3 Discrete Event Control of Manufacturing Systems

D. M. Tilbury
University of Michigan

P. P. Khargonekar
University of Florida

3.1 Introduction

A (discrete part) manufacturing process, whether it be machining or assembly, consists of a sequence of steps that must occur to transform the raw materials into finished parts. A manufacturing system is a set of machines (and humans) along with associated control and information systems protocols that implement the manufacturing process. The steps in the process, often called "operations," are assigned to certain machines. The machines are arranged in a line, and as the part moves along the line, the specified operations are performed on it; at the end of the line, it becomes a finished product. The line of machines may be a physical arrangement, or a virtual "line" where the machines are grouped into cells and an operator or computer guides the parts through the appropriate sequence of machines.

Automated manufacturing systems must perform the same sequence of operations repeatedly. There are two distinct types of control systems in a typical automated manufacturing system: continuous control and discrete event control. Continuous control systems regulate continuous variables such as position, velocity, etc.* Discrete event control correctly sequences the system

*In current technology, continuous control is often implemented using digital computers. In this sense, this type of control is discrete-time digital control. This discrete-time control should not be confused with discrete event control.

operations: do one step after another, perform a specified sequence in the event of a failure, etc. In actual operation, these two types of control systems work concurrently. In this presentation, we will focus on the discrete event control and neglect the interactions between discrete event controllers and the continuous controllers.

In a discrete event framework, the behavior of a manufacturing system is described by a sequence of events, such as the flip of a switch, the push of a button, or the start or end of an operation. These events take the system from one discrete state to another. The state of the manufacturing system is one of a finite set of states, rather than a collection of continuous variables. For example, the discrete event model of a robot gripper may have four states: open, closing, closed, opening; whereas, the continuous model of the gripper would contain position, velocity, and force variables to indicate how wide the gripper is open, how fast it is moving, and the force exerted by the gripper in the closed position.

Because the capital equipment cost for an automated manufacturing system is extremely high, many of these systems typically operate 2 or 3 shifts each day, and 6 or 7 days a week, making reliability extremely important. Thus, in addition to controlling the manufacturing system when it is working well, the discrete event controller must be able to handle various errors. For example, if one machine breaks, the machine before it should stop sending it parts, or if the coolant tank is empty, the spindle should stop drilling. When errors do occur, the discrete event controller should notify an operator by producing some type of error message.

In this chapter, we discuss the problem of discrete event control related to manufacturing systems, how industry currently solves these control problems, current trends in the area, and formal methods that can be used to design and analyze the discrete event control systems used in manufacturing.

3.2 Background on the Logic Control Problems

Discrete event control problems encountered in manufacturing systems consist of the logic and sequence coordination, error recovery, and manual control. These problems are simple in the small view, but extremely complex in the overall picture due to the large number of events that must be coordinated, each with its own input and/or output. For example, a transfer line machining system with ten machining stations can easily contain 10,000 discrete I/O points. Even for such complex manufacturing systems, with thousands of inputs and outputs, the discrete event control is typically written in a low-level programming language. This creates large, unwieldy programs that, although they are intuitive at a very low level, are difficult both to implement and to maintain.

3.2.1 Logic Control Definition

The discrete event control for a manufacturing system controls all of the activity at the machine level as well as the coordination between machines (including material handling). The discrete event controller is also responsible for machine services, such as lubrication and coolant.

Both the discrete event behavior of a manufacturing system and the discrete event controller for the system can be modeled as discrete event systems. Because of the overwhelming complexity of most industrial manufacturing systems, however, the entire possible behavior of the system is rarely described. Typically, only the desired or controlled behavior is specified. In any case, the existing formal methods for analyzing such a combined discrete event system are limited by the computational complexity of dealing with large numbers of states.

A simple block diagram of a manufacturing system with a logic controller is shown in Figure 3.1. The logic controller governs the sequence of the manufacturing process. It controls the system so that the events occur in the specified order in the process, and generate an error event and stops the process in case something goes awry.

Inputs to a discrete event control system consist of proximity and limit switches that indicate the state of the manufacturing system as well as buttons and switches controlled by the operator.

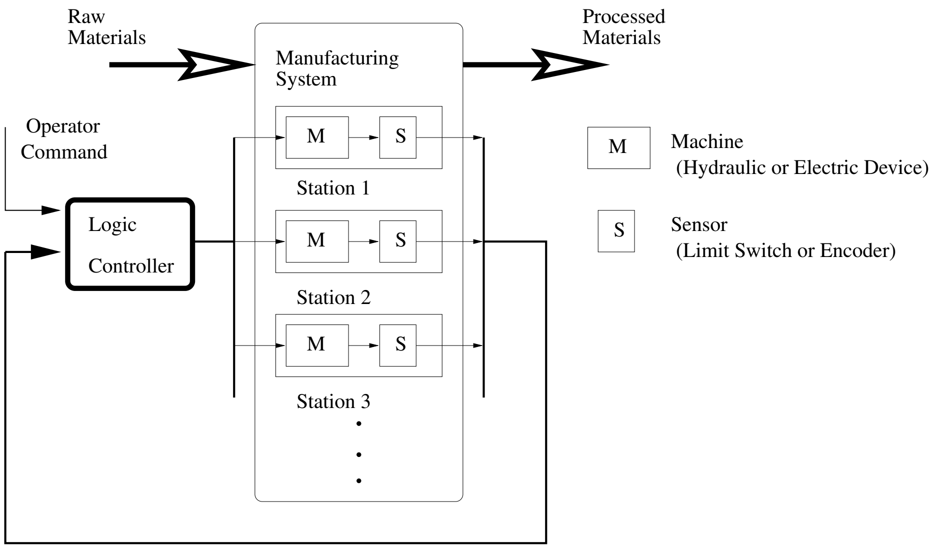


FIGURE 3.1 A block diagram of a manufacturing system with logic control. Raw materials (unfinished parts) enter the system, the machines in the system perform some operations on the parts (such as machining, assembly, etc.), and processed materials (finished or semi-finished parts) leave the system. The logic controller coordinates the operations of the various machines. It is preprogrammed to execute the proper sequence, and also takes some inputs from a human operator. Sensors attached to each machine provide feedback to the logic controller.

The outputs are on/off signals that control valves, motors, and relays as well as lights on the operator interface panel.

3.2.2 Control Modes

The discrete event control for a manufacturing system typically has several different modes. In normal operation, when the system is producing parts, the control operates in the automatic cycle or auto mode. This mode requires little or no operator supervision or intervention, and is the simplest mode to specify and implement. In the event that an error occurs, an operator is usually required to help get the system back to normal operation. The manual modes allow the operator to step through the operation one task at a time or retract slides to allow access to change a tool. Other modes allow the entire operation sequence to be performed only once, or provide diagnostics.

For example, consider a machining system operating in the auto mode. At some point, the tool on the drilling station may break while the system is drilling. The part being worked on will need to be removed, and the machine returned to its default or home position to be ready for the next part. To accomplish this, the operator will first put the machine into manual mode, and will push a sequence of buttons to turn off the power to the spindle, retract the slide, unclamp the part, etc. Then he or she will reach into the machine and physically remove the damaged part and replace the broken tool; hardwired safety interlocks will ensure that the machine cannot operate while the operator is inside the enclosure. Another sequence of buttons will need to be pressed to reset the machine to its home position, and then the operator can switch the machine to the auto mode again. A flow chart depicting this switching of control modes is shown in Figure 3.2.

3.2.3 Logic Control Specification

The sequencing behavior of a manufacturing system can be specified in many different ways. The process plan specifies the operations that must be done to a part to transform it from raw material to a finished product. This plan is generated from the part definition along with the chosen manufacturing

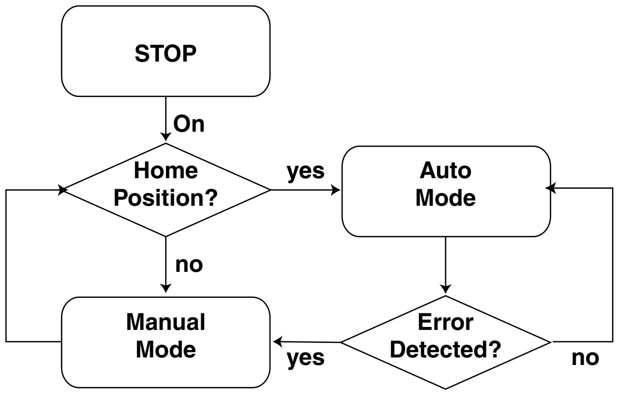


FIGURE 3.2 A flow chart indicating the transitions between auto mode and manual mode. In the manual mode, operator pushbuttons are enabled that can help the operator get the machine back into the home position. The auto mode can only begin when the machine is correctly configured. An error will cause the machine to exit the auto mode and go to the manual mode; an operator is required to fix the machine and help return it to the home position.

processes. If there is only a single sequence in the process, an ordered list of operations will suffice for the logic control specification. Often, however, many tasks must take place simultaneously. The interrelationships and sequential dependencies between these tasks may be specified using a timing bar chart. The tasks to be performed are listed on the vertical axis, and the time taken for each task is represented by a horizontal bar, with the horizontal axis representing time. Dependencies between tasks are indicated by dotted arrows.

A transfer line is a manufacturing system used for high-volume machining operations, for example, automotive engine blocks. Generally, a transfer line is composed of 4 to 12 machining stations; the operation of the system is governed by event sequences within the stations as well as dependencies across the stations. In devising control algorithms for such a machining system, it is necessary to consider not only the sequence of each station but also the correlated sequences of the whole system. An example of a transfer line is shown in Figure 3.3. The system has 15 stations, consisting of 4 mills, 3 clamps, a cradle, and a rotating table. Not all stations are used; the extra space is needed to provide access to the machines for maintenance and repair. The engine blocks move through the machine via a transfer bar from station 1 to station 15. At station 6, they are reoriented.

The timing bar chart shown in Figure 3.4 represents part of the behavior of the high-volume transfer line shown in Figure 3.3. In a transfer line, all of the individual stations must synchronize their operations to the transfer mechanism. Thus, each station has the same amount of time to finish its operation. The total time for operation and transfer is called the cycle time of the transfer line. The causal dependencies of the sequences are represented using the time axis, and the dotted arrows correlate the sequences which depend on each other physically. The timing information of each operation comes from the specifications of the continuous control loops that govern the underlying continuous-time mechanical systems. The timing bar chart shows at a glance the time taken by each task within the cycle time, the time dependencies of tasks, and the total cycle time.

The timing bar chart thus has all the information needed to describe the sequences of tasks that must be performed, and it represents the specification of the operations for the desired process. It is limited by the fact that it only includes the specification for the normal operation of a system, the automatic cycle, or auto mode. The specifications for the other modes of the system (manual, diagnostics, etc.) are rarely described precisely; the control programmer uses experience and intuition to write the logic control for these other modes. Because of this imprecise specification, and the impossibility of foreseeing every possible error that may occur, the logic for the manual modes often requires significant modification during the testing and debug phase.

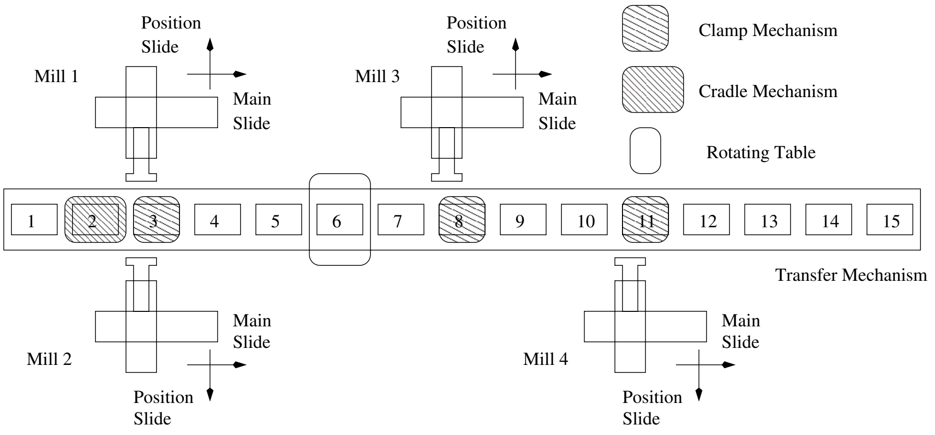


FIGURE 3.3 Sketch of a high-volume transfer line for engine block surface milling. Engine blocks move through the transfer line from station 1 to station 15. The system is composed of four milling machines, a transfer mechanism, and fixture mechanisms. The clamp mechanisms are fixtures for the milling machines and the cradle mechanism prevents interference between mill 2 and the engine block in location number 2. The transfer bar mechanism moves each engine block to next location in each cycle motion. The milling machines start to work after the engine blocks are located properly by the transfer bar mechanism, the cradle mechanism, and the clamp mechanisms.

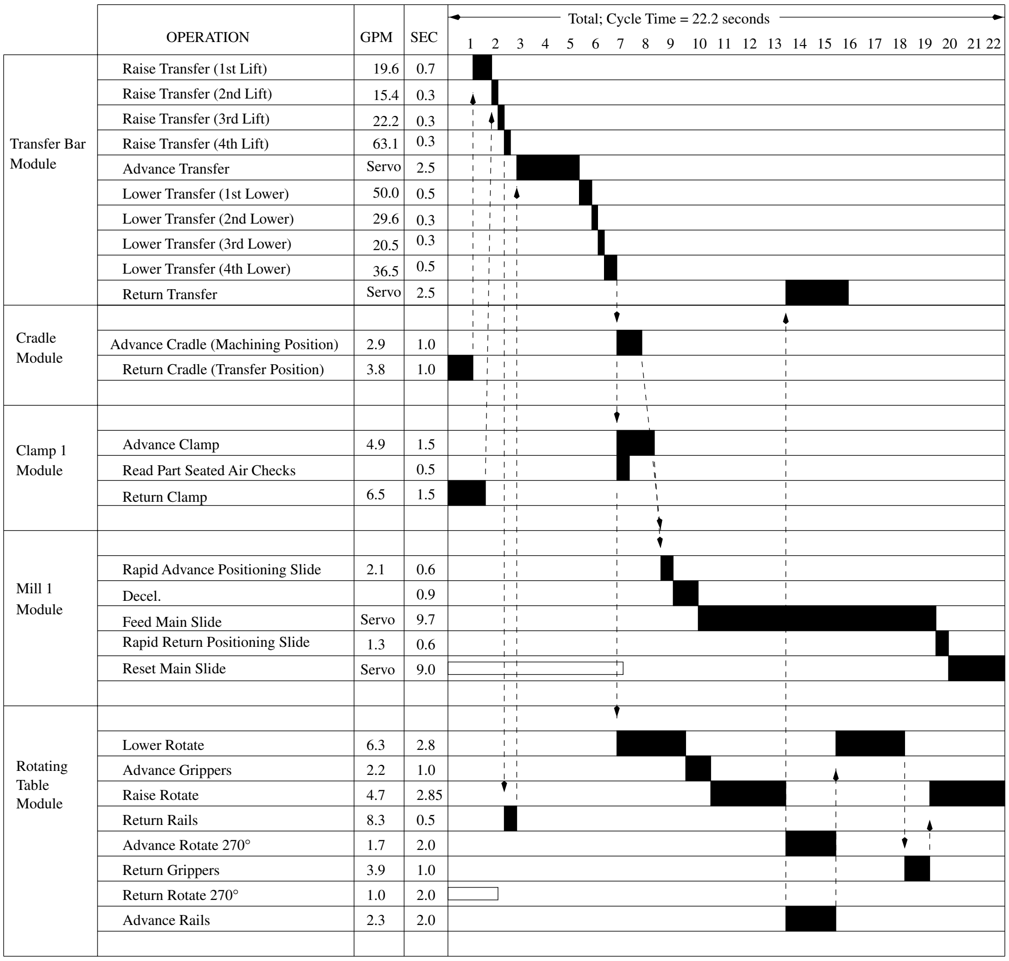
3.2.4 Tasks of a Logic Control Programmer

A major task in the design of manufacturing systems is the design and programming of the logic controllers. A logic control programmer starts from the mechanical definition of the machine and the tasks that it must perform. The inputs to the mechanical system (valves to control coolant and lubrication, motor drives, etc.) are identified, and a set of outputs (limit switches, proximity sensors, etc.) are determined. The total number of inputs and outputs for the system must be known before the control hardware can be specified. It is not uncommon for a machine tool to have 1000 or more I/O points; the complexity is considerable.

Each input and output must be assigned a unique address. Oftentimes, one controller is used for several machines. Even if the logic program is the same for each machine and can be written once and copied, the I/O addresses must be changed — a laborious process. A table of the I/O is maintained to guide the programmer as well as the electrician who will wire everything up.

Once all of the I/O are available, the logic control program must be written. A logic control program may be written as a sequence of if/then rules, or as a flow chart. For example, a logical statement may be "if the part is in place, then engage the clamp." The part is considered to be in place if the appropriate proximity sensor is active, and the clamp is engaged by turning on a solenoid. This statement is implemented in a low-level language as "if the memory location P contains a 1, then write a 1 to the memory location S." It is common for variables to be referred to by their memory locations and not by names; thus, the I/O table must be accurate and up-to-date. Logic control programs may also be written in a flow-chart type program to emphasize the sequential nature of the tasks.

Although each logical statement may be relatively simple, tens of thousands of such statements will be required to make the machine work properly. Also, the logic control program must implement all of the control modes, and it must prevent damage from occurring to the machine. For example, if a drill is extended, the "open clamp" command should be disabled. Other things that must be considered when writing the logic control program include supplying lubrication to a spindle and coolant to a machining operation, checking for availability of hydraulic fluids, as well as all the operator interfaces.



	OPERATION	GPM	SEC
Transfer Bar Module	Raise Transfer (1st Lift)	19.6	0.7
	Raise Transfer (2nd Lift)	15.4	0.3
	Raise Transfer (3rd Lift)	22.2	0.3
	Raise Transfer (4th Lift)	63.1	0.3
	Advance Transfer	Servo	2.5
	Lower Transfer (1st Lower)	50.0	0.5
	Lower Transfer (2nd Lower)	29.6	0.3
	Lower Transfer (3rd Lower)	20.5	0.3
	Lower Transfer (4th Lower)	36.5	0.5
	Return Transfer	Servo	2.5
Cradle Module			
	Advance Cradle (Machining Position)	2.9	1.0
	Return Cradle (Transfer Position)	3.8	1.0
Clamp 1 Module			
	Advance Clamp	4.9	1.5
	Read Part Seated Air Checks		0.5
	Return Clamp	6.5	1.5
Mill 1 Module			
	Rapid Advance Positioning Slide	2.1	0.6
	Decel.		0.9
	Feed Main Slide	Servo	9.7
	Rapid Return Positioning Slide	1.3	0.6
	Reset Main Slide	Servo	9.0
Rotating Table Module			
	Lower Rotate	6.3	2.8
	Advance Grippers	2.2	1.0
	Raise Rotate	4.7	2.85
	Return Rails	8.3	0.5
	Advance Rotate 270°	1.7	2.0
	Return Grippers	3.9	1.0
	Return Rotate 270°	1.0	2.0
	Advance Rails	2.3	2.0

FIGURE 3.4 A portion of the timing bar chart for the transfer line system shown in Figure 3.3. Each operation that must be performed by the system is listed on the left-hand side of the table; the horizontal axis indicates time. The solid lines indicate the amount of time taken by each operation, and the dotted lines indicate causal dependencies between operations. Note that all operations are synchronized to the transfer bar mechanism. The total cycle time is 22.2 seconds.

In any automated manufacturing system, safety is always a primary concern. Typically, the safety circuitry is not programmed into the logic controller but hardwired using relays. An "emergency stop" switch (big red button) is always available; when it is engaged, the machine will stop immediately. The logic control programmer is often responsible for specifying the emergency control logic to be wired by an electrician.

3.3 Current Industrial Practice

Logic controllers for manufacturing systems run on proprietary control systems known as PLCs, or programmable logic controllers.

3.3.1 Programmable Logic Controllers

PLCs are specialized computing devices designed for logic control. They combine a general-purpose microprocessor with discrete I/O capabilities, and are able to handle the thousands of inputs and outputs that are necessary to control a manufacturing system. There are several manufacturers of PLCs, each with their own software tools for programming and slightly different interpretations of the standard languages. Code written for PLCs is not generally portable; a program written for an Allen-Bradley PLC will not run on a Modicon PLC without modification.

PLCs typically operate by reading all of the inputs to a system, then computing all of the logic, then writing all of the outputs. This "scan time" depends on the number of inputs and outputs as well as the complexity of the logic, and may not be repeatable from scan to scan. In addition, the same logical program implemented in a different language or even in the same language on a different platform may require a different scan time. For this reason, it is difficult to achieve guaranteed and repeatable real-time performance with PLCs.

In the early days of automated manufacturing, hardwired relays were used to control the logical behavior of the machines. The logic control "program" was an electromechanical circuit, and programming was done by electricians. When the first microprocessors became available, they were used to replace the unreliable relays. A programming language called "relay ladder logic" was developed to program these early logic controllers. Its graphical interface mimicked the appearance of relays, to make the transition from hardwiring to software easier.

3.3.2 Relay Ladder Logic

Almost 30 years after it was developed, ladder logic remains the industry standard for logic control. Ladder logic is similar to assembly language, the lowest-level programming language commonly used. This makes it easier to implement ladder logic on a microprocessor than it would be to implement a higher-level language. In addition, low-level languages such as assembly and ladder logic give the programmer full control over the instructions being executed on the processor. Programs written in these low-level languages can be made to run very efficiently.

A sample ladder logic program is shown in Figure 3.5. The main elements of ladder logic are normally open contacts, normally closed contacts, and output coils. The relay contacts switch from open to closed or vice versa if the corresponding input terminal or memory location contains a "high" voltage or a "1." Each rung of the ladder implements a simple "if/then" statement. If all of the relays in a rung are closed, then the output coil will be activated. In many implementations of ladder logic, an animated display can tell the programmer or operator which signals are high and which rungs are active, allowing for efficient low-level debugging.

However, because ladder logic is a low-level programming language, the programs for even a relatively small system rapidly become unwieldy (the printout may be several inches high). There is very little support for subroutines or procedures, and no sense of variable "scope." Because all variables are global, it is relatively easy for one part of a large program to mistakenly overwrite or change a variable used by another part of the program. In addition, no facility exists for structured data; only bits and registers are allowed.

Ladder logic has many disadvantages; programs written in ladder logic take longer to develop, are harder to maintain, and are less reusable than equivalent programs written in a higher-level language (such as C++). The most common method for reuse of ladder logic code is to copy the rungs of the ladder from an old program and paste them into a new program. The data I/O address must still be changed to match those of the current project. Databases and libraries can be developed to automate this process, but it is still tedious.

Several alternatives to ladder logic have been proposed. A new standard, the IEC 1131-3,[12,14] includes five distinct languages. One is the familiar ladder diagram; others include structured text, function block diagrams, instruction list, and sequential function charts. Although these languages are based on familiar languages, they have more support for subroutines, parameter passing, limited scope, and strongly typed variables. The standard is intended to allow software written for one brand of PLC to be able to be run on other brands of PLCs.

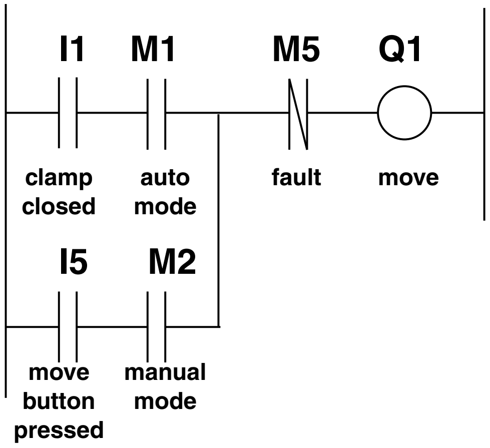


FIGURE 3.5 A sample of relay ladder logic. There are three types of elements: normally open contacts, normally closed contacts, and commands. The I1 and I5 are input signals from a clamp proximity switch and a pushbutton, respectively; the signals M1, M2, and M5 represent memory locations; and Q1 represents an output that may go to a solenoid or a memory location. The ladder diagram implements the following logical statement: "If (((I1 and M1) or (I5 and M2)) and not M5) then Q1;" or equivalently "If the clamp is closed and the system is in auto mode, or the move button is pressed and the system is in manual mode, and there is no fault, then move."

3.3.3 Sequential Function Charts

Sequential Function Chart (SFC) is one of the IEC 1131-3 languages for logic controllers.[12] It is based on Grafcet which was inspired from Petri nets, and thus logic controllers designed using Petri nets (see 3.5.4) can be easily implemented using SFC. Logic control programs can also be written directly in SFC.

Sequential Function Chart and Grafcet are both commonly used in industry along with the ladder diagram.[4,5] SFC programs have two types of nodes: steps and transitions. Steps are represented by squares and initial steps are represented by a double square. The steps in Grafcet can have only one token; in other words, the marking of a step is a Boolean representation. In SFC, a set of simultaneously firable transitions can be fired. It can be shown that a special class of Petri nets (safe marked graphs) is equivalent to SFC.

3.4 Current Trends

3.4.1 Issues with Current Practice

Because logic control programs must be implemented in proprietary programming languages, there is little ability to reuse code (or even library functions) from one project to the next unless the same brand of hardware is used. Even if the same hardware is used, and some code can be reused, the hardware is not inexpensive. Because there is a relatively small market for PLCs, they are expensive compared to more general-purpose computers (such as PCs) with similar performance. Hardware add-ons, such as video cards and networking cards, must be developed for each proprietary architecture and contribute significantly toward the overall cost of a PLC system.

Another major expense associated with discrete event control in a manufacturing system comes from the required electrical wiring. Each limit switch or proximity sensor must have power, and its output must be connected to the PLC. With hundreds or even thousands of I/O points on a typical machine, the labor needed to initially set up this wiring results in a high cost. Additionally, such a mass of wires is extremely difficult to debug — and often the wires do get crossed or connected to the wrong terminal. The PLC and its associated I/O are typically housed in an electrical cabinet near the machine, along with the power supplies, transformers, and motor drives. The floor space consumed by this cabinet is significant.

Most PLC programming languages are fairly low level, requiring many lines of code to implement simple functions. The development time for such programs is relatively long. Some code can be reused mostly through copying and pasting previously written code. Because of the low-level language, all variables are referred to by either their I/O address or their memory address. Thus, if the same function is going to be performed on a different part of the same machine, the same code can be reused, but all of the variable names need to be changed.

In current practice, the logic programs are written while the machine is being built, and are verified on the machine during the ramp-up phase. No method for formally testing the program for correctness exists (although simple tests can be done to find inputs not used or conflicts in the logic program). Some work is currently being done to automatically convert ladder logic into a more formal discrete event system formalism for verification purposes.[24] However, current verification algorithms for discrete event formalisms test all possible combinations of states. With large systems, the number of combinations of states grows too large to feasibly test every combination.

3.4.2 PC-Based Control

There is currently a great deal of interest in moving away from standard logic controllers implemented as ladder logic on a PLC. Both hardware and software are changing. The drivers for this change include price and flexibility. As noted earlier, most PLC systems are proprietary, and even ladder logic programs are not interchangeable between brands. As special-purpose computing devices, PLCs have a relatively small market size. The competition is based on software and support; the hardware commands premium prices. The most likely successor of the PLC is an industrialized version of the desktop PC, which benefits from a large market share to drive down prices for microprocessors, memory, communication peripherals, etc. Because of this intense competition, PCs have much more computational power at a lower cost than PLCs. As the market moves toward general-purpose PCs, programming languages and development tools designed for conventional software will become available. There will certainly be ladder logic implementations on a PC, but more varied programming languages, more powerful and easier to use, will also become viable options. PC-based control will allow the continuous and discrete event control to be integrated on the same computer platform.

3.4.3 Distributed Control

Traditionally, the I/O for an entire machine was brought back to a centralized PLC. Now, distributed systems are being implemented. In a distributed system, a group of smaller PLCs each control a region or subsystem of the machine, and these PLCs communicate and cooperate to control the entire machine. These distributed systems are easier to wire up, and can be designed and debugged in a modular manner.

In some instances, all of the sensors and actuators for a machine may be connected to a sensor or control network. Instead of two or three wires for each sensor, there is one cable which brings both power and a network connection to each sensor. The sensor information is then transmitted to the PLC over the network. Control networks, or sensor networks, are high-bandwidth networks optimized for sending small, periodic packets of information, as opposed to data networks which send large, asynchronous packets of information.[15,21] Currently, these networks are used only to replace the wiring; in the future, each device may also have some embedded intelligence and be able to glean information off the network to determine appropriate control actions.

3.4.4 Simulation

Although some simulation packages have recently become available, control systems for machining systems are typically not verified before they are implemented. A relatively long "cycle and debug" stage in the development process is used to fix most of the problems with the control code. In the past, a transfer line could be expected to build the same parts for 10 or more years. With reduced product lifecycles, the lifetime has been reduced to 5 years or less. Currently, the control system cannot be tested until all of the machinery is in place in the factory setting.

Simulation of the control system combined with the mechanical machine is becoming more common in industry, but is time consuming in terms of both operator setup and computer time. For an unfamiliar system, this may be warranted, but many systems are built as variations of previously built ones, and a reasonable degree of confidence in the correctness of the approach exists.

Several simulation environments are available for production systems, both from universities[20] and commercially.[6,25] A simulation of the manufacturing equipment can be built, and an interface built to the control system. Then the control system can "control" the simulation. Depending on the fidelity and accuracy of the simulation, the control software can be sufficiently tested before it is deployed on the plant floor. Performance can be predicted, and problems with collisions and timing discovered. Some environments provide simple 2-D line graphics; others use 3-D or even virtual reality to animate the manufacturing process.

In addition to control analysis and testing, these simulations have other advantages such as enabling process improvements by the manufacturing engineers (and subsequent changes to the control program) and operator training in a virtual environment. Because the control software and the manufacturing system are so complex, formal verification methods typically fail. However, in a simulation environment, many different test cases can be examined quickly, and some problems can be identified and fixed before they occur on the plant floor.

3.5 Formal Methods for Logic Control

Even though logic controllers are very important in the manufacturing industry, a standard integrated tool does not yet exist that is sufficiently simple to use, powerful, versatile and with which it is possible to carry out systematic analysis and design of discrete event control systems.

3.5.1 Important Criteria for Control

Logic controllers for manufacturing systems must satisfy a given set of criteria. The most important is performing the given task. The task may be defined as a single sequence of events or as an intertwined sequence such as a timing bar chart. It must not be possible for a logic controller to get stuck in a state from which it cannot move; this is formalized as the definition of deadlock-free. The systems must also be reversible, meaning that from any state, they can always return to the initial state with a suitable sequence of events. The time taken to complete one entire cycle of the operation is called the cycle time of the system; this time is often specified in advance (if not, it should be as short as possible while maintaining the desired part quality). In addition to performing the specified task in the automatic mode, the logic controller should contain some diagnostics to detect errors or problems when they occur, and either inform the operator or possibly take action to correct them. The manual modes must allow the operator enough flexibility to control the machine through a pushbutton interface.

3.5.2 Discrete Event Systems

A discrete event system is defined as a dynamic system whose evolution through the state space is defined by the occurence of instantaneous discrete events.[3] Examples of discrete events are the push of a button by an operator, the triggering of a limit switch, the activation of a solenoid, a tool breaking. An event occurs at some discrete moment in time rather than over a time interval.

There has been quite a bit of academic research into the area of discrete event systems, primarily as they relate to computer programming. Most of the control-related research has been based on the framework set up by Ramadge and Wonham,[22,23] in which a controller is defined entirely as a mechanism for the prevention of unacceptable behavior, but not as a method of forcing a discrete event system to complete a task. There has been some research into the concept of forcing events,[1,9] but it is not yet complete. Very little extension of the formal academic theory to real manufacturing systems has occurred. The reason generally given is that as a system grows in complexity, the complexity of a discrete event system associated with the system grows at an exponential rate. This quickly leads to intractable controller verification.

A variety of representations of discrete event systems exist; a few of them are described in this section. Languages are the most general way to express a discrete event system. Any discrete event system can be described using a language, but generally languages are difficult to work with. Finite state machines are a common expression of a discrete event system and have a more well-defined structure than languages; however, they can still be quite complex. Basic descriptions of these representations can be found in Kumar and Garg.[13] Petri nets are another formalism for describing discrete event systems. All of these representations have their advantages and disadvantages. As a rule, the more general the representation of a discrete event system, the more difficult it is to prove desired properties about the system.

The most general representation of a discrete event system is in terms of language theory. The set of events which can occur in a system is denoted by the set $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$. The basis of language theory is the "string," which represents one possible sequence of events which can occur in a discrete event system.

Definition 3.5.1 (Languages) A *string* is an ordered list of events, representing a possible sequence of events in a discrete event system. A *language* is a (possibly infinite) set of strings, representing all possible sequences of events that may occur in a discrete event system.

Given a set of events Σ, the language consisting of all possible strings with elements in this set is denoted Σ^*. Other languages with the same event set are subsets of this. Two strings s and t can be combined by concatenation; $s.t$ denotes the list of events in s immediately followed by the list of events in t.

Thus, a discrete event system with event set Σ has a language L which is a subset of Σ^*, i.e., $L \subseteq \Sigma^*$. Even if Σ is a finite set (which is not necessary), L is often an infinite set. This complexity of enumeration makes language theory difficult to work with from a computational point of view.

Although a language can describe any discrete event system, it is difficult to prove desirable properties of a system from its language definition. For this reason, other modeling formalisms such as finite state machines and Petri nets are more popular than language theory.

3.5.3 Finite State Machines

A finite state machine is a special type of discrete event system in which the event set Σ contains only a finite number of events. In addition, the language of the discrete event system must be describable in terms of the evolution of a state machine with finitely many states.

Definition 3.5.2 (Finite State Machine) A *finite state machine* is a quintuple, $S = \{X, \Sigma, \alpha, x_0, X_m\}$, where:

X	=	The finite set of all states in the FSM
Σ	=	The set of all events recognized by the FSM
α	=	The transition function; $\alpha\colon X \times \Sigma \to X$
x_0	=	The initial state
X_m	=	A set of marked states

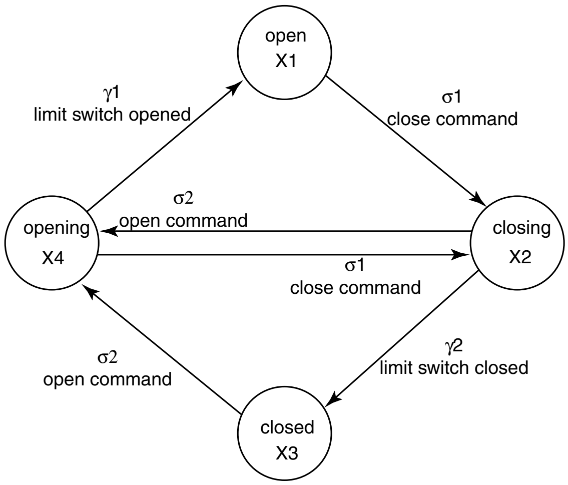


FIGURE 3.6 A finite state machine modeling a gripper or clamp. The system has four states ($X = \{x_1, x_2, x_3, x_4\}$) and four events ($\Sigma = \{\sigma_1, \sigma_2, \gamma_1, \gamma_2\}$).

The transition function α is not generally defined for all possible event/state pairs. At any state, only a subset of the events in Σ can happen. The function α is generally extended recursively to map the set of all states × strings to the set of states as follows. For a string s, a state x, and an event σ, if $\alpha(x, s) = x'$ then $\alpha(x, s.\sigma) = \alpha(x', \sigma)$. This is equivalent to the state reached if all the events in the string are executed sequentially starting from the state x.

The marked states typically represent some desired final states that the finite state machine should reach. If only cyclic behavior is desired, then the initial state may be the only marked state. In other cases, more than one marked state may be used to denote different execution models.

The language admitted by the finite state machine S is denoted $\mathscr{L}(S)$. This is the set of all strings admitted by the finite state machine. The marked language $\mathscr{L}_m(S)$ of the finite state machine can also be defined as the set of all strings which take the initial state to a marked state.

$$\mathscr{L}(S) = \{s \in \Sigma^*: \alpha(x_0, s) = x \in X\}$$

$$\mathscr{L}_m(S) = \{s \in \Sigma^*: \alpha(x_0, s) = x \in X_m\}$$

A finite state machine is generally visualized as a set of nodes representing the states connected by a set of arrows labeled with events representing the transitions. The finite state machine shown in Figure 3.6 can be used as a model of a gripper or clamp. There are four states representing the discrete state of the gripper (x_1 = open, x_2 = closing, x_3 = closed, and x_4 = opening). There are four events in the system. Two of them, σ_1 = close and σ_2 = open represent commands that tell the gripper to change state. The other two, γ_2 = closed and γ_1 = opened, represent limit switches that trip when the gripper has completed its state transition. Not every event is allowed at every state. The state transition function α can be specified by enumeration; it is given in Table 3.1.

TABLE 3.1 The State Transition Function α for the Finite State Machine in Figure 3.6.

α	σ_1	σ_2	γ_1	γ_2
x_1	x_2	–	–	–
x_2	–	x_4	x_3	–
x_3	–	x_4	–	–
x_4	x_2	–	–	x_1

States are listed along the left-hand side, events across the top, and the entries in the table indicate the state that results after an event occurs. Entries marked with a – indicate that the corresponding event cannot occur when the system is in that state.

3.5.3.1 Combinations of Finite State Machines

When broken down into small pieces such as the four-state system described above, discrete event models of manufacturing systems are relatively simple. The complexity arises when many small pieces are put together to form the discrete event model of the entire system. The formal method for combining finite state machines is through parallel composition. For a finite state machine $S = \{X_S, \Sigma_S, \alpha_S, x_{S_0}, X_{S_m}\}$, let the set of events which can occur at a state $x \in X$ be denoted $\Sigma_S(x)$.

$$\Sigma_S(x) = \{\sigma \in \Sigma_S : \alpha(x, \sigma) \in X\}$$

Parallel composition for two finite state machines is then defined as follows.

Definition 3.5.3 (Parallel Composition, ||) Given two finite state machines, A and B:

$$A = \{X_A, \Sigma_A, \alpha_A, x_{A0}, X_{Am}\}$$

$$B = \{X_B, \Sigma_B, \alpha_B, x_{B0}, X_{Bm}\}$$

The *parallel composition* of A and B is defined as:

$$A \,\|\, B = \{(X_A \times X_B), (\Sigma_A \cup \Sigma_B), \alpha, (x_{A0}, x_{B0}), X_{Am} \times X_{Bm}\}$$

where the transition function α of the parallel composition A || B is defined as:

$$\alpha((x_A, x_B), \alpha) = \begin{cases} (\alpha_A(x_A, \sigma), \alpha_B(x_B, \sigma)) & \text{if } \sigma \in \Sigma_A(x_A) \cap \Sigma_B(x_B) \\ (\alpha_A(x_A, \sigma), x_B) & \text{if } \sigma \in \Sigma_A(x_A) \text{ and } \sigma \notin \Sigma_B \\ (x_A, \alpha_B(x_B, \sigma)) & \text{if } \sigma \in \Sigma_B(x_B) \text{ and } \sigma \notin \Sigma_A \\ \text{undefined} & \text{otherwise} \end{cases}$$

In other words, the parallel composition of two machines is equivalent to running both machines simultaneously, with the restriction that events which are elements of both event sets must occur concurrently in both machines. Although the number of events in the combined state machine is (at most) the sum of the number of events in Σ_A and Σ_B, the number of states in the parallel composition is the *product* of the number of states in each state machine *A* and *B*. This leads to the state explosion property as many finite state machines are combined.

3.5.3.2 Supervisory Control of Discrete Event Systems

The most prevalent framework for supervisory control of discrete event systems is that of Ramadge and Wonham.[22,23] In this formalism the set of events Σ is divided into two subsets, labeled Σ_c and Σ_u, called the "controllable" and "uncontrollable" events, respectively. All events must be in one of the two sets, thus $\Sigma = \Sigma_c \cup \Sigma_\sigma$ and $\Sigma_c \cap \Sigma_\sigma = \varnothing$. As suggested by the name, controllable events, can be disabled by the supervisor, which means that any transition labeled with a controllable event can be removed at will. For example, in the simple finite state machine of Figure 3.6, the events $\Sigma_c = \{\gamma_1, \gamma_2\}$ which represent the open and close commands would be considered controllable. The events $\Sigma_u = \{\gamma_1, \gamma_2\}$ which represent the trippings of the two limit switches are influenced by a physical process and would generally be considered uncontrollable.

The supervisor consists of another state machine, which operates on the same set of events as the finite state machine being controlled. Associated with each state in the supervisor finite state machine is a set of controllable events that are enabled when the supervisor is in that state. Only events in this set can be executed by the machine being controlled. At all times every uncontrollable event must be enabled. For example, consider the state machine shown in Figure 3.6 representing a robot gripper combined with a similar one representing the robot moving from one location to another (say to pick and place a part). Once the robot is positioned over the part to be grasped, the supervisor would enable σ_1, representing the close command. Once the event γ_2 has occurred, signaling that the gripper is closed, the supervisor would enable the event corresponding to commanding the robot to move to the desired destination, and so forth. The supervisor keeps track of the state of the system and enables events appropriate to that state.

Another notion of supervisory control of discrete event systems has also been proposed.[1,9] In this framework, controlled events are those that can be forced by the supervisor onto the plant, and will only happen then. Uncontrollable events are those the plant can force on the supervisor. Technically, these notions are equivalent;[2] either can be used to design and analyze finite state machine controllers.

3.5.3.3 Verification of Closed-Loop Behavior

Specification of the desired behavior of control system in the finite state machine framework is generally given in terms of the language L that should be admitted by the finite state machine. The language may be enumerated or specified by another finite state machine. The controller finite state machine will act to disable controllable events that should not happen in certain states.

For a plant P and a controller C, both finite state machines, the controlled (closed-loop) behavior is defined as the parallel composition of the two, $P \parallel C$. The controller should interact with the plant in such a manner that the parallel combination can always reach a marked state; this is formalized by the definition of non-blocking.

Definition 3.5.4 (Non-blocking) A finite state machine S is *non-blocking* if a marked state can be reached from every state in the machine. That is, for every state $x \in$ X, there exists a sequence of events $\mathrm{s} = \sigma_1 \sigma_2 \ldots$ such that $\alpha\,(x, \mathrm{s}) = x_m \in X_m$.

The existence of a non-blocking controller/plant combination that allows all possible uncontrollable events to occur can be determined based on the finite state machine representing the plant and the desired language L.[22] If a non-blocking controller exists, it can be constructed in a straightforward manner as a finite state machine.

Most approaches for verification of finite state machines rely on enumerating all of the states and events to guarantee that an undesirable state is never reached. Even though there are finitely many states and finitely many events, the number of states grows exponentially as more state machines are combined together using parallel composition. Thus, although techniques exist for constructing a supervisory controller given the finite state machine of the plant P and the specified closed-loop behavior (the language L), the large size of the resulting state space limits the sizes of systems that can be handled.

3.5.4 Petri Nets

Petri nets, as graphical and mathematical tools, provide a powerful environment for modeling, formal analysis, and design of discrete event systems. Historically, Carl Adam Petri first developed Petri nets in 1962 as a net-like mathematical tool for the study of communication. Since that time, they have found many uses in a wide variety of applications such as communication protocols, manufacturing systems, and software development. A good survey on properties, analysis, and applications of Petri nets can be found in References 4, 7, 16, and 27.

Petri nets have been used as an analysis tool for event-based systems that are characterized as being concurrent, synchronized, and distributed. Petri nets enable the qualitative and quantitative analysis of an event-based system. The modeled system can be verified to be correct from the qualitative analysis, and the efficiency of the modeled system can be determined from the quantitative analysis.

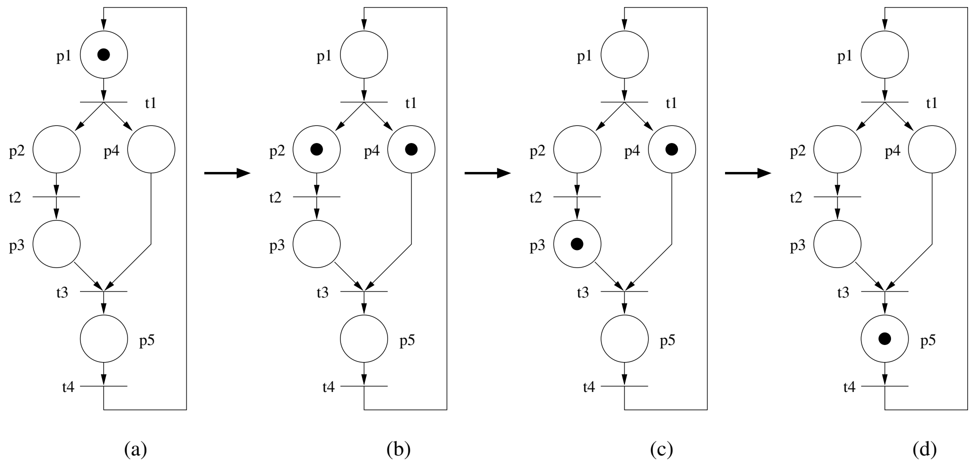


FIGURE 3.7 Marking evolution of an ordinary Petri net: (a) initial marking, (b) firing transition t_1, (c) firing transition t_2, (d) firing transition t_3. In the initial marking (a), there is one token in p_1. Transition t_1 is enabled because all places leading to it are marked; it is the only transition enabled. After transition t_1 fires, the marking becomes that shown in (b). Each place leading out of transition t_1 gets a token. Now transition t_2 is enabled (transition t_3 cannot fire until both places p_3 and p_4 are marked). If more than one transition is enabled at a time, the Petri net exhibits nondeterministic behavior. After t_4 fires, the Petri net returns to its initial marking (a). The Petri net is thus said to be reversible.

Petri net models are used to analyze three important properties of a discrete event system: liveness, safeness, and reversibility. The meanings of these properties for a Petri net for a logic controller are summarized in Reference 26 and are discussed later in this section. By analyzing these properties of the Petri net model, the functional correctness of the logic generated from the Petri net model can be assured.

3.5.4.1 Graphical Representation of Petri Nets

As stated above, a Petri net is a mathematical formalism which has a simple graphical representation. Petri nets consist of two types of nodes: places represented by circles, and transitions represented by bars. Nodes are connected by directed arcs. The dynamics of a Petri net are determined by its initial marking and marking evolution rule. A marking assigns to each place a nonnegative integer and the integer value is graphically represented by the number of tokens in each circle (place). The number of tokens in a place represents the local state of the place and the state of the whole system is defined by the collection of local states of the places. A pictorial example of the evolution of an ordinary Petri net is given in Figure 3.7.

A Petri net and its evolution rule can be represented formally by the following definitions.

Definition 3.5.5 (Petri Nets) A *Petri net* is a four-tuple, $\langle P, T, F, W \rangle$ where:

$Px = x\{p_1, p_2, \ldots, p_n\}$, *a finite non-empty set of places*
$Tx = x\{t_1, t_2, \ldots, t_m\}$, *a finite non-empty set of transitions*
$F \subset (P \times T) \cup (T \times P)$, *the flow relation* (*set of directed arcs*)
$Wx{:}xF \rightarrow Z^{+}$, *the weight function which assigns an integer weight to each arc*

A Petri net is termed *ordinary* if all the arc weights are one. A *marking M* of a Petri net *N* is the assignment of a nonnegative integer to each place. It is an *n*-dimensional state-vector of the Petri net system. A Petri net with the given initial marking is denoted by $\langle N, M_0 \rangle$. The state or marking in a Petri net evolves according to the following transition (evolution) rules:

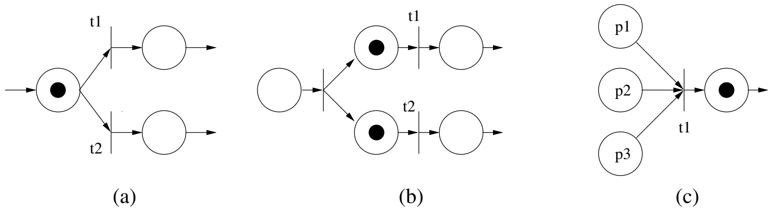


FIGURE 3.8 Some modeling capabilities of Petri nets: (a) conflict: if t_1 fires, t_2 is not enabled and vice versa; (b) concurrency: t_1 and t_2 can be fired independently; (c) synchronization: t_1 synchronizes p_1, p_2, and p_3.

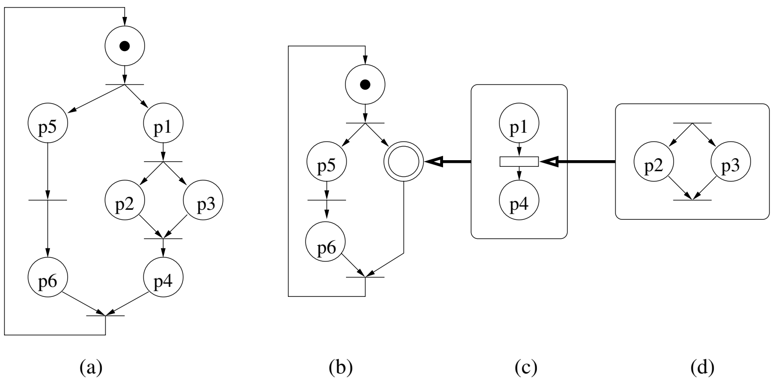


FIGURE 3.9 Hierarchical representation of a Petri net. The original Petri net is shown in (a). The hierarchical reduction (b) uses a double circle place to encapsulate the right branch of the Petri net. In (c), the internal structure of the double circle is shown with the original places p_1 and p_4 with the box transition as another hierarchical level. The internal structure of the box transition is shown in (d) with the original places p_2 and $p_{3.}$

1. A transition *t* is enabled if each input place *p* of *t* is marked with at least as many tokens as the weight of the arc joining them.
2. An enabled transition may or may not fire depending on whether or not the transition (event) actually takes place.
3. A firing of an enabled transition *t* removes $w(p, t)$ tokens from each input place *p* of *t*, and adds $w(t, p)$ tokens to each output place *p* of *t*.

The use of Petri nets in modeling manufacturing systems has several practical features. It can easily model causal dependencies, conflicts, synchronization, mutual exclusion, and concurrency. Some of these modeling capabilities are shown in Figure 3.8. Petri nets also have a locality property on places and transitions which enables hierarchical and modular constructions of complicated systems; a hierarchical representation of a Petri net is shown in Figure 3.9.

In Section 3.2, an example of a timing bar diagram for a transfer line was given. For simplicity, consider only the behavior of mill 1. The Petri net shown in Figure 3.10 describes its behavior as specified by the timing bar chart of Figure 3.4. Here places represent operations and transitions are enabled at the end of operations. A place with the notation "W" represents a waiting place; these places are very useful in modeling synchronization among operations. When the mill Petri net is combined with the clamp Petri net, as shown in the figure, the mill operation "rapid advance" cannot occur until the clamp has advanced due to the synchronizing transition. The two waiting places in the clamp Petri net indicate synchronizations with other parts of the machining system. The tokens are shown in their initial places, representing the starting moment of the timing bar chart.

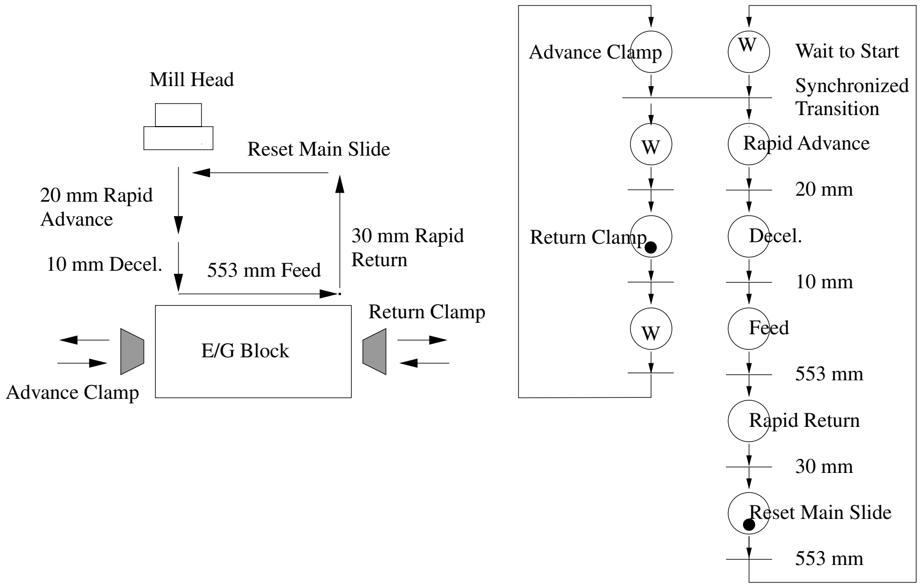


FIGURE 3.10 A Petri net implementation of the controller for mill 1 along with the corresponding clamp.

3.5.4.2 Analysis of Petri Net Models

As mentioned earlier, the qualitative properties that are especially important in Petri net models for manufacturing systems are liveness, boundedness or safeness, and reversibility. The formal definitions of these properties are omitted here, but their general meaning in logic controllers in manufacturing systems can be summarized as follows:[17,26]

1. Boundedness or safeness guarantees the stable behavior of the system without any overflow. The safeness property of the places which represent operations indicates there is no attempt to request execution of an ongoing operation. Another important implication of safeness is the Boolean representation of places, which enables a direct conversion from a Petri net to SFC as shown in Figure 3.11.
2. Liveness is equivalent to absence of deadlocks. This property guarantees that all transitions can be firable and that all operations or conditions represented by places can happen.
3. Reversibility characterizes the recoverability of the initial state from any reachable state of the system. It implies the cyclic behavior of a system and that it will perform its function repeatedly.

Petri net models of logic controllers can be formally analyzed to verify that the boundedness, liveness, and reversibility properties are satisfied. This verification process can guarantee that the corresponding manufacturing system exhibits the desired behavior. There are three approaches to the analysis of these qualitative properties: analysis by enumeration, analysis using linear algebraic techniques, and analysis by transformation.[8,16] The enumeration methods are based on the construction of the reachability graph or the coverability graph of the Petri net. The linear algebraic techniques use the state transition equation to represent the evolution of a Petri net and derive some invariant structures. The transformation method is based on simple reduction rules that preserve the important properties of Petri nets (boundedness, liveness, and reversibility); some simple reduction rules are presented graphically in Figure 3.12. The transformation procedure is iterative and applies the reduction rules until the reduced Petri net becomes irreducible. Generally, the first two techniques are limited by the complexity of the system. Although reduced Petri nets are irreducible, they may not be simple to analyze. One of the first two methods, however, can then

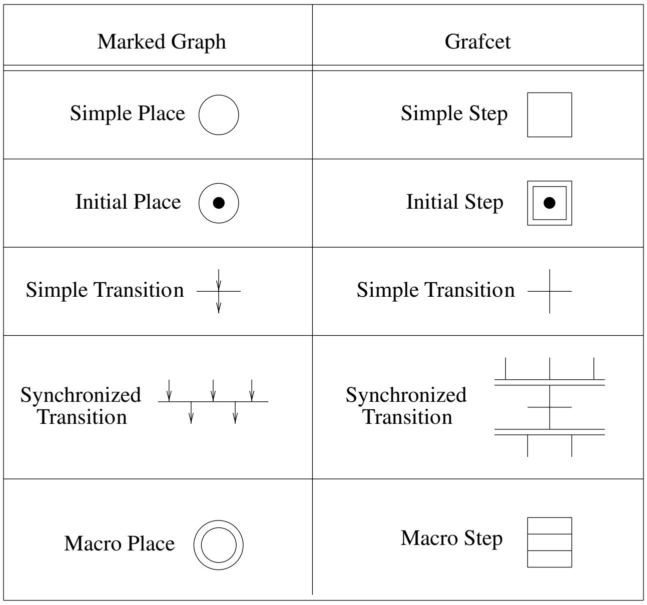


FIGURE 3.11 The conversion rules between a marked graph Petri net and Grafcet or SFC.

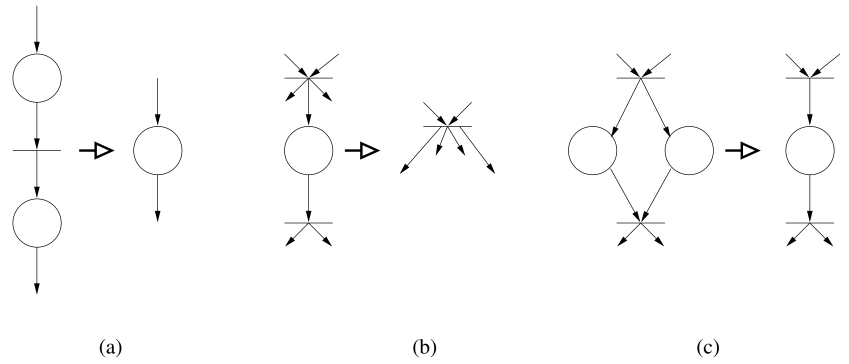


FIGURE 3.12 Simple reductions of Petri net models that preserve the properties of liveness, safeness (or boundedness), and reversibility: (a) fusion of series places, (b) fusion of series transitions, (c) fusion of parallel places.

be applied to the reduced model. In other words, these techniques are complementary and not exclusive.

Petri nets models can be categorized into several subclasses based on their structural characteristics. Analysis techniques are well developed for some subclasses of Petri nets, and the properties of Petri net models can easily be verified using these powerful structural results. For example, a Petri net is said to be *strongly connected* if there exists a directed path (sequence of places and transitions) from every place to every transition, and from every transition to every place. Many properties of Petri nets rely on the definition of *directed circuit*, a sequence of connected places and transitions with the final place being the same as the initial place.

The behavior of many manufacturing systems, including the high-volume transfer line shown in Figure 3.3, can be represented by a subclass of Petri nets called *marked graphs*. In a marked graph, each place p has exactly one input transition and exactly one output transition. Although transitions can have multiple input and output places, the marked graph formulation does not allow for contention for operations; the systems have no conflict and are decision free. The simpler structure of the marked graph allows many theoretical results to be derived.[8] For example, a marked graph is live if and only if its initial marking contains at least one token on each directed circuit. A live marked graph is safe if and only if there is exactly one token on each directed circuit. An initial marking of the marked graph which results in the graph being both live and safe can be found if and only if the graph is strongly connected. And finally, a marked graph is reversible if and only if it is live. Therefore, the verification procedure of safeness, liveness, and reversibility properties of Petri nets representing a manufacturing system and its logic control can be simplified if the system can be modeled using a marked graph or one of the other subclasses of Petri nets with well-understood structural behaviors.

3.6 Further Reading

Industry is currently moving toward open-architecture control systems for manufacturing automation. There are several national and international efforts to define, formalize, and institute an open-architecture standard. OSACA, which began as a European consortium, has focused on developing an open interface standard for proprietary control systems.[19] This open standard allows integration and communication between different systems; OSACA-compliant commercial systems are currently being produced. The North American effort, OMAC, grew out of a specification issued by the "Big 3" auto manufacturers in 1994.[18] The entire control system, from the interface to the factory network down to the servo control algorithm, must be open and user-modifiable. Although this architecture gives much more freedom and flexibility to the end-user, technical and business issues remain to be addressed before it becomes practical. Numerous control system companies promote varying degrees of openness in their products. For up-to-date information, the reader is encouraged to consult the web sites for the various open control consortia as well as the National Industrial Automation Show and Conference (in conjunction with *National Manufacturing Week*) and the International Automotive Manufacturing Conference sponsored by the Society of Automotive Engineers.

This chapter discussed only a few of the PLC languages currently in use in industry. The language of choice in a given factory depends on the industry (automotive, chemical, etc.) as well as the geographic location. An international standard, the IEC 1131, attempts to unify the many languages in use to enable conversion between them. More information can be found in the standard[12] and in textbooks.[14]

The logic controllers discussed in this chapter are typically implemented using digital computers. The field of real-time computer systems is focused on issues of operating systems, networks, applications programming, formal analysis, and design of algorithms with a focus on real-time issues. This area, which is directly relevant to the topic discussed here, is a very active area of research and development. The reader is referred to proceedings of the IEEE Real-Time Systems Symposium for recent developments in this field.

Within the field of control theory, there has been a lot of work on the theory of discrete event systems. This work is focused on fundamental concepts of controllability, observability, controller synthesis, etc. for discrete event systems. In this approach, the system to be controlled is modeled as a finite state machine with discrete event inputs and outputs, and closed-loop specifications are given in terms of the language generated by the machine. The reader is referred to recent books[3,13] for background in this area. The *IEEE Transactions on Automatic Control,* the *Journal of Discrete Event Systems,* and the *Proceedings of the IEEE Conference on Decision and Control should* be consulted for the latest developments in this field.

In this chapter, we have intentionally ignored the interactions between the logic controller and the servo controllers used to control continuous variables such as position, velocity, etc. Hybrid systems is an emerging field of research emplasizing systems that contain both continuous variables and discrete variables. The work in this area is focused on defining appropriate frameworks for analyzing and designing such hybrid systems. The interested reader is referred to the *Proceedings of the Workshop on Hybrid Systems*[10,11] for the latest developments.

Some of the formal methods for discrete event control described in this chapter are also used to control the scheduling of flexible manufacturing systems. Flexible systems, which produce many different types of parts on the same set of machines, are much more complex than the systems described thus far. In addition to the logic control for each machine, a supervisor or scheduler must determine which parts to send to which machine at which time. The supervisor tries to optimize the overall performance of the manufacturing system, but with unknown part mixes and potential machine breakdowns, the problem can become intractable.

Acknowledgments

We were introduced to the problems in discrete event control of manufacturing systems through the Engineering Research Center for Reconfigurable Machining Systems at the University of Michigan. We thank the director and co-director of this center, Yoram Koren and A. Galip Ulsoy, for giving us the opportunity to work in this exciting area. We also acknowledge our students who have worked with us on these problems: Eric Endsley, Dhrubajyoti Kalita, Morrison Lucas, and Euisu Park. In particular, Euisu Park supplied many of the figures used in this chapter. Credit also goes to the control engineers at Lamb Technicon, including Bryan Graham, Mike Griffin, and Matt VanGilder, who have shared their expertise in logic control with us.

References

1. S. Balemi, G. J. Hoffman, P. Gyugyi, H. Wong-Toi, and G. F. Franklin, Supervisory control of a rapid thermal multiprocessor, *IEEE Transactions on Automatic Control*, 38(7), 1040–1059, July 1993.
2. G. Barrett and S. Lafortune, Bisimulation, the supervisory control problem and strong model matching for finite state machines, *Journal of Discrete Event Dynamical Systems*, 8(4), 1998.
3. C. G. Cassandras and S. L. Lafortune, *Introduction to Discrete Event Systems,* Kluwer, Boston, 1999.
4. R. David, Grafcet: A powerful tool for specification of logic controllers, *IEEE Transactions on Control Systems Technology*, 3(3), 253–268, September 1995.
5. R. David and H. Alla, Petri nets for modeling of dynamic systems — A survey, *Automatica*, 30(2), 175–202, 1994.
6. Deneb Robotics, http://www.deneb.com.
7. A. A. Desrochers and R. Y. Al-Jaar, *Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis*, IEEE Press, Piscataway, NJ, 1995.
8. F. Dicesare, G. Harhalakis, J. M. Proth, M. Silva, and F. B. Vernadat, *Practice of Petri Nets in Manufacturing*, Chapman & Hall, New York, 1993.
9. C. H. Golaszewski and P. J. Ramadge, Control of discrete event processes with forced events. In *Proceedings of the IEEE Conference of Decision and Control*, 247–251, December 1987.
10. Hybrid Systems: Computation and control, The First International Workshop, HSCC'98, Berkeley, California, Springer, April 1998.
11. International Conference on Hybrid Systems, *Lecture Notes in Computer Science*, Springer, 1994, 1995, 1996.
12. International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages, IEC Standard 1131, Part 3*, 1993.
13. R. Kumar and V. K. Garg, *Modeling and Control of Logical Discrete Event Systems*, Kluwer, Boston, 1995.
14. R. W. Lewis, *Programming Industrial Control Systems Using IEC 1131-3*, Institution of Electrical Engineers, London, 1995.

15. F.-L. Lian, J. R. Moyne, and D. M. Tilbury, Performance evaluation of control networks: Ethernet, ControlNet, and DeviceNet, *IEEE Control Systems Magazine*, 21(3), 66-83, February 2001.
16. T. Murata, Petri nets: Properties, analysis and applications, *Proceedings of the IEEE*, 77(5), 541–580, April 1989.
17. Y. Narahari and N. Viswanadham, A Petri net approach to the modeling and analysis of flexible manufacturing systems, *Annals of Operations Research*, 3, 449–472, 1985.
18. Open modular architecture controls, http://www.arcweb.com/omac/.
19. Open system architecture for controls within automation systems, http://www.osaca.org.
20. G. Pritschow, A. Storr, and T. Jost, Simulation-based testing environment for master control systems, *Production Engineering*, IV(1), 51–54, 1997.
21. R. S. Raji, Smart networks for control, *IEEE Spectrum*, 31(6), 49–55, June 1994.
22. P. J. G. Ramadge and W. M. Wonham, Supervisory control of a class of discrete event processes, *SIAM Journal of Control and Optimization*, 25(1), 206–230, January 1987.
23. P. J. G. Ramadge and W. M. Wonham, The control of discrete event systems, *Proceedings of the IEEE*, 77(1), 81–98, January 1989.
24. M. Rausch and B. Krogh, Formal verification of PLC programs. In *Proceedings of the American Control Conference*, 234–238, 1998.
25. Sirius Systems, http://www.sirius.com.
26. M. C. Zhou, F. Dicesare, and A. A. Desrochers, A hybrid methodology for synthesis of Petri net models for manufacturing systems, *IEEE Transactions on Robotics and Automation*, 8(3), 350–361, June 1992.
27. R. Zurawski and M. C. Zhou, Petri nets and industrial applications: A tutorial, *IEEE Transactions on Industrial Electronics*, 41(6), 567–582, December 1994.