The International Journal of Flexible Manufacturing Systems, 3 (1991): 231–250
© 1991 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Specification Methods for Material-Handling Control Algorithms in Flexible Manufacturing Systems

TIMOTHY THOMASMA
Department of Industrial and Systems Engineering, University of Michigan–Dearborn, 4901 Evergreen Rd., Dearborn, MI 48128

KURT HILBRECHT
Department of Industrial and Systems Engineering, University of Michigan–Dearborn, 4901 Evergreen Rd., Dearborn, MI 48128

Abstract. Good methods are needed to specify, test, and debug material-handling control logic. This article surveys a number of representative methods for defining and describing control algorithms for programmable material-handling equipment used in flexible manufacturing systems. The methods are evaluated with regard to their suitability for communication between people and as bases for interfaces to automatic program generators. It is concluded that no single method is entirely satisfactory. Three methods (position diagrams, function block diagrams, and operation networks) have potential to be combined into an effective hybrid approach that minimizes the need for the user to switch between various conceptual models.

Key Words: material handling, control algorithms, position diagrams, function block diagrams, operation networks

1. Control software in the flexible manufacturing system life cycle

One of the major hindrances to realizing the potential benefits of flexible manufacturing system (FMSs) is the difficulty of developing the control software that they require. The problem of specifying and developing control software is present throughout the life cycle of the FMS, during its design and implementation, and whenever modifications are made or new components are added.

According to a recent Society of Manufacturing Engineers Blue Book (CASA/SME 1988), simulation is required to determine such things as the optimal number of stations in a flexible manufacturing cell. Because of the dependence of the problem on such a variety of factors—materials, tools, schedules, manning, routing, control—experimentation is needed for its solution. The Blue Book also noted that simulation, including graphic animation, is highly desirable for debugging of complex control logic. Thus, even to model an FMS at early stages of the design effort, the control logic must be specified in the model.

There must also be a good way to specify control algorithms when the system is implemented. The algorithms must be embodied in software for the programmable controllers and computers used for system control. Development of software requires that it first be specified. It often happens that a simulation model is used to design an FMS that incorporates well-constructed control algorithms, but because design and implementation are done by different people, the control algorithms are completely redeveloped when the system

is implemeted (McHaney, 1988). Not only is this a significant reduplication of effort, but there is no mechanism for ensuring that the control algorithms that are implemented are the same as the ones that were designed.

Thus a good method is required for specification of control algorithms in FMSs. A good specification method, when supported by appropriate computer-aided software engineering tools, especially automatic program generators (Rich and Waters, 1988), would provide the following benefits in FMS description, analysis, simulation, and implementation.

- Increased productivity in building simulation models and other models that are capable of incorporating control logic of FMS
- Increased ease and flexibility in modifying control logic in models, thus facilitating experimentation
- Greater degree of correspondence between the control logic that is designed and the control logic that is implemented
- Increased productivity in validating, debugging, and implementing control programs
- Improved communication about control algorithms among the various people involved in design, implementation, and operation of the FMS
- Increased flexibility in adapting control programs to changes in the FMS throughout its life cycle

Aritificial intelligence techniques may in the future provide the basis for control computers that are able to learn how to control FMS cells (Bu-Hulaiga and Chakravarty, 1988). Such computers would be able to design and implement the necessary control algorithms themselves. These systems would, in theory, not require any human involvement in development of control software, not even in its specification. A better understanding of specification methods and tools could contribute to development of control systems that are capable of learning by providing insight into how the required knowledge base should be structured.

The purpose of this article is to survey a number of representative techniques for defining and describing control algorithms for programmable material-handling equipment used in FMSs (robots, automated guided vehicles (AGVs), complex conveyor systems, and automated storage/retrieval systems). Some of these techniques are used informally, while others have been very formally developed and supported by computer-based tools. In section 2 we describe the requirements that a specification method should meet. Existing methods are evaluated against these requirements in section 3. The final section summarizes the strengths and weaknesses of the methods.

2. Desired features of specification methods

In order to provide the benefits listed above, a specification method must be capable of providing a basis for an automatic programming system that can both embed code for modeling control logic into simulation programs and can generate actual control programs for FMS implementation. Our definition of an automatic programming system is adapted from Barstow (1985). An automatic programming system allows a computationally naive user to describe problems using the natural terms and concepts of an application domain in a simple and natural way. An automatic programming system produces programs that run

on real data to effect useful computations that are reliable and efficient enough for routine use. A specification method would serve as the basis for the user interface of such an automatic programming system.

The success of an automatic program generator depends in part on its user interface. If the user interface is badly designed, the process of describing control logic will not be simple or natural. A good user interface provides short learning times, rapid task performance, low error rates, high user satisfaction, and ease of retention (Shneiderman, 1987).

Shneiderman (1983) has identified a class of user interfaces which he calls *direct manipulation* interfaces. They have been highly successful commercially. In addition, the superiority of direct-manipulation interfaces has been verified in laboratory experiments. A notation called *User Action Notation* has been proposed for the specification of direct-manipulation interfaces (Siochi and Hartson, 1989).

Three features characterize a direct-manipulation user interface:

1. There is continuous visible representation of the object and actions of interest
2. The user interacts with the computer by means of physical actions or labeled button presses instead of complex command syntax
3. The system's operations are executed rapidly and are reversible, and the impact of each operation on the object of interest is immediately visible

A direct-manipulation interface is ineffective if the representations of objects and actions are incomprehensible to the user. It has been noted that an object-oriented worldview is very helpful in building simulation models (Burns and Morgeson, 1988; Larkin, Carruthers, and Soper, 1988; Ülgen and Thomasma, 1986). In a computer program based on an object-oriented worldview, all elements of the program, including those that are presented to the user, directly match real-world concepts and entities. Graphical representations and names are chosen with this correspondence always in mind. Design of a direct-manipulation user interface according to an object-oriented worldview increases the likelihood that the representations will be comprehensible.

Whether a computer program is implemented using an object-oriented programming language—i.e., a language that supports abstract data types, polymorphism, and inheritance (Pinson and Wiener, 1988)—is irrelevant to whether or not its user interface is based on an object-oriented worldview. However, programs based on an object-oriented worldview are easier to implement using object-oriented programming languages than other programming languages.

We seek a specification method that has the following features:

1. It is based on an *object-oriented* worldview. Concepts, names, diagram elements, etc. in the methodology have a direct, one-to-one relationship with real-world entities and concepts that are familiar to anyone knowledgeable about the equipment in an FMS.
2. It is *understandable*. The specification method should not require learning concepts and skills that are useful only in connection with that particular technique. When expressed in the language of the technique, specifications should be readily understandable by managers, engineers, technicians, and operators who need to understand the control logic.
3. It is *complete*. The specification technique must be capable of capturing enough information to permit automatic generation of complete control programs.

4. The specification method should be capable of being supported by a computer program that has a *direct-manipulation user interface.*

In the next section, on the basis of the extent to which they incorporate the above features, we evaluate the following methods: problem-oriented languages, function block diagrams, operation networks, time-position diagrams, rule sets with position diagrams, activity-cycle diagrams, Petri nets, and state-transition diagrams.

3. A survey of control-logic specification methods

3.1. An example FMS

In order to assist the reader in making comparisons among methods, we will show how each of them could be applied to specify the control algorithm for the robot in the flexible machining cell pictured in figure 1.

Parts enter the system on an accumulating conveyor and form a queue at the pickup point. Parts are picked in order by the robot at that point and are placed at machine A, machine B, or in storage. Each part that comes into the system has a scheduled processing time on each machine and a routing defined in the part's process plan. This information is made available to the cell controller. The possible part routings within the cell are "in → A → out," "in → B → out," "in → A → B → out," and "in → B → A → out." Storage is used primarily as a place to put parts that need processing on both machines, have been processed on one machine, and are waiting for the other machine to be free for processing.

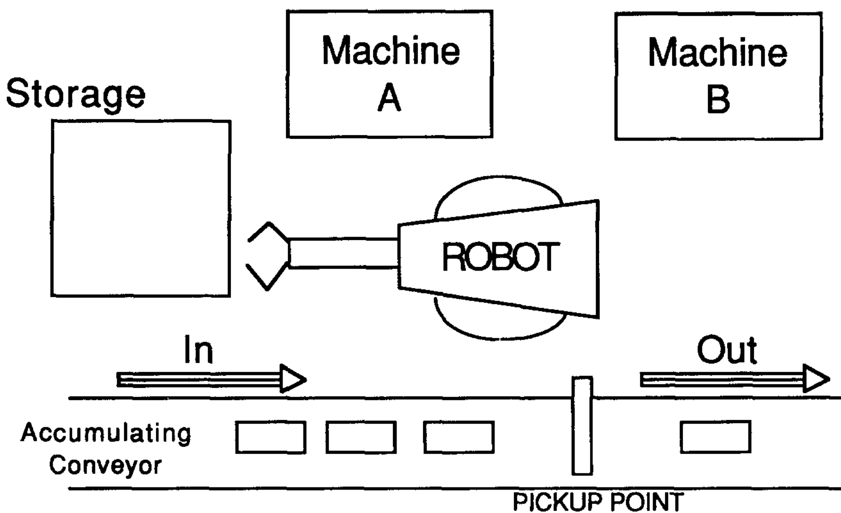


Figure 1. An example flexible manufacturing cell.

Regular preventive maintenance is done on the machines. In addition, both machines are subject to random breakdowns. A machine is always in exactly one of the states "idle," "working," "down," and "finished-part-waiting." When a machine finishes its processing, it can neither be repaired nor can it process a new part until the robot has picked up the finished part.

The robot can pick parts at the pickup point, place parts at Out, and pick and place parts at both the machines and at storage. It can move its gripper (full or empty) from any one of these places to any other. The gripper can hold at most one part at a time. The cell controller knows the remaining routing for each part in storage and the routing for the next part on the conveyor (i.e., the part at the pickup point).

These rules give an overall description of a possible control algorithm for the robot:

1. If a finished part is waiting at a machine, pick it up and take it to whichever place it should go next. If its processing is all done, it should be taken to Out. If it requires processing on the other machine and the other machine is idle, it should be taken there. Otherwise it should be put in storage.

Else:

2. Load idle machines, first from storage, then from the pickup point.

Else:

3. Put parts from the pickup point into storage.

These rules probably do not define a control algorithm that is optimal in any sense, and they are neither complete nor unambiguous. They are characteristic of the sort of natural language description that a manufacturing engineer might give a management scientist who is building a model of the system or to the control engineers responsible for implementing the system.

In the examples that follow, only partial descriptions of some aspects of this or alternative control logics are given. Even for this relatively simple cell, there is no method that can capture all the necessary information and display it in one compact figure.

3.2. Problem-oriented languages

Many programmable controllers and computers used for control can be programmed in ordinary high-level programming languages, like FORTRAN or C (Lukas, 1986, pp. 60–71). Eventually, control algorithms have to be encoded in programming languages for simulation and system implementation, but this must be done using clear specifications as references. In order to make the programming job easier, some vendors provide special programming languages with their controllers or programmable equipment.

A possible language for programming a material-handling robot is illustrated in figure 2. This program instructs the robot to pick up a part from storage and move it to machine A. The first two lines position the robot's gripper in front of storage, at the point with global spatial coordinates $(x, y, z) = (-8, 1, 3)$. There is a part waiting at storage. The next three lines instruct the robot to pick up that part. Line 6 instructs the robot to move its gripper to machine A at coordinates $(-3, 4, 3.5)$. The last three lines instruct the robot to place its part on machine A.

```
OPEN GRIPPER
MOVE TO -8 1 3
EXTEND 0.5
CLOSE GRIPPER
RETRACT 0.5
MOVE TO -3 4 3.5
EXTEND 1
OPEN GRIPPER
RETRACT 1
```

Figure 2. Pick and place operation described in problem-oriented language.

Names and keywords in a good problem-oriented language should be chosen according to an object-oriented world view. This would make the language easier to use than a general-purpose language. But the user will still be faced with all the usual idiosyncrasies common to programming languages. Syntax must be learned and followed precisely. Careful attention must be paid to the order in which statements are written and executed. Problem-oriented languages can be used to completely specify and automatically generate programs for complex control logic, but their use requires all the knowledge that a computer programmer would have. A direct-manipulation interface is not possible with this method. Elements of the control specification are manipulated indirectly by editing a text file.

3.3. Function block diagrams

To build a function block diagram (Lukas, 1986, pp. 49–60), the user is presented with a library of functions that the control system can perform. The functions are represented graphically by symbols or icons. These are chosen from the library and connected together by directed line segments that represent the flow of control signals. In this respect, a ladder diagram or other circuit schematic can be understood as a special case of a function block diagram. Some of the function blocks normally supplied in any system are ones for logic: AND, OR, NOT.

A portion of a robot's control algorithm is pictured in figure 3. The diagram pictures a rule that the robot could use to decide what to do when it has finished moving a part from storage to machine A. If the robot is at machine A, machine A is empty, and the robot has a part, then the robot should begin its place operation immediately.

Function block diagrams are widely used to specify and describe all kinds of control systems. They are used as the graphical language for the HOSE software development environment (Sparks, 1990).

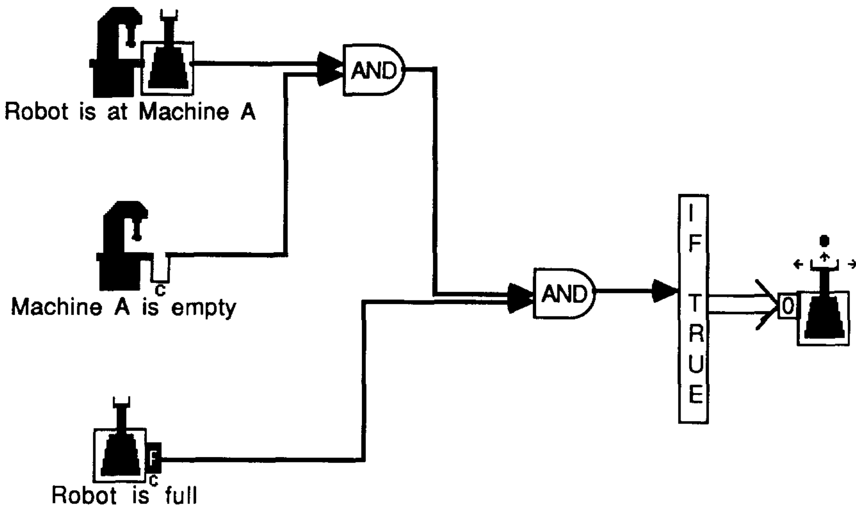


Figure 3. A function block diagram for robot control.

If the blocks in the library are well chosen, function block diagrams can be easy to use and understand. This is true especially if the functions correspond directly to objects that carry out the functions, as they do in an electrical schematic. The information captured on these can be very complete. On the whole, function block diagrams are best for representing complicated decision-making logic. If the control system is large and complex and uses a large number of decision blocks (ANDs, ORs, NOTs), the diagrams can be difficult to read.

3.4. Operation networks

A function block diagram that is used to summarize the sequence of activities in a computer program is commonly known as a flow chart. A flow chart differs from a function block diagram in that the directed line segments between functions represent sequence of activities rather than control signals. Müller (1985) presents a variant of this sort of diagram, which he calls an operation network (figure 4). The functions (operations) in this diagram are "pick," "place," "move," and "manufacture." The diagram is read from left to right. The order of icons in the diagram indicates the order in which operations are done. The diagram can picture operations done sequentially or in parallel, and can portray branching, conjunction, and repetition. Each operation can be further broken down into suboperations.

The diagram in figure 4 defines the following sequence of operations for the cell: 1) The robot places a part at machine A; 2) The robot moves its empty gripper to pickup point In. At the same time, machine A starts processing; 3) The robot picks up a part at In; and 4) The robot moves the part to machine B.

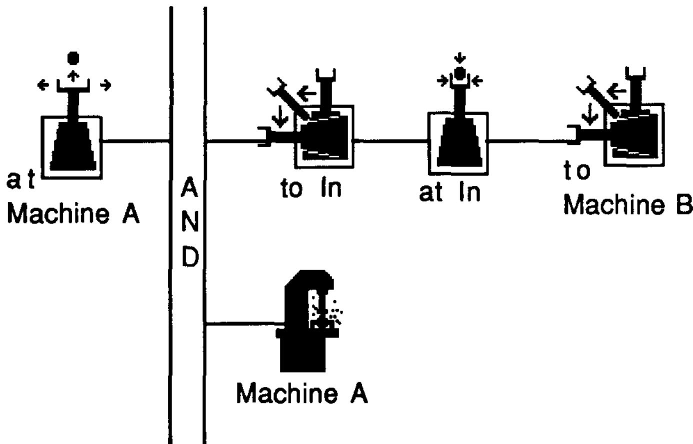


Figure 4. An annotated operation network.

Timing could be easily attached to the operation blocks in these diagrams. Decisions on which branching occurs would have to be added in some way to the description: the operation network does not capture this. The operation blocks match real-world operations very closely. Once it is understood that the positions of the icons indicate temporal precedence relationships among operations, these diagrams are quite easy to understand. The ability to break operations into suboperations makes this appropriate for large systems. This can be well supported by a direct-manipulation user interface. Its drawback is that it does not capture repetition very well or decision rules at all. It also requires some additional labeling to indicate where the moves are from and to.

There is a controller language called Grafcet (Considine and Considine, 1986, pp. 158–159; Savoir, 1986) that is an annotated operations network with extensions. Grafcet adds transitions between the function blocks in the operations network. Function blocks are annotated with programs that implement actions, such as "Move part from pickup point to machine A." Transitions are annotated with programs that are used to determine whether the next function block in the diagram should be executed. Therefore, a function block in a Grafcet diagram is not to be executed until the program (e.g., "Machine A is idle") attached to its preceding transition evaluates to "true." Grafcet diagrams can clearly represent execution of parallel, synchronized processes.

3.5. *Time-position diagrams*

A time-position diagram is a graphic tool commonly used by industrial engineers to coordinate production activities. It is used to help analyze such time-related entities as simultaneous events, sequence of operations, setup times, idle time, and total throughput time. It is widely used in the context of describing robotics work cells. Müller (1985) occasionally uses time-position diagrams in his discussion of material handling. The diagram in figure 5 describes a typical sequence performed by our example robot cell.

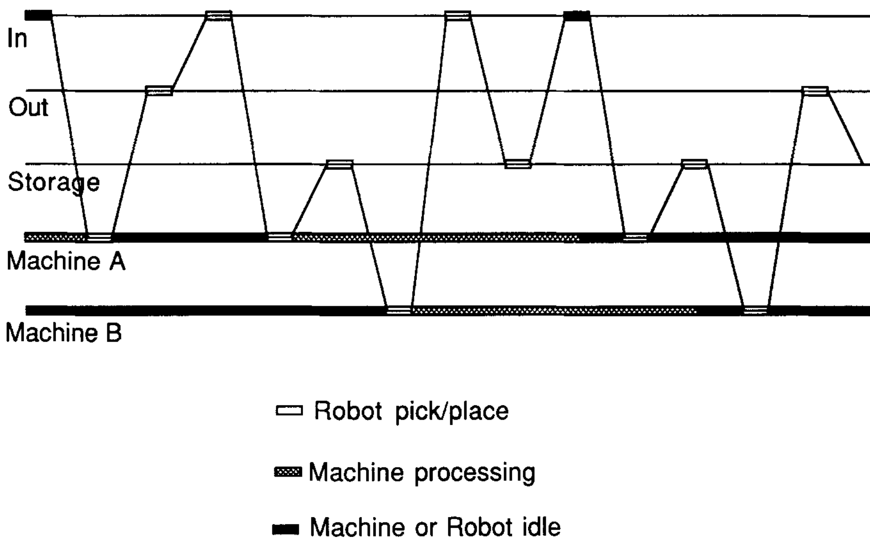


Figure 5. A time-position diagram.

The diagram has a list of positions or operations on the vertical ($y-$) axis. The horizontal ($x-$) axis is time. A horizontal bar, in the case of the robot, represents the robot arm being stationary in a certain position. The length of the bar represents the length of time the arm spends at that point. Lines connecting the bars show transitions from one position to another.

The diagram in figure 5 displays more information than any of the other diagrams presented so far. It shows that at time zero, the robot is waiting at In while machine A is processing a part. The robot then moves to machine A, picks up the finished part, and places it at Out. It then picks a part from In and delivers it to machine A, which begins to process it. The robot gets a part from storage and brings it to machine B, which begins processing it. While both machines are processing, the robot brings a part from In to storage and then waits at In. When machine A is done, the robot brings its finished part to storage and then takes the finished part at machine B to Out. The last diagonal line on the diagram indicates that the robot goes to storage to pick up another part.

Time-position diagrams are very useful for deterministic scheduling, being simple extensions of the familiar Gantt chart (Adelsberger and Kanet, 1989). But they are incomplete descriptions of control logic. They say nothing about what the entities in the system are doing as they move or rest between motions. They have no ability to reflect decision making or adaptation to varying situations in the cell. The diagram is almost useless if the various move times and cycle times are not deterministic. They are also rather abstract, reducing real objects to points along the vertical axis of a chart.

3.6. *Rule sets with position diagrams*

Position diagrams with rule sets attached to decision points are specifically used with material-handling systems, especially AGVs. The position diagram is a map of the physical

routes traveled by the material-handling vehicle. Graphic symbols are placed in positions along the route where special things can occur. Sets of rules that provide the control logic are associated with each of the marked positions, called *decision points*, on the map. These rule sets are usually expressed in terms of a programming language or as productions in an expert system. The PROMOD (1988) simulation system takes this approach in obtaining from the user definitions of material-handling systems.

Figure 6 shows how a rule set with position diagram could be applied to our robot cell. Each position indicates a possible location of the robot's gripper. When the gripper is at the input point, the robot decides what to do next according to the rule set associated with In. If a machine is waiting for a part or waiting to be unloaded, then the robot should service that machine. If both machines are busy, the robot should move a part to storage. Otherwise, the robot should wait for a part to arrive at In on the conveyor.

Note that the positions of the graphic symbols on the diagram could be used to convey distances, thus eliminating the need to specify those particular parameters in the data base of information needed to write a simulation. In fact, the layout could be read from the computer-aided drafting (CAD) package that was used to draw the plant layout when the facility was first planned. The position diagram matches exactly the physical layout of the

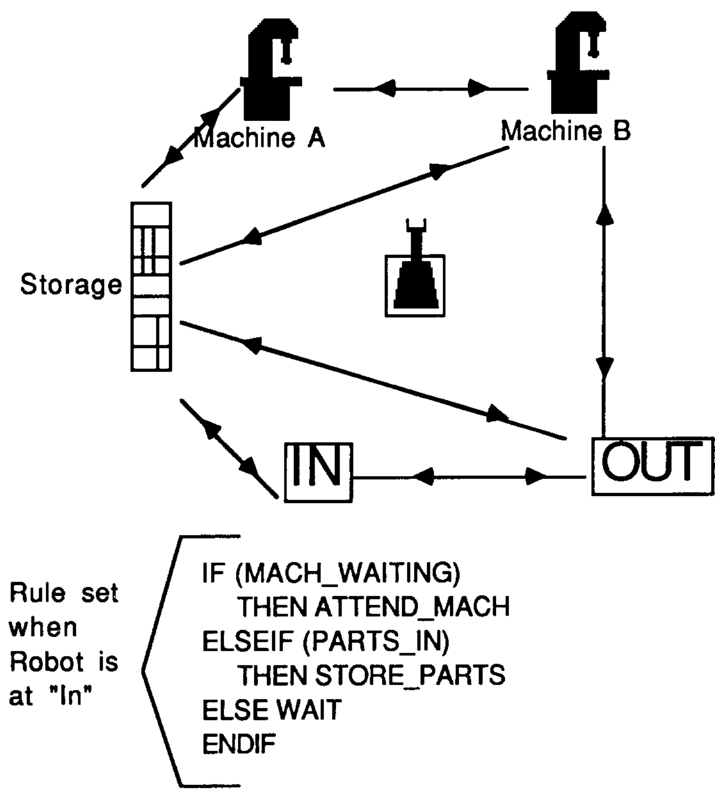


Figure 6. Position diagram with a rule set.

system being controlled, so in that respect it is a strongly objected-oriented approach. However, the decision rules must be written and read by people who can write computer programs or do knowledge engineering. All information needed for simulation or implementation can be captured from the diagram and the rules.

Instead of assigning rule sets to decision points, general decision strategies can be built into the simulation programs and controllers. Such decision strategies could be used to determine such things as which of several waiting parts to pick up first (e.g., take the one that has been in the system longest). A variety of possible decision rules can be made available to the user in a menu (Bachers and Steffens, 1983). The SLAM II simulation language (Pritsker, 1986) takes this approach in its material-handling extension.

3.7. Activity-cycle diagrams

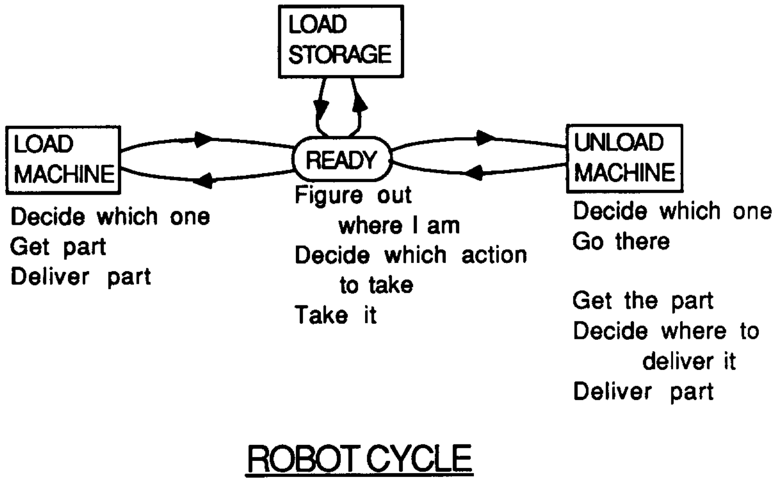
An activity-cycle diagram is used to depict the integration of two or more processes (Pidd, 1988). In these diagrams, rectangles and circles represent active and dead states. Active states represent scheduled service times. In a dead state, an entity waits for something to happen. In general, there is no cooperation between entities of different classes in a dead state. Activities are done in the order indicated by the arrows.

In figure 7, the activity-cycle diagram for the machine shows that it continually repeats this sequence of activities: 1) Wait for the robot to load a part; 2) Process the part; and 3) Wait for the robot to unload the part. States "LOAD MACHINE," "UNLOAD MACHINE," "LOAD STORAGE," are active states of the Robot cycle. "LOAD MACHINE," "UNLOAD MACHINE," "PROCESS PART" are the active states of the machine cycle. The robot is in the dead state "READY" when it is not at one of the active states. A machine is in the "WAIT FOR ROBOT" or "OK" state when not in one of the active states. In the physical world, such states as "OK" and "READY" might not exist. They are placed on the diagram to assure that each active state is followed by a dead state. Annotations under the robot's activities give additional details about the activities and decisions that are made. Figure 8 shows how the cycles for machines A and B would be integrated with the robot's cycle. Both the robot and one of the machines is involved in each "LOAD MACHINE" activity.

There is a computer program that automatically produces simulation programs from activity-cycle diagrams (Hutchinson, 1981). Once a good set of states is chosen to match actual states of equipment in a system, these are fairly easy to understand. The concept of active vs. dead states might be a little difficult to apply in some situations. The diagrams do not capture decision-making behavior or branching.

3.8. Petri nets

Petri nets (Peterson, 1981) were originally conceived to model concurrency in computer systems. Recently, they have been used to study discrete decision and control structures for manufacturing systems. For example, Bruno and Marchetto (1986), Kasturia et al. (1988), and Willson and Krogh (1990) use them to automatically generate programs for process control in manufacturing systems. Wadhwa and Browne (1989) automatically generate simulation programs from them.



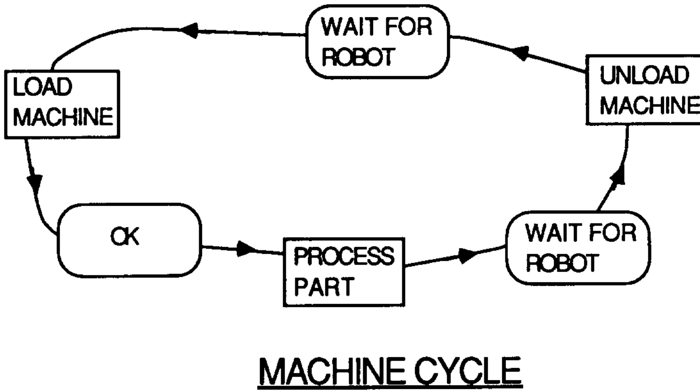


Figure 7. Activity cycle diagrams for robot and machine.

In a Petri net, *transitions* (i.e., operations or events) are representated as short, solid bars and *places* (conditions or states in the process) as labeled circles. Directed arcs connect transitions and places. No two places and no two transitions can be directly connected by one arc. *Tokens* are represented as dots that mark the current states of the process. Transitions fire when all their antecedent places contain tokens. When a transition fires, tokens are removed from all its antecedent places and placed in all its successor places.

Figure 9 shows a portion of a Petri net for our robot cell. To understand it, one must know that to proceed with any transition, the condition preceding it must be satisfied. Thus, in order to begin placing a part on machine A, the robot must have a part that requires processing by machine A and must have its gripper at A. Also, machine A must be idle

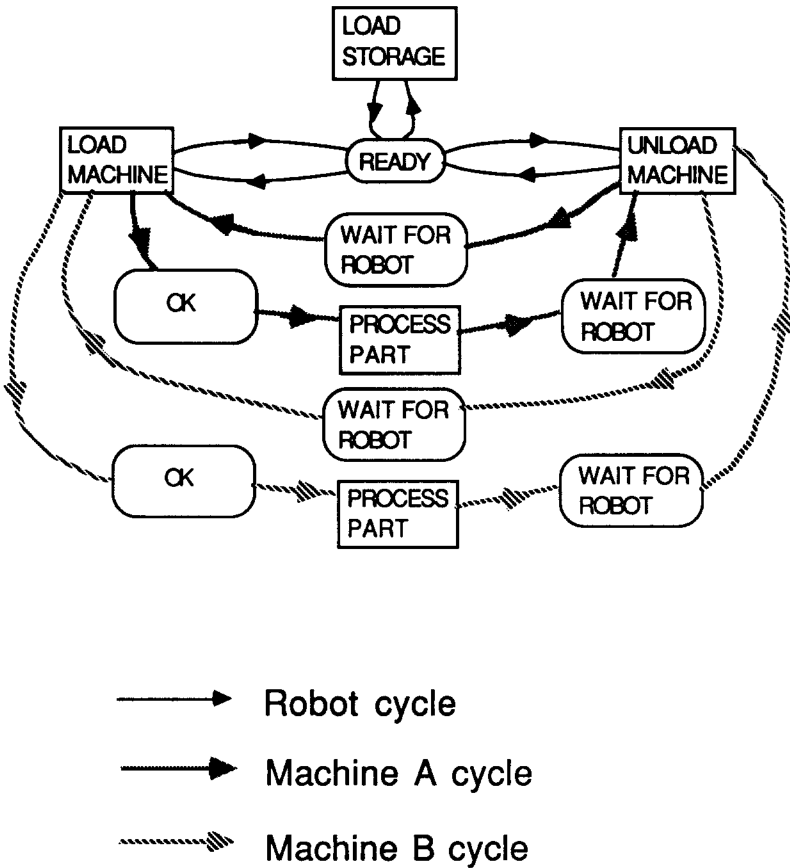


Figure 8. Activity cycle diagram for entire cell.

at that time. In figure 9, there is a token in the place labeled "Machine A idle." When a token appears in the place labeled "Robot has part to process on A and Robot is at A," then the transition will fire and the part can be loaded onto machine A. This will be indicated by removing the tokens and putting a new token into the place labeled "Part being placed in Machine A."

Petri nets convey a clear idea of sequence. They also indicate, in part, what conditions must be met before a given operation can proceed. Through the use of tokens, they can define the state of the system at a given point in time, thus lending themselves to animation. Using and understanding Petri nets requires that states and events in the system be clearly identified and labeled and that labeling be reflected in the Petri net.

Petri nets are seldom used as specification techniques. Although Willson and Krogh (1990) generate programs from them, they use a data base of rules as their user interface. Petri nets are then automatically generated from the rules. The great importance of Petri nets is their usefulness in analysis. Willson and Krogh (1990) and Zhou et al. (1989) use them

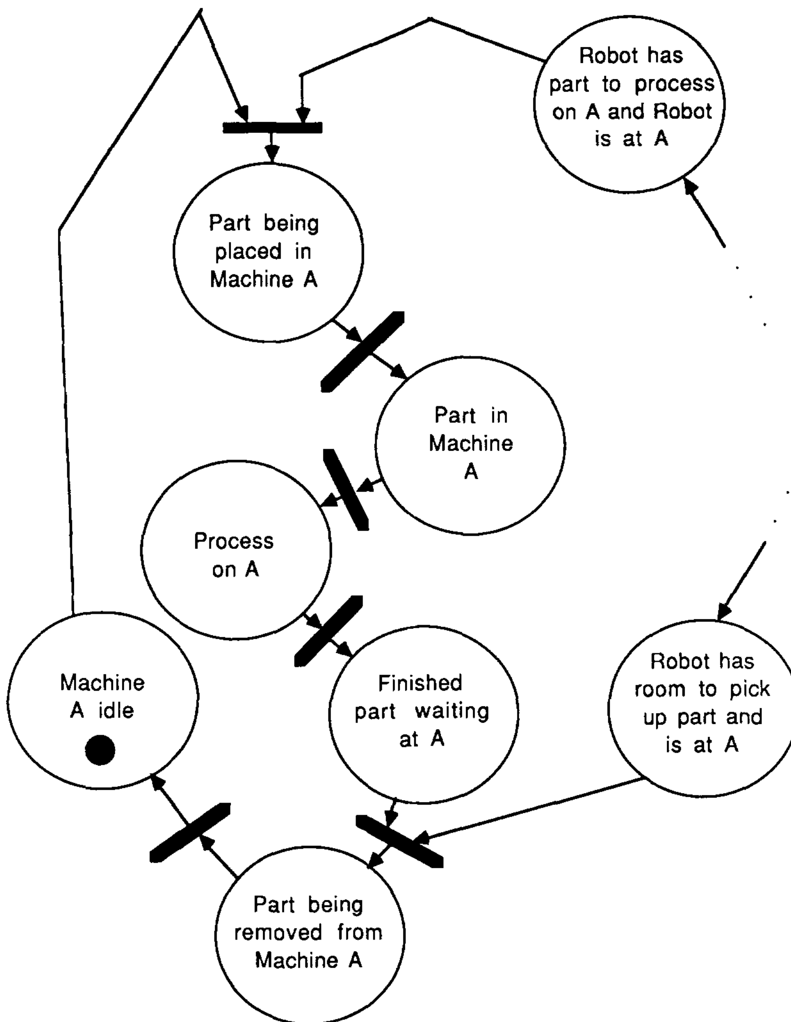


Figure 9. Portion of a petri net.

to determine whether deadlocks are possible and whether it is possible for two incompatible events to occur (safeness). This kind of analysis produces information about a system that is very hard or impossible to obtain by simulation and queueing models.

Petri nets can be difficult to read. Very complicated nets are usually needed to describe systems. It is particularly hard to label places succinctly. Bu-Hulaiga and Chakravarty (1988) find Petri nets hard to modify for reuse when there are structural changes in the manufacturing system being modeled. Wadhwa and Browne (1989) find that producing Petri net models is difficult and that the resulting models are not concise. These and other authors, therefore, have offered a number of extensions of Petri nets in order to increase their expressive power.

Delays can be attached to transitions (DiMascolo et al., 1989) or to tokens in places (Wadhwa and Browne, 1989) to produce timed Petri nets. If these delays are expressed as samples from probability distributions, then the result is a stochastic timed Petri net. This information is important for determining such things as system throughput.

In order to make state descriptions more concise, attributes called *colors* can be assigned to tokens. The resulting formalism is called a colored Petri net (Kasturia et al., 1988). The colored Petri net formalism also allows the modeler to specify functions that control the removal and placement of tokens of various colors when transitions are fired.

Predicates can be attached to transitions (Wadhwa and Browne, 1989), or special decision nodes can be included (Nof et al., 1980) that allows the modeler greater ease in specifying events that are modeled by transition firing. Zhou et al. (1989) and Willson and Krogh (1990) have also offered means of building large Petri nets from smaller Petri net modules.

3.9. State-transition diagrams

Petri nets and activity-cycle diagrams are state-transition diagrams with added structure. An activity-cycle diagram is a state-transition diagram with the added distinction made between "active" and "dead" states. A Petri net is a state-transition diagram plus an explicit way to represent events (*transition*) and current states (*tokens*). In a state-transition diagram, states are represented by labeled circles and transitions by labeled arcs.

The labeling on the arcs in a state transition diagram can be of the form "control signal/output." In this labeling scheme, the control signal is a command containing some information. The output is some additional action that might be taken in addition to changing state. With this labeling, a state-transition diagram can be used to automatically generate a finite-state machine describing the control structures for a computer program (Edelberg, 1988). Drolet and Moodie (1989) have used finite-state machines to automatically produce control programs for a manufacturing cell.

Since they are simpler than activity-cycle diagrams and Petri nets, state-transition diagrams are a little easier to use, but they do not convey as much information as the other diagrams do. All of these three types of diagrams are somewhat abstract in the sense that the positions of entities on the diagram do not have any relationship to positions of objects on the shop floor. This can cause confusion for those who are used to interpreting arrows as indicating material flow or motion from once place to another and circles or other icons as indicating workstations or other physical positions in a factory.

3.10. Other methods

There are some other methods for specifying or programming control logic that, for various reasons, we have not included in our survey.

Ladder diagrams (Lukas, 1986, pp. 55–57) predate the use of computers to control manufacturing processes. They are circuit diagrams for control devices built from relays, timers, contracts, switches, and indicating lights. Ladder diagrams are in widespread use and are well understood among electrical engineers and shop-floor electrical technicians.

However, they are not likely to be helpful to the manufacturing engineer who thinks mostly in terms of flow of materials and how to coordinate the activities of many machines and material-handling devices. Inadequacies of ladder diagrams are recognized by control engineers (Yue and Wong, 1990).

General software specification and design techniques can also be used to specify control logic. These techniques are well surveyed by Martin and McClure (1985); they are primarily concerned with modeling data and transformations of data, and therefore are complementary to the methods surveyed above.

There are also diagrams that are associated with simulation languages such as GPSS. Brandl (1990a) has shown that these diagrams can be treated as varieties of operation networks.

4. Summary

Table 1 summarizes the strengths and weaknesses of the specification methods surveyed above. No one method meets all the requirements set forth in section 2. Petri nets come the closest, but it is generally difficult to describe states (labels for "places") concisely. Some authors define states in terms of patterns of values assigned to instance variables of software objects (Bu-Hulaiga and Chakravarty, 1988) or in terms of histories of events in the system (Naylor and Maletz, 1986). According to these definitions, a great deal of information is needed to describe a state. Also, state transitions are most naturally communicated in terms of animation: changes in visual characteristics over time. The state-transition types of diagrams communicate relationships between states in terms of spatial locations.

Other specification techniques are stronger than Petri nets in some respects. Function block diagrams are better for indicating decision logic. Pure position diagrams (section 3.6) provide the most object-oriented approach, but they are very incomplete. Time-position diagrams give the clearest static view of timing relationships.

Some of these specification methods can be combined very naturally. Function block diagrams can be used as a notation for the rules that accompany position diagrams. An assignment of a rule set to a decision point on a position diagram can be viewed as specification of a decision that must be made upon the completion of a transfer operation. This approach can be generalized to one in which operations are no longer limited to transfer operations. Operations are defined by label, icon, duration, and rules for decisions that are made at the operation's completion. The operation's icon is used in animating the position diagram. Our experience with a system that uses function blocks with position diagrams indicates that it is a useful and informationally complete method (Thomasma et al., 1990a). We have found it cumbersome for describing control logic for large systems, however. Willson and Krogh (1990) observed a similar limitation in their specification method, which is based entirely on rules.

We have found it natural to think in terms of sequences of operations as well as in terms of rules. Operation networks are a natural notation for sequences of operations that a machine or robot frequently executes. Operation networks can also be used to describe process plans for parts. A system based on animated position diagrams, function block diagrams, and

Table 1. Evaluation of specification methods

Technique	Object-oriented worldview	Special knowledge required to use and understand	Information best represented	Completeness of information	Supported by direct-manipulation interface
3.2 Problem-oriented language	Yes, if names and keywords are well chosen	Computer programming	Sequences of operations	Complete	No
3.3 Function block diagram	Yes, if functions can be identified with devices and directed arcs represent something physical	Depends on meanings assigned to function blocks and arcs	Logical decision making, flow of electrical current, other continuous flows	Cannot represent complex material-handling routings and decisions; complete for flow shops	Yes, unless a high degree of annotation is required to capture information completely
3.4 Operation network	Yes, except spatial relationships are used to indicate temporal sequence	Simple definitions of symbols	Sequences of operations	Lacks support for logical decision making (except Grafcet includes some)	Yes (except for annotations in Grafcet)
3.5 Time-position diagram	No	Ability to read a plot of data vs. time	Time-related information	Only deterministic timing and limited position information	No; changes in control logic require total redrawing of chart
3.6 Position diagram with rule sets	Yes, if names and keywords are chosen properly for language in which rules are written	Computer programming or knowledge engineering needed to build rule sets	Position	Complete, but most of the information is expressed in the rules	Only the position diagram
3.7 Activity-cycle diagram	Yes, if well labeled, except spatial relations are used to indicate temporal sequence	Simple definitions of symbols	Sequences of state transitions	Lacks support for logical decision making	Yes, except annotations
3.8 Petri net	Yes, if well labeled, except spatial relations are used to indicate temporal sequence	Simple definitions of symbols; understanding of rules governing firing of transitions	Sequences of state transitions	Supports only AND decisions based on global states	Yes, except annotations required to label places
3.9 State-transition diagram	Yes, if well labeled, except spatial relations are used to indicate temporal sequence	Simple definitions of symbols	Sequences of state transitions	Lacks support for logical decision making	Yes

operation networks is very promising (Thomasma et al., 1990b). Shepherd et al. (1989) and Brandl (1990b) take a similar approach, combining rules and Grafcet diagrams. These systems can be viewed as timed, stochastic Petri nets, with predicates attached to their transitions, that are specially structured for display to users in small, manageable pieces.

We have concentrated in this article on specification methods that capture information at a higher level of abstraction than ladder diagrams or C programming language statements. That is why these methods can serve as user interfaces for programs that can automatically generate the lower-level, detailed control programs. But information of a yet higher level is also needed for effective operation and control of FMSs—for example, overall AGV control strategies, schedules, assignment of tools and operations to machines, and maintenance schedules. The methods we have surveyed are not designed to serve as notations for algorithms for determining this higher-level control information from such inputs as orders, demand forecasts, and process plans.

We envision two ways in which higher-level control information and algorithms can be integrated with the specification methods surveyed in this article. One, which can be implemented using current technology, is to package higher-level algorithms as ordinary computer-program modules that are called instead of rules at decision nodes or transitions.

The other, which requires further research, is to automatically generate Petri nets, Grafcet diagrams, rule sets, state-transition diagrams, and the like (and hence, ultimately, executable control programs), perhaps using artificial intelligence techniques, from such information as overall AGV control strategies, schedules, assignments of tools and operations to machines, maintenance schedules, orders, demand forecasts, and process plans. For this purpose, the specification methods we have surveyed can serve as structures for knowledge representation. Some of these methods already have strong affinities to constructs that are used in artificial intelligence. Rule sets and function block diagrams are similar to the rules in rule bases commonly used as knowledge representations in commercial expert systems. Operation networks and Grafcet diagrams are similar to scripts used in research on machine learning and goal-directed behavior. It should be worthwhile to develop and formalize these specification methods so that they can serve effectively as knowledge representations for artificial intelligence. Achieving that would be a step toward development of control computers capable of constructing their own algorithms for controlling FMS cells.

Acknowledgments

The authors wish to thank Lee Bloomquist of Steelcase, Inc. and Roger McHaney of Control Engineering Company for talking with us about how they develop control logic for material-handling control systems. We also thank the referees for their valuable suggestions. This work was supported by the Research Excellence and Economic Development Fund of the State of Michigan. An abbreviated version of this article was included in the *Proceedings of the Third ORSA/TIMS Special Interest Conference on Flexible Manufacturing Systems.* We thank Elsevier Science Publishers, B.V. for permission to publish this extended version.

References

Adelsberger, H.H. and Kanet, J.J., "The Leitstand—A New Tool in Computer-Aided Manufacturing Scheduling," *Proceedings of the Third ORSA/TIMS Conference on Flexible Manufacturing Systems*, K.E. Stecke and R. Suri (eds.) M.I.T., Cambridge, MA, Elsevier Science Publishers, B.V., Amsterdam, pp. 231-236 (August 1989).

Bachers, R. and Steffens, H., "Development of Strategies for Controlling Automated Material Flow Systems," *Automated Materials Handling: Proceedings of the First International Conference*, London, United Kingdom, pp. 9–20 (April 1983).

Barstow, David R., "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, pp. 1321-1335 (November 1985).

Brandl, Dennis L., "A Language for Real-Time Control Derived from a Discrete Simulation Language," Master's Thesis, California State University, Chico, CA (April 1990a).

Brandl, Dennis L., "Object Structured Design: A Methodology for Analyzing and Constructing Real-Time Systems," *Proceedings of Ninth Control Engineering Conference*, Control Engineering Magazine, Chicago, IL, pp. IV-15–IV-25 (May 1990b).

Bruno, Giorgio and Marchetto, Giuseppe, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, pp. 346–357 (February 1986).

Bu-Hulaiga, Mohammed I. and Chakravarty, Amiya K., "An Object-Oriented Knowledge Representation for Hierarchical Real-Time Control of Flexible Manufacturing," *International Journal of Production Research*, Vol. 26, No. 5, pp. 777-793 (May 1988).

Burns, James R. and Morgeson, J. Darrell, "An Object-Oriented World-View for Intelligent, Discrete, Next-Event Simulation," *Management Science*, Vol. 34, No. 12, pp. 1425-1440 (December 1988).

CASA/SME Technical Council, "CIM Integration Tools for Programmable Controllers and Flexible Manufacturing," *SME Blue Book Series*, Society of Manufacturing Engineers, Dearborn, MI (1988).

Considine, D. and Considine, G., *Standard Handbook of Industrial Automation*, Chapman and Hall, New York (1986).

DiMascolo, Maria, Frein, Yannick, Dallery, Yves, and David, Reneé, "Modeling of Kanban Systems Using Petri Nets," *Proceedings of the Third ORSA/TIMS Conference on Flexible Manufacturing Systems*, M.I.T., Cambridge, MA, Elsevier Science Publishers, B.V., Amsterdam, pp. 307-312 (August 1989).

Drolet, Jocelyn R. and Moodie, Colin L., "A State Table Innovation for Cell Controllers," *Computers and Engineering*, Vol. 16, No. 2, pp. 235–243 (February 1989).

Edelberg, Allen, "The Finite State Program: A Software Design for Real-Time and Event-Driven Programs with CASE Implementation," *Programmer's Update*, pp. 60-70 (November/December 1988).

Hutchinson, G.K., "The Automation of Simulation," *Proceedings of the 1981 Winter Simulation Conference*, Atlanta, Georgia, Institute of Electrical and Electronics Engineers, Piscataway, NJ, pp. 489–495 (December 1981).

Kasturia, E., DiCesare, F., and Desrochers, A., "Real Time Control of Multilevel Manufacturing Systems Using Colored Petri Nets" *IEEE International Conference on Robotics and Automation*, Philadelphia, PA, Institute of Electrical and Electronics Engineers, Piscataway, NJ, pp. 1114–1119 (April 1988).

Larkin, Timothy S., Carruthers, Raymond I., and Soper, Richard S., "Simulation and Object-Oriented Programming: The Development of SERB," *Simulation*, Vol. 51, No. 3, pp. 93–100 (September 1988).

Lukas, Michael P., *Distributed Control Systems: Their Evaluation and Design*, Van Nostrand Reinhold, New York (1986).

Martin, James and McClure, Carma, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, Englewood Cliffs, NJ (1985).

McHaney, Roger (1988), "Bridging the Gap: Transferring Logic from a Simulation into an Actual System Controller," *Proceedings of 1988 Winter Simulation Conference*, San Diego, CA, Institute of Electrical and Electronics Engineers, Piscataway, NJ, pp. 583-590 (December 1988).

Müller, Willi, *Integrated Materials Handling in Manufacturing*, IFS Springer-Verlag, New York, NY (1985).

Naylor, Arch W. and Maletz, Mark C., "The Manufacturing Game: A Formal Approach to Manufacturing Software," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 16, No. 3, pp. 321-334 (May/June 1986).

Nof, S., Whinston, A., and Bullers, W., "Control and Decision Support in Automatic Manufacturing Systems," *AIIE Transactions*, Vol. 12, No. 2, pp. 156-167 (June 1980).

Peterson, J.L., *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, NJ (1981).

Pidd, M., *Computer Simulation in Management Science,* 2nd edition, John Wiley & Sons, New York, NY (1988).

Pinson, L.J. and Wiener, R.S., *An Introduction to Object-Oriented Programming and Smalltalk,* Addison-Wesley, Reading, MA (1988).

Pritsker, A. Alan B., *Introduction to Simulation and SLAM II,* 3rd edition, Systems Publishing Corporation, West Lafayette, IN (1986).

PROMOD, Production Modeling Corporation of Utah, 1875 South State, Suite 3400, Orem, UT (1988).

Rich, Charles and Waters, Richard C., "Automatic Programming: Myths and Prospects," *Computer,* Vol. 21, No. 8, pp. 40–51 (August 1988).

Savoir Grafcet: Preliminary Version, Savoir, 2219 Main Street, Santa Monica, CA (1986).

Shepherd, D., Coote, S., Gallagher, J., Lea, R., Mariani, J., and Scott, A., "An Object Oriented Approach to the High Level Programming of Distributed Process Control Applications," Research Report, Department of Computing, University of Lancaster, Bailrigg, Lancaster, U.K. (1989).

Shneiderman, Ben, "Direct Manipulation: A Step Beyond Programming Languages," *Computer,* Vol. 16, No. 8, pp. 57–69 (August 1983).

Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* Addison-Wesley, Reading, MA (1987).

Siochi, A.C. and Hartson, H.R., "Task Oriented Representation of Asynchronous User Interfaces," *Proceedings of the Conference on Human Factors in Computing,* Austin, TX, Association for Computing Machinery, pp. 183–188 (May 1989).

Sparks, Hugh, "Object Oriented Dataflow Programming Techniques for Industrial Automation," *Proceedings of Ninth Control Engineering Conference,* Control Engineering Magazine, Chicago, IL, pp. IV-27–IV-36 (May 1990).

Thomasma, Timothy, Ülgen, Onur M., and Mao, Youyi, "Tools for Designing Material Handling Control Logic," *Object Oriented Simulation,* A. Guasch (Ed.), Society for Computer Simulation, San Diego, CA, pp. 19–22 (1990a).

Thomasma, Timothy, Mao, Youyi, and Ülgen, Onur M., "Defining Behavior in a Hierarchical Object-Oriented Simulation Program Generator," *Proceedings of the 1990 Summer Computer Simulation Conference,* Calgary, Alberta, Canada, Society for Computer Simulation, San Diego, CA, pp. 266–271 (July 1990b).

Ülgen, Onur M. and Thomasma, Timothy, "Simulation Modeling in an Object-Oriented Environment Using Smalltalk-80," *Proceedings of 1986 Winter Simulation Conference,* San Diego, CA, Institute of Electrical and Electronics Engineers, Piscataway, NJ, pp. 474–484 (December 1986).

Wadhwa, S. and Browne, Jim, "Modeling FMS with Decision Petri Nets," *International Journal of Flexible Manufacturing Systems,* Vol. 1, No. 3, pp. 255–280 (June 1989).

Willson, Reg G. and Krogh, Bruce H., "Petri Net Tools for the Specification and Analysis of Discrete Controllers," *IEEE Transactions on Software Engineering,* Vol. 16, No. 1, pp. 39–50 (January 1990).

Yue, Andrew and Wong, Peng Chiew, (1990), "English Language for Industrial Control Using a Graphical User Interface," *Proceedings of the Ninth Control Engineering Conference,* Control Engineering Magazine, Chicago, IL, pp. I-1–I-5 (May 1990).

Zhou, MengChu, DiCesare, Frank, and Desrochers, Alan, "A Top-down Approach to Systematic Synthesis of Petri Net Models for Manuacturing Systems," *IEEE International Conference on Robotics and Automation,* Scottsdale, AZ, Institute of Electrical and Electronics Engineers, pp. 534–539 (May 1989).