Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

ESTABLISHING AN OBJECT-ORIENTED METHODOLOGY FOR THE SIMULATION AND CONTROL OF INTEGRATED MANUFACTURING SYSTEMS

Joseph G. Macro
Wayne J. Davis

Department of General Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, U.S.A.

Duane L. Setterdahl

Cap Gemini America
5 Westbrook Corporate Center, Suite 600
West Chester, Illinois 60154, USA

ABSTRACT

The current simulation tools which are based upon stochastic queuing network architectures have proven to be inadequate for modeling modern Flexible Manufacturing Systems (FMSs). A Hierarchical, Object-Oriented Programmable Logic Simulator (HOOPLS) is proposed: to model the interaction among the controllers which coordinate the production in an FMS; to address the flow of all entity types including jobs, fixturing, tooling, and information; and to provide direct consideration of the detailed processing plans which govern how resources are employed to manufacture a given item within the FMS.

1 INTRODUCTION

Flexible Manufacturing Systems (FMSs) have evolved from several technological advancements including computer software and hardware, communication networks, robotics and numerically controlled devices. FMSs are being adopted throughout the world to facilitate the manufacture of a wider mix of products and to permit economic production of smaller batch sizes. In a recent study, 75 per cent of the medium- to large-size firms (in France, Italy, Japan, the United Kingdom, the United States and West Germany) reported that they will have installed FMSs by the year 2000 (Mansfield 1993). Although there is a large acceptance of these FMSs, there is a nearly universal complaint among manufacturers that these systems have not achieved the anticipated throughput and productivity.

One source of this discrepancy is that most FMS control architectures evolve as a patchwork of controllers. Manufacturers who do not possess the in-house expertise to design an FMS must seek assistance from outside consultants or system integrators. System integrators, in turn, work with major suppliers of processing equipment who then assemble other manufacturers to provide the essential supporting systems. For example, the vendor of the material handling system (MHS) provides its system with the necessary controller(s) for operation. Since most controllers are proprietary, little or no documentation is provided. A bottom-up (patchwork) design evolves through this procurement procedure. At no point during the process is a complete set of specifications for the individual controllers and their proposed interactions developed. Too often, the first assessment of system integration occurs when the FMS is brought into operation. Then, it is discovered that the integration is incomplete, as there is a propensity for deadlock as well as overlapping (often inconsistent) functionality among the controllers. Given the proprietary nature of the included controllers' logic and the continued absence of simulation tools to model the controller interactions, the task of modifying the controller hierarchy is virtually impossible to address. The modeler typically cannot eliminate the problem(s), and can only document their consequences.

The electronic circuit board FMS developed for the Department of Defense's Rapid Access to Manufactured Parts (RAMP) program clearly demonstrates the bottom-up paradigm. For this FMS, which is now housed at the Naval Air Warfare Center in Indianapolis, over one million lines of computer code were generated to integrate the commercial software packages and controllers for the various processing and material handling equipment. There is a significant overlap in the functionalities for the included commercial software. (Three commercial schedulers are included.) In some cases, the vendors for the controllers have included password protection to protect their proprietary software code, making it impossible to document the logic employed by the controllers. To date, no organization has been able to develop a detailed simulation to project the performance of this FMS, nor has the FMS been brought into full-scale production.

To insure a fully-integrated manufacturing system, a top-down design approach, including detailed specifications of the controllers based upon the desired system integration, is essential. Detailed simulation studies must verify that

these specifications provide the desired coordination among the subsystems by eliminating deadlocks and redundant functionalities. The verified specifications must then be imposed upon each subcontractor to insure that the desired performance will be achieved. Currently, the essential design tools to implement a top-down design do not exist. That is, it is infeasible to accurately simulate the performance of a designed control hierarchy using conventional simulation tools.

Although simulation is the most accepted technology for analyzing FMSs, there are two fundamental flaws limiting its function as a design tool. First, current simulation tools model events associated with entity flows in the system rather than the controller interactions that govern these flows. Second, current simulation tools do not allow for the inclusion of detailed process plans which specify the coordination that must exist among the flow of different entity types during the manufacturing process. (An expanded discussion of these concerns is given in Davis, Setterdahl, Macro, Izokaitis and Bauman (1993)).

2 HIERARCHICAL OBJECT-ORIENTED PROGRAMMABLE LOGIC SIMULATOR

2.1 Message-Based Simulation Modeling

To remedy the inherent deficiencies in SQN-based simulation languages, we are developing a new simulation methodology, termed Hierarchical Object-Oriented Programmable Logic Simulator (HOOPLS). HOOPLS employs the principles of object-oriented programming. Each modeled object represents either a particular control node or resource within the system being modeled. To provide a computer-assisted software engineering environment, a graphical user interface is developed which allows non-experts to construct detailed simulation models using the adopted hierarchical, object-oriented paradigm.

One feature of object-oriented programming is that the objects process information based upon input messages received from other objects and can respond appropriately by sending messages to other objects. HOOPLS uses this object-oriented message passing to mimic the flow of actual control messages passed among the controllers of the modeled manufacturing or assembly system. Our adopted approach to modeling controller interactions is the single most important characteristic of the HOOPLS methodology and separates it from previous object-oriented simulation approaches. Whereas previous simulation approaches have focussed upon modeling discrete-events only, HOOPLS views these events as being consequences of controller interactions.

Using the object-orientation, each programmed object (controller) can be individually tested to insure the code is functioning properly, a feature which is essential in the development of large-scale simulation models. The programmed objects are also reusable, not only within the current simulation, but any future simulations employing similar equipment. In the future, publicly available software libraries of objects (controllers) may be created where modelers can share the code for control objects that they have constructed. Since the manufacturer is responsible in most cases for designing the controller for its equipment, manufacturers could also develop the control object to model its equipment and place it in the library. This library would significantly reduce the effort to develop a simulation model since standardized control objects would exist for each piece of equipment and the major modeling task would be to define and model the control hierarchy that integrates the equipment.

2.2 The Control Architecture

Currently, there is no theoretical or conceptual basis to guide the modeling of control architectures. It is usually left to the modeler to choose the control structure, which is usually centralized, hierarchical, heterarchical, or some hybrid of the three. In the recent literature, there have been various discussions on the appropriateness of these different control architectures in a manufacturing environment (Dilts et al. 1991 and Matsuda and Inoue 1991). The use of a decentralized hierarchy has been established as an effective method of decomposing the complex decision-making problem, particularly in a manufacturing environment where real-time decision making, modularity, and reliability are of great concern. We model the distributed coordination (integrated scheduling and control) of a complex hierarchical discrete-event system using a Recursive Object-Oriented Coordination Hierarchy (ROOCH). We have explicitly designed the ROOCH, an essential component of HOOPLS, to permit a generic coordinate node to be recursively employed at each hierarchical level.

In the real-time decision-making environment, the need for scheduling and control is concurrent. That is, the controlling elements must continuously monitor feedback information from the subordinate subsystems and employ the current control policy to generate the essential coordinating inputs which will direct the future subsystem response. The scheduling decision, on the other hand, is dependent upon the current state of the subordinate subsystems which are influenced by the coordinating inputs. The output of the decision is the control policy which will be implemented to generate the essential coordinating inputs. What evolves is a continuous interaction between scheduling and control which can only be addressed through coordination of both.

2.2.1 The Basic Fractal Unit

The Basic Fractal Unit, introduced by Tirpak et al. (1992), provides the specifications for the fundamental Coordinate Node (object) within the ROOCH. The fractal unit was designed to model the intricacies of a controller, and is based on the following premise: *There is a basic conceptual unit that incorporates a set of functional attributes which must be addressed at every level of a coordination hierarchy.* We have adapted this premise to our purpose of controlling a modern integrated manufacturing system. As shown in Figure 1, the fractal unit has a set of input/output components defined as ports, queues, and inhibit flags. Each unit owns an input queue for incoming entities from its supervisor (next hierarchical unit 'up'), with an inhibit flag to prevent their entry if necessary.

Each unit also owns an output port, which is simply a gateway to its supervisory unit. (Note that the output queue and the output inhibit flag for the unit belong to the supervisor of the unit, and not the unit itself.) Returning to Figure 1, when an entity is placed into the input queue for a given fractal unit, that unit assumes control for the arriving entity. Whenever the fractal unit finishes its assigned tasks for a job or resource entity, the entity is then placed in the output queue. At this point, the fractal unit has no further responsibility for determining the disposition of the entity and the control of the entity is returned to the supervisor. Within each fractal unit there can exist several subordinate units (next hierarchical level 'down'). The modeled fractal unit owns the input port to each of its subordinate units, as well as an output queue and output inhibit flag for each subordinate. Whenever a unit places an entity into the input queue of one of its subordinates it also delegates direct control of that entity to the subordinate.

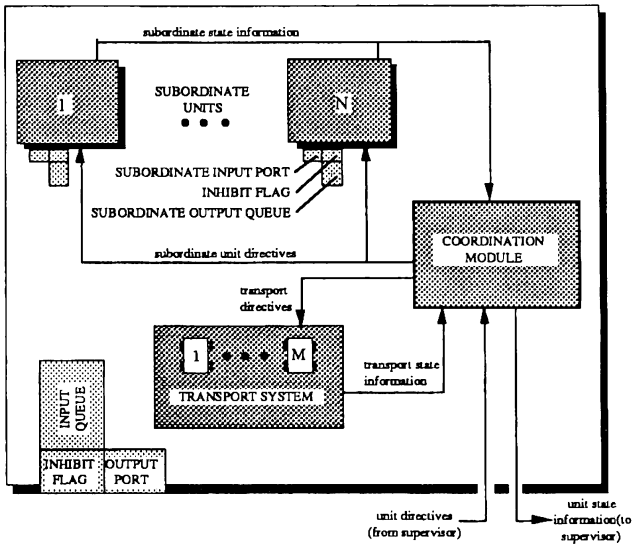

Figure 1. The Basic Fractal Unit

The control of the entity remains with the subordinate until the subordinate places the entity in its output port.

The details of each subordinate unit are hidden from the unit except for strategic state information. Thus, the coordination hierarchy is built by placing layers of coordinators inside other coordinators. The highest level coordinator can have several subordinate units inside it, with each of these subordinates having subordinate units inside (etc.), until the bottom hierarchical level is reached. The control hierarchy can also be expanded upwards indefinitely. For example, in a manufacturing setting one can start by modeling a machine center and its subordinate units, a part loader, a carousel, and a machine spindle. That machine center can then be part of a cell, which in turn can be part of a shop. There can be several shops in a single factory, and several factories comprising each division in a company. Finally, a large corporation may be comprised of several divisions. The basic fractal unit described above allows the modeler to develop simulations at any level, whether the focus lies on the factory floor, or on long-range planning among many geographic points. That is, we have developed a common framework to view a controlled subsystem at any hierarchical level.

To facilitate the flow of entities through layers of the coordination hierarchy each unit contains a transport system. The transport systems are themselves coordinators, but they are not viewed as fractal units since these coordinators can only transport entities from place to place, (i.e. - no physical processing takes place). That is, these can only modify the location and not the state of an entity. Each unit can have several types of transport systems depending on the type of entities that flow through the fractal units. For example, a machine center could have a tool exchanger and tool magazine to store, load and unload the machine spindle, as well as a pallet changer and carousel to store, load and unload parts from the machine. Both the tool handling system and the pallet handling system are subordinate transport systems for the machine center. Each fractal unit must contain at least one transport system to implement the flow of resources to its subordinate unit(s).

Fractal units also contain a coordination module which enables them to schedule and implement assigned tasks upon entities residing within their control. The type of coordination module used does not matter, but it must perform four basic functions: to monitor, assess, analyze and execute (Davis 1992). The coordinator must first be able to monitor its subordinate systems, synthesize this information, and report it back to its supervisor. When a decision is to be made, the coordinator must be able to assess the different control options, analyze each one to determine the appropriate action, and then execute the desired control option. These functions can be supported with the technologies of Petri-nets, Knowledge Base Systems (KBS), or real-time simulation analyses, to list a few.

In summary, the fractal unit provides a single conceptual framework to model the coordinate node at any hierarchical level. The hierarchy is constructed by adding subordinate layers of fractal units inside each fractal unit. The benefits of using fractal units are: (1) models can be constructed at various hierarchical levels and abstraction; (2) the flows of any number of entity types can be considered without modification of the fractal unit; (3) a standard coordination protocol can be developed to depict the chain of commands arising when entities cross fractal unit boundaries; and (4) the control of the included subsystem results from the interaction among the included coordinators and not with the flow of the entities.

2.2.2 Recursive Object-Oriented Coordination Hierarchy

Using the fractal unit described above, the ROOCH was developed as an effective tool for modeling FMSs. The primary motive for developing the ROOCH was to further detail the modeling specifications proposed in the Basic Fractal Unit, and to provide a framework for constructing these models. Each manufacturing system is modeled by its unique ROOCH, which also serves as its coordination hierarchy.

It has been observed that FMSs are composed of equipment which fall into one of three classes of hierarchical elements, or nodes: Transport Nodes, Processing Nodes, and Coordinating Nodes. Each node has prescribed fundamental capabilities based upon its class:

- **Transport Nodes** are responsible for the transportation of Primary Resources (parts, assemblies, etc.) and Supporting Resources (tools, fixtures, NC code, etc.) using Transport Resources (AGVs, conveyors, networks, carousels, etc.).

- **Processing Nodes** represent points (locations) in the manufacturing system where tasks are performed on Primary Resources using various Supporting Resources, as specified by the processing plan.

- **Coordinating Nodes** are responsible for allocating and coordinating the flow of both Primary and Supporting Resources among subordinate nodes toward the completion of assigned processing tasks.

The Coordinating Node is the fractal unit. Every Coordinating Node must have at least one Transport Node to implement the transfer of Primary and Supporting Resources among its other subordinate nodes and, therefore, must also have at least one or more subordinate nodes. Processing Nodes and Transport Nodes cannot have subordinate nodes, which means these always represent a terminal node of a particular ROOCH. In terms of control, this is the hierarchical level at which device-specific interfaces to the controlled unit processors (machines) exist.

The ROOCH defines the role of three node types in the context of a manufacturing system coordination hierarchy, and also defines how different coordination modules may be implemented. It then provides specific rules for the assembly of each of the node types into a decision-making and control hierarchy for the considered system. By formalizing this architecture, it is possible to build coordination hierarchies using graphical objects and linking them together according to the rules of the hierarchy. Our adopted object-oriented view for each node also implies that communication will be through a set of formalized control messages via some message passing facility native to the programming language and network platform chosen. The ROOCH determines the type of control messages each node type will be able to process and the control messages that it will transmit, which will be further detailed below. And finally, the resulting ROOCH explicitly defines the set of coordinators in the adjacent hierarchical levels with which each coordinated object contained in the ROOCH can communicate.

2.3 Control Messages

Given the ROOCH, the communication between coordinators required for production and assembly can be specified. Communication consists of a set of messages that each type of coordinate node sends and receives. Supervisory nodes have no knowledge of the specific logic details of any subordinate tasks, but only know when a task has been accomplished by receiving a response message. Each message initiated between two node types has a specific response, which defines the protocol, or "conversation", between coordinators. In other words, a Coordinating Node would never expect a *taskExecuted()* message from a Transport Node in response to a *pickupItem()* message.

In Figure 2, we define three distinct message types: Resource Messages, Action Messages, and Status Messages. Resource Messages aid in coordinating the flow of resources and making accurate real-time decisions. The resource messages alert a coordinate node that a particular resource has been assigned to it, and this node will be given control of that resource in the future. An action message is an initiating message; a coordinate node determines that a physical action is necessary and an action message is sent. Status messages are responding (feedback) messages and are sent when a subordinate has completed a required task or when a subordinate needs to activate or deactivate its inhibit flag.

The three types of messages play an important role in the communication protocol which defines the hierarchical interactions of the coordinators. A conversation between

coordinators is defined as a set of messages sent between two coordinators which define a specific action. A conversation must include one and only one action message and the appropriate response message. It may also include any number of resource and status messages. Currently, most conversations include only a few messages. However, when the system is developed to the point of real-time scheduling and control, these conversations could be a long dialogue of resource messages and status messages which aid in the determination of the appropriate scheduling and control decisions.

The action message class has been developed, and we believe that although the list of action messages does not contain every possible message, it is a sufficient for most FMSs. The class of action messages can be broken down into three subclasses based on whether the Coordinate node (supervisor coordinator) is communicating with: a transport node, a process node, or another coordinate node. The Coordinating Node communicates with its Transport Node using two basic commands *deliverItem()* and *pickupItem()*. The *deliverItem()* command is used when the transport system already has physical control of an entity. An entity can be delivered to either the Coordinating Node's supervisor or to one of its subordinates. The *pickupItem()* message is similar to the *deliverItem()* message, except the *pickupItem()* message is used when the Coordinating Node does not physically have control of the entity it needs to transport. When a Transport Node has finished transporting a resource to its destination, it issues the appropriate return message: either *itemDelivered()* or *itemPickedup()*. (Note that all the control logic required to move a transporter from one location to another resides within the Transport Node and is appropriately hidden from the Coordinating Node.)

The communication between a Coordinating Node and its subordinate Processing Node is straightforward. Once the Coordinating Node has assembled the essential resources needed to manufacture a part, an *executeTask()* message is sent to the Processing Node. When processing

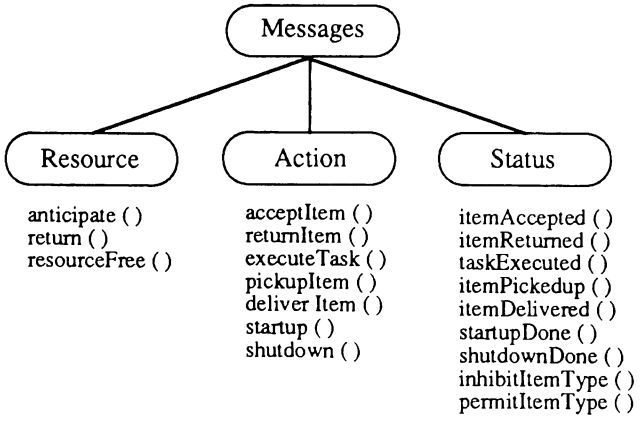

Figure 2. Control Messages

is completed, the Processing Node returns the *taskExecuted()* command. At this point the Coordinating Node can either unload the job from the processor, or load a new set of supporting resources onto the processor to complete another task on the same job.

When the control of an entity is passed from a supervisory Coordinating Node to a subordinate Coordinate Node, either an *acceptItem()* or *returnItem()* message is sent. These messages signal to the subordinate Coordinating Node that the supervisor's Transport Node is ready to be unloaded or loaded. In addition, the control of the supervisor's transport resource is also passed to the subordinate Coordinating Node. This ensures that the transport resource cannot be used for another task while (un)loading operation is being performed. Once the subordinate Coordinating Node has finished its assigned task, the appropriate response message, *itemAccepted()* or *itemReturned()*, is sent to the supervisory Coordinating Node.

We have defined three resource messages that can be sent by a coordinate node only. These messages are *anticipate()*, *return()*, and *resourceFree()*. The *anticipate()* command is sent by a supervisor to alert its subordinate that a particular resource has been assigned to it. The subordinate does not take control of that part until it is physically loaded into it. However, the subordinate coordinators can make decisions based on the fact that it will receive this resource in the near future. The *resourceFree()* command is sent when a subordinate coordinate node no longer needs a resource. The *return()* command is issued when a supervisory coordinate node determines that a resource which is currently being controlled by the subordinate node receiving the message is needed at another location.

These simple messages are a subset of all of the messages that are needed to ensure a smooth communication between coordinators during production or assembly. Developing a standard set of messages with a constant functionality is crucial to implementing a generic coordination hierarchy. We believe that the current set of action messages is sufficient to simulate and control an integrated manufacturing system. However, we concede that there are endless status and resource messages that can be passed among coordinators depending on the level of coordinators' interaction that a system requires. At the lowest level of the hierarchies (the Transport Nodes and Processing Nodes), the internal control messages cannot be standardized due to the limitless possibilities of proprietary material handling and processing devices. These messages must be determined based solely on the system that is being modeled. However, once a set of control messages is determined for a given MHS, these messages can be re-used every time that transport system is employed, due to the object-oriented nature of HOOPLS.

2.4 The Generic Process Plan Representation

To assess both the product and process flexibility for a given FMS, simulation tools must have the ability to directly access detailed processing information associated with each alternative processing sequence for each part to be manufactured in the FMS. Thus, HOOPLS adopts the structure of a generic, object-oriented process representation that supports process flexibility by facilitating the specification of alternative production paths for each part type being produced.

A multi-layered process plan format has been developed by embedding the process flow information inside a product tree representation. The product tree defines all of the component parts and subassemblies that are needed to manufacture or assemble a finished product. Each node in the product tree represents either the finished product, a subassembly, a machined part or a purchased part. Embedded in each product node is a process flow diagram which delineates the alternative production paths needed to produce the associated part.

2.4.1 The Product Tree

The first layer of the graphical process plan representation uses a manufacturing-oriented explosion to decompose the product into subassemblies, machine parts and purchased parts. From this explosion, a manufacturing product tree can be developed (Figure 3). The root of the tree is the finished product while the leaf nodes are purchased parts. The children of a given node are the component parts and subassemblies needed to create that node. These component parts are iteratively consumed in the production of the parent node until the root of the tree is reached and the finished product is complete.

We adopted this product tree representation because of its ability to represent a finished product at various stages of production. All components and subassemblies of the final product are tracked individually until they are consumed. This allows manufactured parts and assembled parts to be modeled similarly, and all product nodes to be represented by the same graphic primitive (Figure 4). Each product node contains scheduling information, a resource list, a component list, and an associated process flow diagram. The scheduling information consists of the lot size, lead time, current inventory level, demand for this item, and scheduled order receipts. Although the data contained in the product node are the inputs required to implement a traditional MRP system, these same inputs may be used in conjunction with various production planning methods including JIT. A resource list details all the possible resources required for completion of a product node. Since multiple process flows may exist, all of the resources may not be necessary for production. The component list specifies the number and type of child product nodes required for manufacture or assembly. The process flow diagram details the alternative processing tasks for completion of that product node.

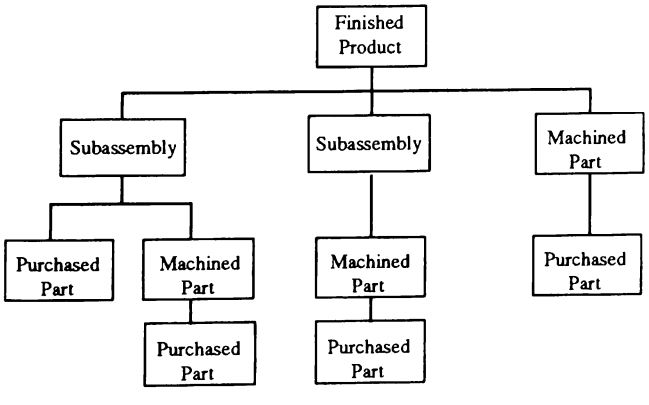

Figure 3. Product Tree Representation

2.4.2 The Process Flow Representation

Our Manufacturing Systems Laboratory (MSL) has also developed simulation models for several different types of FMSs, and the same graphic elements were used to model the product and resource flows. We have determined that there exists a minimal set of graphic primitives that could be used to describe process flows for any manufacturing system. A sample process flow, which utilizes the set of graphic primitives, is shown in Figure 5. This set of primitives includes: a **start node**, a **finish node**, a **branch node**, a **conditional node**, a **resource node** and an **unit process node** (Figure 6). Using these primitives, any flow condition can be represented: processes which can be done using alternative resources where processing need not be sequential ("and" branch); processing which can be completed by one of many stations ("or" branch); one or more processing steps that are repeated several times ("loop" conditional); and flows dependent on the state of the product ("inspect" conditional). Basile (1993) has developed an object-oriented semi-conductor process flow representation using a similar set of graphic primitives. However, this representation lacks the ability to specify process flexibility through alternative processing routes and also the capability of joining separate product flows.

Every process flow diagram begins with a **start node** and concludes with a **finish node**. The **start node** contains

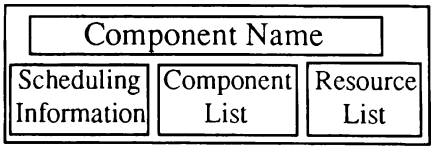

Figure 4. Product Tree Graphic Primitive

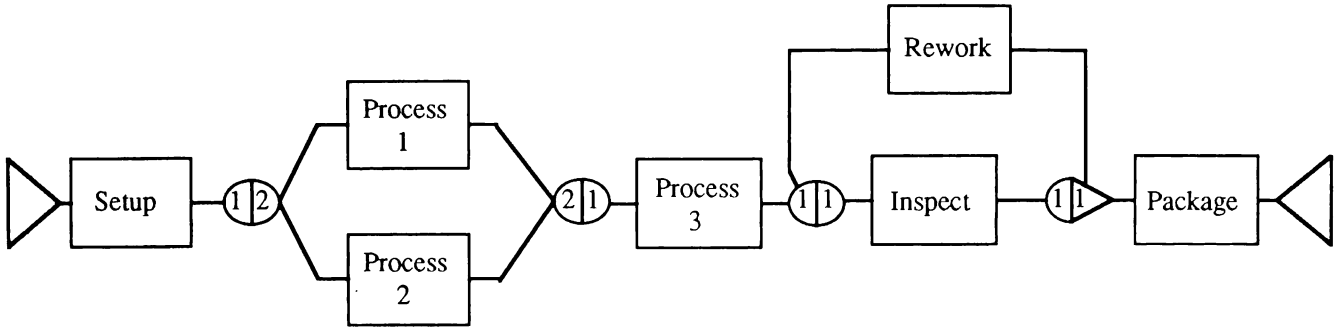

Figure 5. Process Flow Representation

simply the list of items needed to complete the given processing step, along with their respective ID numbers. Detailed information about a given resource (amount, description, etc.) is located in the associated **resource node**. Because a local copy of the process flow is carried with every manufactured entity contained within the product tree (Figure 3), shop-floor data can be recorded for the entity.

Each type of resource in the system is represented by the same graphic primitive. A **resource node** gives the name of the resource, an identification number, and the amount of that resource required for production. As an example, if the resource node represented solder paste, it would contain the amount needed to complete the processing step, or if the resource node represented a tool, this amount would be an estimate of the tool life consumed during the process.

The manner in which entities flow through the system is defined by link nodes, which are the only nodes that can have multiple input or output links. There are two types of link nodes: a **branch node** and a **conditional node**. The **branch node** indicates that there are multiple process paths available and the Coordinator needs to make a decision based on the state of the subsystems it controls. A branch node can be one of two types, an "and" node or an "or" node. For an "and" node, the order in which the alternative processing paths are executed is not crucial, as long as all of the processing paths listed are executed. An "or" node requires that only one of the listed processing paths needs to be executed. Alternatively, **conditional nodes** indicate that the Coordinator makes its decision based on the state of the entity, rather than the state of its subsystems. A conditional node lists the possible processing paths and defines the state of the entity associated with each alternative path. A conditional node is used primarily in two situations - either to model the results of an inspection process or to model a loop in the process flow.

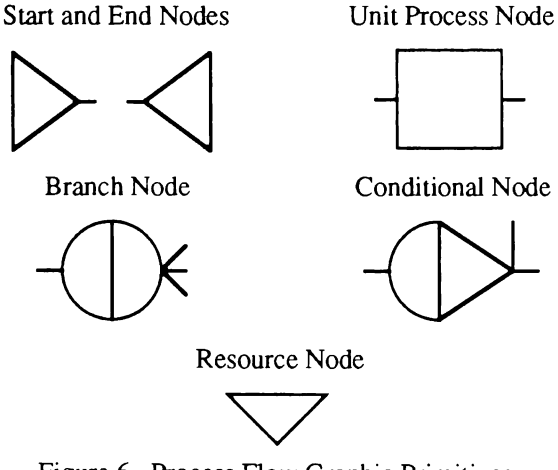

Figure 6. Process Flow Graphic Primitives

2.5 Summary of HOOPLS Methodology

HOOPLS has been explicitly designed to model the coordinators' interactions from both the flow of jobs (parts, assemblies, etc.) and the flow of resources (fixtures, tools, information, AGV's, etc.). The modeled interactions include the explicit exchange of control for each resident job and supporting resource as it is transferred among the coordinate elements within the coordination hierarchy. As a direct result of the object-orientation and formalized coordination hierarchy, the ability to accommodate detailed, flexible processing plans will be greatly enhanced. This latter feature is essential in determining the supporting resources which are required to complete the current processing tasks. HOOPLS is an attempt to provide sophisticated tools to permit accurate modeling of complete operational details, eclipsing the modeling capabilities of existing SQN-based simulation tools.

3 FUTURE RESEARCH

Although the conceptual foundations have been developed for HOOPLS, the implementation of HOOPLS is still very much in its embryonic stage. Preliminary simulations were written in C++, and have shown that the methodology can effectively model a typical FMS. Reusable class libraries are currently under development.

Two projects are also being undertaken to further implement the HOOPLS methodology. A second generation model is being developed for an FMS emulator designed by our MSL, which considers expanded process

plans, tool handling and due date scheduling issues. This second generation model will not only be used for simulations of the emulator, but this same code will be used to physically control the emulator. Our MSL is also developing a HOOPLS model for a semi-automated cell for small batch assembly of circuit boards at the Naval Air Warfare Center (NAWC), Indianapolis.

Concurrent to developing these new HOOPLS models, the programming environment for graphically specifying new HOOPLS models will be developed. The complete environment will include four development frames: the ROOCH Frame, the Control Frame, the Process Plan Frame, and the Experiment Frame. The HOOPLS environment will eventually be an ideal simulation package for the design of modern Integrated Manufacturing Systems.

REFERENCES

Basile, D. 1993. Flow Control for Object-Oriented Semiconductor Process Representations. SRC Technical Report T93009, Semiconductor Research Corporation, Research Triangle Park, NC.

Davis, W. J. 1992. A Concurrent Computing Algorithm for Real-time Decision Making. In the *ORSA Computer Science and Operations Research: New Developments in their Interfaces Conference*, ed. O. Balci, R. Sharda and S. Zenios, 247-266. Pergamon Press, New York.

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

Dilts, D. M., N. P. Boyd and H. H. Whorms. 1991. The Evolution of Control Architectures for Automated Manufacturing Systems. *Journal of Manufacturing Systems*, 10(1):79-93.

Mansfield, E. 1993. The Diffusion of Flexible Manufacturing Systems in Japan, Europe and the United States. *Management Science*, 39(2):149-159.

Matsuda, M. and K. Inoue. 1991. Software Architecture of Autonomous Manufacturing Cells. *Design, Analysis, and Control of Manufacturing Cells*, ASME PED-53:173-183.

Tirpak, T. M., S. M. Daniel, J. D. LaLonde and W. J. Davis. 1992. A Note on a Fractal Architecture for Modelling and Controlling Flexible Manufacturing Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(3):564-567.

AUTHOR BIOGRAPHIES

WAYNE J. DAVIS is a professor of General Engineering at the University of Illinois. His active research areas include simulation, computer-integrated manufacturing, and real-time planning and control of discrete-event systems. He collaborates with the Automated Manufacturing Research Facility and the Electronics Manufacturing Productivity Facility. He is a member of ASME, ORSA and TIMS.

JOSEPH G. MACRO is a doctoral student in the Department of Mechanical and Industrial Engineering at the University of Illinois. He is a recipient of the US Department of Energy Integrated Manufacturing Predoctoral Fellowship.

DUANE L. SETTERDAHL is systems analysis consultant for Cap Gemini America, an enginering consulting firm, in West Chester, Illinois. He received his masters degree from the Department of General Engineering at the University of Illinois.