A Structured Adaptive Supervisory Control Methodology for Modeling the Control of a Discrete Event Manufacturing System

Robin G. Qiu, *Associate Member, IEEE,* and Sanjay B. Joshi, *Member, IEEE*

***Abstract*— Two basic measures, model complexity and model construction efficiency, are usually used to evaluate the implementability (or ease of use in practice) of a methodology for modeling the control of a discrete event manufacturing system (DEMS) on the shop floor. Many well-recognized methods are used to represent and analyze the dynamics of DEMS's, but not many relevant applications have been found in developing control software for the shop floor due to their shortcomings in satisfying these two measures. This paper explores a methodology for modeling the control of a DEMS, which leads to ease of control software development, rather than a new representational/analytical tool, by significantly reducing the model complexity (in terms of the number of required control states) and improving the model construction efficiency. First, an extended finite machine, called a deterministic finite capacity machine (DFCM) with parallel computing capability is developed. Based on DFCM's, the complexity growth function of a DEMS control model is linear in the number of synthesized control components. Then, an automaton structure of a DFCM control model, called structured adaptive supervisory control (SASC), is developed. By referring to supervisory control theory, an SASC model is created with three function layers: acceptance, adaptive supervision, and execution. The well-defined structure ensures that the control model can be constructed systematically.**

I. Introduction

SINCE a large portion of the cost of establishing a discrete event manufacturing system (DEMS) on the shop floor is consumed by its control system [1], significant research has been conducted to develop methodologies for modeling the dynamics of DEMS's [3], [5]–[7], [9], [10], [13], [14], [17]–[20], [25]. Existing approaches include Markov chains, queuing theory, Petri nets, and supervisory control theory [11]. Developed models using these approaches are widely applied for analyzing the behaviors of DEMS's, but very few of them have been transferred into manufacturing control systems due to their limited implementability (or ease of use in practice) on the shop floor [2], [16], [24], [25]. For a methodology to be applicable to developing control software on the shop floor it must support construction of a control model in an efficient and effective manner. The implementability of a methodology for the shop floor is usually evaluated by the following two basic measures:

1) Model Complexity—a quantitative measure. The complexity of a designed control model (in terms of the number of control states) should not exceed the limits of practical implementation.
2) Model Construction Efficiency—a qualitative measure. A methodology should be able to support systematic construction of a control model from start to finish without iterating through phases of trial, analysis, and redesign. The resulting design should satisfy all the specified control objectives.

Currently, Petri nets and supervisory control theory are popular approaches used to model and analyze the controls of DEMS's [6], [9], [10], [13], [17], [18], [25]. Because of the combinatorial explosion of solution complexity when the dynamics of a DEMS is modeled using these theories [7], [11], [18], [19], they are typically limited to creating control models of simple systems, such as small or medium-sized systems with a fixed part-mix and given processing routes [6], [13], [25]. For example, when modeling the control of a two-machine, two-robot, two-buffer, and two-part-type system using supervisory control theory, the size of the potential control state space is in excess of 10^{28} states [24]. Although the size of a Petri net model in terms of the number of control states can be controlled (i.e., it could grow linearly in the number of control components), the size of the reachability graph used for analysis to attain the final control model grows exponentially [7]. Therefore, the effort in resolving a control synthesis problem using either Petri nets or supervisory control theory can be extremely complex and easily go beyond a practitioner's ability.

The control model of a DEMS using Petri nets is based on trial, analysis, and redesign to converge to a model with the desired properties [6]. There has been some effort to investigate efficient modeling techniques using Petri nets [13], [25], but no methodology for systematically modeling the dynamics of a large-scale DEMS on the shop floor has been established. In contrast, supervisory control theory provides a systematic approach from start to finish, and unlike Petri nets it does not require iterating through phases of trial, analysis, and redesign, to modeling the control of a DEMS. Since supervisory control theory is a modeling methodology developed from the synthesis of control theory and automata

Manuscript received April 15, 1996; revised September 16, 1996; May 12, 1999; and July 20, 1999. This work was supported in part by NSF Presidential Young Investigator Award DDM9158042. This paper was recommended by Associate Editor C. Hsu.

R. G. Qiu is with the Factory Systems Division, Kulicke & Soffa Industries, Inc., Willow Grove, PA 19090 USA (e-mail: rqiu@eng.kns.com).

S. B. Joshi is with the Department of Industrial and Manufacturing Engineering, Pennsylvania State University, University Park, PA 16802 USA.

Publisher Item Identifier S 1083-4427(99)08394-0.

theory [17], [18], it guarantees that the designed model for controlling a DEMS yield the desired properties.

Although supervisory control theory provides a systematic method to model the control of a DEMS, the control-space explosion problem still limits its shop floor applications [2], [7], [24]. Hence, a methodology applicable to the manufacturing shop floor for the modeling and control of a DEMS, which leads to ease of software development rather than a general representational and analytical tool, is worth further exploration. In this paper, a two step approach is used to explore such a methodology by addressing these basic requirements carefully.

First, a modified finite machine, called deterministic finite capacity machine (DFCM), is systematically developed to model the dynamics of a DEMS. Automata and language theory is used to provide a firm mathematical foundation to study the logical behavior of a deterministic event system. Using these theories, the structural and behavioral properties of the formal model of a deterministic event system can be precisely defined and analyzed. Thus, like supervisory control theory, automata and language theory is used as the basis for the theoretical development of DFCM's. By capturing the specific characteristics of a manufacturing control system and combining the technological advances in multi-process operating systems, a DFCM is developed with the capability of running multiple computations in parallel. Consequently, the control-state-space explosion problem is resolved successfully.

Secondly, an automaton structure of a DFCM control model called structured adaptive supervisory control (SASC) is developed for describing the dynamics of a DEMS. By referring to supervisory control theory, an SASC model is defined with three function layers: acceptance, adaptive supervision, and execution. The well-defined structure ensures that the SASC model can be constructed systematically.

The remaining paper is organized as follows. Section II analyzes the specific characteristics of manufacturing automation, from which the requirements for control are derived. Section III reviews the basic terminology and notation of the theory of automata and languages and provides preliminaries for this paper. Section IV develops the theory of deterministic finite capacity machines. Section V systematically investigates the structured adaptive supervisory control. Section VI shows a typical implementation of the developed methodology. Finally, conclusions of this paper are presented in Section VII.

II. Specific Characteristics of Manufacturing Automation

A DEMS is composed of finite asynchronous equipment components [4], [6], [11], [17]–[19], [25]. The control state space of the synthesized control model of a DEMS suffers from exponential growth in the number of components [19]. Take for example, an abstracted manufacturing system consists of m machines. The controller for this system must be formed in such a way that all the control states of individual asynchronous machines are synthesized and the desirable control properties are acquired. Consequently, a combinatorial explosion in terms of the number of control states arises naturally as the number of machines and parts increases [11], [15], [18]. That is, the size of the control space will be the product of $|M_1|\times|M_2|\times\cdots|M_m|$, where M_i for $1 \le i \le m$ is a component of the DEMS, and $|M_i|$ represents the cardinality of the control state space of component M_i. Obviously, as the size of the DEMS grows, the control problem of the DEMS will eventually become too large and practically unresolvable on the shop floor with the enabling technologies (although it is resolvable in theory). This type of system and explosion is classified as an NP-hard problem and cannot be resolved optimally without a new approach to reduce the size of constrained control state space [22].

Delving into the operations of a DEMS reveals some unique characteristics, which are worthy of consideration during the design of a control model. All the asynchronous events (except those concerning the loss or recovery of machine capability in a DEMS) are associated with on-line part states. In other words, a machine state or event can be mapped into a part state, while the part state is normally easy to trace [20]. This characteristic in manufacturing automation is called *part traceability*. The dynamics of a DEMS can be then modeled by describing all the possible part states instead of all the possible machine states within the DEMS. As a result, the control issue of a DEMS becomes one that all the on-line part states should be dynamically and cooperatively changed as desired.

A part advancing through a DEMS can be described by its part flow—a diagram showing the sequence of part states required to process this part within the DEMS [16]. If a language L is used to represent all the legal sequences of part states for these desirable families of parts manufactured in a DEMS, a string from the language L provides one legal sequence of part states for a particular part. For each part, there exists at least one string in L describing how to make it. In terms of modeling the control of the DEMS using automata theory, when only one part enters the DEMS, a recognizer G for the language L should describe (recognize) the trace of the part advancing through the DEMS. But if the DEMS is machining multiple parts, all the separate part traces are then required to be recognized simultaneously. According to this observation, a control system can be considered as a recognizer G_s capable of recognizing a shuffled language L_s, where $L_s = \{x: x = x_1 \|_s \cdots \|_s x_i \cdots \|_s x_m$ for $1 \le i \le m$ and $x_i \in L\}$, $\|_s$ is the natural shuffle operation, and m is the maximum number of parts being machined within the DEMS.

Theoretically, when m languages are shuffled and n is the average number of control states required for a recognizer to recognize a language, the number of control states required for a recognizer to recognize the shuffled language is $O(n^m)$. In other words, even though the dynamics of a DEMS is described by the set of all the part traces (instead of the set of all the legal machine-event sequences), the complexity (in terms of the number of control states) of the DEMS recognizer G_s could be the same as that of a control system recognizing all the machine-event sequences.

However, it is worth noting that all the strings (part traces) of a DEMS are from the same language L representing all possible part-flows within the DEMS. As discussed, it is possible to construct a new recognizer G_s which recognizes

the shuffled language L_s. But instead, one could form a coordinator which coordinates all the necessary recognizers G_i for $1 \leq i \leq m$, where $G_i = G$, and each G_i is required for tracing an individual part ($x_i \in L_i \equiv L$). (A part is physically traceable.) Thus, the control model space of a DEMS will avoid the control-state-space explosion problem associated with constructing a recognizer for L_s. It is the use of the concepts of coordination of multiple computations and part traceability that provides the foundation for the solution approach, presented in this paper, to modeling the control of a DEMS on the shop floor.

III. Terminology and Notation [8], [12]

A *machine* $\mathcal{M}$ is an ordered tuple of *devices* d_i for $i = 1, 2, \ldots, k$, specifically denoted by $[d_1, d_2, \ldots, d_k]$. The class of devices consists of control, input, output, stack, queue, tape, etc. A machine type is defined by a fixed combination of different devices. If machine $\mathcal{M}$ has no storage devices other than the control, it is called a *finite machine*. A machine state is called a *configuration* of a machine, which is the aggregate of all information stored by the machine's devices. Formally, a configuration is a k-tuple of states $C = (c_1, c_2, \ldots, c_k)$; it specifies that the state of device d_i is c_i for $i = 1, 2, \ldots, k$.

A program $\mathcal{P}$ for a machine $\mathcal{M}$ comprises an *initializer* α, a *terminator* ω, and a finite *instruction* set $\mathbf{I}$ for machine $\mathcal{M}$.

An initializer, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$, for machine $\mathcal{M}$ is an injection which maps an argument to the initial configuration of machine $\mathcal{M}$. Typically $x \in \Sigma^*$, where Σ is an input alphabet, denotes an argument to program $\mathcal{P}$, and $C_0 = (c_{1,0}, c_{2,0}, \ldots, c_{k,0})$ denotes the initial configuration for machine $\mathcal{M}$. The relation α is defined as

$$x\alpha = (x\alpha_1, x\alpha_2, \ldots, x\alpha_k), \quad \text{i.e. } x \overset{a}{\mapsto} C_0.$$

Similarly, a terminator, $\omega = (\omega_1, \omega_2, \ldots, \omega_k)$, for machine $\mathcal{M}$ is a partial function which determines whether a language is accepted or whether a transduction is completed. It is worth noting that no result is produced unless all devices are in a final state. We denote $y \in \Delta^*$ as some result, where Δ is an output alphabet, and $C_f = (c_{1,f}, c_{2,f}, \ldots, c_{k,f})$ as a final configuration for machine $\mathcal{M}$. Then if the terminator ω maps the configuration C_f to some result y, the relation ω is defined as

$$C_f\omega y \Leftrightarrow c_{1,f}\omega_1 y \quad \text{and} \quad c_{2,f}\omega_2 y \quad \text{and} \quad \cdots \quad \text{and} \quad c_{k,f}\omega_k y, \quad \text{i.e., } C_f \overset{\omega}{\mapsto} y.$$

An operation for each device of machine $\mathcal{M}$ is designated by an instruction. An instruction maps a configuration C of machine $\mathcal{M}$ to another configuration C'. This mapping can be completed in one step iff there exists some instruction $\pi \in \boldsymbol{I}$ such that $C\pi C'$. This mapping can be simply denoted as $C\Pi C'$ where $\Pi = \cup_{\pi \in I}\pi$. If a finite number of steps is required to complete this composite relation, it is denoted as $C\Pi^* C'$. A *computation* is the sequence of instructions $\langle\langle \pi_1, \pi_2, \ldots, \pi_n, \ldots \rangle\rangle$ executed when program $\mathcal{P}$ is run on machine $\mathcal{M}$. Therefore, a complete computation of $\mathcal{P}$ on argument x with result y exists iff an $x\alpha\Pi^*\omega y$ also exists. The relation $\alpha o \Pi^* o \omega$, which relates arguments of $\mathcal{P}$ to results of $\mathcal{P}$, is called as the *transfer relation* of $\mathcal{P}$, and is denoted by

$$\tau = \alpha o \Pi^* o \omega, \quad \text{i.e. } x \overset{\tau}{\mapsto} y.$$

As long as program $\mathcal{P}$ is deterministic and has no null instructions, the τ of $\mathcal{P}$ is a partial function.

If a program is faced with a choice of which instruction to perform next or whether to continue or terminate, the program is *nondeterministic*. The program is *deterministic* only if all the behavior of the program is precisely determined.

When a program $\mathcal{P}$ is installed on a finite machine $\mathcal{M}$, machine $\mathcal{M}$ is an *operational finite machine* [5], which can be formally denoted as

$$M = (Q, \Sigma, \Delta, \mathbf{I}, q_0, Q_f),$$

where

Q	finite control set;
Σ	input alphabet;
Δ	output alphabet,
$\boldsymbol{I}$	finite instruction set;
$q_0 \in Q$	initial control state;
$Q_f \subseteq Q$	final control set.

If a program $\mathcal{P}$ on machine $\mathcal{M}$ is deterministic and only determines the membership of an input string, i.e., $x \in L(M) \Leftrightarrow x \overset{\tau}{\mapsto}$ ACCEPT, $L(M)$ is used to denote the language accepted by program $\mathcal{P}$ on machine $\mathcal{M}$ and machine M is called as a *deterministic finite acceptor*.

If a deterministic finite acceptor replaces its input by an output, then the acceptor becomes a *deterministic finite generator*. The generator is defined to be a deterministic automaton by including the marking concept [17], $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is the state space, Σ is the alphabet or set of output symbols σ, $\delta : \Sigma \times Q \to Q$ is the transition function (pfn), $q_0 \in Q$ is the initial state, and $Q_m \subseteq Q$ is the set of marked states. The language generated by G is

$$L(G) = \{s \in \Sigma^* : \delta(s, q_0) \text{ is defined}\}.$$

The language marked by G is

$$L_m(G) = \{s \in L(G) : \delta(s, q_0) \in Q_m \text{ and } \omega \text{ is well-behaved}\}.$$

If program $\mathcal{P}$ on machine $\mathcal{M}$ is deterministic and maps an input string to an output string in one complete computation, i.e. $x \overset{\tau}{\mapsto} y$, $x \in \Sigma^*$, $y \in \Delta^*$, program $\mathcal{P}$ on machine $\mathcal{M}$ completes a finite transduction and machine M is called a *deterministic finite transducer*.

If machine M has a complete computation on argument x, then machine M *halts* on x. If machine M is *blocked*, the current configuration of machine M is neither in the domain of any instruction nor in the domain of ω.

IV. Finite Capacity Machines

A. Basic Concepts and Representations

For a finite machine a task $t \in \mathcal{T}$ (a set of tasks) is a specified string which requires computing to determine whether it belongs to the domain of a language. A *process* is

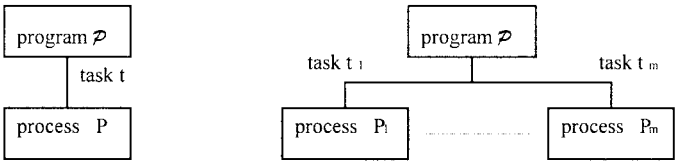


Fig. 1. Single process versus multiple processes on a finite machine.

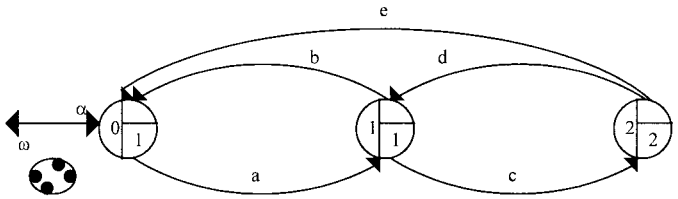


Fig. 2 State transition diagram of a finite capacity machine.

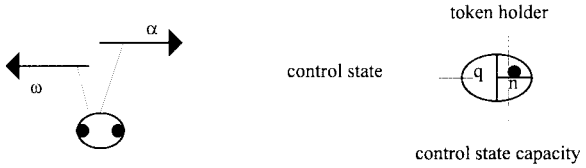


Fig. 3. Basic representations.

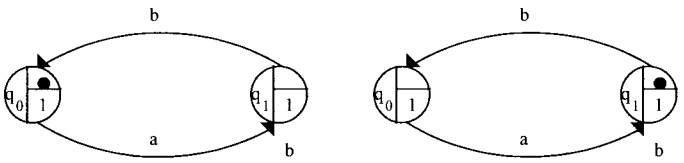


Fig. 4. One basic transition.

a program whose computation has started but not terminated. Using the analogy of computer operating systems, MS-DOS supports one process, while UNIX supports multiple processes. As a finite machine can have only one process at a time [Fig. 1(a)], a finite machine should be able to be extended into a new finite capacity machine capable of running multiple processes simultaneously [Fig. 1(b)].

A finite capacity machine can be considered as an aggregate of multiple identical traditional finite machines. To a finite capacity machine, each finite machine seems like a process; each process has its unique input and output devices [16]. To identify a process, a digitized token is used. The formal definition of a finite capacity machine is then given as follows.

Definition 1: A *finite capacity machine* (FCM) is formally defined by

$$M = \left(Q, Q', Q_t^m, \Sigma^m, \Delta^m, \boldsymbol{I}, q_0, Q_f\right)$$

where

- Q finite control set, which includes all the main control states for M;
- Q' extra finite control set, called *state-availability control* set, $q'_q \in Q' = \{0,1\}$, $q \in Q$. If $q'_{q_1} = 1$, then the control state q_1 is available for a transition from q, otherwise control state q_1 is not available;
- Q_t^m multiple-extra control set, called *digitized token control* set, which is the Cartesian product of all the digitized token control sets, $Q_t^m = \times_{i=1}^m Q_t^i$, $q_{tq}^i \in Q_t^i = \{0,1\}$, $q \in Q$. If $q_{tq}^i = 1$ for $i = 1, \ldots, m$, then there is a digitized token in state q, otherwise the ith token is not in state q;
- Σ^m multiple-input alphabet, the Cartesian product of all the inputs, i.e., $\Sigma^M = \times_{i=1}^M \Sigma_i$, $\Sigma_i \subseteq \Sigma$ (an input alphabet) for $i = 1, \ldots, m$;
- Δ^m multiple-output alphabet, the Cartesian product of all the outputs, i.e., $\Delta^m = \times_{i=1}^m \Delta_i$, $\Delta_i \subseteq \Delta$ (an output alphabet) for $i = 1, \ldots, m$;
- $\boldsymbol{I}$ instruction set;
- $q_0 \in Q$ initial control state;
- $Q_f \subseteq Q$ final control set.

Like a finite machine, a finite capacity machine can also be graphically represented by a state transition diagram. The diagram consists of a finite number of nodes and a finite number of directed arcs. All the arcs except these representing the initializer and the terminator are interpreted as instructions $\pi \in \boldsymbol{I}$. All the nodes are interpreted as the main control states $q \in Q$. A typical state transition diagram for a machine capable of running four processes is shown in Fig. 2. The nodes labeled 0, 1, 2 correspond to the main control states of the machine. Node 0 is the initial and final state. The meanings of these symbols in this diagram will be explained shortly.

If each of the main control states is considered as a type of resource whose capacity is limited, then a computation can be interpreted as a sequence of resource uses. Let the capacity of a resource $(q_i \in Q)$ be r_i $(r_i \geq 1)$ for $i = 1, \ldots, n$, where $n = |Q|$. The finite machine capacity will be $R = \sum_{i=1}^n r_i$. Therefore, the number of allowed processes should be not greater than the finite machine capacity, i.e., $m \leq R$.

A process obtains a unique digitized token when the process is initialized by the initializer α, and returns the token when the process is terminated by the terminator ω [Fig. 3(a)]. A token booth is used as the provider and collector of tokens. The number of tokens in the token booth can be any positive number not greater than R. Fig. 3(b) shows the three components of a node: control state, token holder, and control state capacity. The capacity of a token holder is equal to its control state capacity, i.e., resource capacity. A basic transition of an FCM M is shown in Fig. 4, where M completes an instruction

$$\pi = \left\{q_0 \to q_1, a, \left(q_{q_0}^i = 0, q_{q_1}^i = 1\right) \to \left(q_{q_0}^i = 1, q_{q_1}^i = 0\right), \right.$$
$$\left. q_{tq_o}^i \to q_{tq_i}^i\right\}, \quad \text{for some } i = 1, \ldots, m.$$

If the configuration of machine M is represented by $C = (Q, Q', Q_t^m)$ [8], then this transition changes the configuration from $(q_0, (0,1), (1,0))$ to $(q_1, (1,0), (0,1))$. A transition can be fired from q_0 to q_1 iff q_1 is available, i.e., $q'_{q_1} = 1$.

The use of the token and availability concepts may make an FCM look similar to a Petri net. In fact, a token in an FCM is totally different from one in Petri nets. In a Petri net, a token represents the availability of a place. When a token exists in a place, it means a condition in the place is satisfied

and ready for next transition. A transition can be fired iff all the required conditions for firing it are satisfied. Tokens are unceasingly diverged or converged during computation [6], [13], [25]. Thus, the physical meaning of a token also varies with places. But in an FCM a token represents a physical entity (e.g., part in a DEMS). It never changes its physical representation during computation. A token also identifies a computation process. The concept of a process tracing the entity (token) in an FCM is unique, and neither a Petri net nor a supervisory control model includes this concept.

B. Coordination of Computing Processes

When only one individual process is run on an FCM M, machine M will function exactly the same as a finite machine. However when multiple processes are computing simultaneously, the processes compete for resources with each other. The fixed capacity of a token holder will cause conflicts if too many processes attempt to transfer into the same control state. It is obvious that effective coordination between different processes can avoid these conflicts. By referring to the coordination theory for intelligent machines [23], a method to coordinate different processes is studied. To investigate how processes can coordinate with each other, it is necessary to understand how an individual process works in a parallel-computing environment.

The definition of an individual process on an FCM M is as follows.

Definition 2: One process P_i for some $1 \le i \le m$, running on a finite capacity machine M, can be formally defined by

$$P_i = \left(Q, Q', Q_t^i, \Sigma^i, \Delta^i, \mathbf{I}, q_0, Q_f\right)$$

where Q, Q', $\mathbf{I}$, q_0, Q_f are defined the same as these in definition 1

- Q_t^i extra control set, called a digitized token control set, $q_{tq}^i \in Q_t^i = \{0,1\}$, $q \in Q$;
- Σ^i input alphabet;
- Δ^i output alphabet.

If only a deterministic FCM (DFCM) is considered, then the behavior of a process running on the DFCM M is precisely determined by the instruction set $\boldsymbol{I}$ for $1 \le i \le m$. (Although a nondeterministic finite machine can simplify machine construction and provide sufficient problem-solving information, it is impossible to physically build an operational machine based on a nondeterministic finite machine.) The instruction set $\boldsymbol{I}$ [8], [12], [17] can be further defined as

$$\boldsymbol{I} = \begin{cases} \delta : \Sigma^i \times Q \times Q_t^i \to Q \times Q_t^i, & \text{state transition function (pfn),} \\ g : \Sigma^i \times Q \to \Delta^i, & \text{output function (pfn).} \end{cases} \tag{1}$$

An augmented state transition function $\delta_a : \Sigma^i \times Q \times Q' \times Q_t^i \to Q \times Q' \times Q_t^i$ can be defined according to

$$\delta_a\left(\sigma, q \times q_q' \times q_{tq}^i\right) = \begin{cases} \delta\left(\sigma, q \times q_{tq}^i\right) & \text{if the codomain of } \delta_a : \Sigma^i \times Q \times Q' \times Q_t^i \\ & \quad \to Q \times Q' \times Q_t^i \text{ satisfies } q_q' = 1, \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2}$$

For each complete computation of a process on the DFCM M, one can say that an assigned task is performed successfully, i.e., there is a

$$x \overset{\tau}{\mapsto} y, \quad \tau = \alpha o \Pi^* o \omega$$

where $x \in (\Sigma^i)^*$, $y \in (\Delta^i)^*$, and $\Pi = \cup_{\pi \in I} \pi$.

Note that a state transition occurs iff $\delta_a : \Sigma^i \times Q \times Q' \times Q_t^i \to Q \times Q' \times Q_t^i$ for $1 \le i \le m$ is defined. For a multiple process FCM, if a requested state satisfies $q_q' = 0$, it simply means that the requested state is occupied by some other process. In other words, the requesting process should be kept waiting in its current state until the requested state is released, i.e., q_q' turns to 1.

In the case of a tie (multiple processes requesting the same resource simultaneously), the conflict can be resolved using their token values as their priorities. For instance, a process with the highest priority will execute first. The priority setting can be transformed into a scheduling problem (beyond the scope of this paper). The token value of a process can be modified internally or externally without affecting the computation of the process. Therefore, it is the existence of a transition-availability control set and the well-defined augmented state transition function that ensure that all the processes started can run in a coordinated manner.

After the concept and coordination mechanisms of processes have been exploited, a DFCM M can be simply defined by

$$M = (P_1 \parallel P_2 \parallel \cdots \parallel P_m)$$

where $\parallel$ is a parallel line which indicates the concurrence of distinct processes. The configuration of the DFCM M will then be described as

$$C = C_1 \langle + \rangle C_2 \langle + \rangle \cdots \langle + \rangle C_m,$$

where $\langle + \rangle$ is a disjoint configuration addition through distinct processes. More explicitly, if $C = (Q, Q', Q_t^m)$, then

$$C = \left(\begin{pmatrix} q_i \\ q_j \\ \cdot \\ \cdot \\ \cdot \\ q_k \end{pmatrix}, \left(q_{q_0}', q_{q_1}', \ldots, q_{q_n}'\right), \begin{pmatrix} (e_{10}, e_{11}, \ldots, e_{1k}) \\ \cdot \\ \cdot \\ \cdot \\ (e_{m0}, e_{m1}, \ldots, e_{mk}) \end{pmatrix} \right)$$

where, $e_{ij} \in (0,1)$, $\sum_{j=0}^{k} e_{ij} = 1$, $1 \le i \le m$, $\sum_{i=0}^{m} e_{ij} = 1$, $1 \le j \le k$, $1 \le i, j, k \le n = |Q|$.

By summarizing the above discussions, the following four remarks are provided to elucidate the operational properties of a DFCM:

Remark 1: A transition of a process on a DFCM can be made iff the transition is defined and the next state is available.

Remark 2: $q' \in Q' = \{0,1\}$ can be set internally or externally. When the number of tokens in control state q_i for $i = 1, \ldots, n$ is equal to the capacity of its token holder, i.e. $\sum_{j=1}^{m} q_{tq_i}^j = r_i$, $q_{q_i}' \in Q'$ is set to 0 internally. $q_{q_i}' = 0$ will be held until one token is passed on to another control state. Besides, if the token holder loses its capability, as in the case of machine breakdown, q_{q_i}' can be set to 0 externally. In this case, $q_{q_i}' = 0$ will be held until its capability is recovered (fixed).

Remark 3: $q_t^i \in Q_t^i = \{0,1\}$ for $i = 1, \ldots, m$ keeps track of the ith token. The token has a unique identification number (e.g., a valid integer) given when the ith process gets started. The token moves to next node iff the associated transition has been completed. The location of a token shows the state of a process.

Remark 4: Whenever a process is initialized, it obtains a token. A process will terminate when it completes its computation. At that time, the process returns its token. If no tokens are available in a DFCM, the DFCM cannot accept any more tasks. Since the number of tokens is finite, the capacity of a DFCM is therefore limited.

C. Language Representations

If a program $\mathcal{P}$ on machine $\mathcal{M}$ is deterministic and each process P_i of a DFCM M for $1 \le i \le m$ only determines the membership of an input string, i.e.

$$x \in L(P_i) \Leftrightarrow x \overset{\tau}{\mapsto} \text{ACCEPT},$$

$L(P_i)$ is used to denote the language accepted by process P_i, $L(M)$ to denote the language accepted by program $\mathcal{P}$ on machine $\mathcal{M}$, and machine M is called a *deterministic finite capacity acceptor* (DFCA).

If a program $\mathcal{P}$ on machine $\mathcal{M}$ is deterministic and each process P_i of a DFCM M for $1 \le i \le m$ tests the membership of an input string within a finite computation, i.e.

$$x \in L(P_i) \Leftrightarrow x \overset{\tau}{\mapsto} \text{ACCEPT}, \quad \text{and}$$
$$x \notin L(P_i) \Leftrightarrow x \overset{\tau}{\mapsto} \text{REJECT}$$

$L(P_i)$ is used to denote the language recognized by process P_i, $L(M)$ to denote the language recognized by program $\mathcal{P}$ on machine $\mathcal{M}$, and machine M is called a *deterministic finite capacity recognizer* (DFCR).

If program $\mathcal{P}$ on machine $\mathcal{M}$ is deterministic and each process P_i maps an input string to an output string in one complete computation, i.e. $x \overset{\tau}{\mapsto} y$, $x \in \Sigma^*$, $y \in \Delta^*$, program $\mathcal{P}$ on machine $\mathcal{M}$ completes a finite transduction and machine M is called a *deterministic finite capacity transducer* (DFCT).

For each process P_i for $1 \le i \le m$ of a DFCA, if the process finishes a complete computation, then an assigned task is successfully completed. The language (set of tasks) accepted by the process of the DFCA is

$$L(P_i) = \{x : x \in (\Sigma^i)^* \text{ and } x \overset{\tau}{\mapsto} \text{ACCEPT}\}.$$

Proposition 1: The language of a process P_i for $1 \le i \le m$ of a DFCM M is equivalent to the language of the DFCM M.

Proof: Based on Definition 1, the language of one process P_i has the same language as that of another process P_j $(i \ne j, 1 \le i \le m, 1 \le j \le m)$ since all the processes are running the same program on M

$$L(P_i) = L(P_j) \quad \text{for } 1 \le i, \quad j \le m.$$

Therefore,

$$L(M) = \bigcup_{1 \le i \le m} L(P_i)$$
$$L(M) \equiv L(P_i) \quad \text{for } 1 \le i \le m.$$

Since a DFCM can be considered as an aggregate of multiple deterministic finite state machines, the structure of the DFCM can be further simplified for the purpose of analysis. Based on Proposition 1, the structure of a DFCM can be defined as the structure of an one-process DFCM but with a token control set. Thus, the language of a DFCM then can be defined in a concise manner.

Definition 3: The language that can be accepted by a DFCA can be defined as the language accepted by $M = (Q, Q', Q_t^m, \Sigma, \boldsymbol{I}, q_0, Q_f)$, i.e.

$$L(M) = \{x : x \in \Sigma^* \text{ and } x \overset{\tau}{\mapsto} \text{ACCEPT}\}.$$

When L physically represents all the process plans required to produce parts, a string in L is a sequence of operations required for completing a part. Since each type of part can be made by following different process plans, a task equivalence class or coset in terms of process plans (strings) can be identified according to the following two definitions [8], [16]. More specifically, all of the possible processing alternatives defined for processing a part will be included in a task equivalence class. Therefore, no matter which alternative is chosen from this partitioned coset, its computation can be completed and the task can be performed successfully.

Definition 4 (Task Equivalence): Let L be a language of an FCM. Two strings x_1 and x_2 are task equivalence with respect to L (denoted as $x_1 \overset{t}{\approx}_L x$) if

$$(\forall u \in \Sigma^*)(\forall v \in u \,\|_s \varepsilon^*)[x_1 u \in L \Leftrightarrow x_2 v \in L],$$

where ε represents the set of characters physically completing none of useful tasks (e.g., a part being sent to a buffer and waiting for further processing or inspection).

Definition 5 (Task Equivalence Class): Let L be a language of an FCM. The class of task equivalence will be $[x] = \{x' : x' \overset{t}{\approx}_L x\}$.

Theorem 1: If L is a shuffled language of $L_1, L_2, \ldots, L_m$, where $L_i = L_j$ for $1 \le i, j \le m$ and L is recognized by a deterministic finite recognizer M, then there exists a DFCR R which is at least as powerful as the recognizer M.

Proof:

1) For simplicity, first assume $m = 2$, then $L = L_1 \,\|_s\, L_2$ $(L_1 = L_2)$ is defined by

$$L = \{s: s = x_1 y_1 \ldots x_i y_j \ldots x_n y_n, x = x_1 \ldots x_i \ldots x_n,$$
$$y = y_1 \ldots y_j \ldots y_n, \text{ and } x \in L_1, y \in L_2\}$$

where x, y can be strings of any length, x_i, y_j for $1 \le i, j \le n$ can be either characters or empty. Furthermore, assume that M recognizes L. Then R can be constructed

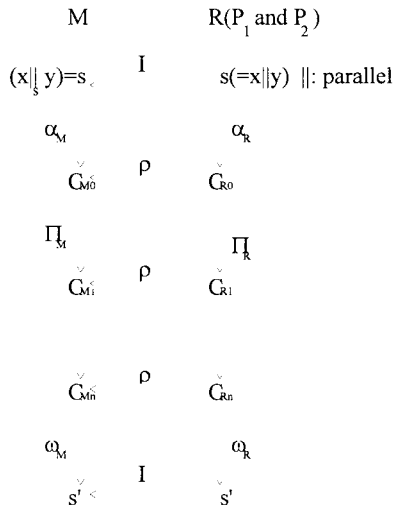


Fig. 5. R lockstep simulates M.

as a two-process (P_1 and P_2) DFCR (R's program P recognizes L_1 or L_2, $L_1 = L_2$). The representation relation ρ of R and M configurations can be defined as

$$C_R \rho C_M = \left(\binom{q_i}{q_j}, (q_{q_0}, q_{q_1}, \ldots, q_{q_n}), \left(\begin{matrix} (0,0,\ldots,1_{i\text{th}},\ldots,0) \\ (0,0,\ldots,1_{j\text{th}},\ldots,0) \end{matrix} \right) \right) \rho q.$$

If x_i for $1 \le i \le n$ is always marked as a red character, and y_j for $1 \le j \le n$ as a blue character, then whenever R receives a red character, P_1 computes; otherwise P_2 computes (Fig. 5). If M accepts s, then R accepts both x and y; if M rejects s, then R rejects both x and y.

2) In general, m can be any integer number. By marking all the characters of a string from a different language in a unique color and constructing R using the above techniques, R can be formed, which simulates M in a lockstep manner. Therefore, there exists a DFCR R which is at least as powerful as M [8].

Theorem 2: When a DEMS is modeled by a DFCR, the number of control states in the DFCR grows linearly in the number of resources of the DEMS.

Proof: Note that each of the main control states in a DFCR can be considered as a kind of resource, then a computation can be interpreted as a sequence of resource uses. Assume a DFCR is defined as $M = (Q, Q', Q_t^m, \Sigma, \boldsymbol{I}, q_0, Q_f)$. If $|Q| = n$, then n indicates the number of resources. According to Proposition 1, $L(M) = L(P_i)$ for $1 \le i \le m$. The proof of this theorem can be also divided into two steps.

1) For simplicity, first assume $m = 2$, then the shuffled language of L_1 and L_2 will be $L_1 \|_s L_2 = \cup_{x \in L_1, y \in L_2} x \|_s y$. If the recognizer of L_1 has n_1 control states and the recognizer of L_2 has n_2 control states, the recognizer of $L_1 \|_s L_2$ will need $O(n_1 n_2)$ control states according to Myhill–Nerode and Pumping theorems [8]. If $L_1 = L_2$, then $n_1 = n_2 = n$. Since M is capable of computing $x \in L_1$ and $y \in L_2$ simultaneously without shuffling x and y as one input string, the size of M in terms of the number of control states is $O(n)$ rather than $O(n^2)$. When n increases, the size of M increases with n proportionally, i.e., $O(n)$.

2) In general, if a recognizer needs to recognize m languages, i.e. the shuffle of $L_1 \|_s L_2 \cdots \|_s L_i \cdots \|_s L_m$ for $1 \le i \le m$, it will requires for $O(n^m)$ control states, where n is the average number of control states for the recognizer of L_i for $1 \le i \le m$. M is capable of computing m strings by running m processes simultaneously. These m processes are running in a coordinated manner so that all the computations will be complete if those strings are legal. Thus M is constructed to recognize L_i ($L_i = L_j$ for $1 \le i, j \le m$) rather than $L_s = L_1 \|_s L_2 \cdots \|_s L_i \cdots \|_s L_m$ for $1 \le i \le m$. Therefore, when n increases, the size of M still increases proportionally with n, i.e., $O(n)$.

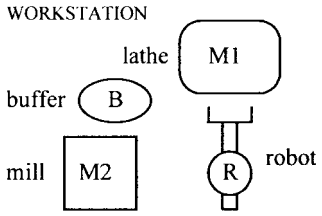


Fig. 6. Typical workstation.

D. Adjustable Transitions

Assume that each node in a DFCA state transition diagram represents one resource in a DEMS. When $q'_{q1} = 0$, the control state q incident to control state q_1 will be held. In other words, since the transition of control state q is not available, the process will be put in its state waiting for the resource. If a resource is in full operations and all the users of other resources are in waiting states, then no additional state transition can occur. This situation results in the low utilization of the resource, potentially even a system deadlock for the DEMS if the resource is pivotal, such as a robot, and any other kind of material handler. To avoid this undesirable situation, a distinguishable subset of the input alphabet Σ_a is defined. Like supervisory control [17], [18], an operation $(\sigma, q \times q', q_1 \times q'_1)$ is an adjustable operation if $\sigma \in \Sigma_a \subset \Sigma$. An adjustable operation is graphically denoted as $- \mapsto$. Let

$$\Gamma = \{1\}^{\Sigma_a} \cup \{0\}^{\Sigma - \Sigma_a}$$

be the set of assignments to the elements of Σ. Then a total function $\gamma : \Sigma \to \{0, 1\}$ holds, which is an adjustable pattern.

Example 1: An automated manufacturing workstation consists of two machines M_1 and M_2, one robot R and one buffer B (Fig. 6). Assume that each of these resources has a capacity of one and the robot can access all the other resources. The controller of this workstation can be then constructed as a DFCA whose state transitions are shown in Fig. 7. The state transitions (a, b, c, d, e, and f) physically correspond to all

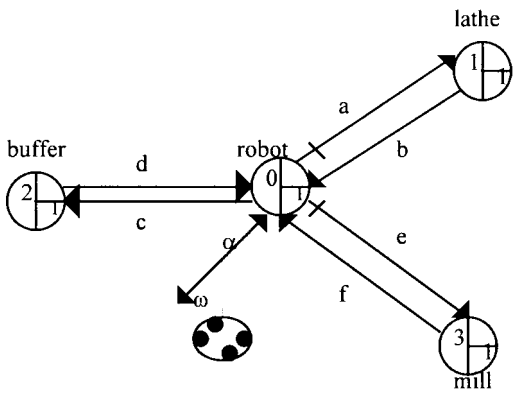


Fig. 7. State transitions of example 1's controller.

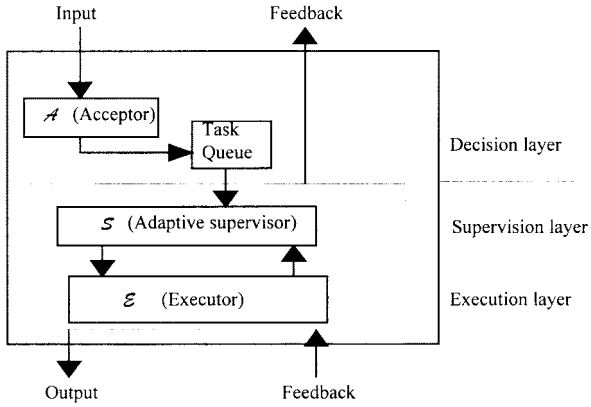


Fig. 8. Well-structured SASC model.

the possibilities of part-flows within the workstation during production.

Formally, the controller is defined by $M = (Q, Q', Q_t^m, \Sigma, \delta_a, q_0, Q_f)$, where

Q $= \{0,1,2,3\}$;
Q' $= \{0,1\}$;
Q_t^m 4×4 token matrix;
Σ $= \{a,b,c,d,e,f\}$;
Σ_a $= \{a,e\}$;
$\Sigma - \Sigma_a$ $= \{b,c,d,f\}$;
δ_a defined as (2);
q_0 $= 0$;
Q_f $= \{0\}$;
ε $\{c,d\}$;
Γ $= \{1\}^{\Sigma_a} \cup \{0\}^{\Sigma-\Sigma_a}$;
$L(M)$ $= (ab \cup cd \cup ef)^*$.

Let us see how the adjustable pattern works in the example. If the lathe has a part, then $q'_{q1} = 0$. When a new part requests an operation on the lathe, the robot will first pick up the part; then before sending the part to the lathe, the robot checks if the transition is available. Since the lathe has a part, the check returns false. To avoid this new part from blocking other operations, the robot should be released. In this case, transition "a" should be adjustable, so the robot can be released by putting the part onto the buffer.

When $\gamma(\sigma) = 1$, $\sigma \in \Sigma_a$ is adjustable. But σ will be adjusted iff it is necessary. Furthermore, if any operation of a part has been adjusted, the task should not be changed. Another concept termed as substitution under a task is defined below, which assures that a part can be completed even though some operations of the part have been changed during production.

Definition 6 (Substitution under a Task): For $x \in \Sigma^*$ and $y \in \Delta^*$, a transfer relation τ can be substituted under a task by a τ' iff $x \overset{\tau}{\mapsto} y \Leftrightarrow x' \overset{\tau'}{\mapsto} y$ if $x' \in [x]$, where $[x] = [x' : x' \overset{t}{\approx}_L x]$.

Proposition 2: Substitutions under a task assure that the assigned task will be successfully complete, no matter how many transitions have been adjusted within its computation.

Proof: The class of regular languages is closed under substitutions [12]. If the process of a computation is dynamically adjusted by observing substitutions under a task, the computation retains the task in an unchanged state.

More specifically, let $x_0 = \alpha_0\beta_0 \cdots \zeta_0$ and $\Rightarrow$ indicates immediate derivation due to a substitution under a task. If

$$x_0 \Rightarrow x_1 \Rightarrow \cdots \Rightarrow x_n$$

then for every i $(1 \le i \le n)$, there exist α_i, $\beta_i, \cdots \zeta_i$, with $x_i = \alpha_i\beta_i \cdots \zeta_i$ such that

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n,$$
$$\beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_n,$$
$$\cdots\cdots\cdots\cdots\cdots$$
$$\zeta_0 \Rightarrow \zeta_1 \Rightarrow \cdots \Rightarrow \zeta_n.$$

As long as each derivation is taken according to Definition 6, n can be any finite integer.

V. Structured Adaptive Supervisory Control

A DFCM can be used to model the control of a DEMS. A method is needed to construct DFCM control models, which are guaranteed to function as desired and be easily transformed into an operational control system. By referring to supervisory control theory [17], [18], an automaton structure called structured adaptive supervisory control (SASC) for manufacturing systems is developed (Fig. 8). To take advantage of object-oriented programming techniques, an SASC model is defined to have an object structure. The interface for an SASC model (the only way to connect an SASC controller to its outside world components) includes an input, an output, and feedback. The input receives incoming messages from a task-requesting unit and returns controller states to the task-requesting unit. The output sends out commands to a task-executing unit and receives feedback from the task-executing unit. All the embedded methods can be classified into three categories: decision, supervision, and execution.

The execution layer consists of an executor. In terms of formal languages, its function can be interpreted as accepting a language, and its structure can be obtained from a deterministic finite capacity acceptor. In terms of the physical operations, the executor executes all the assigned executable operations;

when an unexecutable event occurs, it requests an adjustment from its adaptive supervisor.

The supervision layer consists of an adaptive supervisor. The adaptive supervisor conceptually functions as a recognizer, and is constructed from a deterministic finite capacity transducer. In terms of physical operations, it dynamically supervises the behavior of the executor, coordinates all the running processes on the executor, thus guarantees the completion of all the dispatched tasks.

The decision layer includes an acceptor and a task queue. The acceptor is constructed from a deterministic finite capacity transducer, and performs two functions:

1) checking the capability of the controller to complete an incoming task;
2) taking action by accepting the task and mapping it into a task in an executable format, or by rejecting the task if it is beyond the controller capability.

The existence of an acceptor ensures that all the computing processes on the executor will be nonblocking. The task queue simply collects all the accepted tasks and dispatches them to the lower layers optimally.

A. Executor

The behavior of a DEMS can be completely described by its structured event-graph at the part-flow level [15], [20], which forms the state transition diagram of the DEMS. The event-graph can be then transferred into a DFCA. Formally an executor is defined as $M_e = (Q_e, Q'_e, Q^m_{et}, \Sigma_e, \delta_e, q_{e0}, Q_{ef})$, where

Q	finite control set;
Q'_e	transition-availability control set;
Q^m_{et}	token control set;
Σ_e	input alphabet;
δ_e	transition function;
$q_0 \in Q$	initial control state;
$Q_f \subseteq Q$	final control set.

The accepted language is

$$L(M_e) = \{x : x \in \Sigma_e^* \text{ and } x \overset{\tau}{\mapsto} \text{ACCPET}\}.$$

As discussed in Section IV, a distinguishable subset of the input alphabet Σ_{ea} can be defined. An operation $(\sigma, q_e \times q'_e, q_{e1} \times q'_{e1})$ is an adjustable operation if $\sigma \in \Sigma_{ea} \subset \Sigma_e$. Let

$$\Gamma = \{1\}^{\Sigma_{ea}} \cup \{0\}^{\Sigma_e - \Sigma_{ea}}$$

be the set of assignments to the elements of Σ_e. A function $\gamma: \Sigma_e \to \{0,1\}$ holds, which is an adjustable pattern. Furthermore, an augmented transition function $\delta_{ea}: \Sigma_e \times \{Q_e \times Q'_e\} \to \{Q_e \times Q'_e\}$ (pfn) can be defined according to

$$\delta_{ea}(\sigma, q_e \times q'_{eq}) = \begin{cases} \delta_e(\sigma, q_e), & \text{if the codomain of } \delta_{ea} : \Sigma_e^i \times Q_e \times Q'_e \\ & \quad \to Q_e \times Q'_e \text{ satisfies } q'_{eq} = 1, \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{3}$$

The executor for *Example 1* can be constructed as a DFCA whose state transitions are correspondingly shown in Fig. 7. All the state transitions $\{a, b, c, d, e, f\}$ physically correspond to all the possible part flows during production. Formally, the executor is defined as $M_e = (Q_e, Q'_e, Q^m_{et}, \Sigma_e, \delta_{ea}, q_{e0}, Q_{ef})$ where

Q_e	$= \{0,1,2,3\}$;
Q'_e	$= \{0,1\}$;
Q^m_t	4×4 token matrix;
Σ_e	$= \{a,b,c,d,e,f\}$;
Σ_{ea}	$= \{a,e\}$;
$\Sigma_e - \Sigma_{ea}$	$= \{b,c,d,f\}$;
δ_{ea}	defined as (3);
q_{e0}	$= 0$;
Q_{ef}	$= \{0\}$;
Γ	$= \{1\}^{\Sigma_{ea}} \cup \{0\}^{\Sigma_e - \Sigma_{ea}}$.

Consequently, the accepted language for executor M_e is known

$$L(M_e) = \{x : x \in \Sigma_e^* \text{ and } (x, q_{e0} \times q'_{e0}) \overset{\tau}{\mapsto} (\Lambda, q_{ef} \times q'_{ef})\}$$

where Λ stands for an empty string, i.e.,

$$L(M_e) = (ab \cup cd \cup ef)^*.$$

For example, a part p1 needs three machining operations in the workstation according to the process plan, <<M1-turning, M1-turning, M2-milling>>. This process plan belongs to the domain of traditional manufacturing process plans which calls out routing sequences. To execute this process plan on M_e, it should be converted into a string $x =$ '$ababef$' such that $x \in L(M_e)$. The class of task equivalence for part 1 is

$$[x_{\text{p1}}] = (cd)^*ab(cd)^*ab(cd)^*ef(cd)^*$$

i.e., any string from $[x_{\text{p1}}]$ describes part p1 task. For executor M_e, a complete computation of such a string from $[x_{\text{p1}}]$ physically indicates the task completion of part p1.

B. Adaptive Supervisor

An adaptive supervisor oversees the behavior of the executor M_e. The adaptive supervisor can be formally defined as a filter which is a particular DFCT, $M_s = (Q_s, Q'_s, Q^m_{st}, \Sigma_s, \Delta_s, \delta_s, g_s, q_{s0}, Q_{sf})$, where Q_s, Q'_s, Q^m_{st}, Σ_s, Δ_s, q_{s0}, Q_{sf} are defined the same as before, $\delta_s: \Sigma_s \times Q_s \to Q_s$ is the state transition function (pfn), $g_s: \Sigma_s \times Q_s \to \Delta_s$ is the output function (pfn), and whose transfer relation τ is given by

$$x\tau = \begin{cases} x, & \text{if } x \in L_e, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Obviously, $L \cap L_e = L\tau$ is regular.

Let $Q^m_{et} = Q^m_{st}$ and couple process p_{si} of supervisor M_s to process p_{ei} of executor M_e for $1 \le i \le m$ by a lockstep loop in such a way that a currently computing instruction can be completely computed on M_s iff its output is currently executable on M_e; otherwise, the instruction will be temporarily blocked. A coupler $\Psi = (T, \amalg)$ is required, which functions in a lockstep loop manner such that an operation on M_s and its corresponding operation on M_e can be stepwise coordinated.

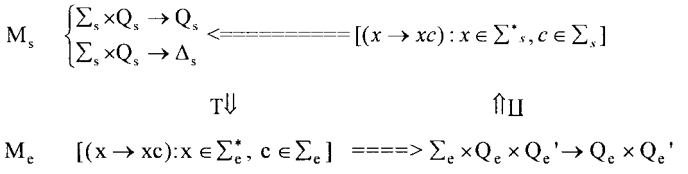


Fig. 9. M_s and M_e are coupled by a lockstep loop.

Formally a coupler $\Psi = (T, \Pi)$ is defined as a coupling function by

$$T : \begin{cases} \Sigma_s \times Q_s \to Q_s \\ \Sigma_s \times Q_s \to \Delta_s \end{cases} \Rightarrow [(x \to xc) : x \in \Sigma_e^*, c \in \Sigma_e] \quad (4)$$

$$\Pi : (\Sigma_e \times Q_e \times Q_e' \to Q_e \times Q_e') \Rightarrow [(x \to xc) : x \in \Sigma_s^*, c \in \Sigma_s]. \quad (5)$$

T is a downward lockstep feed which guarantees the output of M_s accepted stepwise by M_e; Π is an upward lockstep feedback which guarantees that the next operation on M_s can be computed iff the operation on M_e has been completed. The function of a coupler is graphically shown in Fig. 9.

In a DEMS, a currently running operation can possibly be blocked (or conflict with others) due to the existence of concurrent and asynchronous operations. In order for executor M_e to complete its temporarily blocked running process, the adaptive supervisor M_s has to be capable of replacing its program (making a dynamic adjustment) by another in such a way that M_e will still successfully execute its running task under the supervision of M_s.

The state feedback map ϕ is defined as $\phi: Q_s \to \Gamma \times Q_e$ which is a function that maps supervisor state q_s into adjustable pattern γ and M_e's state q_e. Define $\phi(q_s) = 1$ for each $\gamma(\sigma) = 1$ and $q_e = 0$, which simply means that a block or conflict occurs in the current computing process on M_e; otherwise, $\phi(q_s) = 0$. When $\phi(q_s) = 1$, the supervisor has to adaptively replace its program by another such that the new program will map a newly coordinated operation sequence from the current task equivalence class.

As discussed previously, the completion of a task is defined by the completion of a computation on a machine. A formal definition for task completion during an adaptively supervised computation can correspondingly be defined as: a task is completely done iff the coupler $\Psi = (T, \Pi)$ of M_s and M_e has simultaneously led M_s and M_e into their final states during their computations.

For the workstation in *Example 1*, its adaptive supervisor can be constructed as $M_s = (Q_s, Q_s', Q_{st}^m, \Sigma_s, \Delta_s, \delta_s, g_s, q_{s0}, Q_{sf})$, whose instructions are shown in Fig. 10.

Without loss of generality, assume that only one adjustable transition arc is considered. In this case, for instance 'a', the language accepted by this controller is still described by

$$L(M_e) = \{x : x \in \Sigma_e^* \text{ and } (x, q_{e0} \times q_{e0}') \overset{\tau}{\mapsto} (\Lambda, q_{ef} \times q_{ef}')\}, \text{ i.e.}$$
$$L(M_e) = (ab \cup cd \cup ef)^*.$$

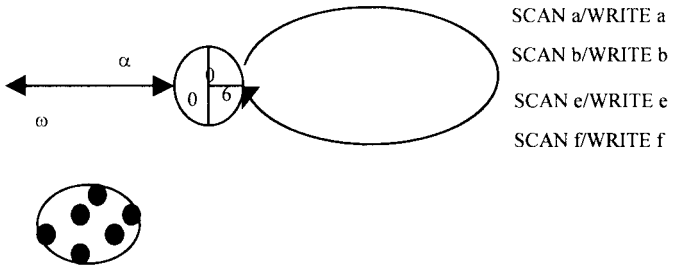


Fig. 10. State transitions of the supervisor.

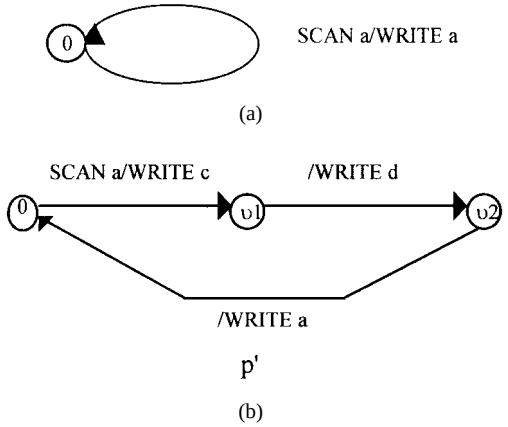


Fig. 11. A subprogram substitutes an instruction.

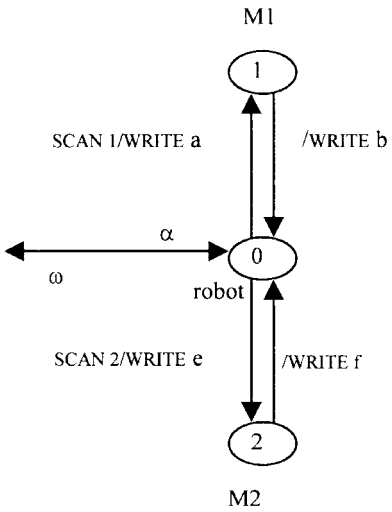


Fig. 12. State transitions of the acceptor.

The equivalence class for part p1 is $[x_{\mathrm{p1}}] = (cd)^*ab(cd)^*abef(cd)^*$. It is obvious that for real manufacturing setuations the Kleene-closure $(^*)$ will be interpreted as either zero or one. Any number greater than one will be redundant.

If a state feedback map is $\phi(q_s) = 1$, the adaptive supervisor can switch computation from p to p'. Since only one adjustable operation is considered, there is only one instruction [Fig. 11(a)] which will be substituted by a subprogram (Fig. 11(b)) when $\phi(q_s) = 1$. Apparently, any string from $[x_{\mathrm{p1}}] = (cd)^*ab(cd)^*abef(cd)^*$ can be computed completely by both M_s and M_e.

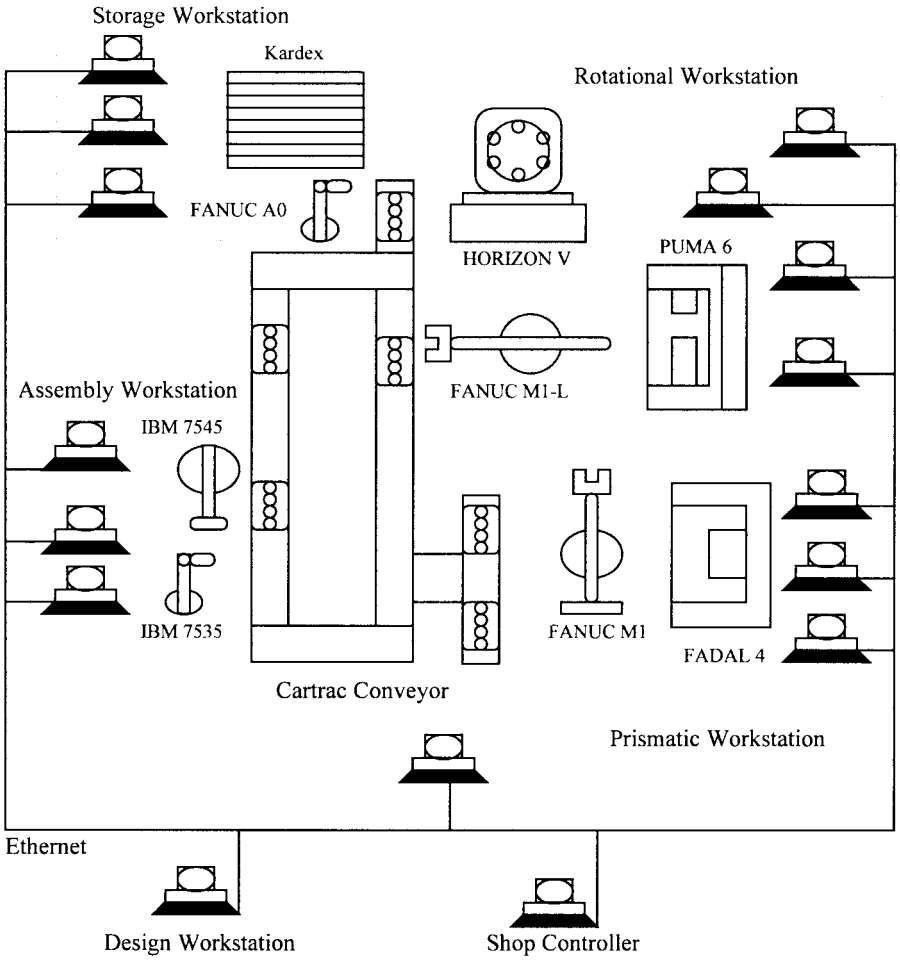


Fig. 13. FMS in the CIM lab at Pennsylvania State University.

C. Acceptor

If a string x (a given process plan) is mapped into another string y which belongs to the accepted language of executor M_e, string x (or string y) can be computed completely on both M_s and M_e. However, if there is no such mapping, the process computing string x will eventually be blocked. To guarantee that no computing process will be blocked, an acceptor is used to check whether the mapping of an input string is complete. The acceptor resides on the top of M_s. A string accepted by the acceptor is a computable task; otherwise, the string is not computable and should be rejected.

An acceptor can be formally defined as a standard deterministic finite transducer [8], $M_a = (Q_a, \Sigma_a, \Delta_a, I_a, q_{a0}, Q_{af})$, which maps an input string, $x \in L_i$ (a set of strings incoming messages formatted in traditional process plans), to an output string, $y \in L_e$ (a set of strings formatted in modified process plans and executable for the executor), by a complete computation, i.e.

$$x \overset{\tau}{\mapsto} y \Leftrightarrow x \overset{\alpha}{\mapsto} (q_{a0}, x, \Lambda) \overset{\Pi^*}{\mapsto} (q_{af}, \Lambda, y) \overset{\omega}{\mapsto} y,$$

$$\tau = \alpha o \Pi^* o \omega.$$

An acceptor for *Example 1* can be constructed as $M_a = (Q_a, \Sigma_a, \Delta_a, I_a, q_{a0}, Q_{af})$, whose instructions are shown in Fig. 12. If character 0 stands for the robot, 1 for M1, and 2 for M2, then $\Sigma_a = \{0, 1, 2\}$. The given traditional process plan, <<M1-turning, M1-turning, M2-milling>>, for part p1 can be accepted and mapped into a modified and executable process plan by this acceptor, i.e. $x \overset{\tau}{\mapsto} y$, or $\{112\} \overset{\tau}{\mapsto} \{ababef\}$ such that $y \in L(M_e)$.

VI. A Case Study

The flexible manufacturing system (FMS) in the CIM Lab at Penn State is a typical DEMS, which includes three numerical control (NC) machines, five robots, a material transport system, and a warehouse system (Fig. 13). Each computer controls a piece of equipment. Based on the manufacturing functionality and control architecture [21], the DEMS is divided into five workstations: a rotational workstation, a prismatic workstation, an assembly workstation, a material transport workstation, and a storage workstation.

The rotational workstation consists of a Daewoo Puma numeric control turning machine, a Pratt & Whitney Horizon V NC vertical milling machine, a Fanuc M1-L robot, and an intermediate buffer space containing five slots. The prismatic workstation consists of a Fadal NC milling machine

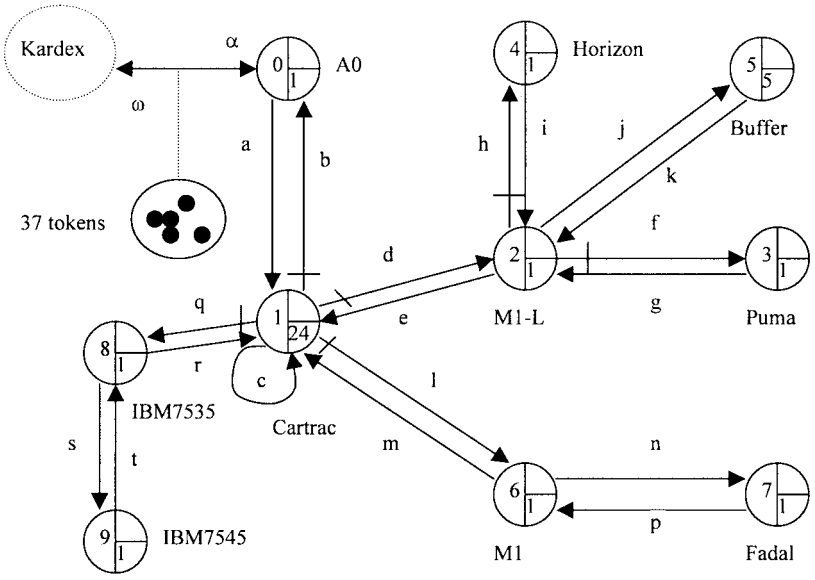


Fig. 14. State transition diagram of the FMS executor.

and a Fanuc M1 robot. The assembly workstation consists of an IBM7545 robot and an IBM7535 robot. The IBM 7545 robot performs assembly operations, while the IBM 7535 robot performs only part delivery tasks. The AS/RS workstation consists of a Kardex tray-based vertical automatic storage system and a Fanuc A0 robot. The material transport workstation consists of a mini-Cartrac conveyor system which physically ties all the other constituent workstations into an integrated manufacturing system. All the parts (raw material, partially finished pieces, and finished pieces) are transported between different workstations by the Cartrac system. In the Cartrac system, six carts are available. Each cart has four slots. Whereas, all the above processing machines and robots are capable of dealing with one part at a time. The capacity for this system can be summed as $R = \sum_{i=1}^{n} r_i = 37$.

A part advances through different workstations based on a given process plan. A part can be completely processed iff its process plan is successfully executed by the FMS control system. Carts in the Cartrac system and the buffer in the rotation workstation can be used as intermediate storage spaces between processes. In addition, some parts may have alternative process routes during production. Therefore, part-state transitions from Cartrac to M1, to M1-L, or to IBM7535 and from M1-L to Horizon, or to Puma are adjustable.

The control model of this FMS is developed using the presented SASC automaton structure. First, the state transition diagram of the executor is constructed (Fig. 14). Formally, the executor M_e is defined as $M_e = (Q_e, Q'_e, Q^m_{et}, \Sigma_e, \delta_{ea}, q_{e0}, Q_{ef})$
where

$$\begin{aligned}
Q_e &= \{0,1,2,3,4,5,6,7,8,9\};\\
Q'_e &= \{0,1\};\\
Q^m_t &\quad 37 \times 10 \text{ token matrix},\\
\Sigma_e &= \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,p,q,r,s,t\};\\
\Sigma_{ea} &= \{b,d,f,h,l,q\};\\
\Sigma_e - \Sigma_{ea} &= \{a,c,e,g,i,j,k,m,n,p,r,s,t\};
\end{aligned}$$

$$\delta_{ea}(\sigma, q_e \times q'_{eq}) = \begin{cases} \delta_e(\sigma, q_e), & \text{if the codomain of } \delta_{ea} : \Sigma^i_e \times Q_e \times Q'_e \\ & \quad \to Q_e \times Q'_e \text{ satisfies } q'_{eq} = 1, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$\begin{aligned}
q_{e0} &= 0;\\
Q_{ef} &= \{0\};\\
\Gamma &= \{1\}^{\Sigma_{ea}} \cup \{0\}^{\Sigma_e - \Sigma_{ea}}.
\end{aligned}$$

Consequently, the accepted language for M_e is known

$$L(M_e) = \{x : x \in \Sigma_e^* \text{ and } (x, q_{e0} \times q'_{e0}) \overset{\tau}{\mapsto} (\Lambda, q_{ef} \times q'_{ef})\}$$

where Λ stands for an empty string, i.e.,

$$\begin{aligned}
L(M_e) = (ac^*b \cup ac^*((d(hi \cup jk \cup fg)^*e)^*\\
\cup (l(np)^*m)^* \cup (q(st)^*r)^*)^*c^*b)^*.
\end{aligned}$$

As an example, consider the following four parts made in the FMS: p1, p2, p3, and p4. Their process plans along with alternatives are: p1 {Alt1: Puma → Horizon, Alt2: Horizon → Puma, Alt3: Fadal → Puma}; p2 {Fadal}; p3 {Fadal → Puma → Horizon}; p4 {Alt1: Fadal → Puma → IBM7545, Alt2: Horizon → Puma → IBM7545}. Based on the definition of task equivalence, their task equivalence classes are then defined by

$$\begin{aligned}
[x_{\mathrm{p1}}] &= ac^*(d(jk)^*fg(jk)^*hi \cup d(jk)^*hi(jk)^*fg\\
&\quad \cup lnpmc^*d(jk)^*fg)(jk)^*ec^*b,\\
[x_{\mathrm{p2}}] &= ac^*lnpmc^*b,\\
[x_{\mathrm{p3}}] &= ac^*lnpmc^*d(jk)^*fg(jk)^*hi(jk)^*ec^*b, \quad \text{and}\\
[x_{\mathrm{p4}}] &= ac^*(lnpmc^*d \cup d(jk)^*hi)(jk)^*fg(jk)^*ec^*qstrc^*b,
\end{aligned}$$

where $[x_{\mathrm{p1}}]$, $[x_{\mathrm{p2}}]$, $[x_{\mathrm{p3}}]$, and $[x_{\mathrm{p4}}] \in L(M_e)$.

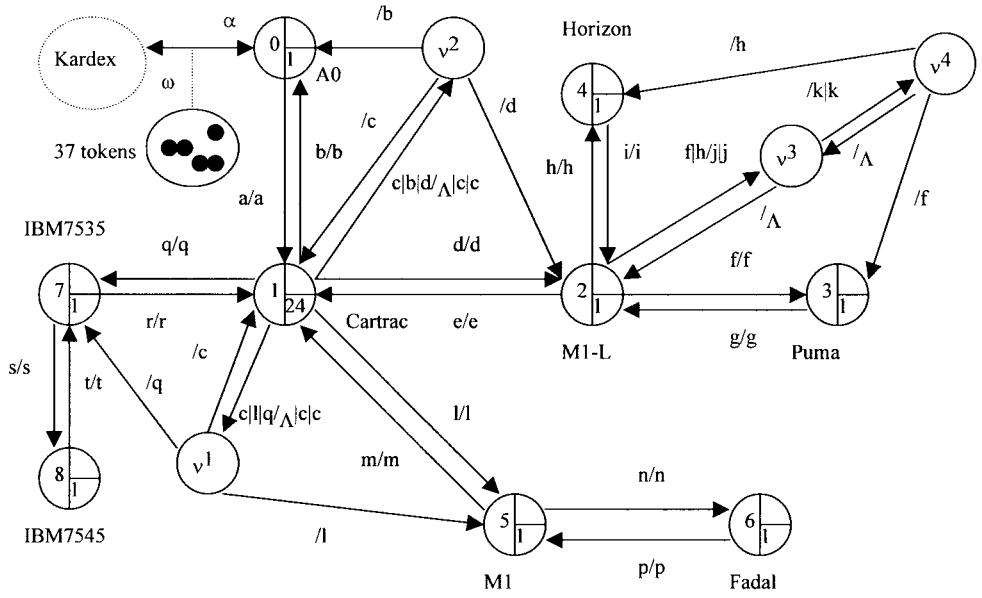


Fig. 15. State transition diagram of the FMS adaptive supervisor.

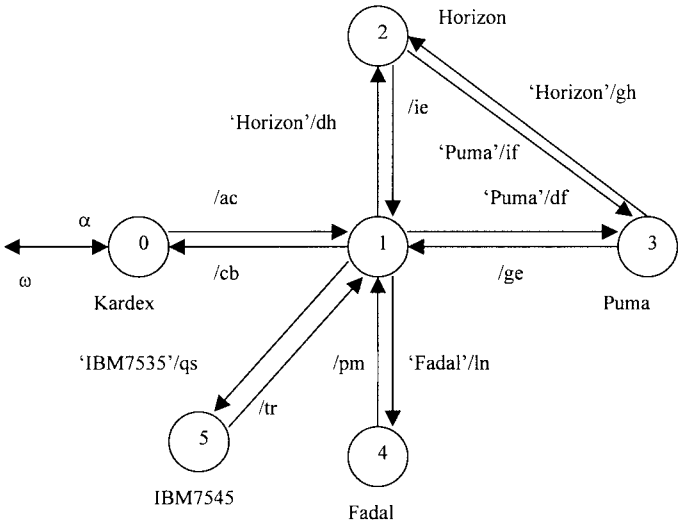


Fig. 16. State transition diagram of the FMS acceptor.

Then, the adaptive supervisor M_s is constructed. As discussed, the following substitutions under tasks are valid in the FMS: $b \Leftrightarrow c^*b$, $d \Leftrightarrow c^*d$, $1 \Leftrightarrow c^*1$, $q \Leftrightarrow c^*q$, $f \Leftrightarrow (jk)^*f$, and $h \Leftrightarrow (jk)^*h$. Thus M_s can be constructed as $M_s = (Q_s, Q_s', Q_{st}^m, \Sigma_s, \Delta_s, \delta_s, g_s, q_{s0}, Q_{sf})$, whose instructions are shown in Fig. 15. Intermediate states $\nu 1$, $\nu 2$, $\nu 3$, and $\nu 4$ are defined, which accomplish all the necessary task adjustments under a coupler $\Psi = (T, \amalg)$ defined in Eqs. (4) and (5).

The last step requires the construction of the acceptor, which is constructed as $M_a = (Q_a, \Sigma_a, \Delta_a, I_a, q_{a0}, Q_{af})$, whose instructions are shown in Fig. 16. The example process plans: p1 {Alt1: Puma → Horizon, Alt2: Horizon → Puma, Alt3: Fadal → Puma}, p2 {Fadal}, p3 {Fadal → Puma → Horizon}, p4 {Alt1: Fadal → Puma → IBM7545, Alt2: Horizon → Puma → IBM7545} can be transferred into $[x1] = \{acdfghiecb, acdhifgecb, aclnpmdfgecb\}$, $x2 = \{aclnpmcb\}$, $x3 = \{aclnpmdfghiecb\}$, $[x4] = \{aclnpmdfgeqstrcb, acdhifgeqstrcb\}$. Apparently, $[x1] \subseteq [x_{\mathrm{p1}}]$, $x2 \in [x_{\mathrm{p2}}]$, $x3 \in [x_{\mathrm{p3}}]$, and $[x4] \subseteq [x_{\mathrm{p4}}]$ are held.

This case study demonstrates

1) number of control states in an SASC model grows linearly in the number of constituent machines, i.e., $\Theta = O(n)$, where n is the number of machines;
2) SASC model can be constructed systematically.

The detailed process plan specification and production operations of the FMS are given in Qiu [16]. The control model has been successfully transferred into control software, which controls the FMS as desired [16].

VII. Conclusion

In this paper, a methodology potentially applicable to the shop floor for modeling the control of a DEMS has been studied. Using the concepts of coordination of multiple computations and part traceability, the methodology was presented as a two step approach.

First, a modified finite machine (DFCM) was developed, which can be used to model certain discrete event manufacturing systems and make sure that the complexity of the constructed control model is linear to the constituent machines. Secondly, based on the concept of DFCM, a well-defined automaton structure SASC is developed, which systematically guides the construction of a DFCM control model for a discrete event manufacturing system. By controlling all the on-line part states, the SASC model controls all the machines on the shop floor.

However, the presented methodology requires further study for applications in assembly lines, where parts can be merged. In addition, the methodology could be inapplicable to modeling the control of a nonpart oriented manufacturing shop floor. As discussed, the DFCM is originated from the concept of part traces. A nonpart manufacturing system loses part traceability.

In addition, note that the presented methodology provides a structured solution to develop control software for manufacturing systems rather than a general representational/analytical tool, such as the Petri net [25] and Ramadge–Wonham's supervisory control theory [17].

ACKNOWLEDGMENT

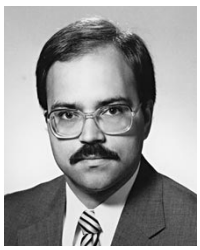
The authors would like to thank Dr. R. A. Wysk, Department of Industrial and Manufacturing Engineering, Pennsylvania State University, Dr. Y. C. Ho, Division of Applied Science, Harvard University, and Dr. M. W. Wonham, Department of Electrical and Computer Engineering, University of Toronto, for their discussions, comments, and suggestions.

REFERENCES

[1] R. U. Ayres, "Technology forecast for CIM," *Manuf. Rev.*, vol. 2, no. 1, pp. 43–52, 1989.
[2] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin, "Supervisory control of a rapid thermal multiprocessor," *IEEE Trans. Automat. Contr.*, vol. 38, pp. 1040–1059, July 1993.
[3] F. Biemans and P. Blonk, "On the formal specification and verification of CIM architectures using LOTOS," *Comput. Ind.*, vol. 7, pp. 491–504, 1986.
[4] C. G. Cassandras, *Discrete Event Systems: Modeling and Performance Analysis*. Homewood, IL: Irwin, 1993.
[5] J. K. Chaar, "A methodology for developing real-time control software for efficient and dependable manufacturing systems," Ph.D. dissertation, Univ. Michigan, Ann Arbor, 1990.
[6] A. A. Desrochers and R. Y. Al-Jaar, *Applications of Petri Nets in Manufacturing Systems*. New York: IEEE Press, 1995.
[7] Y. Dotan and D. Ben-Arieh, "Modeling flexible manufacturing systems: The concurrent logic programming approach," *IEEE Trans. Robot. Automat.*, vol. 7, pp. 135–148, Feb. 1991.
[8] R. W. Floyd and R. Beigel, *The Languages of Machines: An Introduction to Computability and Formal Languages*. New York: Computer Science, 1994.
[9] A. Giua and F. DiCesare, "Petri net structured analysis for supervisory control," *IEEE Trans. Robot. Automat.*, vol. 10, pp. 185–195, Apr. 1994.
[10] A. Giua and F. DiCesare, "Decidability and closure properties of weak Petri net languages in supervisory control," *IEEE Trans. Automat. Contr.*, vol. 40, pp. 906–910, May 1995.
[11] Y. C. Ho, *Discrete Event Dynamic Systems, Analyzing Complexity and Performance in the Modern World*, Y.-C. Ho, Ed. New York: IEEE Press, 1992.
[12] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.
[13] E. Kasturia, F. DiCesare, and A. Desrochers, "Real time control of multilevel manufacturing systems using colored Petri nets," in *Proc. IEEE 1988 Int. Conf. Robotics Automation*, vol. 2, pp. 1114–1119.
[14] A. W. Naylor and M. C. Maletz, "The manufacturing game: A formal approach to manufacturing software," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-16, pp. 321–334, May/June 1986.
[15] R. G. Qiu and S. B. Joshi, "Structured adaptive supervisory control of a flexible manufacturing system," in *Factory Automation Intelligent Manufacturing'96*, pp. 800–809, May 1996.
[16] R. Qiu, "Modeling and control of a flexible manufacturing system using deterministic finite capacity automata," Ph.D. dissertation, Penn. State Univ., University Park, PA, 1996.
[17] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Contr. Optimiz.*, Vol. 25, no. 1, pp. 206–230, Jan. 1987.
[18] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, pp. 81–98, Jan. 1989.
[19] P. J. Ramadge, "The complexity of some basic control problems for discrete event systems," *Advanced Computing Concepts and Techniques in Control Engineering*, M. J. Denham and A. J. Laub, Eds. Berlin, Germany: Springer-Verlag, 1988.
[20] J. S. Smith, "A formal design and development methodology for shop floor control in computer integrated manufacturing," Ph.D. dissertation, Penn. State Univ., University Park, PA, 1992.
[21] J. S. Smith, W. C. Hoberecht, and S. B. Joshi, "A shop floor control architecture for computer integrated manufacturing," *IIE Trans.*, vol. 28, no. 10, pp. 783–794, 1996.
[22] T. J. Tsitsiklis, "On the control of discrete-event dynamic systems," *Math. Contr., Sig. Syst.*, no. 2, pp. 95–107, 1989.
[23] F. Y. Wang and G. N. Dsaridis, "A coordination theory for intelligent machines," *Automatica*, vol. 26, pp. 833–844, Sept. 1990.
[24] R. A. Williams, B. Benhabib, and K. C. Smith, "A hybrid supervisory control system for flexible manufacturing workcells," in *Proc. IEEE Inte. Conf. Systems, Man, Cybernetics*, 1994, vol. 3, pp. 2551–2556.
[25] M. C. Zhou and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Norwell, MA: Kluwer, 1993.

Robin G. Qiu (M'95–A'96) received the M.S. and B.S. degrees from Beijing Institute of Technology, Beijing, China, and the Ph.D. degree in computer science and industrial engineering from Pennsylvania State University, University Park, in 1996.

His disciplines cover industrial and manufacturing engineering, computer science and engineering, electrical engineering, and mechanical engineering. He is currently a Research Scientist at Kulicke and Soffa Industries, Inc., Willow Grove, PA. He has more than ten years of working experience in the field of computer-integrated manufacturing systems. He has had more than 20 articles published or presented in journals or conferences. He is actively serving as a referee and panelist for several international journals and USA governmental agencies. His interests include control of automated manufacturing systems, information technology, enterprise resource planning, and manufacturing execution planning.



Sanjay B. Joshi (S'92–M'96) received the B.S. degree from the University of Bombay, Bombay, India, the M.S. degree from the State University of New York, Buffalo, and the Ph.D. degree in industrial engineering from Purdue University, West Lafayette, IN.

His research and teaching interests are in the area of CAD/CAM with specific focus on computer aided process planning, control of automated flexible manufacturing systems, and rapid prototyping and tooling. He is currently Professor of industrial and manufacturing engineering at Pennsylvania State University, University Park. He is currently Department Editor for *Process Planning—IIE Transactions on Design and Manufacturing*, and also serves on the editorial board of *Journal of Manufacturing Systems*, *Journal of Intelligent Manufacturing*, and *Journal of Engineering Design and Automation*.

Dr. Joshi is the recipient of several awards, including the Presidential Young Investigator Award from NSF in 1991, Outstanding Young Manufacturing Engineer Award from SME in 1991, and Outstanding Young Industrial Engineer Award from the IIE in 1993.