Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

A SIMULATION-BASED CONTROLLER BUILDER FOR FLEXIBLE MANUFACTURING SYSTEMS

Fernando G. Gonzalez

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
1406 W. Green Street
Urbana, Illinois 61801 USA

ABSTRACT

This paper discusses a new simulation and control prototype software package that can be employed as a controller for a simple FMS. In developing this software, we attempted to achieve a compromise between the programming flexibility of a general-purpose language like C++ and the convenience of a simulation language like SIMAN® (see Pegden et al. [1995]). In particular, we desire to provide standard programming features within the model such as recursive function calls while providing a single model which can be used to simulate and control a real-world system. To test the efficacy of the developed simulation tool, we developed a simulation model which can control a physical emulator of an FMS which has been constructed in our research laboratory.

1. INTRODUCTION

When writing a simulation model, one can either employ a dedicated simulation language like SIMAN or a general-purpose programming language like C++. The simulation language has the advantage of being easy to use which usually hastens model development. Consider the SIMAN® simulation language. It provides a set of basic modeling elements which can quickly be assembled to build your model. It also provides the essential software elements that are needed to execute the model and to perform the requisite statistical analyses. If one elects to program a simulation model using a general-purpose language, then one must also provide the software needed to execute these auxiliary functions.

However, simulation language like SIMAN have their limitations. SIMAN is not a general purpose programming language like C++, and therefore, cannot provide full modeling flexibility. One is limited to programming within the constraints of the set imposed by the provided modeling elements. For many modeling situations, the provided set of modeling elements is sufficient, and using a simulation language like SIMAN is an optimal choice over developing the simulation model in a language like C++. However, to model more complex systems like an FMS, the set of modeling elements provided by most simulation languages impose severe modeling constraints, particularly when one attempts to assess the impact of the control architecture upon the system. Using a general programming language becomes essential. (Davis et al. [1993] discussed in detail the numerous restrictions that entity flow-based simulation languages impose upon the modeling of an FMS. Mize et al. [1992] further discuss the inaccuracies that ensue from the use of current simulation languages to model an FMS.) Although SIMAN and other simulation languages do permit C subprograms to be included within the model, there is only so much that one can accomplish with these patches. Furthermore, these patches invariably make the model more difficult to verify.

This capability to control an FMS using simulation has already been demonstrated in the literature. Smith et. al. [1994] presented their RapidSIM effort where they enhanced SIMAN with additional constructs to make it possible to implement the physical control of equipment. They added a TASK block and modified the DELAY, ROUTE, MOVE and TRANSPORT blocks. The modeling approach discussed in this paper has some similarities to the approach discussed by Smith et al. It too employs a basic set of SIMAN-like modeling elements and introduces new blocks to manage the modeled system. The presented approach also expands the general modeling flexibility of the language to permit standard features such as recursive function calls to be employed in the modeling. This added flexibility is essential when modeling complex systems such as an FMS.

In this paper a set of objects are developed to provide the minimum necessary modeling elements that one must use in every simulation and controller regardless of the methodology being used. Care has been taken to ensure that the software does not impose on the methodology the designer may have. That is, whether one chooses to write the controller using an entity-flow based method or a message-flow based method, this software gives the flexibility to implement either one.

For this study, the modeled and controlled system will be a physical emulator for an FMS which we have constructed. The study will also demonstrate some new features which could be provided within simulation languages to increase their modeling flexibility.

For discussion purposes in this paper, an event represents the action taken between simulated or actual physical events. For example, a delay event is the action taken when a delay is finished, and a QUESEIZE event is the action taken when the resources needed for executing the event are allocated. Please note also that time does not advance while an event is being executed.

2. THE EXPLORED MODELING FRAMEWORK

The goal for this research effort was to develop a simulation approach using C++ which could provide the modeling convenience of a simulation language like SIMAN while providing the additional capability needed to control a real-world system. This paper concentrates first on defining a set of basic modeling elements using C++ and then adapts these blocks to model the controller of an FMS. Each included modeling element is represented as an object in C++. As with most simulation approaches, this approach is also event driven, and an execution object manages the simulation by the sequential processing of events. An event list is maintained where the scheduled events are stored in chronological order based upon the time that they are scheduled to occur.

In the development of this modeling environment, only a minimal set of modeling elements have been included. The decision of which elements were to be provided was based upon whether the object required an event to be scheduled on the event list or requires the allocation of resources. Since the proposed modeling environment will provide an increased capacity to include C++ programming elements, we decided to provide for the other modeling requirements, like the TALLY block, by defining a library of C++ functions. Some of the more trivial modeling elements like the COUNT, BRANCH and ASSIGN are not implemented since they can be trivially implemented in C++.

A simulation executive object manages the chronologically-ordered event list, controls the processing of the events and manages the allocation of resources. After each event is processed, the executive object pops the event with the smallest event time off of the event list and then invokes the proper function to manage its execution. Given this streamlined approach, there are two basic types of events that occur: QUESEIZE, where the event occurs as a result of resources becoming available and DELAY where the event occurs when a delay is finished. There are also other types of events that provide an expanded modeling flexibility for this simulation environment which are not considered in most simulation tools including the PENDING, GOSUB, and RETURNTO elements. These will be discussed later.

Each modeling element object is instantiated from an object subclass for the element type. A feature of all of these element subclasses is there ability to store (queue) the pointer of the entity which enters the element. Within a system's model, a given modeling element may be used several times. For example, a particular model may have several DELAY elements. Each use of an element within a model will assign a unique Label which will also be attached to every event that is instantiated using that element. For example, the seventh QUESEIZE employed in the model would be assigned the label QUESEIZE7. A DELAY32 label would be attached to the thirty-second DELAY element included in the model. Whenever an entity enters a given modeling element an event object of the particular type may be instantiated, depending on whether the element is of the type that needs to schedule an event. Among the event object's attributes are its event label (e.g. DELAY32) and the scheduled event time. For example, an event may be DELAY37, 3:15. This means that a DELAY37 event is to occur at 3:15. This simulates the completion of the process that is being modeled with the DELAY37 element.

To enhance the capability of the user to include C++ functions, either from the modeling tools library or a special user-provided function, the modeling tool makes a significant deviation from the approach employed in most simulation. Usually, the list of modeling elements included within the model specification are assumed to be sequentially attached to each other unless otherwise specified via modifiers such as DETACH or NEXT(Label). That is, the entity will usually flow from one model element to the next model element in the model's statement. Under our framework, this assumption is not made. Instead, the modeler can insert a set of C++ statements in-line between any two modeling elements. The C++ code can then perform any function upon the entity that is required, including tally statistics from its attribute list or return resources allocated to the entity. Note that neither of these actions has consequences upon the evolution of time. The modeling element that follows the last statement of the inserted C++ code will then receive the entity after the intervening C++ statement have been executed upon it. An event ends when the next modeling element is encountered. At this point, the entity is stored inside of the modeling element's object until the event associated with it occurs. When this event occurs, the entity is removed from the element's object and processing of the event starts. The following is an example of a DELAY37 event. Notice that the first step in the event is to remove the entity from the DELAY37 element and the last step is to store the entity inside of the next element's object.

```
case DELAY37:
    ent = Motor_d37.remove();
                    .
                    .
                    .
    get_machine_q42.insert(ent);
    break;
```

Note: Motor_d37 is the name of the DELAY37 modeling element's object and likewise get_machine_q42 is the name of the QUESEIZE42 element's object.

The following provides a brief outline on how modeling element objects and the simulation executive object are implemented. The DELAY block is an object that consists of a priority queue used to store the position of the entity's pointer, a pointer to a random number generator that can generate random numbers from a prespecified probability distribution, and an event label. The priority among the entities in the queue is based upon the time that the entity is due to be removed from the DELAY block. That is, its order is determined by the entities' remaining delay time. This way the order of the entities in the queue is synchronized with the order of the DELAY events that are scheduled for that particular DELAY block. The random number generator is used to compute the delay time. Each entity entering a delay block is given a random number from this generator and is used as the delay time. When an entity is inserted into a DELAY block, the block generates the delay time, computes the time when the delay will finish, stores the pointer to the entity in its queue using the time it will finish as the priority and finally schedules a delay event for that particular DELAY block in the event list. The element label associated with the modeled element in the model statement is also stored as an attribute within the instantiated entity object. This permits the executive program to know at what location the entity object currently resides in model network.

The QUESEIZE element also contains a queue to store the pointer of the involved entity object and an element label, but no random number generator. When an entity is inserted into the QUESEIZE block, it simply puts the entity into the rear of the queue. No event is scheduled since the entity is waiting for resources to become available. After the execution of every event, the executive object checks each QUESEIZE element to see if any waiting entity needs the available resources to continue. If so, a QUESEIZE event is scheduled to occur immediately. That is, it schedules it into the event list with the current time as the time to occur.

The CREATE element is similar to the DELAY but functions automatically without an entity flowing into the element. When a create event occurs, it automatically creates a new create event object and schedules it on to the event list. Since a new create event is scheduled every time one occurs, there is always a create event scheduled to occur in the future. This models the constant flow of entities entering a system. Within the C++ library, there is also a function which can create a duplicate entity at any point while a given entity is flowing through the modeled network.

The resource object is used to manage the resources. Only the executive object can interact with this object. This private object is invoked by the QUESEIZE element when it either checks or seizes some resources or by the Release function when it releases some resources.

The executive object starts its cycle by processing the event object from the event list with the minimum scheduled event time. It begins the processing of each event object by removing it from the event list. The simulation time is then set to the event's scheduled event time. After the model executes the event, it is possible that resources may have been released, so the executive object checks each QUESEIZE element to see if any entity can now allocate the required resources. If so the corresponding QUESEIZE event is scheduled to occur at the current simulation time and is placed in the front of the event list. After the event object is processed, it is destructed, and the next event is removed form the event list thus starting a new cycle.

In Program 1, an example of a simulation using this developed simulation tool is provided. This model simulates a simple M/M/1 queue where the arrival rate is LAMDA and the service rate is MU. This simulation estimates the average queue size.

3. FURTHER MODEL ENHANCEMENTS

In addition to implementing the basic set of SIMAN like blocks, additional modeling elements not found in most simulation language have been added to enhance the software's capacity to provide expanded control over the flow of entities and to permit the model to function to as a controller of the real system.

3.1 Flow Control Enhancements

To provide enhanced control over the flow of entities, several new features have been incorporated. Each modeling element object contains three public functions. The first function is the Insert function. This function permits the modeler through the use of C++ code to insert an entity into any other modeled element object within the model. The inserted entity object would then begin to flow from its new location. The second function is the Remove function. It permits the modeler to remove an entity object from the modeled element object where it currently resides. The modeler can then Insert the element elsewhere or the entity will continue to

```
#define      SERVER   0

int          total_ent;
ResourceStruct   resources[50] = { "Server", 1};
RandomExp    arrival(1/LAMBDA),
             service_time(1/MU);
QueSeize     server_q1("Server Que", SERVER,1);
Release      server_r1;
Delay        service_d1( &service_time);
Create       arrivals_a1(&arrival);
Tally        TIS_t1("time in sys.");

void
perform(int  EventNumber)
   {
   Entity *ent;

   switch (EventNumber)
      {
      case CREATE1:
           if (get_TNOW() > 150)
                break;
           ent = arrivals_a1.create();
           ent->attr[0] = get_TNOW();
           server_q1.insert(ent);
           break;

      case QUESEIZE1:
           ent = server_q1.remove();
           service_d1.insert(ent);
           break;

      case DELAY1:
           ent = service_d1.remove();
           server_r1.release();
           TIS_t1.tally( get_TNOW() -
                ent->attr[0]);
           total_ent++;
           delete ent;
           break;

      default:
           cout << "Oops. Did a boo boo\n";
           break;
      }
   }

void
main()
   {
   InitializeResources(resources);
   total_ent = 0;
   sim( arrival.get() );
   PrintResults();
   }
```

Program 1: Sample Model for M/M/1 Queue

flow from the position where the Remove function was invoked. A third public function, Display, provides the pointers of the entity objects that currently reside at a given modeled element object. It returns a null pointer if no entity objects currently reside at that element.

There are times in a simulation that a section of the model needs to be repeated several times. For example, in our emulator of an FMS, the submodel that models the movement of the automated guided vehicle (AGV) is repeated several times through out the model, i.e. once to take the part to the machine and then once again to pick up the part and so on. Languages like SIMAN provide little support for recursive submodel calls. The submodels simply allow one to organize the model into groups of blocks. These submodels always return entities to a specific location. In order to support recursive submodel calls, two new blocks have been added. The GOSUB and the RETURNTO blocks allow a submodel to be called from anywhere in the model and always returns to the location where it was called, much like the recursive function calls in C or any general programming language.

The GOSUB block is used to call a submodel. An example is gosub(ent,MOVE,7), where ent is the pointer to the entity, MOVE is the name of the submodel and 7 is the return address or the return event. In this example, the entity is passed to the block via the variable ent that points to the entity, the first event in the submodel is called MOVE and the next event to execute when the submodel is done is called RETURN7. The RETURNTO element is the last block in the submodel. It sends the entity back to the return event. An example is returnto(ent), where ent points to the entity. Parameters can be passed to the submodel by using a space in the entity object called param. This is like the space for the attributes but is designated for parameter passing.

The ability to return the entity to the location where it was called from is provided by storing the return address inside of the entity itself. Each entity has a return address stack. Since there may be several entities within the submodel at the same time, each entity must hold there own return address unlike a general programming language that usually has only a single process operating and therefore, uses the system stack to hold the return address.

When GOSUB is called, it stores the entity in a holding area and pushes its return address onto the entities return address stack. It then sets a flag which signals the executive object that an event must be executed before finishing the cycle and provides it with a pointer to the next event, the first event in the submodel. The RETURNTO block works in much the same way, but it pops the return address off of the return address stack in the entity and uses this address to tell the executive object where to go to next. The following is an example of a section of a model where the recursive submodel called MOVE_AGV is called to move the AGV from the machine to the port. It has just finished turning the machine's motor needed to load the AGV with the part. Notice the passing of the parameters through the "param" spots 5 and 6. Param 5 holds the location of the AGV and param 6 hold the destination.

```
case PENDING35:
        ent = turn_motor_p35.remove();
        Motor_r9.release();
        ent->param[5] = PORT;
        ent->param[6] = MACHINE;
        gosub(ent,MOVE_AGV,7);
        break;

case RETURN7:
        ent = get_ent();
            .
            .
            .
case MOVE_AGV:
        ent = get_ent();
        move_from = ent->param[5];
        move_to =ent->param[6];
            .
            .
            .
        returnto(ent);
        break;
```

3.2 System Control Enhancements

The original desire for developing this simulation tool was to permit the simulation model to function as the controller for the system. To accomplish this goal, the controller must send commands to the physical machine to execute a task. A new model element called the PENDING block has been included to allow the model to synchronize itself with the hardware.

The execution of the PENDING event requires a major redefinition of the manner in which the executive object manages the simulation. First, the clock must run in real-time instead of simulated time. That is, instead of extracting the next event from the event list and setting the current time to the time that event is to occur as is done in simulation, the controller waits until that time arrives before it executes the event. So if the event is to occur at 5:00 the controller waits until 5:00 before it executes it as opposed to simply setting the current time to 5:00. The controller looks at the computer system's clock to get the current time. When a PENDING element is active, the simulation object must wait for a feedback signal from the hardware to indicate that the requested task has been completed. Hence the occurrence of the PENDING event cannot be scheduled. While the executive object is waiting for the PENDING event to occur, the real-time clock will continue to advance. The executive object also knows that there are additional scheduled events on the calendar. Since one or more of these events could occur before the PENDING event occurs, it is essential that the simulation executive object continuously compare the real-time clock against the earliest event time for an event object on the event list. Whenever the real-time equals the earliest event time, the associated event is processed.

When a machine is simulated, the DELAY block calculates the time it will be when the task is complete based on its duration and schedules a "task is finished" event for that time. However, when the machines actually exist and the model is to employ control of the system, the controller sends a signal to the machine to perform the task. It then waits until the machine signals that it has completed the task before the "task is finished" event is executed. The PENDING block in control mode does precisely that. However when a PENDING block is executed the event is put into a list of events that are pending for signals from the hardware before it can execute. This list is called the pending event list. So instead of scheduling events onto an event list where each event has an associated time of occurrence, the PENDING block puts events into a pending event list where there is no time associated with the events. In this list, the events simply wait until the hardware signals the controller that the requested task is done. The corresponding event is then pulled off of the list and executed.

Since the model is used for simulation as well as for control, the driver runs in two modes; simulation and control mode. In the simulation mode the clock runs in simulated time and the PENDING block operates in an identical fashion to the DELAY block. In this mode, there is no difference between the DELAY and PENDING block. In the control mode the system clock runs in real-time, and the PENDING block performs as indicated above. So one only sets the mode of operation, and the model either controls the system or simulates it. Note, in control mode one can still use the DELAY block. It always operates as a DELAY block but the delay is measured in real-time, not in simulated time. It is only the PENDING block that switches modes. This is useful when controlling an emulator where some of the machines exists and must be controlled while others are simulated. Also note that one must introduce the code to send commands to the hardware and provide checks so that the commands are not sent in simulation mode. This requirement is easily achieved in C++. The following example shows a section of the model communicating a command to turn the motor by 1 unit after seizing the motor.

```
case QUESEIZE9:
        ent = Motor_q9.remove();
        if (control_mode)
            send_msg("TURN");
        turn_motor_p35.insert(ent);
        break;
case PENDING35:
        ent = turn_motor_p35.remove();
```

4. THE FMS EMULATOR.

To develop an educational and research laboratory for the coordination of discrete-event systems, the Manufacturing Systems Laboratory at the University of Illinois has constructed an FMS emulator (see Davis et al. [1994]). To insure safety and economy, all physical processing has been omitted. We have also constructed physical analogs for the material handling systems (MHSs). The controller for this system is built using this package.

The schematic for the constructed FMS emulator is depicted in Figure 1. Figure 2 provides a photograph of the emulator. The emulator has four Processing Centers, numbered 1 through 4. Each Processing Center (PC) is a cell containing one primary subordinate processing resource (the Unit Process) and a dedicated MHS.

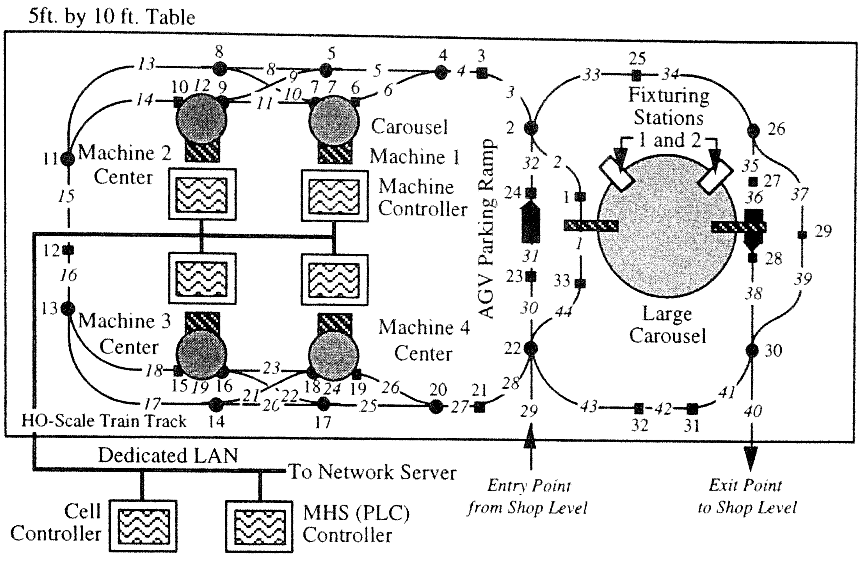

Figure 1: Schematic Layout for FMS Emulator

In Figure 3, we show the constructed PC. Its MHS is represented by a carousel with six electromagnets which

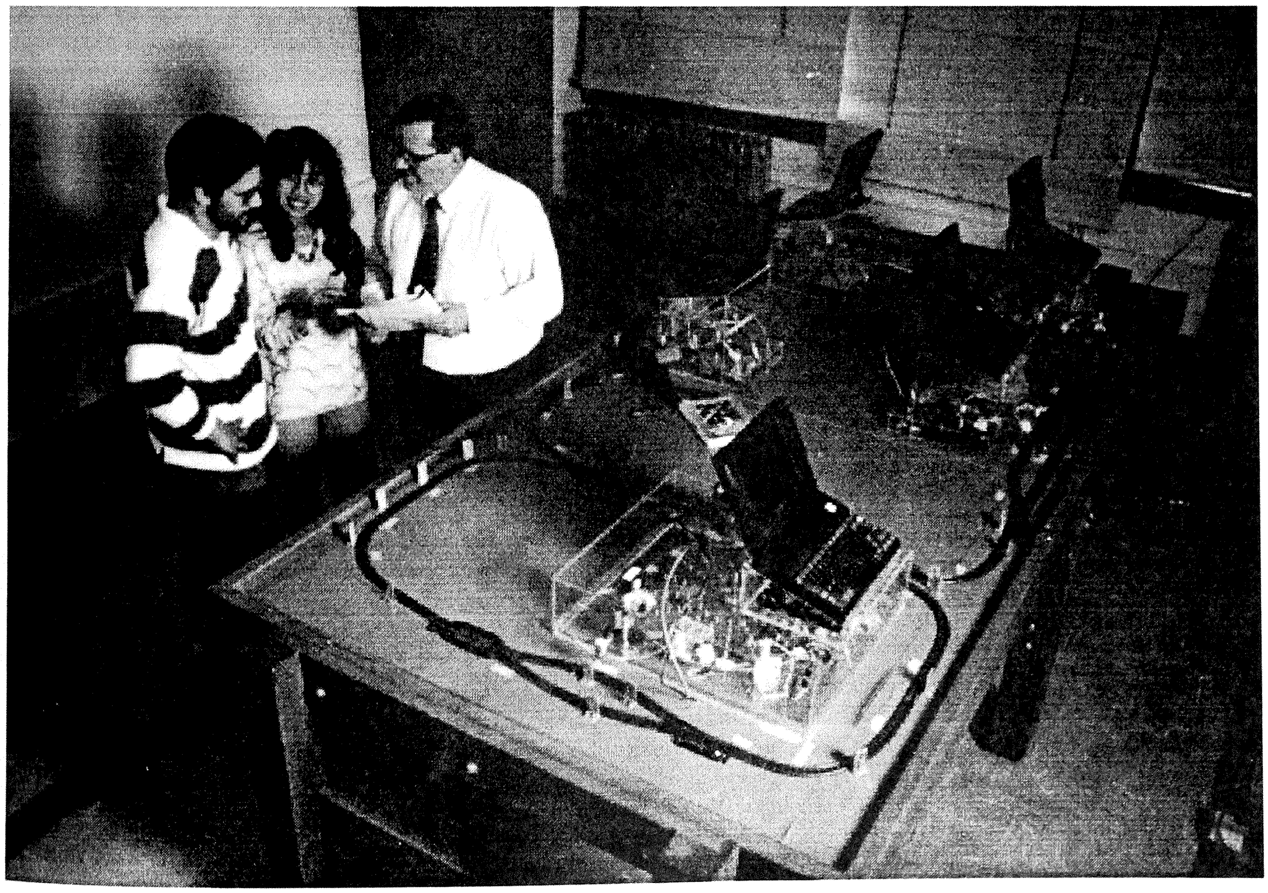

Figure 2.: FMS Emulator under Construction

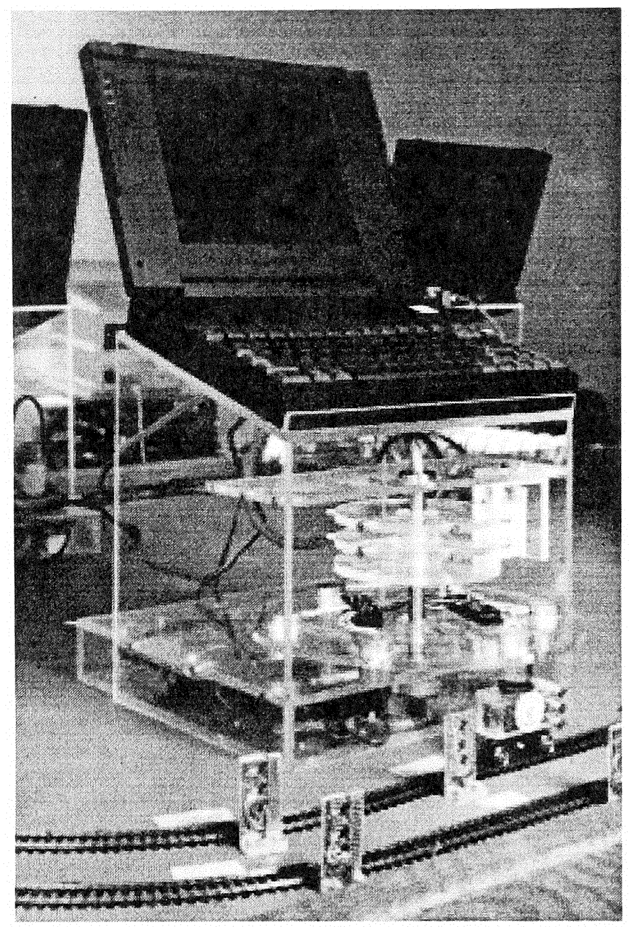

Figure 3: The Emulated Processing Center

hold the jobs (represented by color-coded steel washers) residing in either the input or output queues of the PC. The movement of the carousel is controlled by a dedicated Programmable Logic Controller (PLC). The PC controller is a lap-top UNIX workstation which is placed on a rostrum atop each PC. This controller is connected to the MHS PLC via a dedicated RS 232 link, and its display provides the summary status for the MHS.

The PC Controller summarizes each task as it is being implemented by the subordinate unit process. Future plans provide for the inclusion of another Unix computer at each PC to animate the operation of the unit processes. Software packages such as Deneb Robotics' Virtual NC can provide the desired animation for machining processes. (These packages are very expensive and are not critical to current implementation of the emulator.)

Within the emulated FMS (see Figure 2), another subordinate process is the Fixturing Center (FC) which also represents a cell. The structure and emulation of the FC is very similar to that of a PC. The FC has a dedicated MHS consisting of a primary carousel capable of holding sixteen jobs and two smaller carousels for loading and unloading jobs from the AGVs. The movement of these carousels is controlled by a dedicated PLC. The FC has two fixturing positions which represent the unit processes. A dedicated lap-top UNIX workstation provides the real-time status information for each fixturer. This same workstation also issue control messages to the dedicated PLC for the FC's MHS.

The final subordinate process is the Cell MHS. Automated Guided Vehicles (AGVs) are employed as the MHS and are emulated with a HO-scale electric trains. The train layout is diagrammed in Figure 1 and pictured in Figure 2. In this layout, there are over forty track segments which can be individually powered. Sensory switches are provided on each track segment to detect the presence of an AGV (see Figure 3). A Petri net has been developed to insure that no more than one AGV ever occupies a single track segment at a time.

A dedicated PLC receives directives from the Material Handling controller to move a given AGV from one location to another using the incorporated Petri net logic. The dedicated PLC returns the location of each AGV as it enters each track segment to the Material Handling controller. The Material Handling controller is also responsible for determining which of the pending material handling transfers will be processed and which AGV will be assigned to complete the requested transfer.

The cell controller is implemented by another UNIX workstation. It is connected to each of the subordinate PCs, FC and MHS controllers (and a network server) via an ether network. Various commands and feedback informations flow across this network. The role of the cell Controller is to orchestrate the flow of the entities of all types (jobs as well as supporting resources) among the subordinate processes within the cell.

5. CONCLUSION

Presented here is a simulation and control software that has the following properties:

1. The simulation is written in C++ providing the full modeling flexibility of a general purpose language.

2. The set of basic SIMAN like blocks are available to provide ease in modeling.

3. The package itself is a C++ library of routines and objects. The package is portable and may be run on any platform that has a C++ compiler.

4. This package supports subroutines not just submodels. Subroutines can be called from anywhere in the model

and returns to where they were called. You can even call them recursively. Submodels do not do this.

5. The package can run in "control" mode. In this mode, with the hardware drivers written, your simulation can actually control the system. The simulated-time becomes real-time and the event list includes a pending event list where the events are waiting for signals from the hardware drivers. You can write a controller for your system using SIMAN like blocks.

The system was also successfully tested on the physical FMS emulator. Future research is being done to build an intelligent controller for this emulator using real-time simulation and controller concurrently. That is, the controller will run real-time simulations concurrently in order to provide feed-forward information used for making intelligent control decisions.

REFERENCES

Davis, W. J., D. Setterdahl, J. Macro, V. Izokaitis, and B. Bauman. 1993. Recent Advances in the Modeling, Scheduling and Control of Flexible Automation. *Proc. of the 1993 Winter Simulation Conference*, eds. G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles, 143-155, The Society for Computer Simulation, San Diego, CA.

Davis, W. J., B. Bauman, J. Macro and D. Setterdahl. 1994. Constructing an Emulator for Research and Education in the Control of Flexible Automation. *Proceedings of the ORSA Technical Section on Manufacturing Management Conf.*, eds. J. Buzacott and C.A. Yano, 151-157.

Mize J. H., H. C. Bhuskute, and M. Kamath. 1992. Modeling of Integrated Manufacturing Systems. *IIE Transactions* , 24(3):14-26.

Pegden, C.D., R.E. Shannon and R. P. Sadowski. 1995. Introduction to Simulation using SIMAN (second edition). McGraw-Hill, New York.

Smith, J. S., R. A. Wysk, D. T. Sturrock, S. E. Ramaswamy, G. D. Smith and S. B. Joshi. 1994. Discrete Event Simulation for Shop Floor Control. *Proc. of the 1994 Winter Simulation Conference*, Eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A.F. Seila, 962-969.

AUTHOR BIOGRAPHY

MR. FERNANDO G. GONZALEZ is doctoral student in Electrical and Computer Engineering at the University of Illinois. He received his B.S. in computer science and M.S. in electrical engineering at Florida International University. His current research addresses the real-time management of discrete-event systems. He is a recipient of an NSF Support for Under Represented Groups in Engineering (SURGE) Fellowship and a GTE Minority Fellowship..