PERGAMON

Computers & Industrial Engineering 38 (2000) 107–123

computers & industrial engineering

www.elsevier.com/locate/dsw

A generic event control framework for modular flexible manufacturing systems

Jonghun Park [a,*], Jinwoo Park [b], Jongwon Kim [c]

[a] *School of Industrial and Systems Engineering, Georgia Institute of Technology, 765 Ferst Drive, Atlanta, GA 30332-0205, USA*

[b] *Department of Industrial Engineering, Seoul National University, Seoul, South Korea*

[c] *School of Mechanical and Aerospace Engineering, Seoul National University, Seoul, South Korea*

Abstract

A generic event control framework is presented for a class of modular flexible manufacturing systems (FMSs). A modular FMS is characterized as a set of flexible workstations inter-connected by a material handling system consisting of a transporter and a stocker. Many of the existing FMS implementations in industry fall into this class. Furthermore, the modularity also allows the capability to effectively model a complex FMS by decomposing it into several modular FMSs. In order to achieve reconfigurability of control system for various FMS implementations and control policies, the proposed control framework is defined as a set of distributed resource controllers and a central system supervisor coordinating them. The resource controllers are further classified into workstation, transporter, and stocker controllers. As the controllers exchange a series of events according to pre-defined protocols, they are modelled as event handlers in which control actions are made based on the event occurrences. Specifically, for each controller, an event-based control structure specified in terms of generic logical and performance control functions, is presented. Since the proposed framework is defined by the use of the interacting distributed processes with the well-defined protocols and computationally efficient algorithms, it is expected that the framework is easily implementable for most industrial FMSs. Finally, the performance and reconfigurability is demonstrated by the distributed simulation from which we can also verify the correctness of the proposed algorithms.

Keywords: Flexible manufacturing system; Control algorithms; Event control

* Corresponding author. Tel.: +1-404-894-4202.
E-mail address: jonghun@isye.gatech.edu (J. Park).

0360-8352/00/$ - see front matter
PII: S0360-8352(00)00032-2

1. Introduction

Flexible manufacturing systems (FMSs) are promising means to meet the increasing needs for agility of modern manufacturing systems. Also, flexible automation is often an inevitable choice when the parts are too large or small to be handled by a human operator, requiring automation of material handling functions. In an FMS, finite resources such as buffers, tools, pallets, and material handling devices are shared by jobs, and therefore control system plays an important role as it is responsible for allocating the resources to concurrently competing jobs so that logical correctness and high performance can be achieved. However, the control system design for FMSs is a complex task since it requires a clear understanding of the underlying concurrent and asynchronous system dynamics. For this reason, many researchers have proposed various formal models to facilitate the development of control software. In Naylor and Volz (1987) give an excellent discussion about the formalism necessary to develop a generic control software, and present a conceptual modeling framework for integrated manufacturing system characterized as an assembly of software/hardware components. A series of rigorous research results for automating control software development process has been presented by a group of researchers in Joshi, Mettala and Wysk (1992), Smith and Joshi (1994), Joshi, Mettala, Smith and Wysk (1995) and Smith, Hoberecht and Joshi (1996). Specifically, Joshi et al. (1992) propose an approach to automate the development of control software by use of a context-free grammar. Smith and Joshi (1994) present a message-based part state graph (MPSG) to represent the execution module of shop floor controller as a communicating finite state machine. A graph-based model equivalent to pushdown automata is proposed as a formal model of flexible manufacturing cell in Joshi et al. (1995). Further approaches along this line of research can be found in Daltrini and Kumara (1996), where a Moore machine is extended to provide a generic representation of distributed and timed control environments. Other important formal models which have been widely accepted for FMS control are Petri nets and their variants. Ezpeleta and Colom (1997) present an automatic synthesis method based on a colored Petri net. Venkatesh and Zhou (1998) propose a combined approach using object modeling diagrams and Petri nets so that object-oriented modeling helps to support reusability and extensibility of the control software. Detailed descriptions on applying Petri nets to FMS control are given in Zhou and Venkatesh (1999) and Zhou and Dicesare (1993). Finally, supervisory control theory pioneered by Ramadge and Wonham (1989) has also been applied to synthesize the control software. In particular, an implementation methodology which utilizes the supervisory control theory in conjunction with programmable logic controller, is presented in Lauzon, Mills and Benhabib (1997), and a method to rapidly prototype the control software using the supervisory control theory is developed by Qiu and Joshi (1996).

While the approaches based on the above mentioned formal models are usually generic enough to be applied to any kind of FMS configurations, the complexity in modeling and computation has been recognized as the main barrier for their successful implementations in the large-scale, real-world FMSs. Hence, motivated by the need for readily implementable control systems for industrial, large-scale FMSs, we take a different approach in this paper. We confine our modeling domain to a class of FMSs which is characterized as *modular*, and propose an event control framework by specifying a set of easily implementable and

computationally efficient distributed algorithms (Shatz, 1993). The proposed framework is *generic* in the sense that, once implemented, a specific FMS configuration, job description, and control policies are provided as input specifications to the framework, therefore it allows rapid prototyping of the control software for various FMS implementations producing arbitrary part mix. Accordingly, the framework also ensures reconfigurability to the changes in the FMS layout, part spectrum, and control policies in order to minimize reprogramming efforts.

The paper is organized as follows: Section 2 introduces the notion of modular FMS, and presents a control architecture consisting of a system supervisor and a set of resource controllers. The concrete control algorithms are given in Section 3. Section 3.1 specifically, establishes the underlying modeling framework, and the algorithms for the resource controllers and the system supervisor are detailed in Sections 3.2 and 3.3, respectively. The performance and reconfigurability of the proposed approach is demonstrated in Section 4. Section 5 concludes the paper with a discussion of possible future extensions.

2. The control architecture

The basic model of the *modular* FMS considered in this work is depicted in Fig. 1. It consists of a set of flexible *workstations*, inter-connected with a material handling system. Each workstation is further characterized as a set of *processors* (depicted as circles in Fig. 1) with well-defined processing capabilities, and a set of *buffers* (depicted as rectangles in Fig. 1) accommodating jobs waiting to be (or being) served by the processors. For example, in Fig. 1, workstation 1 has two buffers and one processor. The material handling system consists of a *transporter* capable of moving jobs between any pair of workstations typically through a pre-

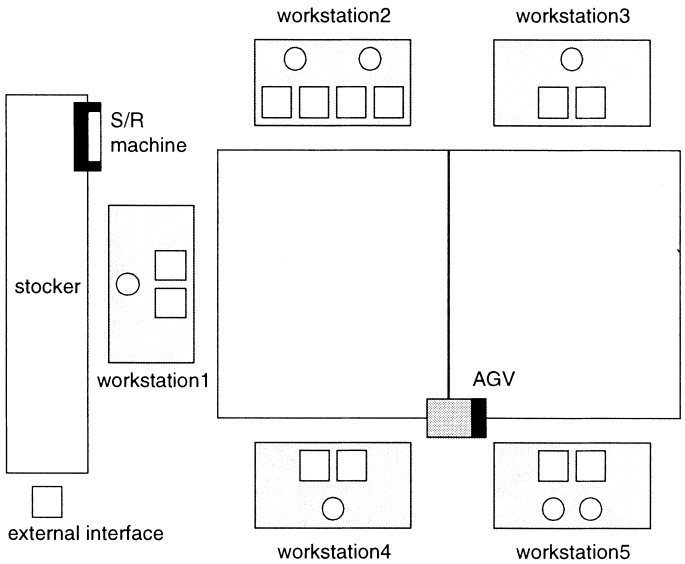

Fig. 1. The modular flexible manufacturing system model.

defined network of transportation paths (An automated guided vehicle system (AGVS) is a representative example), and a *stocker* that accommodates jobs arriving to and leaving from an FMS. Hence, the stocker constitutes the interface between the considered modular FMS and the rest of the production environment. Jobs processed in the system are initially induced to it through a *load/unload* station, interfacing the stocker with the other FMS components. Subsequently, each job visits a sequence of workstations, determined by matching the distinct processing needs of the job with the processing capabilities of the various workstations. The finished job is again stored into the stocker. This routing sequence is known as *process plan*. In case each job has a set of alternative process plans, the process plan is assumed to be determined before the job is induced into the system. We believe that the above characterization is generic for encompassing most current implementations of the flexibly automated manufacturing systems, and their underlying operational characteristics. For example, in a semiconductor wafer fab environment (Atherton & Atherton, 1995), a bay can be considered as a modular FMS consisting of a set of workstations and an intra-bay material handling system, and the entire fab can then be modeled as an upper-level modular FMS in which each bay corresponds to a workstation inter-connected with an inter-bay material handling system. Hence, the modularity allows the capability to effectively model and control a large-scale FMS through decomposing it to several component subsystems.

Having established the necessary structure required to describe the behavior of the modular FMS considered in this paper, we propose a control framework consisting of a *system supervisor* and a set of concurrent *resource controllers*, each managing a set of specific FMS devices. The proposed control architecture is depicted in Fig. 2 where each circle represents an independently running process. The underlying motivation for this process decomposition is to achieve reconfigurability of control system as well as to deal with concurrency efficiently. That is, different types of asynchronous events generated by job arrivals, workstations, and material handling devices are taken care of separately by their corresponding processes. In addition, as long as the resource controllers follow a pre-defined set of protocols, they become functionally

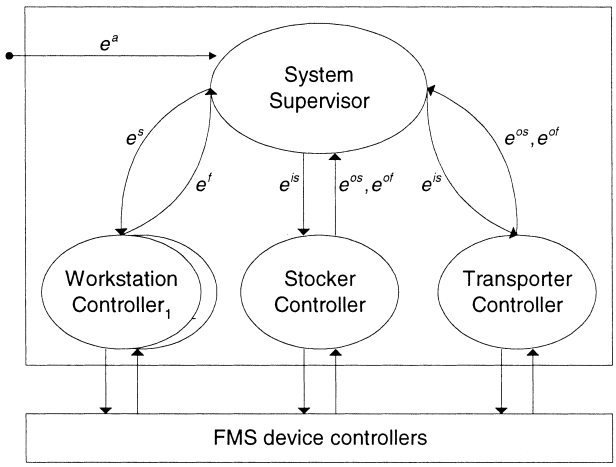

Fig. 2. The proposed hierarchical control architecture.

independent so that each of them can be added, changed or removed as necessary with minimal changes to the rest of the system.

The system supervisor is responsible for managing job flow through the system, i.e. the induction of new jobs into the system, and their dispatching to the required workstations. On the other hand, the detail operations of the workstation, transporter, and stocker are controlled by their corresponding resource controllers, namely *workstation controller*, *transporter controller*, and *stocker controller*. Furthermore, the resource controllers are in subordinate relationship with the system supervisor, in the sense that (i) their operations are triggered by commands issued by the system supervisor, and (ii) they provide feedback information to it regarding the processing status of their task assignments. These lower-level resource controllers supervise the operation of the hardware devices in their domain, by exchanging a series of communication messages with these devices. Hence, the control architecture is *hierarchical* since the system supervisor performs the role of a centralized event controller which establishes logically correct behavior of an FMS, and coordinates a set of workstations and material handling devices. It is also *distributed* as each resource controller is defined as an independently running process.

3. The control algorithms

3.1. The modeling framework

Control requirements for the system supervisor can be broadly classified into either *logical* or *performance oriented*. The logical control concerns with ensuring logically correct behavior with respect to the given control specifications such as deadlock freedom, mutual exclusion, and avoidance of forbidden states (Moody & Antsaklis, 1998). On the other hand, the performance control attempts to optimize overall system performances such as flow time or makespan by determining the sequence and timing of individual operations. Our perspective on the relation between the logical and performance control is layered. That is, in every FMS state, the logical control provides a set of *admissible* next events, and then the performance control decides the most attractive one from the set according to the performance measure. Hence, the logical control establishes the feasibility of events, and therefore it is important to have the least restrictive (i.e. maximally permissive) logical control so that the largest possible admissible set is obtained at every FMS state.

In this paper, we limit our discussion about the logical control problem to the deadlock avoidance which is an essential requirement for establishing unmanned operation of FMSs. However, we mention that our proposed framework is able to accommodate other control specifications easily by redefining the admissibility[1]. The deadlock avoidance problem in an FMS has been actively investigated by many researchers using different formalisms. Some representative results include Banaszak and Krogh (1990), Reveliotis, Lawley and Ferreira (1997), Fanti, Maione, Mascolo and Turchiano (1997), Lawley, Reveliotis and Ferreira (1998) and Reveliotis, Lawley and Ferreira (1998). Among those, we adopt the *resource allocation*

[1] The admissibility will be formally defined later in the section.

system (RAS) theory presented by Reveliotis et al. (1998) in order to formally present the proposed framework. According to the RAS theory, a modular FMS RAS is defined by a set of workstations, denoted by $\mathcal{W} = \{W_i, i = 1, \ldots, m\}$, and a set of job types, $\mathcal{J} = \{J_j, j = 1, \ldots, n\}$. Each workstation W_i is characterized by its *buffering capacity* $b(W_i)$ (the number of buffers) and *processing capacity* $p(W_i)$ (the number of processors). Job type J_j is defined by a sequence $\langle J_{jk},\ k = 1, \ldots, l(J_j)\rangle$, where J_{jk} is the kth job stage of job type J_j and $l(J_j)$ is the number of job stages for job type J_j. Let $w(J_{jk})$ specify the workstation required for executing the job stage J_{jk}. The route of job type J_j is then defined by a sequence $\langle w(J_{jk}), k = 1, \ldots, l(J_j)\rangle$. The state s of a modular FMS, is defined as a collection of each workstation state which is further characterized by the number of buffers allocated to job stages. Formally, s is the collection of workstation states $s_{W_i}, i = 1, \ldots, m$, where s_{W_i} is the matrix of dimension $n \times l_{\max}$, and $l_{\max} \equiv \max_j\{l(J_j)\}$. The value of the (p, q)th element, $s_{W_i}(p, q)$, is the number of buffers allocated to job instances of stage J_{pq} at workstation W_i.

As an example, consider a small FMS depicted in Fig. 3 which is composed of three workstations of unit buffering capacity. There are two job types J_1 and J_2 requiring the routes $\langle W_1, W_2, W_3\rangle$ and $\langle W_3, W_2, W_1\rangle$, respectively. The stacker crane is assumed to be capable of transferring jobs between any locations in the FMS. The entire state transition diagram (Kozen, 1997) for the FMS is depicted in Fig. 4, where $s^i, i = 1, \ldots, 26$, stands for an instance of state. A *transition* is defined as an ordered state pair (s^i, s^j) represented as an arc in Fig. 4, and each transition is associated with its start and finish events. It is easy to see that the state in Fig. 4 evolves according to the three types of events, namely, *job loading* (retrieval of a job from the stocker to a workstation), *job advancing* (a job transfer between workstations), and *job unloading* (storage of a job from a workstation to the stocker). A state marked black in Fig. 4 is a *deadlock* state where no further state advancement is possible. An *unsafe* state which unavoidably leads to a deadlock state is marked gray. Finally, a state circled with a dashed line is an unreachable state if an FMS starts with the initial state s_0. Hence, at any state $s^i, i = 1, \ldots, 26$, all the transitions leading to unsafe or deadlock states should be prohibited in order to avoid a system deadlock, and the resulting safe transitions are characterized as *admissible*. Formally, in order to avoid deadlock in the least restrictive way, the system operation should be confined to the maximal communicating class that contains s_0. Let us denote this class by Q_s. Then every state $s \in Q_s$ is characterized as safe, while states s belonging to the complement of Q_s, denoted by Q_u, are characterized as unsafe. Therefore, the

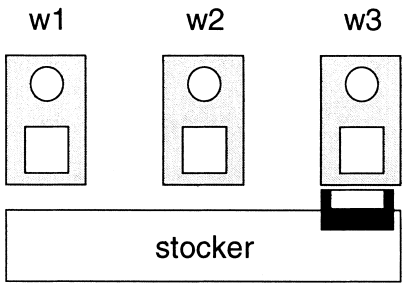

Fig. 3. Example FMS.

objective is to prevent the system from entering the unsafe region of its state space, Q_u. However, in general, the admissibility test to decide whether or not a state $s \in Q_u$ is an $\mathcal{NP}$-complete problem which is not computationally efficient (Reveliotis et al., 1998). As a result, with a possible sacrifice of optimality, we use a class of polynomially computable deadlock avoidance policies proposed in the past research which are generally suboptimal but guarantee deadlock free operation (Banaszak & Krogh, 1990; Fanti et al., 1997; Reveliotis et al., 1997; Lawley et al., 1998; Reveliotis et al., 1998). For instance, the policy RUN (Resource Upstream Neighborhood) (Reveliotis & Ferreira, 1996), checks the admissibility by use of a system of linear inequalities, and the policy presented by Reveliotis et al. (1997) is a polynomially computable optimal policy for the special RAS classes where $b(W_i) \geq 2$ for all $i = 1, \ldots, m$. Since each policy provides its own test condition to decide whether state s belongs to Q_u, we characterize this test as a predicate safe(s). Therefore, at any state s^i, as long as the system supervisor permits the transition (s^i, s^j) only if the predicate safe $(s^j) = \text{TRUE}$, no deadlock will occur.

Formally, let e^s_{jk} (respectively, e^f_{jk}) denote the start (respectively, finish) event of job stage J_{jk}, and e^a_j (respectively, e^d_j) denote the arrival (respectively, departure) event of job type J_j. From now on, when we refer to an event associated with specific job instance ζ_x instead of a job type, we use the subscript x in the place of j :e^s_{xk}, for instance. Similarly, the definitions of simple functions such as $w(\cdot)$ and $l(\cdot)$ are also extended for the job instances: That is, $w(\zeta_{xk})$

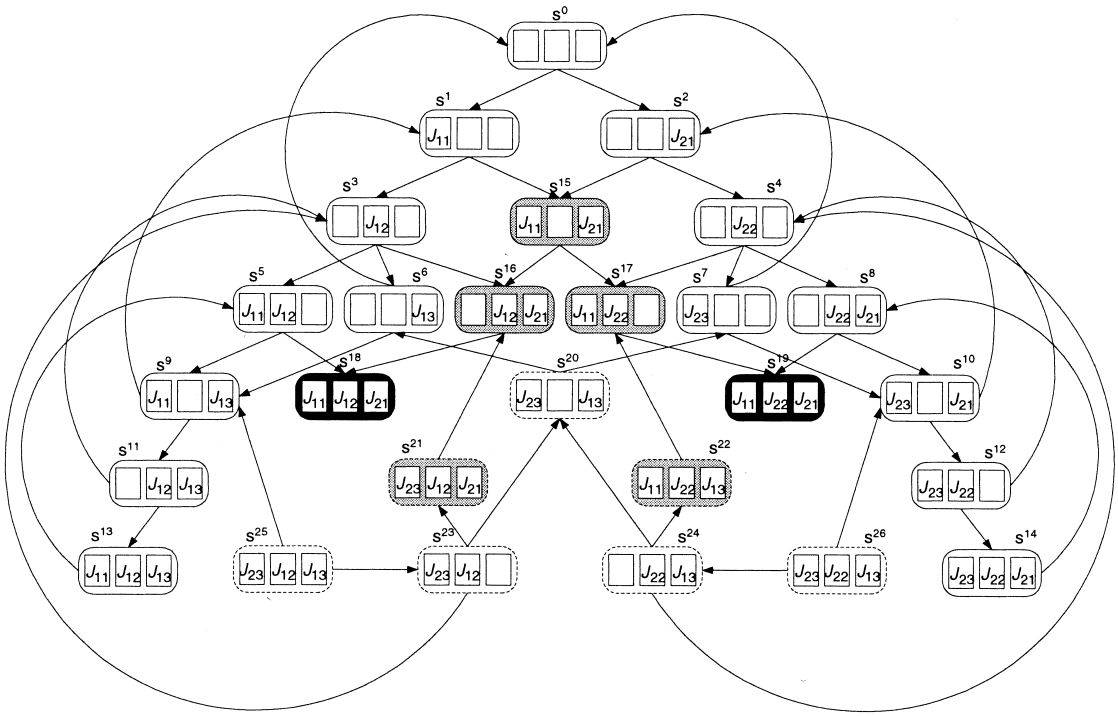

Fig. 4. State transition diagram for the example FMS.

represents the workstation required for stage k of ζ_x, and $l(\zeta_x)$ represents the number of job stages defined for ζ_x. e^s_{jk} is the only event type that can be enforced by the system supervisor. Thus, it follows that the FMS behavior should be controlled by *enabling* or *disabling* e^s_{jk} based on the observation of occurrences of other event types e^f_{jk}, e^a_j, and e^d_j. Job stage start event, e^s_{xk}, is said to be *feasible* at state s if $\sum_p \sum_q s_{w(\zeta_{xk})}(p, q) < b(w(\zeta_{xk}))$. Feasibility implies that one buffer can be accommodated for a job instance in consideration. Hence, if e^s_{xk} is not feasible, ζ_x is characterized as *blocked*, and it should stay at its current workstation $w(\zeta_{x,k-1})$. Next, if we denote the set of feasible job stage start events at state s by $\mathscr{F}(s)$, the *admissibility* of each event in $\mathscr{F}(s)$ is decided by looking up the *one-step ahead* state which is defined as follows:

Definition 1. *The one-step ahead FMS state from state s by event* e^s_{jk}, *denoted by* $\tilde{s}(e^s_{jk})$, *is defined as*:

$$\begin{aligned}
&\tilde{s}_{W_i}(j, k-1) \leftarrow s_{W_i}(j, k-1) - 1, && \text{if } W_i = w\left(J_{j,k-1}\right) \text{ and } k > 1\\
&\tilde{s}_{W_i}(j, k) \leftarrow s_{W_i}(j, k) + 1, && \text{if } W_i = w\left(J_{jk}\right) \text{ and } k \le l\left(J_j\right)\\
&\tilde{s}_{W_i}(j, k) \leftarrow s_{W_i}(j, k), && \text{otherwise}
\end{aligned}$$

That is, job stage start event, e^s_{xk}, is identified as *admissible* at state s if $e^s_{xk} \in \mathscr{F}(s)$ and safe $(\tilde{s}_f(e^s_{jk})) = \text{TRUE}$, where j is the job type of job instance ζ_x. Clearly, e^s_{xk} is admissible when $k = l(\zeta_x) + 1$, since the job is unloaded from an FMS and this transition is always safe. Hence, the system supervisor makes *state-based* decisions, and it bipartites set $\mathscr{F}(s)$ into the set of admissible events and the set of inadmissible events at every state s. It should be noted that only two states (s and $\tilde{s}$) are required for admissibility decision. Accordingly, the whole state space (such as Fig. 4) does not have to be generated nor stored.

On the performance side, a job stage start event is characterized as *schedulable* if its immediate execution is necessary according to some performance measure in consideration. Hence, the schedulability decision of an event is independent of logical control decision. When every admissible event is immediately schedulable, we characterize such a scheduling policy as *non-idling*. However, in some cases (for instance, in flow rate-based performance control models by Connors, Feigin and Yao (1994) and Gershwin (1989, 1994)) it is necessary to delay triggering time of an admissible event intentionally in an attempt to optimize overall system performance (e.g. average waiting time or inventory level). In this case, dispatching of an admissible event should be subject to the permission of the scheduling policy, and the system supervisor enables a job stage start event only if it is schedulable as well as admissible. For instance, the schedulability decision for e^s_{x1} corresponds to on-line job loading problem, and the event becomes *dispatchable* if it is admissible and granted by the scheduling policy. Hence, if an FMS is operated under the constant WIP (Work In Process) scheduling policy (Hopp & Spearman, 1996), say c jobs, more than c jobs will not be induced into the system regardless of their admissibility. On the other hand, if a non-idling releasing policy is used, the policy becomes pure *push* type.

The dispatchability is formally characterized by the following predicate definition:

Definition 2. *Given job stage start event,* e^s_{xk}, *the predicate* $\text{dispatchable}(e^s_{xk}) = \text{TRUE}$, *if* $e^s_{xk} \in \mathscr{F}(s)$, $\text{safe}(\tilde{s}_f(e^s_{jk})) = \text{TRUE}$, *and the scheduling policy permits immediate dispatching of* e^s_{xk}

at s, where j is the job type of job instance ζ_x. *Also, if* $k = l(J_j) + 1$, dispatchable(e^s_{xk}) = TRUE. *Otherwise*, dispatchable(e^s_{xk}) = FALSE.

Furthermore, when there is a conflict among dispatchable job stage start events, it should be resolved by the scheduling policy which selects an event according to some predefined *dispatching rule*. That is, given an available workstation buffer, the dispatching rule decides which pending job stage should advance to it. Examples of the conventional dispatching rules include FCFS (First Come First Served), SPT (Shortest Processing Time), and LBFS (Last Buffer First Served) (Baker, 1998).

Before the detail algorithms of the system supervisor and resource controllers are presented, we briefly outline the basic structure of each algorithm in Fig. 5. Each control algorithm is presented as an *event handler* which exerts necessary control actions according to the event observed. The event types perceived by each controller are already shown in Fig. 2. In the subsequent sections, we first present the algorithm of the workstation controller which is the most primitive component in the modular FMS, and it is followed by the algorithms of the transporter and stocker controllers. Finally, the system supervisor algorithm for coordination of the resource controllers is given in Section 3.3.

3.2. The resource controllers

3.2.1. The workstation controller

There are m workstations controllers, where m is the number of workstation in an FMS. Since the behavior of each workstation controller is identical, discussion is based on the ith workstation, $i = 1, \ldots, m$. Once a job is transferred to one of the buffer position of a workstation, it is assumed that there is no further job flow inside the workstation. Otherwise, the situation can then be modeled by (i) decomposing the workstation into several other workstations, and (ii) considering the parent workstation as a modular FMS. Under this

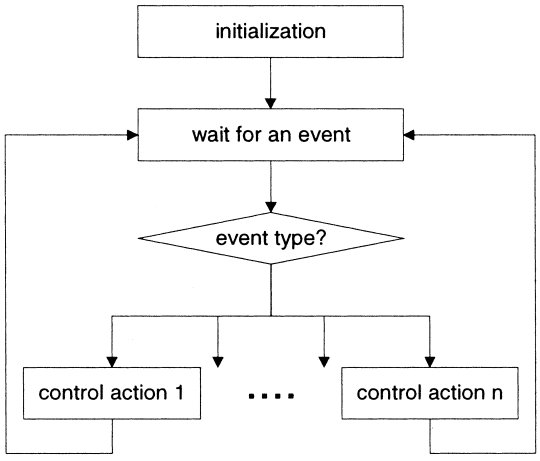

Fig. 5. Basic algorithm structure for event handling.

assumption, no blocking is possible, and every job stage start event is admissible once a job is at the workstation. Each buffer is of unit capacity, and all processors are assumed to be functionally equivalent. Hence any processor can serve a job in the workstation. However, in case there is a difference among the processors in terms of processing time and quality, the preference should be reflected in the *processor scheduling rule* which selects a processor from a set of idle processors whenever there are more than one processors capable of processing a job.

The responsibilities of the workstation controller include (i) reaction to the job stage start event dispatched from the system supervisor, and (ii) notification of the job stage finish event when it is observed from one of its processors. Specifically, in the proposed framework, event e^s_{xk} is dispatched to the workstation controller whenever the job instance ζ_x waiting for its kth job stage is available at one of the workstation's buffers. Therefore, if we denote the number of idle processors by π (its initial value is $p(W_i)$), on event e^s_{xk}, the workstation controller starts processing for the kth stage of job ζ_x if $\pi \geq 1$. Otherwise, e^s_{xk} is added to event queue $\mathcal{Q}_{W_i}$ maintained by the workstation controller. On the other hand, when the workstation controller receives job stage finish event, e^f_{xk}, from one of its processors, it immediately notifies it to the system supervisor, and commands another job stage start event from $\mathcal{Q}_{W_i}$ as long as the queue is not empty. Selection of an event is based on *job scheduling rule* which selects the next job to be processed from $\mathcal{Q}_{W_i}$ whenever a processor becomes idle. Conceptually, design of an optimal processor and job scheduling rule corresponds to solving a parallel machine scheduling problem (Baker, 1974, 1998), since no consideration is necessary for finite buffering and job transfer time. However, an on-line scheduling algorithm should be devised in order to cope with the dynamic job arrivals. Finally, we mention that, the effect of the job scheduling rule of the workstation is more local than that of the system supervisor, both rules should be consistently defined. Now, the control algorithm of the workstation controller is presented as follows:

- e^s_{xk} from the system supervisor: If $\pi \geq 1$, select one processor based on the processor scheduling rule, command the processor to start an operation for ζ_{xk}, and $\pi \leftarrow \pi - 1$. Otherwise, add e^s_{xk} to $\mathcal{Q}_{W_i}$.
- e^f_{xk}: Notify e^f_{xk} to the system supervisor. $\pi \leftarrow \pi + 1$. If $\mathcal{Q}_{W_i}$ is not empty, select one event (denoted by e^s_{yl}) according to the job scheduling rule, command processing for ζ_{yl}, remove e^s_{yl} from $\mathcal{Q}_{W_i}$, and $\pi \leftarrow \pi - 1$.

3.2.2. The transporter controller

Whenever event e^s_{xk} becomes dispatchable, the system supervisor immediately requests the transporter controller a transfer of job ζ_x from $w(\zeta_{x,\,k-1})$ (referred to as a source) to $w(\zeta_{x,\,k})$ (referred to as a destination). Transferring a job from one buffer location to another consists of two phases: when the transporter is assigned a transport operation, it makes an unloaded trip to the source from its current location (defined as *inbound travel*), picks up the job at the source, makes a loaded trip to the destination, and places the job at the destination (these last three operations are collectively defined as *outbound travel*). Let $\bar{e}^{is}_{xk}$ (respectively, $\bar{e}^{if}_{xk}$, $\bar{e}^{os}_{xk}$, $\bar{e}^{of}_{xk}$) denote the inbound start (respectively, inbound finish, outbound start, outbound finish) event associated with the transfer of job instance ζ_x waiting for its stage k. Note that the feasibility

of event $\bar{e}^{is}_{xk}$ is already guaranteed by the system supervisor, and therefore no deadlock will occur during a job transfer.

The basic structure of the algorithm is similar to that of the workstation controller, except the scheduling rule for conflict resolution if there are more than one transfer requests waiting for the transporter. This rule is referred to as *transfer scheduling rule*, i.e. it decides, given an idle transporter, which pending transfer request should be selected next. Examples of the transfer scheduling rules are FCFS, STT (Shortest Travel Time) and LWT (Longest Waiting Time) (Ganesharajah, Hall & Sriskandarajah, 1998).

The control algorithm of the transporter controller is presented below. We assume that the system supervisor requests a job transfer by dispatching event $\bar{e}^{is}_{xk}$ to the transporter controller.

- $\bar{e}^{is}_{xk}$ dispatched from the system supervisor: If the transporter is idle, set the transporter state busy and command $\bar{e}^{is}_{xk}$. Otherwise, add $\bar{e}^{is}_{xk}$ into event queue $\mathcal{Q}_t$.
- $\bar{e}^{os}_{xk}$: Notify $\bar{e}^{os}_{xk}$ to the system supervisor.
- $\bar{e}^{of}_{xk}$: Notify $\bar{e}^{of}_{xk}$ to the system supervisor. Set the transporter state idle. If $\mathcal{Q}_t$ is not empty, select an event (denoted by $\bar{e}^{is}_{yl}$) from $\mathcal{Q}_t$ according to the transporter scheduling rule, set the transporter state busy, command $\bar{e}^{is}_{yl}$, and remove $\bar{e}^{is}_{yl}$ from $\mathcal{Q}_t$.

3.2.3. The stocker controller

The algorithm for the stocker controller is explicitly defined in this section for the case where loading and unloading operations are performed by a specific device such as a S/R machine (or stacker crane) instead of a transporter. For example, in Fig. 1, material transfers between the stocker and workstation 1 are made by the S/R machine and not by the AGV. In this case, the system supervisor should dispatch the corresponding events $\bar{e}^{is}_{x1}$ and $\bar{e}^{is}_{xk}$ (when $k = l(\zeta_x) + 1$) to the stocker controller of which the control algorithm is defined as follows:

- $\bar{e}^{is}_{x1}$ dispatched from the system supervisor (This corresponds to a retrieval request.): If S/R machine is idle, set S/R machine state busy and command the retrieval of ζ_x. Otherwise, add $\bar{e}^{is}_{x1}$ to event queue $\mathcal{Q}_r$.
- $\bar{e}^{is}_{xk}$ $(k = l(\zeta_x) + 1)$ dispatched from the system supervisor (This corresponds to a storage request.): If S/R machine is idle, set S/R machine state busy and command the storage of ζ_x. Otherwise, add $\bar{e}^{is}_{k}$ to event queue $\mathcal{Q}_s$.
- $\bar{e}^{os}_{xk}$ $(k = l(\zeta_x) + 1)$ from S/R machine (This event occurs when S/R machine picks up job $\zeta_x$and starts storage operation.): Notify $\bar{e}^{os}_{xk}$ to the system supervisor.
- $\bar{e}^{of}_{x1}$ (respectively, $\bar{e}^{of}_{xk}$ $(k = l(\zeta_x) + 1)$) from S/R machine (This is the retrieval finish (respectively, storage finish) event.): Notify the received event to the system supervisor. Set S/R machine state idle. If $\mathcal{Q}_r$ is not empty and the performance control policy of the stocker controller decides the retrieval mode, select an event (denoted by $\bar{e}^{is}_{y1} \in \mathcal{Q}_r$) according to the retrieval scheduling rule, command the retrieval of ζ_y, and remove $\bar{e}^{is}_{y1}$ from $\mathcal{Q}_r$. If the performance control policy decides the storage mode, select an event (denoted by $\bar{e}^{is}_{yl} \in \mathcal{Q}_s$) according to the storage scheduling rule, command the storage of ζ_y, and remove $\bar{e}^{is}_{yl}$ from $\mathcal{Q}_s$. If the event is commanded, set S/R machine state busy.

The scheduling rules for the retrieval and storage decide which pending request should be

processed next. Some examples include FCFS and Nearest Neighbor rules presented by Han, McGinnis, Shieh and White (1987). For the sake of simplicity, we omitted the procedure of updating the current state of storage positions in the above algorithm (e.g. storage location i has job instance ζ_x, etc.) Therefore, event handling for $\bar{e}^{os}_{x1}$ is not included in the algorithm, since it is not necessary to notify this event to the system supervisor for control purpose.

3.3. The system supervisor

The system supervisor is responsible for inducing and advancing jobs. On job arrival event, e^a_x, it decides whether the event corresponding to the induction of job ζ_x (i.e. e^s_{x1}) is dispatchable. Also, whenever a job stage finish event is notified from a workstation controller (i.e. e^f_{xk}), it decides whether advancing the job to the next job stage is dispatchable or not by testing dispatchable($e^s_{x,\,k+1}$). In both cases, if the event is not dispatchable, it is inserted to event queue $\mathcal{Q}_\Sigma$ so that it can be considered again at future event occurrences. On the other hand, if the event is dispatchable, the system supervisor grants the occurrence of event e^s_{x1} (respectively, $e^s_{x,\,k+1}$) by dispatching $\bar{e}^{is}_{x1}$ (respectively, $\bar{e}^{is}_{x,\,k+1}$) to the transporter or stocker controller. Every time an inbound start event $\bar{e}^{is}_{xk}$ is dispatched, system state s is updated by function advance($\bar{e}^{is}_{xk}$) defined as follows:

Definition 3. advance(e): *advances FMS state s by the following rules*:

if $e = \bar{e}^{is}_{xk}$,

$s_{W_i}(j,k) = s_{W_i}(j,k) + 1$, *if* $W_i = w(\zeta_{xk})$ *and* $k \le l(\zeta_x)$

if $e = \bar{e}^{os}_{xk}$,

$s_{W_i}\big(j,(k-1)\big) = s_{W_i}\big(j,(k-1)\big) - 1$, *if* $W_i = w\big(\zeta_{j,k-1}\big)$ *and* $k > 1$

where j *is the job type of* ζ_x.

In Definition 3, a destination buffer is committed when the system supervisor dispatches a job transfer. This is crucial in order to handle the time delay between dispatching an event start and the actual start of the event. That is, if event $\bar{e}^{is}_{xk}$ is dispatched, $w(\zeta_{xk})$ needs to be allocated to job instance ζ_x immediately even though ζ_x is not indeed at $w(\zeta_{xk})$. This is to prevent accidental over-allocation of $w(\zeta_{xk})$ by another job instance. Therefore, at the time of dispatching $\bar{e}^{is}_{xk}$, ζ_x is considered to be allocated to both of $w(\zeta_{x,\,k-1})$ and $w(\zeta_{xk})$. Then, $\bar{e}^{os}_{xk}$ will occur eventually, and therefore ζ_x is de-allocated from $w(\zeta_{x,\,k-1})$ at this time by invoking advance($\bar{e}^{os}_{xk}$). Furthermore, event $\bar{e}^{os}_{xk}$ implies that state is changed: i.e. one buffer at $w(\zeta_{x,\,k-1})$ becomes empty. Therefore, on $\bar{e}^{os}_{xk}$, the system supervisor decides (based on the dispatching rule) which job stage in event queue $\mathcal{Q}_\Sigma$ should be advanced to the empty buffer, if $\mathcal{Q}_\Sigma$ is not empty. Finally, $\bar{e}^{of}_{xk}$ will be notified to the system supervisor, which means job ζ_x has now arrived at $w(\zeta_{xk})$, $k = 1,\ldots,l(\zeta_x)$. In this case, the system supervisor sends event e^s_{xk} to the workstation controller responsible for $w(\zeta_{xk})$, notifying the job arrival.

Hence the event types observed by the factory supervisor are e^a_x, e^f_{xk}, $\bar{e}^{os}_{xk}$, and $\bar{e}^{of}_{xk}$. Note that, for the purpose of the logical and performance control, other event types such as e^s_{xk}, $\bar{e}^{is}_{xk}$, and $\bar{e}^{if}_{xk}$ need not to be observed in the proposed framework, although they may be important for other purposes such as system monitoring. The control algorithm of the system supervisor is defined as follows:

- e^a_x: If dispatchable(e^s_{x1}) = TRUE, dispatch $\bar{e}^{is}_{x1}$ to the corresponding resource (transporter or stocker) controller, and advance($\bar{e}^{is}_{x1}$). Otherwise add e^s_{x1} to $\mathcal{Q}_\Sigma$.
- e^f_{xk}: If dispatchable($e^s_{x,\,k+1}$) = TRUE, dispatch $\bar{e}^{is}_{x,\,k+1}$ to the corresponding resource (transporter or stocker) controller, and advance($\bar{e}^{is}_{x,\,k+1}$). Otherwise add $e^s_{x,\,k+1}$ to $\mathcal{Q}_\Sigma$.
- $\bar{e}^{os}_{xk}$:
  1. advance($\bar{e}^{os}_{xk}$).
  2. Build the set of admissible events $\mathcal{A}_\Sigma$ from $\mathcal{Q}_\Sigma$ by testing admissibility of an event in $\mathcal{Q}_\Sigma$.
  3. while ($|\mathcal{A}_\Sigma| \geq 1$) do
     (a) select an event (denoted by e^s_{yl}) from $\mathcal{A}_\Sigma$ according to the dispatching rule.
     (b) if dispatchable (e^s_{yl}) = TRUE, dispatch $\bar{e}^{is}_{yl}$, advance($\bar{e}^{is}_{yl}$), remove e^s_{yl} from $\mathcal{Q}_\Sigma$, and $\mathcal{A}_\Sigma \leftarrow \varnothing$.
     (c) else remove e^s_{yl} from $\mathcal{A}_\Sigma$.
- $\bar{e}^{of}_{xk}$: If $k \leq l(\zeta_x)$, notify e^s_{xk} to the workstation controller responsible for $w(\zeta_{xk})$.

For event $\bar{e}^{of}_{xk}$, if $k = 1$, it implies that ζ_x is retrieved from the stocker, and if $k = l(\zeta_x) + 1$, it means that ζ_x is stored into the storage system. Notice that if the last job stage is finished (i.e. e^f_{xk}, where $k = l(\zeta_x)$), $e^s_{x,\,k+1}$ is recognized as dispatchable by definition, and $\bar{e}^{is}_{x,\,k+1}$ is dispatched for storage to the stocker. Now, since event $\bar{e}^{of}_{xo}$ is not notified to any of the workstation controllers, if $o = l(\zeta_x) + 1$, it is clear that e^s_{xo} (where $o \geq l(\zeta_x) + 1$) is not generated from the workstation controllers. Therefore, only $l(\zeta_x)$ number of job finish events are notified to the system supervisor regarding job instance ζ_x.

4. Performance analysis

In this section, performance and reconfigurability of the proposed framework are demonstrated by simulation results for two different FMS configurations. The simulation is conducted as follows: We first implement the proposed control framework using Java language. Job arrivals, workstations, and material handling devices are simulated by concurrently running processes in such a way that the processes generate the same event streams as the corresponding real entities do. Hence, the whole simulation becomes distributed, and it is performed by constituent processes exchanging a series of messages with the control system. Therefore, the simulation itself can serve as a verifier for the correctness of the proposed framework, and the framework becomes a control system if we replace the simulating processes by the real entities.

We first consider the configuration depicted in Fig. 3 with a slight modification: We assume that additional buffer is installed for W_2, hence $b(W_1) = b(W_3) = 1$, $b(W_2) = 2$, and $p(W_1) = p(W_2) = p(W_3) = 1$. Two cases are considered for this configuration in order to demonstrate the reconfigurability with respect to different job mix: In Case A, two job types J_1 and J_2 are

supported with their corresponding routes $W_1(10)\rightarrow W_2(15)\rightarrow W_3(5)$ and $W_3(15)\rightarrow W_2(5)\rightarrow W_1(10)$, where the number in the parenthesis represents an average processing time. Each processing time for a job stage is assumed to follow an exponential distribution. In Case B, one more job type J_3 in addition to J_1 and J_2 is introduced. The route for J_3 is re-entrant: $W_1(5)\rightarrow W_2(5)\rightarrow W_3(5)\rightarrow W_1(10)\rightarrow W_2(10)\rightarrow W_3(10)$. Clearly, in both cases, the system is deadlock prone due to the counter flow characteristics of the job routes, and we use a disjunction of deadlock avoidance policy RUN presented by Reveliotis and Ferreira (1996) for the instantiation of predicate safe(). Simulation is performed for the period during which three hundred job instances for each job type arrive to the FMS. The inter-arrival time distributions for job types J_1, J_2, and J_3 are assumed to follow exponentially with parameters 30, 30 and 60, respectively. As discussed in Section 3.1, in this particular FMS configuration, the stacker crane is capable of transferring jobs between workstation buffers, hence the transporter is not modelled. FCFS rule is used for both storage and retrieval scheduling. The following table shows average makespan results under the various dispatching rules.

	FCFS	LCFS	SPT	LPT	LBFS	Random
Case A	10,262	10,233	10,409	10,562	10,649	10,483
Case B	18,745	19,036	19,022	19,114	19,265	18,836

Next, we consider the configuration depicted in Fig. 1 where $b(W_1) = b(W_3) = b(W_4) = b(W_5) = 2$, $b(W_2) = 4$, $p(W_1) = p(W_3) = p(W_4) = 1$, and $p(W_2) = p(W_5) = 2$. Two cases are considered for this configuration: In Case C, two job types J_1 and J_2 are supported with the corresponding routes $W_1(1)\rightarrow W_2(10)\rightarrow W_3(5)\rightarrow W_4(10)\rightarrow W_5(10)\rightarrow W_1(1)$, and $W_1(1)\rightarrow W_5(15)\rightarrow W_4(10)\rightarrow W_3(10)\rightarrow W_2(20)\rightarrow W_1(1)$. In Case D, two more job types J_3 and J_4 in addition to J_1 and J_2 are introduced. The routes for J_3 and J_4 are $W_1(1)\rightarrow W_2(5)\rightarrow W_4(5)\rightarrow W_1(1)$, and $W_1(1)\rightarrow W_3(5)\rightarrow W_5(10)\rightarrow W_1(1)$, respectively. It is clear that this system is also susceptible to deadlock. Hence, by noting that $b(W_i)\geq 2, i = 1,\ldots,5$, we use the optimal deadlock avoidance policy presented by Reveliotis et al. (1997) for the instantiation of predicate safe().

Simulation is performed for the period during which three hundred job instances for each job type arrive to the FMS. The inter-arrival time distributions for job types J_1, J_2, J_3, and J_4 are defined as exponential with parameters 25, 30, 25 and 30, respectively. The processing time for each job stage is assumed to follow exponential distribution. Finally, the scheduling rules used for transfer, storage, and retrieval are FCFS. Note that in this configuration, W_1 is defined at the beginning and end of routes for all job types to represent loading and unloading operations. The following table shows average makespan results under the various dispatching rules.

	FCFS	LCFS	SPT	LPT	LBFS	Random
Case C	8,681	10,733	9,410	9,449	8,543	9,159
Case D	10,237	11,842	10,898	10,691	11,537	11,423

Overall simulation results show that the performance of a specific FMS configuration can be improved if an appropriate dispatching rule is used. However, one cannot expect that a single dispatching rule always dominates others since the performance of a rule depends on the FMS configuration, part mix produced, logical control policy, and scheduling rules employed by the resource controllers. Therefore, a rigorous simulation study is required to determine the best scheduling rules for a specific configuration before the final control system is implemented.

5. Conclusion

We presented a generic event control framework in which different FMS configurations and control policies can be accommodated. Development of the proposed framework was motivated by the need for easily implementable control systems for large-scale, industrial FMSs. Hence, instead of taking formal model-based approaches which are complex for modeling and control synthesis, we proposed an architecture and computationally efficient algorithms by limiting our modeling domain to the class of modular FMSs. A modular FMSs is defined as a set of workstations inter-connected by a material handling system consisting of a transporter and a stocker. We believe that many of the existing FMS implementations in industry fall into this class. Furthermore, the modularity also allows the capability to effectively model a complex FMS by decomposing it into several modular FMSs.

The proposed control architecture consists of a set of resource controllers managing specific FMS devices, and the system supervisor coordinating them. The resource controllers are further classified into the workstation, transporter, and stocker controllers. As the controllers exchange a series of events based on the pre-defined protocols, they are defined as the event handlers in which control actions are made on the event occurrences. Specifically, for each controller, the event-based control structure and the generic logical and performance control functions are identified. Since the proposed framework is defined in terms of a set of distributed processes with the well-defined protocols and algorithms, it is expected that the framework is easily implementable for most industrial implementations. The performance and reconfigurability is demonstrated by a distributed simulation from which we can also verify the correctness of the proposed algorithms. As a further extension, we are pursuing the extension of the proposed framework to the configurations where there are more than one transporters. Also, the incorporation of more sophisticated scheduling policies (e.g., Connors et al., 1994; Gershwin, 1989) into our framework is a part of our future work.

Acknowledgements

This work was supported by Ministry of Education through Inter-University Semiconductor Research Center (ISRC 96 - E - 1065) in Seoul National University.

References

Atherton, L. F., & Atherton, R. W. (1995). *Wafer fabrication: factory performance and analysis*. Dordrecht: Kluwer Academic Publishers.

Baker, A. D. (1998). A survey of factory control algorithms that can be implemented in a multi-agent heterarchy: dispatching, scheduling, and pull. *Journal of Manufacturing Systems*, *17*(4), 297–320.

Baker, K. R. (1974). *Introduction to Sequencing and Scheduling*. New York: Wiley.

Banaszak, Z. A., & Krogh, B. H. (1990). Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. *IEEE Transactions on Robotics and Automation*, *6*(6), 724–734.

Connors, D., Feigin, G., & Yao, D. (1994). Scheduling semiconductor lines using a fluid network model. *IEEE Transactions on Robotics and Automation*, *10*, 88–98.

Daltrini, A. M., & Kumara, S. R. T. (1996). FMS modeling using timed extended moore machine network. In *Proceedings of the Japan–USA Symposium on Flexible Automation*.

Ezpeleta, J., & Colom, J. M. (1997). Automatic synthesis of colored petri nets for the control of fms. *IEEE Transactions on Robotics and Automation*, *13*(3), 327–337.

Fanti, M. P., Maione, B., Mascolo, S., & Turchiano, B. (1997). Event-based feedback control for deadlock avoidance in flexible production systems. *IEEE Transactions on Robotics and Automation*, *13*, 347–363.

Ganesharajah, T., Hall, N. G., & Sriskandarajah, C. (1998). Design and operational issues in agv-served manufacturing systems. *Annals of Operations Research*, *76*, 109–154.

Gershwin, S. B. (1989). Hierarchical flow control: a framework for scheduling and planning discrete events in manufacturing systems. *Proceedings of the IEEE*, *77*, 195–209.

Gershwin, S. B. (1994). *Manufacturing systems engineering*. Englewood Cliffs, NJ: PTR Prentice Hall.

Han, M. H., McGinnis, L. F., Shieh, J. S., & White, J. A. (1987). On sequencing retrievals in an automated storage/retrieval system. *IIE Transactions*, *19*(1), 56–66.

Hopp, W. J., & Spearman, M. L. (1996). *Factory physics: Foundations of manufacturing management*. Irwin: Homewood.

Joshi, S. B., Mettala, E. G., Smith, J. S., & Wysk, R. A. (1995). Formal models for control of flexible manufacturing cells: Physical and system model. *IEEE Transactions on Robotics and Automation*, *11*(4), 558–570.

Joshi, S. B., Mettala, E. G., & Wysk, R. A. (1992). Cimgen — a computer aided software engineering tool for development of FMS control software. *IIE Transactions*, *24*(3), 84–97.

Kozen, D. C (1997). *Automata and computability*. Berlin: Springer Verlag.

Lauzon, S. C., Mills, J. K., & Benhabib, B. (1997). An implementation methodology for the supervisory control of flexible manufacturing cells. *Journal of Manufacturing Systems*, *16*(2), 91–101.

Lawley, M., Reveliotis, S., & Ferreira, P. (1998). A correct and scalable deadlock avoidance policy for flexible manufacturing systems. *IEEE Transactions on Robotics and Automation*, *14*(5), 796–809.

Moody, J. O., & Antsaklis, P. J. (1998). *Supervisory control of discrete event systems using petri nets*. In *The kluwer international series on discrete event dynamic systems*. Dordrecht: Kluwer Academic Publishers.

Naylor, A. W., & Volz, R. A. (1987). Design of integrated manufacturing system control software. *IEEE Transactions on Systems, Man, and Cybernetics*, *17*(6), 881–897.

Qiu, R. G., & Joshi, S. B. (1996). Rapid prototyping of control software for automated manufacturing systems using supervisory control theory. *ASME, Manufacturing Engineering Division*, *4*, 95–101.

Ramadge, P. J. G., & Wonham, W. M. (1989). The control of discrete event systems. *Proceedings of the IEEE*, *77*, 81–98.

Reveliotis, S. A., & Ferreira, P. M. (1996). Deadlock avoidance policies for automated manufacturing cells. *IEEE Transactions on Robotics and Automation*, *12*(6), 845–857.

Reveliotis, S. A., Lawley, M. A., & Ferreira, P. M. (1997). Polynomial complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Transactions on Automatic Control*, *42*(10), 1344–1357.

Reveliotis, S. A., Lawley, M. A., & Ferreira, P. M. (1998). Structural control of large-scale flexibly automated manufacturing systems. In C. T. Leondes, *Computer aided and integrated manufacturing systems: Techniques and applications*. London: Gordon and Breach.

Shatz, S. M. (1993). *Development of distributed software: concepts and tools*. New York: Macmillan.

Smith, J. S., Hoberecht, W. C., & Joshi, S. B. (1996). A shop floor control architecture for computer integrated manufacturing. *IIE Transactions*, *28*(10), 783–794.

Smith, J.S., & Joshi, S.B. (1994). Message-based part state graphs (MPSG): a formal model for shop floor control. Technical report, Department of Industrial Engineering, Texas A and M University.

Venkatesh, K., & Zhou, M. (1998). Object-oriented design of fms control software based on object modeling technique diagrams and petri nets. *Journal of Manufacturing Systems*, *17*(2), 118–136.

Zhou, M., & Dicesare, F. (1993). *Petri net synthesis for discrete event control of manufacturing systems*. Dordrecht: Kluwer Academic Publishers.

Zhou, M., & Venkatesh, K. (1999). *Modeling, simulation, and control of flexible manufacturing systems: A petri net approach*. Singapore: World Scientific.