.g., human supervisors, databases and knowledge-bases, may be modeled in a similar manner in order to facilitate seamless interaction with their physical counterparts during emulation. The differing emphasis on statistics in simulation and emulation is accommodated by separating the statistics collection task from the reporting task. Individual (manufacturing) objects have methods to collect statistics upon themselves, but do so only when directed by a system-level experiment object which does the statistics reporting. Links/coupling between objects are not defined at the object definition level, instead a system object establishes the relationships between instances of the objects needed in a particular model.

The dual-purpose modeling structure developed in this research lends structure to the emulation development task. Additionally, it also provides a means to alleviate the problem of emulation verification and validation. This task is confounded by the inability to dynamically evaluate the emulation since the control software that drives it is under development. A method to rapidly craft a driver for the purpose of testing the emulation was developed. The viability of the emulation development method was demonstrated in a prototype system for the Flexible Machining and Assembly System at Virginia Tech. The models were created from object classes developed in the MODSIM II simulation language.

This research also examined the issue of actually employing the emulation, as a substitute for the physical system, in order to verify the functionality of the shop control software. A solution was sought to the problem of having to observe the emulation for extended time periods, in the hope of detecting errors. Two techniques for faster-than-real-time (FRT) emulation--delay-scaling and mixed-mode emulation--were developed and implemented. Experiments were conducted to study their ability to compress emulated shop operation into small periods of wall-clock time. The mixed-mode emulation technique was found to have the potential to reduce run times by well over 50% while still retaining the sequence of activities observed in conventional emulation.

6.2 Contributions

Four distinct contributions have been made in this research:

First and foremost, this research has developed and demonstrated a method for emulation model development for shop control software testing. While the use of software emulation for testing controllers at lower levels of the manufacturing control hierarchy has been raised from the level of academic research to industrial application, emulation for testing shop-level controllers has not even been a subject of extensive research. Furthermore, none of the published research explicitly discusses *how to* develop an emulation for the purpose. The model development steps and modeling guidelines presented in this dissertation represent an answer to the question, "How do I go about developing an emulation model?", and can thus guide future emulation development efforts. In the past, the tasks of simulation and emulation model

development have been viewed as being separate and non-overlapping. In this research, the simulation model is the first step towards the emulation model and represents a unique approach to emulation model development. Since a substantial part of the emulation model development is done during the development of the simulation model used for testing shop control strategies, the result is a savings in time and effort.

The second contribution is the use of OOM to link simulation and emulation models. An OO modeling structure developed in this research which has the dual perspective of simulation and emulation. There are no published accounts of OOM for emulation systems. Specifically, the research demonstrated that the controller-interaction modeling perspective is suitable for both simulation and emulation models. As part of the modeling guidelines, this research also provided a conceptual controller model to help get away from the job-flow perspective mind set and assist in implementing the controller-interaction perspective.

The third contribution is a method for validating the emulation. During the testing process it is highly desirable that all unexpected behavior be a result of errors in the control software and not the emulation. However, the critical task of emulation verification and validation is confounded by the inability to dynamically evaluate the emulation since the control software that drives it is under development. This research presented a method for dynamic verification, validation and testing of the emulation by moving through the steps of hi-fi simulation and first-cut emulation. The unique approach to rapidly craft a driver to stimulate the emulation model is enabled by the object-oriented modeling framework on which the overall emulation development method is based.

The fourth contribution is in emulation application. Due to its control horizon, it is obviously impractical to test the shop control software in real-time. None of the research in manufacturing emulator development discusses the ability to test the control systems in faster-than-real-time mode. This research investigated the implementation issues for a method to reduce the real time needed to test the extended operation of the shop

control software. In particular, a technique which uses a dual mode of time-advance was developed and its viability demonstrated.

6.3 Recommendations for Future Research

This research has demonstrated the feasibility of a dual-purpose modeling structure, based on the object-oriented paradigm, for simulation and emulation. The next step is to build upon the modeling guidelines presented in Chapter Three and create a formal method, or an architecture, for specifying the system from the controller-interaction perceptive and for designing the objects for the dual purpose of simulation and emulation.

The research has discussed the conceptual feasibility of more than one external entity interacting with the emulation model, e.g., the shop control computer and a human supervisor. Additional work is needed to gain a better understanding of the implementation issues and of implications for the testing process.

In addition to methods for emulation model development, an integrated testing environment should be the subject of future research. This environment is envisioned to provide specific tools and functions to assist in testing and debugging similar to those found in integrated simulation environments. The following aspects have the potential for much conceptual research:

Incorporating error detection knowledge into the emulation: During simulation or emulation development one becomes aware of potential errors and builds alarms into object methods. Detection of system level errors and incorporation of this knowledge into the emulation software is a more challenging but potentially more rewarding task. Research is needed to find methods for the developer to easily transfer knowledge such as that of illegitimate system states, expected response and/or expected response time, etc. into the emulation.

Given the value of active testing with emulation, i.e., fault introduction, research on specification of an appropriate software structure to define experimental scenarios for runs with emulation models is mandated. This will require an understanding of what constitutes a "typical" scenario for emulation testing.

The mixed-mode emulation technique has lot of potential for significant savings of testing times. The technique can be refined so that switching between time-advance modes is done more "intelligently." Also, research can be done to make the method applicable to more general situations, taking into account idiosyncrasies of different control methods.

It is important to note here that while literature on emulator development and application in the manufacturing arena is sparse, there is much that can be gained from other domains. Hardware-in-the-loop (HIL) simulation is common in the development of software to control military systems [Fayad et al., 1992], and Man-in-the-loop (MIL) simulation is used for training humans in decision making [Hopkinson & Sepulveda, 1995]. Mechanisms for faster-than-real-time emulation can draw from research in distributed and parallel simulation [Fujimoto, 1995].

7. REFERENCES

1. Adiga, S. and C. R. Glassey (1991): "Object-Oriented Simulation to Support Research in Manufacturing", *International Journal of Production Research*, Vol. 29, No. 12, 1991, pp. 2529-2542.

2. Ashfal, C. R. and A. Balagamwala (1990): "SIMLOG: A PROLOG-based Simulator for Industrial Logic Control Systems", *Computers & Industrial Engineering*, Vol. 19, No. 1-4, 1990, pp. 195-199.

3. Balci, O. (1995): "Principles and Techniques of Simulation Validation, Verification, and Testing", In: *Proceedings of the 1995 Winter Simulation Conference*, (Eds.: C. Alexopoulos, K. Kang, W. R. Lilegdon and D. Goldsman), 1995, pp. 147-154.

4. Balci, O. (1994): "Validation, Verification, and Testing Techniques Throughout the Life Cycle of a Simulation Study", In: *Proceedings of the 1994 Winter Simulation Conference*, (Eds.: J. D. Tew, S. Manivannan, D. A. Sadowski and A. F. Seila), 1994, pp. 215-220.

5. Barnichon, D., C. Caux and M. Gourgand (1990): "Methodology for Manufacturing System Performance Analysis Using a SIMAN-Petri Nets Coupling", In: *Intelligent Process Control and Scheduling: Discrete Event Systems. Proceedings of the 1990 European Simulation Symposium*, (Eds.: G. C. Vansteenkiste et al.), SCS, San Diego, CA, 1990, pp. 148-152.

6. Bastos, J. M. and N. Shires (1987): "Factory Simulation for Planning and Control", In: *Proceedings of the 3rd International Conference on Simulation in Manufacturing - SIM 3*, IFS, Bedford, UK, 1987, pp. 51-66.

7. Beizer, B. (1990). Software Testing Techniques, Second Edition, Van Nostrand Reinhold, New York, NY, 550 pages.

8. Belanger, R. (1990): "MODSIM II--A Modular Object-Oriented Language", In: *Proceedings of the 1990 Winter Simulation Conference*, (Eds.: O. Balci, R. P. Sadowski and R. E. Nance), 1990, pp. 118-122.

9. Ben Hadj-Alouane, N., J. K. Chaar and A. W. Naylor (1990): "The Design and Implementation of the Control and Integration Software of a Flexible Manufacturing System", In: *Proceedings of the First International Conference on Systems Integration*, 1990, pp. 494-502.

10. Bhuskute, H. C., M. N. Duse, J. T. Gharpure, D. B. Pratt, M. Kamath and J. H. Mize (1992): "Design and Implementation of a Highly Reusable Modeling and Simulation Framework for Discrete Part Manufacturing Systems", In: *Proceedings of the 1992 Winter Simulation Conference*, (Eds.: J. J. Swain, D. Goldsman, R. C. Crain and J. R. Wilson), 1992, pp. 680-688.

11. Bijou, J. M., M. Y. P. Courvoisier, H. Demmou, C. Desclaux, J. C. Pascal and R. J. Valette (1987): "A Methodology of Specification and Implementation of Distributed Discrete Control Systems", *IEEE Transactions on Industrial Electronics*, Vol. IE-34, No. 4 (November), 1987, pp. 417-421.

12. Bilberg, A. and L. Alting (1990): "Simulation, a Tool for Developing FMS Control Programs", In: *Proceedings of the 28th International MATADOR Conference*, 1990, pp. 155-162.

13. Bischak, D. P. and S. D. Roberts (1991): "Object-Oriented Simulation", In: *Proceedings of the 1991 Winter Simulation Conference*, (Eds.: B. L. Nelson, W. D. Kelton and G. M. Clark), 1991, pp. 194-203.

14. Blair, E. L. and S. Selvaraj (1989): "DISC++: A C++ Based Library for Object Oriented Simulation", In: *Proceedings of the 1989 Winter Simulation Conference*, (Eds.: E. A. MacNair, K. J. Musselman and P. Heidelberger), 1989, pp. 301-305.

15. Bloom, H. M., C. M. Furlani and A. J. Barbera (1984): "Emulation as a Design Tool in the Development of Real-Time Control Systems", In: *Proceedings of the 1984 Winter Simulation Conference*, (Eds.: S. Sheppard, U. Pooch and D. Pegden), 1984, pp. 627-636.

16. Bodner, D. A., S. J. Dilley, S. Narayanan, U. Sreekanth, T. Govindraj, L. F. McGinnis and C. M. Mitchell (1993): "Object-oriented Modeling and Simulation of Automated Control in Manufacturing", In: *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, 1993, pp. 83-88.

17. Booch, G. (1994). Object-Oriented Analysis and Design with Applications, Second Edition, Benjamin/Cummings Publishing Co., Redwood City, CA, 589 pages.

18. Bradshaw, W. W. (1987): "Using Simulation to Upgrade a Warehouse Control System: TI Case Study", *CIM Review*, Vol., No. Winter 1987, 1987, pp. 631-638.

19. Cecil, J. A., K. Srihari and C. R. Emerson (1992): "A Review of Petri-Net Applications in Manufacturing", *The International Journal of Advanced Manufacturing Technology*, Vol. 7, No. 3, 1992, pp. 168-177.

20. Chanchien, S. W., L. Lin and D. Sun (1995): "A Dynamic Control Model of Flexible Manufacturing Cells Using the Information Processing Object Hierarchy", *International Journal of FAIM*, 1995.

21. Chandra, J. and J. Talavage (1991): "Intelligent Dispatching for Flexible Manufacturing", *International Journal of Production Research*, Vol. 29, No. 11, 1991, pp. 2259-2278.

22. Chen, Y. and C. Wongladkown (1991): "A Real-Time Control Simulator Design for Automated Manufacturing Systems Using Petri Nets", In: *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, 1991, pp. 2542-2547.

23. Clark, G. M. and D. H. Withers (1989): "Architecture for an Integrated Simulation/CIM system", In: *Proceedings of the 1989 Winter Simulation Conference*, (Eds.: E. A. MacNair, K. J. Musselman and P. Heidelberger), 1989, pp. 942-948.

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

25. Cosic, K. (1992): "Real Time Simulation in Control System Design and Testing", In: *Proceedings of the 1992 American Control Conference*, IEEE, Piscataway, NJ, 1992, pp. 1971-1975.

26. Courvoisier, M., R. Valette, J. C. Pascal, D. Barbalho, Y. Baudin and K. Benzakour (1987): "Distributed Emulation of Flexible Manufacturing Systems", In: *Proceedings of - IECON 87*, 1987, pp. 957-964.

27. Courvoisier, M., J. M. Bijou, R. J. Valette, C. Desclaux and K. Benzakour (1984): "The S.E.CO.I.A. project", In: *Proceedings of the 6th European Conference on Electrotechnics - EUROCON 84*, 1984, pp. 1-4.

28. Davis, W. J., J. Macro and D. Setterdhal (1994): "An Object-oriented, Coordination-based Simulation Model for the RAMP Flexible Manufacturing System", In: *Proceedings of the Fourth International Factory Automation and Integrated Manufacturing (FAIM) Conference*, (Eds.: W. G. Sullivan and M. M. Ahmad), Bergell House, Inc, New York, NY, 1994, pp. 148-157.

29. Davis, W. J., D. Setterdhal, J. Macro, V. Izokaitis and B. Bauman (1993): "Recent Advances in the Modeling, Scheduling and Control of Flexible Automation", In: *Proceedings of the 1993 Winter Simulation Conference*, (Eds.: G. W. Evans, M. Mollaghasemi, E. C. Russell and W. E. Biles), 1993, pp. 143-155.

30. Desrochers, A. A. (1990): "Modeling and Control Using Petri Nets", In: *Modeling and Control of Automated Manufacturing Systems*, (Ed.: A. A. Desrochers), IEEE Computer Society Press, Washington, DC, 1990, pp. 239-251.

31. Doyle, R. J. (1990): "Object-oriented Simulation Programming", In: *Proceedings of the SCS Multiconference on Object Oriented Simulation*, (Ed.: A. Guasch), SCS, San Diego, CA, 1990, pp. 1-6.

32. Drolet, J. R., C. L. Moodie and B. Montreuil (1991): "Object Oriented Simulation With SMALLTALK-80: A Case Study", In: *Proceedings of the 1991 Winter Simulation Conference*, (Eds.: B. L. Nelson, W. D. Kelton and G. M. Clark), 1991, pp. 312-322.

33. Duffie, N. A. and R. S. Piper (1987): "Non-Hierarchical Control of a Flexible Manufacturing Cell", *Robotics & Computer Integrated Manufacturing*, Vol. 3, No. 2, 1987, pp. 175-179.

34. Duffie, N. A. and V. V. Prabhu (1994): "Real-time Distributed Scheduling of Heterarchical Manufacturing Systems", *Journal of Manufacturing Systems*, Vol. 13, No. 2, 1994, pp. 94-107.

35. Duffie, N. A., R. Chitturi and J. Mou (1988): "Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities", *Journal of Manufacturing Systems*, Vol. 7, No. 4, 1988, pp. 315-328.

36. Duggan, J. and J. Browne (1988a): "An AI based Simulation for Production Activity Control Systems", In: *Proceedings of the 4th International Conference on Simulation in Manufacturing - SIM-4*, IFS Publications, Bedford, UK, 1988a, pp. 177-193.

37. Duggan, J. and J. Browne (1988b): "ESPNET: Expert System-based Simulator of Petri Nets", *IEE Proceedings-D*, Vol. 135, No. 4 (July), 1988b, pp. 239-247.

38. Ennulat, H. W. (1992): "Emulation Framework for Testing Higher Level Control Methodology", M.S. thesis, Virginia Tech, Blacksburg, VA, 1992, 131 pages.

39. Entsminger, G. (1990). The Tao of Objects: A Beginners Guide to Object-Oriented Programming, M&T Books, Redwood City, CA, 249 pages.

40. Erickson, C., A. Vandenberge and T. Miles (1987): "Simulation, Animation, and Shop-Floor Control", In: *Proceedings of the 1987 Winter Simulation Conference*, (Eds.: A. Thesen, H. Grant and D. W. Kelton), 1987, pp. 649-653.

41. Fabian, M. and B. Lennartson (1992): "Control of Manufacturing Systems: An Object Oriented Approach", In: *Proceedings of the 7th IFAC Symposium on Information Control Problems in Manufacturing Technology (INCOM 92)*, (Ed.: M. B. Zaremba), Pergamon Press, Oxford, England, 1992, pp. 47-52.

42. Fayad, M. E., L. J. Hawn, M. A. Roberts and J. W. Schooley (1992): "Hardware-in-the-loop (HIL) Simulation: An Application of Colberts's Object-Oriented Software Development Method", In: *Proceedings of Tri-Ada '92*, (Ed.: C. B. Engle Jr.), ACM, New York, NY, 1992, pp. 176-188.

43. Fishwick, P. A. (1992): "SimPack: Getting Started with Simulation programming in C and C++", In: *Proceedings of the 1992 Winter Simulation Conference*, (Eds.: J. J. Swain, D. Goldsman, R. C. Crain and J. R. Wilson), 1992, pp. 154-162.

44. Fujimoto, R. M. (1995): "Parallel and Distributed Simulation", In: *Proceedings of the 1995 Winter Simulation Conference*, (Eds.: C. Alexopoulos, K. Kang, W. R. Lilegdon and D. Goldsman), 1995, pp. 118-125.

45. Galgocy, C. B. (1986): "Integrating a Flexible Manufacturing System with Programmable Controller Ladder Logic through Simulation", M.S. thesis, Virginia Tech, Blacksburg, VA, 1986, 208 pages.

46. Gaskins, R. J. and J. M. A. Tanchoco (1989): "AGVSim2-a Development Tool for AGVS Controller Design", *International Journal of Production Research*, Vol. 27, No. 6, 1989, pp. 915-926.

47. Glassey, C. R. and S. Adiga (1989): "Conceptual Design of a Software Object Library for Simulation of Semiconductor Manufacturing Systems", *Journal of Object Oriented Programming*, Vol. 2, No. 4 (Nov/Dec), 1989, pp. 39-43.

48. Godio, C. and A. Vignale (1987): "Plant Emulators for Control Software Test", In: *Proceedings of the 3rd International Conference on Simulation in Manufacturing - SIM-3*, (Ed.: G. F. Micheletti), IFS Publications, Bedford, UK, 1987, pp. 137-147.

49. Govindraj, T., L. F. McGinnis, C. M. Mitchell, D. A. Bodner, S. Narayanan and U. Sreekanth (1993): "OOSIM: A Tool for Simulating Modern Manufacturing Systems", In: *Proceedings of the 1993 NSF Design and Manufacturing Systems Grantees Conference*, 1993, pp. 1055-1062.

50. Govindraj, T., L. McGinnis, C. M. Mitchel and L. K. Platzman (1990): "Manufacturing Simulation Using Objects", In: *Proceedings of the 1990 Summer Computer Simulation Conference*, (Eds.: B. Sveerk and J. McRae), SCS, San Diego, CA, 1990, pp. 219-224.

51. Guo, D., D. H. Norrie and O. R. Fauvel (1990): "Object-Oriented Flexible Manufacturing System Simulation", In: *Proceedings of the 1990 Summer Computer Simulation Conference*, (Eds.: B. Sveerk and J. McRae), SCS, San Diego, CA, 1990, pp. 225-230.

52. Hadavi, K., M. S. Shahraray and K. Voigt (1990): "ReDS--A Dynamic Planning, Scheduling, and Control System for Manufacturing", *Journal of Manufacturing Systems*, Vol. 9, No. 4, 1990, pp. 332-344.

53. Harmonosky, C. M. (1990): "Implementation Issues Using Simulation for Real-Time Scheduling, Control, and Monitoring", In: *Proceedings of the 1990 Winter Simulation Conference*, (Eds.: O. Balci, R. P. Sadowski and R. E. Nance), 1990, pp. 595-598.

54. Harmonosky, C. M. and D. C. Barrick (1988): "Simulation in a CIM Environment: Structure for analysis and Real-Time Control", In: *Proceedings of the 1988 Winter Simulation Conference*, (Eds.: M. Abrams, P. Haigh and J. Comfort), 1988, pp. 704-711.

55. Hatvany, J. (1985): "Intelligence and Cooperation in Heterarchic Manufacturing Systems", *Robotics & Computer Integrated Manufacturing*, Vol. 2, No. 2, 1985, pp. 101-104.

56. Hennell, M. A., D. Hendley and I. J. Riddell (1987): "Automated Testing Techniques for Real-Time Embedded Software", In: *ESEC '87: 1st European Software Engineering Conference*, (Eds.: H. K. Nichols and D. Simpson), Springer-Verlag, Berlin, Germany, 1987, pp. 244-253.

57. Hitchens, M. W. and T. K. Ryan (1989): "Direct Connect Emulation and the Project Life Cycle", In: *Proceedings of the 1989 Winter Simulation Conference*, (Eds.: E. A. MacNair, K. J. Musselman and P. Heidelberger), 1989, pp. 843-847.

58. Hopkinson, W. C. and J. A. Sepulveda (1995): "Real-Time Validation of Man-in-the-Loop Simulations", In: *Proceedings of the 1995 Winter Simulation Conference*, (Eds.: C. Alexopoulos, K. Kang, W. R. Lilegdon and D. Goldsman), 1995, pp. 1250-1256.

59. Johnson, M. E., L. Thompson and R. Fontaine (1992): "An Integrated Simulation and Shop-Floor Control System", *Manufacturing Review*, Vol. 5, No. 3, 1992, pp. 158-165.

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

61. Joines, J. A., K. A. Powell and S. D. Roberts (1992): "Object-Oriented Modeling and Simulation with C++", In: *Proceedings of the 1992 Winter Simulation Conference*, (Eds.: J. J. Swain, D. Goldsman, R. C. Crain and J. R. Wilson), 1992, pp. 145-153.

62. Jones, A. T. and C. R. McLean (1985): "A Proposed Hierarchical Control Model for Automated Manufacturing Systems", *Journal of Manufacturing Systems*, Vol. 5, No. 1, 1985, pp. 81-95.

63. Joshi, S. B. and J. S. Smith (1992): "Intelligent Control of Manufacturing Systems", In: *Intelligent Design and Manufacturing*, (Ed.: A. Kusiak), John Wiley & Sons, Inc, 1992, pp. 491-520.

64. Kockerbeck, G. and O. Schlictherle (1990): "Real-time Simulation as an Instrument for Validation of Control Software, Training and Process Visualisation Shown at an AGVS", In: *Proceedings of the 1990 European Simulation Multi-conference*, (Eds.: W. Frish et al.), SCS, San Diego, CA, 1990, pp. 380-384.

65. Krogh, B. H., R. Willson and D. Pathak (1987): "Automatic Generation of Control Programs for Discrete Manufacturing Processes", *Annual Research Review*, 1987, pp. 21-31.

66. Kuhn, R. (1989): "Generating Extended State Transitions from Structured Specifications for Process Control Systems", *Software Engineering Journal*, Vol. 4, No. 5, 1989, pp. 283-291.

67. Law, A. M. and W. D. Kelton (1991). Simulation Modeling and Analysis, Second Edition, McGraw-Hill, Inc., New York, 759 pages.

68. Lin, L., M. Wakabayashi and S. Adiga (1994): "Object-oriented Modeling and Implementation of Control Software for a Robotic Flexible Manufacturing Cell", *Robotics & Computer Integrated Manufacturing*, Vol. 11, No. 1, 1994, pp. 1-12.

69. Little, M. C. and D. L. McCue (1994): "Construction and Use of a Simulation Package in C++", *C User's Journal*, Vol. 12, No. 3, 1994.

70. Lomov, G. and D. Baezner (1990): "A Tutorial Introduction to Object-Oriented Simulation and Sim++", In: *Proceedings of the 1990 Winter Simulation Conference*, (Eds.: O. Balci, R. P. Sadowski and R. E. Nance), 1990, pp. 149-153.

71. Macro, J., W. J. Davis and D. Setterdhal (1994): "Establishing an Object-Oriented Methodology for the Simulation and Control of Integrated Manufacturing Systems", In: *Proceedings of the 1994 Winter Simulation Conference*, (Eds.: J. D. Tew, S. Manivannan, D. A. Sadowski and A. F. Seila), 1994, pp. 954-961.

72. Manivannan, S. and J. Banks (1992): "Design of a Knowledge-Based On-Line Simulation System to Control a Manufacturing Shop Floor", *IIE Transactions*, Vol. 24, No. 3, 1992, pp. 72-83.

73. Manivannan, S. and J. Banks (1991): "Real-time Control of a Manufacturing Cell using Knowledge-Based Simulation", In: *Proceedings of the 1991 Winter Simulation Conference*, (Eds.: B. Nelson, W. D. Kelton and G. M. Clark), 1991, pp. 251-260.

74. Marcus, R. (1990): "Simulation-Based Planning, Scheduling, and Control of Job Shops", *CIM Review*, Vol. 7, No. 1--Fall 90, 1990, pp. 44-49.

75. McHaney, R. (1988): "Bridging the Gap: Transfering Logic from a Simulation into an Actual System Controller", In: *Proceedings of the 1988 Winter Simulation Conference*, (Eds.: M. Abrams, P. Haigh and J. Comfort), 1988, pp. 583-590.

76. Mecker, S. S. (1989): "A Simulator for Ladder Logic Debugging", M.S. thesis, Virginia Tech, Blacksburg, VA, 1989, 204 pages.

77. Miles, T. I. (1989): "Using Discrete-Event Computer Simulation to Test Control Systems", In: *Proceedings of the 1989 Winter Simulation Conference*, (Eds.: E. A. MacNair, K. J. Musselman and P. Heidelberger), 1989, pp. 848-858.

78. Mize, J. H., H. C. Bhuskute, D. B. Pratt and M. Kamath (1992): "Modeling of Integrated Manufacturing Systems Using an Object-Oriented Approach", *IIE Transactions*, Vol. 24, No. 3, 1992, pp. 14-25.

79. Montreuil, B., P. Lefrancois and S. Harvey (1995): "Organism-Oriented Models of Manufacturing Systems", *Journal of Intelligent Manuacturing*, 1995.

80. Mujtaba, M. S. (1992): "Systems with Complex Material and Information Flows", In: *International Conference on Object-Oriented Manufacturing Systems*, 1992, pp. 188-193.

81. Muller, B. and A. Neumann (1992): "Development and Testing of Control Software for Flexible Manufacturing Systems", *Systems Science*, Vol. 18, No. 3, 1992, pp. 51-65.

82. Narayanan, S., D. A. Bodner, U. Sreekanth, T. Govindraj, L. F. McGinnis and C. M. Mitchell (to appear): "An Object-Based, Diriect-Image Approach to the Modeling and Simulation of Manufacturing Systems", *International Journal of Production Research*, to appear.

83. Narayanan, S., D. A. Bodner, U. Sreekanth, T. Govindraj, L. F. McGinnis and C. M. Mitchell (1994): "Research in Object-Oriented Manufacturing Simulations: An Assessment of the State of the Art", *Manufacturing Review*, 1994.

84. Narayanan, S., D. A. Bodner, C. M. Mitchell, L. F. McGinnis, T. Govindraj and L. K. Platzman (1992a): "Object-oriented Simulation to Support Modeling and Control of Automated Manufacturing Systems", In: *Proceedings of the 1992 Western Multiconference*, SCS, San Diego, CA, 1992a, pp. 59-63.

85. Narayanan, S., D. A. Bodner, U. Sreekanth, S. J. Dilley, T. Govindraj, L. F. McGinnis and C. M. Mitchell (1992b): "Object-Oriented Simulation to Support Operator Decision Making in Semiconductor Manufacturing", In: *Proceedings of the 1992 International Conference on Systems, Man, and Cybernetics*, 1992b, pp. 1510-1515.

86. Ogle, M. K. (1994): "The Interaction between Control and Model Elements in a Three Level Simulation Control Hierarchy", Ph.D. thesis, Arizona State University, Tempe, 1994, 179 pages.

87. Ogle, M. K., T. G. Beaumariage and C. A. Roberts (1991): "The Separation and Explicit Declaration of Model Control Structures in Support of Object-Oriented Simulation", In: *Proceedings of the 1991 Winter Simulation Conference*, (Eds.: B. L. Nelson, W. D. Kelton and G. M. Clark), 1991, pp. 1173-1179.

88. O'Grady, P. J. (1986). Controlling Automated Manufacturing Systems, Kogan Page, London, UK, 111 pages.

89. Onosato, M. and K. Iwata (1992): "VirtualWorks: Building a Virtual Factory with 3-D Modelling and Object Oriented Programming Techniques", In: *Proceedings of the 7th IFAC Symposium on Information Control Problems in Manufacturing Technology (INCOM 92)*, (Ed.: M. B. Zaremba), Pergamon Press, Oxford, England, 1992, pp. 281-286.

90. Payne, J. E. and R. Mills (1992): "A Distributed Simulation Environment for Simulating Manufacturing Processes", In: *Proceedings of the 1992 Summer Computer Simulation Conference*, (Ed.: P. Luker), Society for Computer Simulation, San Diego, CA, 1992, pp. 1169-1173.

91. Pegden, C. D. (1982). Introduction to SIMAN, Systems Modeling Corporation, State College, PA.

92. Pobanz, N. E. and J. R. Hernandez (1989): "Dynamic Simulation - An Engineering Tool for Advanced Control System Development/Checkout", *Advances in Instrumentation and Control: Proceedings of the ISA 1989 International Conference and Exhibit*, Vol. 44, No., 1989, pp. 805-816.

93. Quinn, E. B. (1985): "A Simulation Based System for Automatic Development and Testing of AGV Control Software", In: *Proceedings of the 3rd International Conference on AGV Systems*, IFS Publications, Bedford, UK, 1985, pp. 219-227.

94. Quirk, W. J., ed. (1985). Verification and Validation of Real-Time Software, Springer-Verlag, Berlin, Germany, 245 pages.

95. Rihar, M. (1994): "The Software Simulator as an Effective Tool for Testing Control Algorithms", *Simulation*, Vol. 63, No. 1, 1994, pp. 6-14.

96. Roberts, C. A., T. G. Beaumariage, Y. Dessouky and M. K. Ogle (1991): "Object Oriented Simulation Tools Necessary For A Flexible Batch Process Management Architecture", In: *Proceedings of the 1991 Winter Simulation Conference*, (Eds.: B. L. Nelson, W. D. Kelton and G. M. Clark), 1991, pp. 323-330.

97. Roberts, S. D. and J. Heim (1988): "A Perspective on Object-Oriented Simulation", In: *Proceedings of the 1988 Winter Simulation Conference*, (Eds.: M. Abrams, P. Haigh and J. Comfort), 1988, pp. 277-281.

98. Rogers, P. and R. Gordon (1993): "Simulation for Real-Time Decision Making in Manufacturing Systems", In: *Proceedings of the 1993 Winter Simulation Conference*, (Eds.: G. W. Evans, M. Mollaghasemi, E. C. Russell and W. E. Biles), IEEE, Piscataway, NJ, 1993, pp. 866-874.

99. Santamarina, G., C. Chen and S. Lee (1991): "An Application of C++ to Manufacturing Systems Control", *Computers & Industrial Engineering*, Vol. 21, No. 1-4, 1991, pp. 565-570.

100. Sargent, R. G. (1994): "Verification and Validation", In: *Proceedings of the 1994 Winter Simulation Conference*, (Eds.: J. D. Tew, S. Manivannan, D. A. Sadowski and A. F. Seila), 1994, pp. 77-87.

101. Sarin, S. C. and R. R. Salgame (1990): "Development of a Knowledge-based system for Dynamic Scheduling", *International Journal of Production Research*, Vol. 28, No. 8, 1990, pp. 1499-1512.

102. Schmidt, R., A. Schurholz and M. Ruger (1989): "Create! - Simulation Aided Development of Control Software for Automated Material Flow Systems", In: *Proceedings of the 1989 Summer Computer Simulation Conference*, (Ed.: J. K. Clema), SCS, San Diego, CA, 1989, pp. 64-69.

103. Schurholz, A. and B. Noche (1987): "Application of a Simulator for the Development and Check of Controlling Software", In: *Proceedings of the 1987 European Simulation Multi-conference*, 1987, pp. 126-130.

104. Schutz, W. (1993). The Testability of Distributed Real-Time Systems, Kluwer Academic Publishers, Boston, MA, 144 pages.

105. Schwetman, H. D. (1990): "Introduction to Process*Oriented Simulations and CSIM", In:* Proceedings of the 1990 Winter Simulation Conference, (Eds.: O. Balci, R. P. Sadowski and R. E. Nance), 1990, pp. 154-157.

106. Shewchuk, J. P. and Tien-C. Chang (1991): "An Approach To Object-Oriented Discrete-Event Simulation Of Manufacturing Systems", In: *Proceedings of the 1991 Winter Simulation Conference*, (Eds.: B. L. Nelson, W. D. Kelton and G. M. Clark), 1991, pp. 302-311.

107. Shires, N. (1988): "On-Line Simulation and Monitoring for Real-Time Decision Support in Manufacturing", In: *Proceedings of the 4th International Conference on Simulation In Manufacturing*, 1988, pp. 117-126.

108. Siggard, K. and L. Alting (1991): "Methodology for Test and Validation of Shop Floor Control Systems", In: *Flexible Automation and Information Management 1991*, (Eds.: M. M. Ahmad and W. G. Sullivan), CRC Press, Boca Raton, FL, 1991, pp. 952-960.

109. Smith, J. S. (1992): "A Formal Design and Development Methodology for Shop Floor Control in CIM", Ph.D. thesis, The Pennsylvania State University, College Park, 1992, 260 pages.

110. Smith, J. S. and S. B. Joshi (1992): "Object-Oriented Development of Shop Floor Control Systems for Computer Integrated Manufacturing", In: *Proceedings of the International Conference on Object-Oriented Manufacturing Systems*, (Ed.: D. H. Norrie), 1992, pp. 152-157.

111. Sreekanth, U., S. Narayanan, D. A. Bodner, T. Govindraj, C. M. Mitchell and L. F. McGinnis (1993): "A Specification Environment for Configuring a Discrete-Part Manufacturing System Simulation Infrastructure", In: *Proceedings of the 1993 International Conference on Systems, Man, and Cybernetics*, 1993.

112. Sue-Tang, J., G. J. Savage, R. P. Picard and K. Pritchard (1987): "Debugging Programmable Logic Controller Programs by Emulation", *Modeling and Simulation*, Vol. 18, No. 4, 1987, pp. 1217-1221.

113. Thomasama, T. and J. Madsen (1990): "Object Oriented Programming Languages for Developing Simulation-Related Software", In: *Proceedings of the 1990 Winter Simulation Conference*, (Eds.: O. Balci, R. P. Sadowski and R. E. Nance), 1990, pp. 482-485.

114. Thomasama, T. and O. Ulgen (1988): "Hierarchical, Modular Simulation Modeling in Icon-Based Simulation Program Generators for Manufacturing", In: *Proceedings of the 1988 Winter Simulation Conference*, (Eds.: M. Abrams, P. Haigh and J. Comfort), 1988, pp. 254-261.

115. Thompson, M. (1994): "Expanding Simulation Beyond Planning and Design", *Industrial Engineering*, Vol. 26, No. 10 (October), 1994, pp. 64-66.

116. Valette, R., M. Courvoisier, J. M. Bijou and J. Albukerque (1983): "A Petri Net based Programmable Logic Controller", In: *Proceedings of the IFIP Conference on Computer Applications in Production and Engineering*, North Holland, New York, NY, 1983, pp. 103-116.

117. Villarroel, J. L. and P. R. Muro-Medrano (1994): "Using Petri Net Models at the Coordination Level for Manufacturing Systems Control", *Robotics & Computer Integrated Manufacturing*, Vol. 11, No. 1, 1994, pp. 41-50.

118. Voller, L. A. and P. L. Webster (1991): "Emulation is New Diagnostic Tool for Dynamic Testing", *Industrial Engineering*, Vol. 23, No. 11 (December), 1991, p. 14.

119. Waye, D. (1988): "SIMSMART: Dynamic Simulation for Engineering Design, Operator Training and Automated Control of Industrial Processes", *Advances in Instrumentation and Control: Proceedings of the ISA 1988 International Conference and Exhibit*, 1988, pp. 83-93.

120. Weil, M. (1993): "Harnischfeger Takes Control with Emulation Modeling", *Managing Automation*, 1993.

121. Yim, D. and T. A. Barta (1994): "A Petri Net-Based Simulation tool for the Design and Analysis of Flexible Manufacturing Systems", *Journal of Manufacturing Systems*, Vol. 13, No. 4, 1994, pp. 251-261.

122. Zeigler, B. P. (1985): "System-Theoretic Representation of Simulation Models", *IIE Transactions*, Vol. 16, No. 1, 1985, pp. 19-34.

VITA

Malay Ashvin Dalal was born on Oct. 20. 1965 in Bombay, India. He received his B.Engg. degree in Industrial and Production Engineering from M.S.R. Institute of Technology, Bangalore University, India; in 1988. Shortly thereafter, he enrolled at Virginia Polytechnic Institute and State University and received his M.S. and Ph.D. degrees in Industrial and Systems Engineering in 1991 and 1996 respectively. His teaching experience includes Laboratory Instructor for the Industrial Automation course, Instructor in SQC for the NSF Young Scholars Program. While pursuing his Ph.D., he gained valuable consulting and research experience on a process control experimentation project, spread over 3 years, at the Chemical Products Division of Sandvik Rock Tools in Bristol VA. The project included developing software for integrated PC-based automated acquisition and analysis of process and quality data. His major research interests include simulation--object-oriented and conventional, intelligent production control, artificial intelligence and machine learning. He has been a member of Alpha Pi Mu, the Industrial Engineering honor society since 1991.

Following the successful defense of his dissertation, Dr. Dalal joined Knowledge Based Systems, Inc. (KBSI) in College Station, TX. As Research Scientist, his responsibilities include research and development of software tools for knowledge-based "intelligent" process modeling and simulation, and industrial consulting.

He can always be contacted through the following address: C-11 Elco Arcade, Hill Road, Bandra, Bombay 400050. India.