Deadlock Avoidance Policies for Automated Manufacturing Cells*

Spiridon A. Reveliotis and Placid M. Ferreira†

Department of Mechanical and Industrial Engineering

University of Illinois at Urbana-Champaign

Urbana, IL 61801

Abstract

Although the typical process-layout manufacturing environment is susceptible to deadlocks, the problem of deadlock resolution in this context has only lately been undertaken by the scientific community. Previous studies have found that deadlock avoidance methodologies seem to be the most appropriate for this particular context. Unfortunately, in the general case, these methods suffer from high computational complexity which results in heuristic solutions and/or reduced performance. Taking the position that any solution to the problem should be scalable and provably correct, this paper proposes an analytical framework for designing deadlock avoidance policies for a subclass of Resource Allocation Systems (RAS). Specifically, this subclass is characterized by the fact that jobs in the system are defined by deterministic job-step sequences with every step in the sequence requiring a single unit of the system resources. Job-step models are appropriate for the study of the deadlock problem in the context of the Automated Manufacturing Cell (AMC). The functionality of the framework is demonstrated by using it to prove the correctness of two avoidance policies: the Resource Upstream Neighborhood (RUN) policy, developed in the authors' research program and first presented in [10], and the B-K Deadlock Avoidance

*To appear in the IEEE Trans. on Robotics & Automation.

†Corresponding author

Algorithm (BKDAA), presented in [3]. An interesting relationship existing between RUN and BKDAA policies is also identified and explored.

NOTATION

$\mathcal{R} = \{R_1, \ldots, R_m\}$: system resources

$\mathcal{R}^s$ / $\mathcal{R}^u$: set of shared / unshared resources (cf. Section 5.2)

$\mathcal{O}_i = \{o_{i1}, \ldots, q_{i,k(i)}\}$: e-operations supported by resource R_i

C_i: capacity of resource R_i

$JT_j =< o_{j1}, \ldots, o_{j,l(j)} >$: job type j

$\mathcal{J} = \{JT_1, \ldots, JT_n\}$: set of job types supported by the considered AMC configuration

$\mathcal{R}(i, j)$: set of resources capable of supporting the j-th operation of job type JT_i

$\mathcal{JR}_j = < JR_{j1}, \ldots, JR_{j,l(j)} > \in \mathcal{JR} \subseteq \mathcal{R}(j, 1) \times \ldots \times \mathcal{R}(j, l(j))$: a feasible route for job type JT_j

$\texttt{Sup}(R_k)$: the set of route stages supported by resource R_k (also, $|\texttt{Sup}(R_k)| \equiv Q_k$)

$\mathbf{s}(t) = s^i \in \mathcal{S}$: RAS state ($s^0$ is the initial empty and idle state of RAS)

E: the event set of the FSA modeling the AMC operation

E^*: the set of all finite sequences (strings) of events of E, including the empty string (Kleene Closure of E)

$\mathcal{F}(s^i)$: the set of feasible events in state s^i

$\mathcal{S}_r$ / $\mathcal{S}_{\bar{r}}$ / $\mathcal{S}_s$ / $\mathcal{S}_{\bar{s}}$: the subset of reachable / unreachable / safe / unsafe AMC states

$s^i \leftarrow s^j$: state s^i is reachable from state s^j

$\mathcal{P}: \mathcal{S} \rightarrow 2^E$: an avoidance policy, i.e., a function defining the enabled event set at every state

$\mathcal{S}_r(\mathcal{P})$ / $\mathcal{S}_s(\mathcal{P})$ / $\mathcal{S}_a(\mathcal{P})$: policy-reachable / policy-safe / policy-admissible sates (the latter is defined primarily for Ω policies)

δ_{ikt}: an indicator (binary) variable on whether job instance j_i is supported by resource R_k at the t-th stage of the system operational evolution (starting from a given initial state $\mathbf{s}(0)$)

$\mathtt{sr}(i, k, p_i)$: the successor resource function, returning the type required by job j_i for its next processing step, given that it is currently allocated one unit of resource R_k for the execution of its p_i-th processing step

$\mathtt{UN}(R_k)$: the Upstream Neighborhood of resource R_k (cf. Definition 9 in the discussion of RUN policy, Section 5.1)

$z_j^k = s_j^k u_j^k$: the k-th zone of job type JT_j, consisting of the shared segment s_j^k and the unshared segment u_j^k (cf. the discussion of BKDAA in Section 5.2)

$\mathcal{T}(R_k, (j, q))$: the subsequence of resources which support the steps in the shared subzone of step (j, q), that follow step (j, q) (cf. the discussion of BKDAA in Section 5.2)

1 Introduction

The problem of deadlocks that arises when two or more concurrently executing processes interact through a set of possible events has been studied by the scientific community for more than twenty years. Following [20], we can abstractly describe a *deadlock* as the situation in which there exists a set of processes such that every process in the set is waiting for an event that can only be caused by another process in the set. Typically, the aforementioned events are either (i) the acquisition and release of resource units shared by the process set, or (ii) the exchange of messages, the content of which is indispensable to the receiving process in order to successfully run to completion. In this paper, we consider only interaction through type (i) events. Since the resources involved are returned to the system after they have been used by the acquiring processes, they are characterized as *reusable*. The abstracted system of interacting processes and the set of shared resources will be characterized as a *Resource Allocation System (RAS)*.

In [15, 5] the necessary and sufficient conditions for the development of a deadlock in a RAS are identified. These conditions suggest possible methodologies for addressing the problem of deadlock. These methodologies can be broadly classified as: **(i) Prevention**, **(ii) Detection & Recovery**, and **(iii) Avoidance**. *Prevention* methods stipulate that the RAS is *designed* to be deadlock-free by ensuring that the entire set of necessary conditions for deadlock cannot be simultaneously satisfied at any point of the RAS operation. This requirement is met by imposing a number of restrictions on how a process can request the required resources. As a result, such methods are overly conservative, generally leading to significant underutilization of system resources. *Detection & Recovery* methods allow the RAS to enter a deadlock state. A monitoring mechanism is used to detect a deadlock when it occurs and a resolution algorithm *rolls* the system *back* to a safe state by appropriately preempting resources from a number of deadlocked processes. By definition, a state is *safe*, if there exists a sequence of resource acquisition and release operations that allows all the processes in the system to run to completion [3]. Detection & Recovery approaches afford maximum flexibility to the RAS, at the expense of continuous monitoring for deadlocks and reduced productivity when the system is rolled back during recovery. This paper will focus on the third class of methods, i.e. avoidance methods.

Avoidance methods address the problem of deadlocks by controlling how resources are granted to requesting processes. They use *on-line feedback* about the RAS state w.r.t. the allocation of the system resources. Given the current RAS state, a process request is granted only if the resulting state is safe. In principle, such a scheme leads to informed decisions about the safety of a resource allocation operation, allowing the RAS to achieve maximum flexibility. However, two important assumptions qualify this statement: First, it is assumed that the algorithm has *complete* knowledge of future behavior of processes w.r.t. resource allocation, i.e. the algorithm uses information about the current allocation of the system resources, as well as, the sequence of additional allocation (request / release) operations that every process, presently in the RAS, must undergo to reach completion. Secondly, it is assumed that the algorithm can resolve the safety of a considered state of resource allocation in *real-time*. The first assumption cannot be satisfied by all RAS environments, since detailed information about subsequent process behavior might not be available either because it is too costly to obtain, or because it is inherently nondeterministic. Furthermore, even if the aforementioned information were available, it is shown in [13, 2] that in the general case, the problem of determining the safety of a RAS state is NP-complete. An important implication is that some flexibility, and possibly performance, must be sacrificed when using the avoidance methodology for an arbitrary RAS.

Banker's Algorithm [14], for example, the most widely-used avoidance algorithm, stipulates knowledge of the maximum number of units of every resource type in the system that each process is ever going to use. When the safety of granting a process request is analyzed, the *worst-case* scenario, in which all processes simultaneously require their maximum allowable claims for resources to proceed forward, is assumed. Granting a process request is safe only when it is possible to meet that request and all previously described maximal process requests. The consideration of the maximal process requests, combined with the reusable nature of the system resources, makes the number of available units of every resource type increase monotonically between the service of these maximal requests. According to [22], this monotonicity makes backtracking unnecessary in the search for a safe allocation sequence and gives Banker's Algorithm a complexity of $O(rn \log n)$, where r is the number of resource types and n is the number of processes. [13] shows how the Banker's Algorithm can be generalized in order to deal with other special problem structures.

Banker's algorithm does not consider the particular order in which each process acquires and releases resources. The algorithm was primarily developed for use in computing environments [14] where it is impractical to expect an operating system to keep track of all possible threads for every single running process. However, there are modern technological applications in which the interacting processes are much better structured. A typical example is the *Automated Manufacturing Cell (AMC)*. An AMC consists of a number of workstations / machines, each one able to perform a set of elementary operations, and a number of transport units (Material Handling System, MHS) which carry the various workpieces among the workstations. The cell concurrently produces a number of parts, each of which is fabricated by a predetermined sequence of elementary operations (*process route*), and corresponds to a RAS process. Each of the workstations and the MHS can execute a number of elementary operations simultaneously. The maximum number of operations that can be simultaneously executed by a workstation / MHS defines its *capacity*. A complete operational model for the considered AMC is presented in Section 2.

Under this model, the AMC is susceptible to deadlocks where the finite *operational space* of the workstations and of the MHS are the resource(s) shared among the concurrently executed processes. Actually, the abstracted RAS model is a particular case of the *Job Step Model*, proposed in [22] for the study of the problem of deadlock avoidance in multiprocess systems of sequential processes. In the past, the manufacturing community has tended to ignore the deadlock problem since the presence of human operators allowed for an easy solution, namely the *preemption* of a number of deadlocked jobs to auxiliary buffers so that the remainder could proceed (*manual* detection & recovery). However, the emerging manufacturing paradigm places increased emphasis on flexibility, calling for rather arbitrary process routes through the system. At the same time, there is an increased focus on reduced production lead times and low work in process (WIP). Computerized shop floor control is the enabling factor for successfully meeting these requirements. [7, 6]. In this context, *it is crucial that computerized manufacturing system* *controllers implement automated solutions to the deadlock problem.* A second justification for a more systematic approach is that deadlocks are in fact constraints imposed by the plant structure on operational policies. Therefore, *to achieve integrated performance optimization, operational policies must consider the effect of AMC deadlock.* For these reasons, there is a resurgent interest in the problem of deadlocks occuring in job step models.

In one of the first papers to deal with deadlock in manufacturing systems, [3], the authors remark that avoidance seems to be the most appropriate method for handling deadlocks in an FMS environment. Furthermore, they claim that avoidance methods developed by the computing community are not efficient for FMS applications, since they ignore available information about the process structures and are thus "unduly conservative". A new deadlock avoidance policy, to be called the *(B-K) Deadlock Avoidance Algorithm (BKDAA)*, is proposed. BKDAA expands on the idea underlying Banker's algorithm by exploiting the fact that certain resources are used exclusively by a single processing step. These exclusive resources can function as buffers dedicated to the related processes, thus allowing for the decomposition of the deadlock resolution problem over segments of process routes, and the design of an efficient control algorithm. Using formal Petri Net (PN) modeling, the authors show that under BKDAA, the system will never enter a deadlock or a *restricted*[1] deadlock.

A different avoidance policy, called the Resource Upstream Neighborhood (RUN) policy, has been developed in the authors' research program and was first presented in [10]. This policy is based on the observation that if there are certain resources with higher capacity than others in the RAS, they can function as temporary buffers for the jobs that they are currently holding or for some jobs on route towards them. The idea is formalized by the introduction of the concept of the *Resource Upstream Neighborhood (RUN)*, which gives the policy its name. A formal definition of RUN is given in Section 5.1. Even though [10] provides considerable experimental evidence that under RUN the system will never enter a deadlock or a restricted deadlock, no formal proof is provided. A formal proof for the correctness of RUN policy is

[1]A *restricted* deadlock is defined as a RAS state in which progress is inhibited because of the constraints imposed by the avoidance policy itself. [3]

presented in Section 5.1 of this paper.

[23] uses PN modeling for the analysis of deadlocks in FMS's by exploiting the concept of *model invariants*. The proposed control policy performs a look-ahead search over the set of states reachable from the current state in a certain number of steps and selects as the next system step the one least likely to lead to deadlock. Under this policy, deadlocks are possible, even though the probability of their occurrence is reduced.

A different approach to the problem is taken in [25]. Here, the authors suggest that a deadlock detection algorithm, presented in [24], be applied to the state that results if a considered request is granted. The request can be granted only if the resulting state is not a deadlock state. This approach is, however, susceptible to restricted deadlocks. Furthermore, the volume of information processed by the deadlock detection algorithm in [24] grows exponentially with the size of the RAS configuration.

Finally, Petri Net modeling and synthesis has also been used for deadlock *prevention* during the system design phase. Under these methods, the AMC is modeled by a *composite* PN model, and its *liveness* [19] properties are analyzed. Characteristic examples of this approach are presented in [9, 12]. The development of a PN model for a complex system from the synthesis of a number of elementary modules (Elementary Control Tasks) is detailed in [9, 8]. Furthermore, the liveness properties of the composite model are studied. From the performed analysis, guidelines to be observed by the system designer are extracted. Similarly, in [12], an analytical characterization of the reachable states (*markings* [19]) of models composed from elementary *state machines* [19] is obtained. Then, the use of this characterization in facilitating the study of the *nonblocking* [21] (or, essentially deadlock-free) properties of the models is demonstrated. It should be mentioned that nonblocking behavior is one of the most fundamental requirements underlying a whole body of theory dealing with the analysis of the *logical* behavior of Discrete Event Systems (DES), known as *Supervisory Control* [21].

With the exception of [3] and [10], the approaches discussed above suffer from problems arising from the NP-complete nature of the state-safety decision problem. Therefore, with these solutions either (i) there is no guarantee that the system will never enter deadlock or restricted deadlock, or (ii) if such a guarantee is provided, it is obtained by (potentially) excessive computational cost (*required computation is exponential in the size of the RAS configuration*). In the first case, we say that the avoidance policy is *not provably correct* and in the second case that it is *not scalable*. It is our position that *deadlock avoidance policies for future technological systems must be, both, provably correct and scalable*. Note that correctness in the context of this discussion implies only that states characterized as safe are indeed safe. There might exist a subset of safe states which will not be recognized as such by a provably correct and scalable avoidance policy. Actually, this is the price one must pay (the compromise one must make) to obtain scalable policies because of the NP-complete nature of the underlying decision problem. Provably correct policies that successfully recognize all safe states of the RAS, are, by definition, *optimal*.

The policies presented in [3] and [10] are two examples of provably correct and scalable policies. Another provably correct and scalable policy is presented in [17]. The policy correctness is established on the condition that a resource request is granted only if the resulting state is *demonstrably* safe. To overcome the problem of the NP-completeness of the state-safety decision problem, and hence ensure the algorithm scalability, it is suggested that the dispatching scheduling policy applied to the RAS operation should guide the search for a safe allocation sequence. Specifically, the RAS operation is simulated by first ordering the executed jobs according to a given dispatching rule, and then trying to advance them in that order as much as possible. The starting RAS state is safe if the state (*marking*) resulting from this simulation covers the *minimal marking* required to keep the RAS live [19].

This paper suggests a framework for developing provably correct and scalable avoidance policies for the aforementioned AMC model. Section 2 provides a formal description of the AMC operation and establishes its correspondence to a Finite State Automaton (FSA) [16, 4]. In Section 3, the AMC model's deadlock characteristics are studied with the aid of the FSA State Transition Diagram (STD). Section 4 provides a new characterization of AMC state safety which serves as the basis of a new framework for systematic development of avoidance policies. Section 5 provides two examples of the framework functionality. First, the framework is used to formally prove the correctness of the Resource Upstream Neighborhood (RUN) avoidance policy [10]. Then, it is used to provide an alternative proof for the correctness of BKDAA [3]. Furthermore, an interesting relationship existing between RUN and BKDAA policies is brought out at the end of this section. Finally, Section 6 summarizes the paper results, draws conclusions, and discusses possible extensions and generalizations of the underlying ideas.

2 The AMC Operational Model

In this section, we describe the operation of the AMC at the level of detail required to extract the underlying RAS model. We also show that this model corresponds to a Finite State Automaton (FSA) [16, 4].

Resources, Jobs and Routes For the purposes of our study, an AMC is considered as a collection of workstations each capable of performing some fixed set of e(lementary)- operations. We shall call these workstations the cell *resources* and denote them as elements of the set $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$, where the cell has m resources. Furthermore, let $\mathcal{O}_i = \{o_{i1}, o_{i2}, \ldots, o_{i,k(i)}\}$ denote the set of e-operations supported by resource $R_i, i = 1, \ldots, m$, and $\mathcal{O} = \bigcup_{i=1}^{m} \mathcal{O}_i$.

It is possible that some workstations of the AMC are capable of executing a number of operations simultaneously. We capture this feature by defining the function $C(R_i) : \mathcal{R} \rightarrow N$ with $C(R_i) \equiv C_i$ being the *capacity* of resource R_i, i.e. the number of e-operations that it can run simultaneously. Let $R_{ik}, k = 1, \ldots, C_i$, define the k-th unit of capacity of resource R_i.

A *job type* supported by the AMC is a sequence of e-operations, i.e. $JT_i =< o_{i1}, o_{i2}, \ldots, o_{i,l(i)} >$. Let $\mathcal{J} = \{JT_1, JT_2, \ldots, JT_n\}$ denote the entire set of job types supported by the AMC. Given a certain job type JT_i, let $\mathcal{R}(i,j) = \{R_k : o_{ij} \in \mathcal{O}_k, k = 1, \ldots, m\}$ denote the set of resources capable of performing its j-th operation. Then $\mathcal{JR}_i \in \mathcal{R}(i,1) \times \mathcal{R}(i,2) \times \ldots \times \mathcal{R}(i,l(i))$ denotes a possible *route* in which an instance of this job type can be realized through the AMC, provided that it can be supported by the topology of the AMC transportation system. Let $\mathcal{JR}$ denote the set of possible routes a job instance can take through the AMC, and $r = |\mathcal{JR}|$. In this paper we consider only *static* routing schemes, i.e., the route of a job instance is determined before the job enters the AMC. Therefore, the elements of $\mathcal{JR}$ are resource sequences of the format

$$\mathcal{JR}_j =< JR_{j1}, JR_{j2}, \ldots, JR_{j,l(j)} >, \; j = 1, \ldots, r$$

and they characterize completely a process type of the underlying RAS.

The AMC State Given the definition of the job routes, let us define the set of route stages / steps *supported* by resource $R_k, \; k = 1, \ldots, m$, to be

$$\mathtt{Sup}(R_k) = \{(j, q), \; j = 1, \ldots, r, \; q = 1, \ldots, l(j) : R_k = JR_{jq}\}$$

i.e. the set of all route steps that have to be processed by resource R_k. Let $Q_k \equiv |\mathtt{Sup}(R_k)|$. We assume that the elements of $\mathtt{Sup}(R_k), k = 1, \ldots, m$, have been ordered in some arbitrary way so that they can be set in correspondence to the sequence $< 1, \ldots, Q_k >$. We shall denote the ith route step in that ordering by $\mathtt{Sup}(R_k)[i]$.

Definition 1 *The* state *of resource $R_k, k = 1, \ldots, m$, at time t is the vector $\mathbf{s}_{R_k}(t)$ with Q_k components. The value of the ith component, $\mathbf{s}_{R_k}(t, i)$, is equal to the number of units of resource R_k allocated to job instances executing the route step $\mathtt{Sup}(R_k)[i]$ at time t.*

It can be shown that the number of distinct possible states for resource R_k is $N(Q_k, C_k) = (Q_k + C_k)!/(C_k!Q_k!)$.

The AMC state is defined as follows:

Definition 2 *The* AMC state *at time t, $\mathbf{s}(t)$, consists of the concatenation of its resource states $\mathbf{s}_{R_k}(t), \; k = 1, \ldots, m$. By convention, the order of the concatenation is that defined by the resource numbering.*

From the above definition and the previous remarks on the number of the resource states, it follows that the number of the AMC states is $Z = \prod_{k=1}^{m} N(Q_k, C_k)$. Let $\mathcal{S}$ denote the entire set of the distinct AMC states. $\mathcal{S}$ will be called the AMC *state space*. Then we establish an one-to-one correspondence $\mathtt{Sgn}() : \mathcal{S} \to \{0, 1, \ldots, Z-1\}$ with $\mathtt{Sgn}(\mathbf{s}) = i$. We shall call $\mathtt{Sgn}(\mathbf{s})$ the state *signature* and denote a state $\mathbf{s}$ with $\mathtt{Sgn}(\mathbf{s}) = i$ as s^i. Finally, the AMC *trajectory* w.r.t. time is the function $\mathtt{tr}() : \Re \to \{0, 1, \ldots, Z-1\}$ with $\mathtt{tr}(t) = \mathtt{Sgn}(\mathbf{s}(t))$.

The AMC State Transitions The AMC state can change in one of the following three ways: **(i) Loading a new job:** This is possible if there exists an idle unit R_{rk} of resource R_r and a job-route $\mathcal{JR}_j$ with $JR_{j1} = R_r$. **(ii) Advancing a partially processed job:** Job j with job route $\mathcal{JR}_j$, currently located at unit R_{sk} of resource $R_s = JR_{jq},\ q \neq l(j)$, having completed its processing step on that resource, is transferred to a *currently idle* unit R_{rt} of resource $R_r = JR_{j,(q+1)}$. **(iii) Unloading a finished job:** A job j with route $\mathcal{JR}_j$, which is currently located at resource R_{sk} with $R_s = JR_{j,l(j)}$ having completed its processing (last) step on that resource, is removed from the AMC.

Furthermore, it is assumed that state transitions take place instantaneously and only if they have their preconditions satisfied. Executable transitions will be characterized as *feasible*. The AMC controller selects the transition to be executed next among the set of feasible transitions; detailed implementation of the selection scheme is an open research issue. Selection of a feasible transition by the controller and its execution by the AMC will be called an *event* in the AMC operation. Since we are mainly concerned with deadlock-free operation of the AMC, only the *event-sequencing* is relevant to the performed analysis while detailed event-timing is insignificant. In other words, it is assumed that the controller can defer making a decision until all transitions that are potentially feasible from the current state have been enabled. Therefore, we suppress explicit consideration of time from further analysis, which takes us to the realm of *Logical Analysis and Control of Discrete Event Systems* [4, 21].

The AMC Operational Model as an FSA In the following, we assume that the reader is familiar with the concept of the Finite State Automaton (FSA) and its related concepts [16, 4]. Then, it can be seen that the AMC operational model corresponds to an FSA with *event set* E consisting of all the possible route steps executed in the AMC [2], the *set of states* S corresponding to the set of AMC states, and the *state transition function* $f : S \times E \to S$

[2]Following standard notation in automata theory and formal languages, E^* denotes the *Kleene Closure* of event set E, i.e., the set of all finite sequences of events from set E, including the *empty* sequence ϵ.

defined by

$$f(s^i, e) = \begin{cases} \texttt{succ}(s^i, e), & \text{if } e \in \mathcal{F}(s^i) \\ s^i, & \text{o.w.} \end{cases}$$

where $\mathcal{F}(s^i)$ is the set of feasible events in state s^i and the function $\texttt{succ}(s^i, e)$ returns the state that results when event $e \in \mathcal{F}(s^i)$ takes place with the AMC being in state s^i. Finally, the *initial* and *final* states are identified by state s^0, the state in which the AMC is idle and empty of jobs. The *language accepted by the AMC* consists of those input (controller command) strings that, starting from the empty state, leave the AMC in the same idle condition. In a physical interpretation, these strings correspond to complete production runs.

An Example We conclude this section with an example which will highlight the previously introduced concepts. It will also facilitate the discussion on AMC deadlocks and avoidance policies that follows in the next section.

The AMC to be analyzed consists of 3 workstations, i.e. $\mathcal{R} = \{R_1, R_2, R_3\}$, with $C_i = 1,\ i = 1, 2, 3$. The jobs supported by the cell can take only one of the 2 following routes: $\mathcal{JR}_1 =< R_1,\ R_2,\ R_3 >$ and $\mathcal{JR}_2 =< R_3,\ R_2,\ R_1 >$. This dataset completes the AMC description. From this we obtain: $\texttt{Sup}(R_1) = \{(1,1), (2,3)\}$, $\texttt{Sup}(R_2) = \{(1,2), (2,2)\}$, $\texttt{Sup}(R_3) = \{(1,3), (2,1)\}$. Thus, the dimension of the AMC state $\mathbf{s}$ is equal to 6 (2×3). The size of the state space of every resource $R_i,\ i = 1, 2, 3$, is equal to $N(2, 1) = 3$, while the size of the AMC state space is $Z = 3 \times 3 \times 3 = 27$. The state signatures run from 0 to 26, with $s^0 =< 0\ 0\ 0\ 0\ 0\ 0 >$ being the initial empty state. Table 1 enumerates the AMC state transition function and Figure 1 provides the corresponding *State Transition Diagram (STD)*.

3 AMC Deadlock and Avoidance Policies

We next consider the deadlock properties of the AMC operational model and define deadlock avoidance policies. The STD of the previous example is used for an initial introduction to these concepts.

$i: \ s^i$	State Vector	Successor States
0	0 0 0 0 0 0	1, 18
1	0 0 0 0 0 1	3, 19
2	0 0 0 0 1 0	0, 20
3	0 0 0 1 0 0	4, 9, 21
4	0 0 0 1 0 1	10, 22
5	0 0 0 1 1 0	3, 11, 23
6	0 0 1 0 0 0	2, 7, 24
7	0 0 1 0 0 1	25
8	0 0 1 0 1 0	6, 26
9	0 1 0 0 0 0	0, 10
10	0 1 0 0 0 1	1, 12
11	0 1 0 0 1 0	2, 9
12	0 1 0 1 0 0	3, 13
13	0 1 0 1 0 1	4
14	0 1 0 1 1 0	5, 12
15	0 1 1 0 0 0	6, 11, 16
16	0 1 1 0 0 1	7
17	0 1 1 0 1 0	8, 15
18	1 0 0 0 0 0	6, 19
19	1 0 0 0 0 1	7, 21
20	1 0 0 0 1 0	8, 18
21	1 0 0 1 0 0	22
22	1 0 0 1 0 1	
23	1 0 0 1 1 0	21
24	1 0 1 0 0 0	20, 25
25	1 0 1 0 0 1	
26	1 0 1 0 1 0	24

Table 1: Example: The AMC State Transition Function

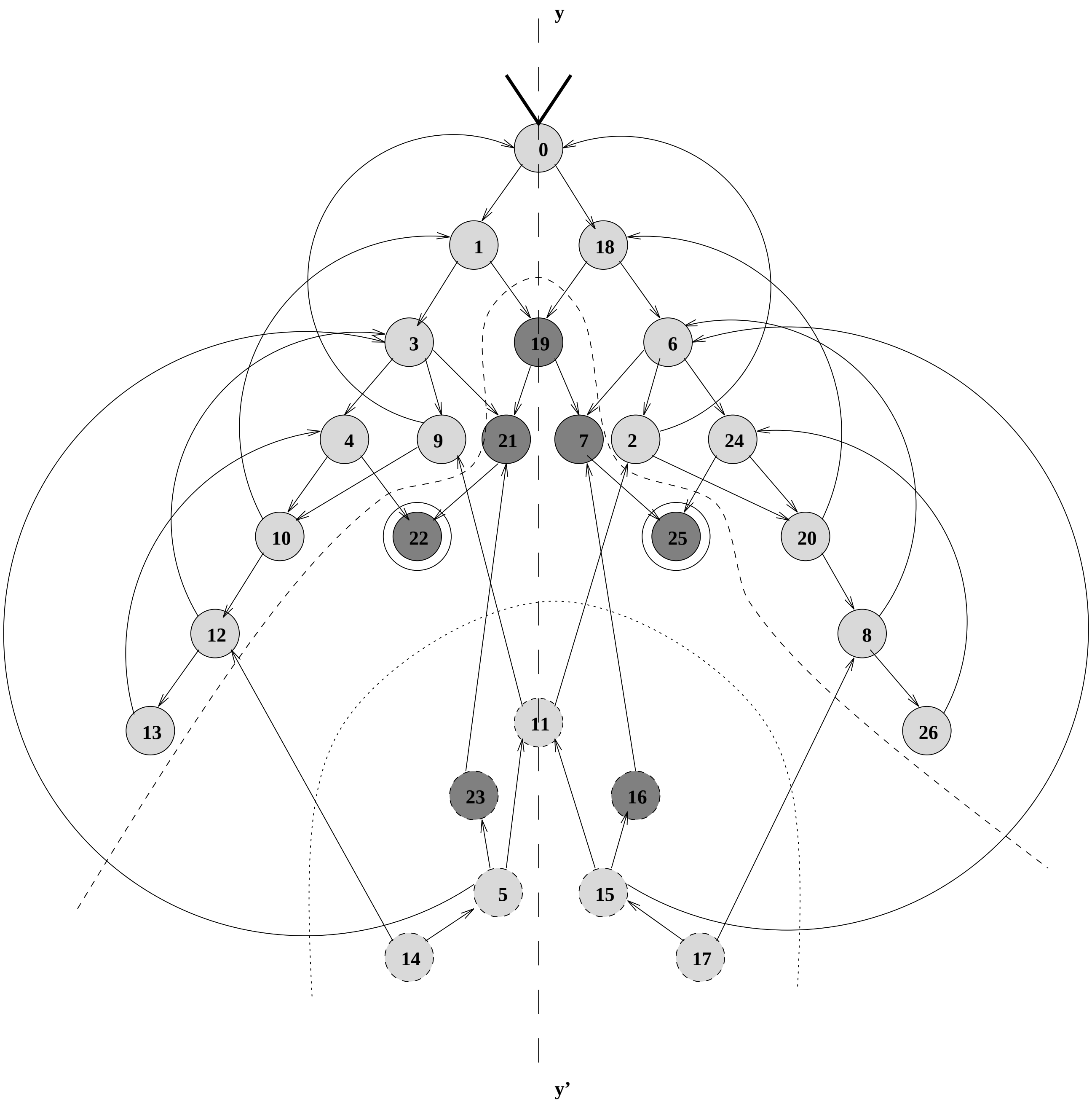

Figure 1: Example: The AMC State Transition Diagram

State Reachability and Safety A *directed path* in the STD of Figure 1 represents a feasible sequence of events in the AMC. We are mainly interested in paths that start and finish in the empty state s^0. Notice that there is a subset of nodes for which there is no directed path from state s^0; these are shown as dashed nodes in the STD. This implies that when the cell is started from empty state, (under normal [3] operation) the states (resource allocation) represented by the dashed nodes will never occur. These states will be referred to as *unreachable.* The remaining states are feasible states under normal operation and will be called *reachable* states. The set of reachable states will be denoted by $\mathcal{S}_r$ and the set of unreachable states will be denoted by $\mathcal{S}_{\bar{r}}$. Notice that $\mathcal{S}_{\bar{r}} = \mathcal{S} \backslash \mathcal{S}_r$. Formally,

Definition 3 *State s^i is* reachable from *state s^j, denoted $s^i \leftarrow s^j$, iff there exists a sequence of events that can bring the cell from state s^j to state s^i. In the FSA notation,*

$$\forall s^i, s^j \in \mathcal{S},\ s^i \leftarrow s^j \Longleftrightarrow \exists u \in E^* :\ f(s^j, u) = s^i$$

Furthermore, a state $s^i \in S$ is a reachable state, *iff $s^i \leftarrow s^0$.*

Another important classification of the STD nodes / states results from the following observation: there are states from which the empty state s^0 is reachable by following a directed path of the STD, and states for which this is not possible. In the STD of Figure 1, the former states are lightly shaded while the latter are heavily shaded. If the AMC enters any of the heavily shaded states it will never be able to complete all running jobs, i.e. become idle and empty. For this reason, the heavily shaded states are characterized as *unsafe*, while the lightly shaded states, which provide accessibility to state s^0, are characterized as *safe*. The set of safe states is denoted by $\mathcal{S}_s \subseteq \mathcal{S}$ and the set of unsafe states is denoted by $\mathcal{S}_{\bar{s}}$. Again, it holds $\mathcal{S}_{\bar{s}} = \mathcal{S} \backslash \mathcal{S}_s$. Formally,

Definition 4 *State s^i is a* safe *state iff state s^0 is reachable from state s^i. A state which is not safe will be called an* unsafe *state. In the FSA notation,*

$$\forall s^i \in \mathcal{S},\ \texttt{safe}(s^i) \Longleftrightarrow \exists u \in E^* :\ f(s^i, u) = s^0 \Longleftrightarrow s^0 \leftarrow s^i$$

[3] By normal it is meant that the AMC operation obeys the stipulations of Section 2.

Furthermore, we extend the characterization of safety to AMC transitions emanating from safe states, by characterizing them as safe if they result in a safe state; mathematically,

$$\forall s^i \in \mathcal{S}_s,\ \forall e \in \mathcal{F}(s^i),\ \mathtt{safe}(e/s^i) \Longleftrightarrow f(s^i, e) \in \mathcal{S}_s$$

Finally, we denote the intersection of any two classes resulting from the previous two classifications by $\mathcal{S}_{xy}$ where $x = r, \bar{r}$, and $y = s, \bar{s}$.

AMC Deadlock Deadlocks are the natural reason for the existence of the unsafe states in the AMC operation. A formal characterization of the AMC deadlock is given with the following definition, while the two subsequent propositions establish the previous remark.

Definition 5 *Let*

- $\mathtt{sl}(R_j)$ *denote a function that returns the number of idle units of resource* R_j *at the time that the function was invoked, i.e. the* slack *capacity of resource* R_j*,*
- $\mathtt{jr}(R_{jq})$ *denote a function that returns the job route of the job occupying the resource unit* R_{jq}*, and*
- $\mathtt{st}(R_{jq})$ *denote a function that returns the processing step executed by the job occupying the resource unit* R_{jq}*.* [4]

Then, state s^i *is a (partial)* deadlock *if there exists a subset of resource types* $\mathcal{DR}^i \subseteq \mathcal{R}$ *s.t.*

1. $\forall R_j \in \mathcal{DR}^i,\ \mathtt{sl}(R_j) = 0$
2. $\forall R_{jq}, q = 1, \ldots, C_j$ *with* $R_j \in \mathcal{DR}^i,\ (\mathtt{st}(R_{jq}) \neq l(\mathtt{jr}(R_{jq}))) \wedge (JR_{\mathtt{jr}(R_{jq}),\mathtt{st}(R_{jq})+1} \in \mathcal{DR}^i)$

In words, an AMC state is a deadlock if a subset of its resources is filled to capacity and every job holding a unit of those resources requires transfer to another resource of that group for its next processing step.

Proposition 1 *An AMC deadlock is an unsafe state.*

[4]Functions $\mathtt{jr}()$ and $\mathtt{st}()$ are undefined if R_{jq} is idle when the functions are invoked.

Proof: One of the requirements for a transition to be feasible is that the receiving resource has an idle unit. Furthermore, resource units become idle only when the occupying job proceeds to its next processing step or leaves the cell. Neither is possible for the resources in subset $\mathcal{DR}^i$. Thus, the jobs accommodated in $\mathcal{DR}^i$ cannot proceed to completion. $\diamond$

Proposition 2 *In the AMC-STD, every directed path that starts from an unsafe state results in a deadlock.*

Proof: Suppose not. Then there exists a path of events in which the successive states either have at least one idle unit for every resource type, or if $\mathcal{R}' \subseteq \mathcal{R}$ consists of resources filled to capacity, it is possible to identify a subset $\mathcal{R}'' \subseteq \mathcal{R}'$ with at least one job ready to leave the cell or requiring a processing unit in $\mathcal{R} \backslash \mathcal{R}'$. This implies that at every successive state, there exists at least one job of those currently in the cell able to proceed. Given the finiteness of the job routes and assuming that no new jobs are loaded in the cell, the cell is going to be empty after a sufficient number of events. But this contradicts the definition of unsafeness! $\diamond$

It should be noticed that the property described in Proposition 2 holds only in the context of the considered RAS model; actually it results from the fact that process behaviors are completely deterministic in this context. In more general RAS models, like a RAS modeling a computer system environment, unsafe states do not necessarily imply deadlocks. Unsafe states in those systems are states which possess a nonzero probability of leading to deadlock. This broader definition of unsafeness reflects the fact that the avoidance methodology is in general based on a worst-case analysis of the resource-allocation situation.

Deadlock Avoidance Policies In the following, we assume that the AMC always undergoes normal operation so that only reachable states occur. In the FSA terminology, the subgraph consisting of the reachable states $\mathcal{S}_r$ and the arcs emanating from them is called the *reachability graph* of the FSA. The problem undertaken in this work is the development of control strategies that will restrict the operation of a given AMC by limiting it to its reachable and safe subspace $\mathcal{S}_{rs}$. Practically, we seek to identify an appropriate set of feasible transitions which when removed from the STD (or equivalently, *disabled* by the AMC controller), render the unsafe subspace $\mathcal{S}_{r\bar{s}}$ unreachable from state s^0. At the same time, we must ensure that every state s^i in the remaining graph (i.e., *reachable under the control policy*), is still safe (i.e., there exists a directed path *in the remaining graph* leading from state s^i to s^0). States which are reachable under a deadlock avoidance policy and from which progress is inhibited by the policy-imposed constraints and not by the RAS structure, are characterized as *restricted* deadlocks in the deadlock literature [3].

Formally, an avoidance policy for the AMC is defined as follows:

Definition 6 *An* avoidance policy $\mathcal{P}$ *for the AMC is a function*

$$\mathcal{P}:\ \mathcal{S} \to 2^E,\ \ \mathcal{P}(s^i) = \{e \in \mathcal{F}(s^i):\ e \textit{ is selected by the policy}\}$$

Events $e \in \bigcup_i \mathcal{P}(s^i)$ *are called the* (policy-)enabled *events.*

An avoidance policy $\mathcal{P}$ is *correct*, if it is *deadlock and restricted deadlock-free*. From the discussion above, this requirement is met by ensuring that every state s^i which is reachable when the AMC is operated under the policy, is safe (considering operation under the policy). Formally,

Definition 7 *Given an avoidance policy* $\mathcal{P}$*, let* $s^i \overset{\mathcal{P}}{\leftarrow} s^j$ *denote the fact that state* s^i *is reachable from state* s^j *through an event sequence which comprises policy-enabled events only. Let* $S_r(\mathcal{P}) = \{s^i:\ s^i \overset{\mathcal{P}}{\leftarrow} s^0\}$ *and* $S_s(\mathcal{P}) = \{s^i:\ s^0 \overset{\mathcal{P}}{\leftarrow} s^i\}$*. Then, policy* $\mathcal{P}$ *is* correct *iff* $S_r(\mathcal{P}) \subseteq S_s(\mathcal{P})$*.*

Finally, a correct avoidance policy $\mathcal{P}^*$ is *optimal* if it disables the *minimum* set of events that is required in order to ensure correctness. This is equivalent to the requirement that the policy restriction on the S_{rs} subspace of the AMC disables *only* those actions that result in unsafe states. Formally,

Definition 8 *The correct avoidance policy* $\mathcal{P}^*$ *is* optimal *iff*

$$\forall s^i \in \mathcal{S}_{rs},\ \forall e \in \mathcal{F}(s^i),\ e \in \mathcal{P}^*(s^i) \Longleftrightarrow f(s^i, e) \in \mathcal{S}_s$$

Two implications of this characterization of the optimal policy are that, (i) for a given AMC configuration, the optimal avoidance policy $\mathcal{P}^*$ is *unique*, and (ii) $\mathcal{S}_r(\mathcal{P}^*) = \mathcal{S}_{rs}$. Establishing the optimal control policy $\mathcal{P}^*$ is equivalent to removing from the reachability graph those transition arcs that belong to the *cut*[5] $[\mathcal{S}_{rs}, \mathcal{S}_{r\bar{s}}]$. For example, in the STD of Figure 1, the optimal control policy $\mathcal{P}^*$ consists of removing the arcs that emanate from lightly shaded solid nodes and cross the twisted dashed line. Finally, since the inclusion of a transition to the optimal avoidance policy $\mathcal{P}^*$ depends on the safety of the successor state, it follows that obtaining policy $\mathcal{P}^*$ is an NP-hard problem [11].

Typically, a deadlock avoidance policy is expressed by a set of logical and/or algebraic constraints imposed on the AMC operation. Under such a formalism, the scalability requirement for the policy implies that *satisfaction of the policy constraints by an AMC transition can be verified in effort polynomial in the number of AMC workstations and job routes*. The correctness requirement implies the provision of a formal proof establishing that *for every state reachable by the AMC when it is operated under the policy, there exists a sequence of events that can take it to the empty state without violating, both, the AMC event-feasibility constraints (cf. section on AMC transitions) and the constraints imposed by the policy (cf. the previous remark on restricted deadlocks).*

4 A New Characterization of the State-Safety Problem and its Implications in the Design of Avoidance Policies

This section presents another characterization of the state safety decision problem. The significance of this characterization is that it serves as a framework for the analysis of the correctness of a certain class of deadlock avoidance policies. This class of policies can be formulated as *a set of inequalities which constrain the allocation of system resources at any single instance*. These constraints are *"static"*, i.e. they consider only the present status of resource allocation and not its history. AMC states corresponding to resource allocation that satisfies the policy constraints will be characterized as *admissible* under the policy. An AMC event is enabled under such a policy, iff the resulting AMC state is admissible. We shall refer to this class of avoidance

[5]For a definition of this concept refer to [1].

policies as the class Ω. Given a policy $\mathcal{P} \in \Omega$, let $\mathcal{S}_a(\mathcal{P})$ denote the set of AMC states which are admissible under policy $\mathcal{P}$. It should be obvious that for a given policy $\mathcal{P} \in \Omega$, $\mathcal{S}_r(\mathcal{P}) \subseteq \mathcal{S}_a(\mathcal{P})$. The second part of this section discusses how the specification of policies in class Ω, coupled with the proposed safety characterization, can facilitate the analysis of correctness of any policy from this class. Finally, notice that the avoidance policy presented in [17] is an example of a correct and scalable policy not belonging in class Ω since state admissibility is determined on the basis of the *explicit construction of a safe resource allocation sequence*.

The proposed state-safety characterization Let us consider the AMC state $s^i = \mathbf{s}_0$ in which J_0 jobs, $\{j_1, j_2, \ldots, j_{J_0}\}$, are currently loaded in the cell. Obviously, $J_0 \leq \sum_{k=1}^{m} C_k$, the total capacity of the AMC. For uniformity of presentation, we include an extra resource R_{m+1} with $C_{m+1} = \infty$. This resource accommodates all jobs exiting the cell. Hence, all job routes are augmented by one step: $JR_{j,l'(j)} = R_{m+1},\ j = 1, \ldots, r$, where $l'(j) = l(j) + 1$. Let $U_{it}, i = 1, \ldots, J_0$, denote the resource unit occupied by job j_i at step t. Then, assuming that state $\mathbf{s}_0$ is safe, the total number of steps required for running all the currently loaded jobs to completion, is $T_t = \sum_{i=1}^{J_0}[l'(\mathtt{jr}(U_{i0})) - \mathtt{st}(U_{i0})]$, where $\mathtt{jr}(U_{i0})$ and $\mathtt{st}(U_{i0})$ are the functions described in Definition 5. Finally, let $\{\delta_{ikt}\}$ denote a set of *binary* variables with

$$\delta_{ikt} = \begin{cases} 1 & \text{if job } j_i \text{ occupies a unit of resource } k \text{ at step } t \\ 0 & \text{o.w.} \end{cases}$$

where $i \in \{1, \ldots, J_0\}$, $k \in \{1, \ldots, m+1\}$ and $t \in \{0, \ldots, T_t\}$.

State $\mathbf{s}_0$ is safe iff the following system in variables δ_{ikt} is feasible:

$$\delta_{ik0} = \begin{cases} 1 & \text{if } \exists q \in \{1, \ldots, C_k\} :\ U_{i0} = R_{kq} \\ 0 & \text{o.w.} \end{cases} \qquad \forall i, k \tag{1}$$

$$\sum_{i=1}^{J_0} \delta_{ikt} \leq C_k, \quad \forall k, t \tag{2}$$

$$\sum_{k=1}^{m+1} \delta_{ikt} = 1, \quad \forall i, t \tag{3}$$

$$\sum_{t=0}^{T_t} \delta_{i,l'(i),t} \geq 1, \quad \forall i \tag{4}$$

$$\delta_{ikt} - (\delta_{ik(t+1)} + \delta_{i,\mathtt{sr}(i,k,p_i),(t+1)}) \leq 0, \quad \forall i, k(\neq m+1), t \tag{5}$$

$$\delta_{ikt} + \delta_{i,\mathtt{sr}(i,k,p_i),(t+1)} + \sum_{j=1}^{J_0} \delta_{j,\mathtt{sr}(i,k,p_i),t} \leq C_{\mathtt{sr}(i,k,p_i)} + 1, \quad \forall i, k(\neq m+1), t \tag{6}$$

$$\delta_{ikt} \in \{0, 1\}, \quad \forall i, k, t \tag{7}$$

In the above equations, $\mathtt{sr}(i, k, p_i)$ is a function returning the resource type required by job j_i for its next processing step, given that it is currently allocated one unit of resource type k for the execution of its p_i-th processing step ($\mathtt{s}$-uccessor $\mathtt{r}$-esource). Equation 1 introduces the description of the initial state $\mathbf{s}_0$ into the program and it represents the set of initial conditions. Equation 2 states that no resource type (cell workstation) can hold more jobs than its capacity. Equation 3 expresses the fact that every job instance always requires one complete unit of operational space. Equation 4 simply states that for a cell state to be safe, it is necessary that every job executes its last step at some point over the considered horizon. Given the convention introduced above, this guarantees that the job has been run to completion and left the cell. Equation 5 represents the precedence constraints introduced by the job routes, i.e. every movement of a job in the cell must obey its routing sequence. Equation 6 imposes the requirement that a job can proceed to its next step only if there is at least one unit of available capacity from the resource type required during that step. Finally, Equation 7 states the binary nature of the δ variables.

The state-safety characterization as a correctness-analysis tool Since the solution of the system of Equations 1 - 7 is an NP-complete problem [16, 11], this formulation is of no aid in the design of an algorithmic solution to obtaining the optimal avoidance policy. However, it can provide a *framework* for systematic development of *suboptimal* avoidance policies which belong to class Ω, and are provably correct. The underlying idea is to incorporate in the above set of equations the extra set of "static" constraints on resource allocation that express the considered avoidance policy. For the expanded set of constraints the following proposition holds: [6]

Proposition 3 *Let $\mathcal{P} \in \Omega$. If the existence of a set of variables $\{\delta_{ik\tau}\}$ satisfying Equations 2, 3, 7, and the policy constraints for step $t = \tau$ implies the existence of a set of variables $\{\delta_{ik(\tau+1)}\}$ satisfying the same set of equations for step $t = \tau + 1$, and the combined set of variables $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ satisfies Equations 5 and 6, policy $\mathcal{P}$ is* correct.

Proof: The correctness of the policy is proved by showing that under the condition in the statement of the proposition, there exists a set of variables $\{\delta_{ikt}\}$ satisfying Equations 2 - 7 and the policy constraints for all i, k and $t \in \{\tau, \ldots, T_t\}$. Then, for any state $s^i \in \mathcal{S}_r(\mathcal{P})$, there exists a sequence of policy-enabled events that can lead the AMC to the empty state s^0, and thus, $s^i \in \mathcal{S}_s(\mathcal{P})$.

Without loss of generality, let us assume that $T_t = \tau + n$, i.e. n more steps are required for processing the workload implied by the pricing of variables $\delta_{ik\tau}$. From the description of the AMC operation, it can be seen that $n \in N$ and it is finite for a given set $\{\delta_{ik\tau}\}$. Recursive application of the condition in the statement of Proposition 3 implies the existence of a set sequence $< \{\delta_{ik(\tau+1)}\}, \ldots, \{\delta_{ik(\tau+n=T_t)}\} >$ such that each set $\{\delta_{ikt}\},\ t \in \{\tau+1, \ldots, \tau+n\}$, satisfies Equations 2, 3, 7 and the policy constraints, while the combined set of variables $\{\delta_{ik(t-1)}, \delta_{ikt}\}$ satisfies Equations 5 and 6. Finally, the fact that $\{\delta_{ikt}\},\ t \in \{\tau+1, \ldots, \tau+n\}$ satisfy Equation 5 together with the assumption that $T_t = \tau + n$, imply that Equation 4 is satisfied by $\cup_{t=\tau}^{\tau+n}\{\delta_{ikt}\}$ (remember that one step corresponds to the advancement of a single job to its next processing step). $\diamond$

Notice that analysis of a policy $\mathcal{P} \in \Omega$ in the context of the framework established by Proposition 3 resolves simultaneously the problem of policy correctness with respect to, both,

[6] For avoidance policies in class Ω, a stronger definition of correctness can be given, requiring that *a policy $\mathcal{P} \in \Omega$ is correct iff $\mathcal{S}_a(\mathcal{P}) \subseteq \mathcal{S}_s(\mathcal{P})$*. It can be easily seen that the proof given in the text establishes the validity of Proposition 3 even in the case that the above definition of correctness is used. Furthermore, under this modified definition of correctness, the reverse of Proposition 3 is also true, i.e., for every *correct* policy $\mathcal{P} \in \Omega$ and every AMC state $s^i \in \mathcal{S}_a(\mathcal{P})$, there exists *at least one* policy-enabled event concerning the advancement of an already loaded job to its next processing stage. (Otherwise, $\mathcal{S}_a(\mathcal{P}) \not\subseteq \mathcal{S}_s(\mathcal{P})$, which implies that $\mathcal{P}$ is not correct.)

deadlocks and restricted deadlocks. Two examples of applying the proposed system of equations as a correctness-analysis tool are given in the next section.

5 Examples of the Framework Functionality

This section uses the system of Equations 1 - 7 and proposes to formally prove the correctness of the avoidance policies presented in [10] and [3]. Both of these policies are shown to belong to class Ω, by restating them as a set of "static" constraints to be added to the system of Equations 1 - 7. Proofs are then developed as discussed in the previous section. In fact, comparison of these proofs demonstrates the ability of the proposed framework to systematize the analysis of correctness for the considered class of avoidance policies. Finally, at the last part of the section, it is shown that the two avoidance policies considered generally admit different subspaces of the system safe space, and it is discussed how this property can be used in order to obtain better scalable approximations of the optimal policy $\mathcal{P}^*$.

5.1 The Resource Upstream Neighborhood (RUN) Avoidance Policy

The *Resource Upstream Neighborhood (RUN)* avoidance policy is based on the idea that if there are some resources with higher capacity than others in the system, then they can function as temporary buffers for the jobs that they support. For the formal statement of the RUN avoidance policy, we introduce the concept of the *resource upstream neighborhood*:

Definition 9 *The* upstream neighborhood *of resource R_k is defined as*

$$\begin{aligned}\mathtt{UN}(R_k) = \{R_q,\ q=1,\ldots,m : \exists (j,t) \in \mathtt{Sup}(R_k),\\ (R_k = \mathtt{sr}(JR_j, \mathtt{sr}(JR_j, \mathtt{sr}(JR_j, \ldots \mathtt{sr}(JR_j, R_q, t-n), \ldots), t-2), t-1) \equiv \mathtt{sr}^{(n)}(JR_j, R_q, t-n))\\ \wedge (C_k \geq C_{\mathtt{sr}^{(l)}(JR_j, R_q, t-n)},\ \forall l \in \{0, \ldots, n-1\})\}\end{aligned}$$

Furthermore, we say that a job instance j_i is in the upstream neighborhood of resource R_k, and we write $j_i \propto \mathtt{UN}(R_k)$, iff j_i is allocated one unit of resource $R_q \in \mathtt{UN}(R_k)$ and its subsequent route satisfies the conditions w.r.t. resources R_k and R_q, stated in the equation above.

In words, the upstream neighborhood of resource R_k is the union of *resource sequences* that precede resource R_k in the route steps that it supports, and they have the property that every resource R_q in them has capacity $C_q \leq C_k$.

RUN Given this definition, RUN requires that

$$\sum_{l \in \mathtt{UN}(R_k)} \sum_{i: j_i \propto \mathtt{UN}(R_k)} \delta_{ilt} \leq C_k, \quad \forall k, t \tag{8}$$

In words, RUN requires that *the number of jobs in the upstream neighborhood of a resource never exceeds the capacity of that resource.* The following theorems establish its correctness.

Theorem 1 *If there exists a set of variables $\{\delta_{ik\tau}\}$ satisfying Equations 2, 3, and 7, together with Equation 8, for step $t = \tau$, then there exists a set of variables $\{\delta_{ik(\tau+1)}\}$ satisfying the same set of equations for step $t = \tau + 1$. Furthermore, the combined set of variables $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ satisfies Equations 5 and 6.*

Proof: In order to prove this theorem, we show that if there exists a set of variables $\{\delta_{ik\tau}\}$ satisfying Equations 2, 3, 7 and 8, then at least one job can proceed in the cell, according to the job advancement scheme defined in Section 2. In this way, Equations 3, 7 are satisfied automatically (by construction) by the variable set $\{\delta_{ik(\tau+1)}\}$, while the combined set $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ satisfies Equation 5. It remains to be shown that Equations 2 and 8 will be satisfied by $\{\delta_{ik(\tau+1)}\}$, and Equation 6 is satisfied by the combined set $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$.

Case 1: $\exists j_i, \ C_{\mathtt{sr}(i,k,p_i)} \geq C_k$

In words, there exists a job j_i in the AMC waiting for a resource R_q with C_q greater than or equal to the capacity C_k of its current resource R_k. In this case, $\{\delta_{ik(\tau+1)}\}$ can be obtained from $\{\delta_{ik\tau}\}$ by letting job j_i proceed to its successor resource R_q. This step is legitimate since

1. *Equation 8 is satisfied by $\{\delta_{ik(\tau+1)}\}$.* This is true since only resources supporting subsequent steps of job j_i can have their corresponding Equation 8 affected by this movement. For these resources, however, $C_k \leq C_q$ implies that if R_q is in the upstream neighborhood of such a resource so is R_k. As a result, the net sum of jobs in the upstream neighborhood remains unaffected for these resources.

2. *Satisfaction of Equation 8 by $\{\delta_{ik(\tau+1)}\}$ implies the satisfaction of Equation 2 by these variables.* To see this, note that every resource is contained in its own upstream neighborhood.

3. Finally, *satisfaction of Equation 2 for R_q by $\{\delta_{ik(\tau+1)}\}$ implies the satisfaction of Equation 6.*

Case 2: $\forall j_i,\ C_{\mathtt{sr}(i,k,p_i)} < C_k$

In this case, all jobs are waiting for resources with smaller capacity than that of the resource they are currently occupying. Let R_{min} be one of the resources holding jobs at step $t = \tau$, s.t. $C_{min} = \min\{C_k : R_k \text{ contains some jobs}\}$; this set is finite since job instances and resource types are finite. Consider a job j_i in R_{min}. This job waits for resource R_q which is idle, since by (the case) assumption $C_{\mathtt{sr}(i,R_{min},p_i)} < C_{min}$, and R_{min} is the resource of smallest capacity containing jobs. Then it is obvious that proceeding job j_i to resource R_q satisfies Equations 2 and 6; it remains to be shown that taking this step satisfies Equation 8.

Suppose that by taking this step, Equation 8 is violated for $\{\delta_{ik(\tau+1)}\}$. This implies that resource R_q belongs in the upstream neighborhood of a resource R_b which is filled to capacity at step $t = \tau$, and job j_i enters this neighborhood for the first step at step $t = \tau + 1$. By the definition of R_{min}, it follows that $C_{min} \leq C_b$. Furthermore, by the definition of the upstream neighborhoods, it follows that all resources R_l in the segment of route JR_i contained between R_{min} and R_b have capacity $C_l \leq C_b$ (since R_q, which succeeds R_{min} in that chain, belongs to the upstream neighborhood of R_b). This implies that R_{min} should be in the upstream neighborhood of R_b w.r.t. job j_i, and thus, job j_i does not enter the upstream neighborhood of resource R_b for the first time at $t = \tau + 1$. $\diamond$

Theorem 2 *RUN is* correct.

Proof: Just notice that Theorem 1 establishes that the assumption stated in Proposition 3 holds in the case of the RUN avoidance policy. $\diamond$

Since this paper focuses on the functionality of Equations 1 - 7 as an analytical tool for analysis and design of correct deadlock avoidance policies, we shall not expand on other properties of RUN. The policy is easily shown to be scalable [18]. Furthermore, implementational considerations and the policy effect on the AMC performance are under investigation. Some preliminary results are reported in [10].

5.2 The B-K Deadlock Avoidance Algorithm (BKDAA)

In the introductory section, we saw that the B-K Deadlock Avoidance Algorithm (BKDAA), presented in [3], was the first scalable and provably correct policy for manufacturing systems. Here, we restate the policy in the context of the proposed AMC operational model, and then we provide an alternative proof of its correctness, which makes use of the framework discussed in Section 4. Note that this proof is of significantly reduced length and complexity when compared to the original proof provided in [3]. Furthermore, it should be noticed that the structure of this proof is identical to the previous proof that establishes the correctness of the RUN policy. This demonstrates that the proposed framework is indeed a methodological approach for dealing with this type of problems.

Shared and Unshared Resources and Production Zones For the discussion of the BKDAA policy, the following concepts are required [3]: First, we define an *unshared resource* as a resource which supports only a single processing step, i.e. resource $R_k,\ k = 1,\dots,m$, is unshared iff $|\texttt{Sup}(R_k)| = 1$. Let us denote the set of unshared resources by $\mathcal{R}^u (\subseteq \mathcal{R})$. Then, $\mathcal{R}^s = \mathcal{R} \backslash \mathcal{R}^u$ defines the set of *shared resources* [7]. Given a process route $JR_j,\ j = 1,\dots r$, consider the *maximal* route segments consisting of successive unshared resources, in the sense that the immediately preceding and succeeding resources in the route are either shared or do not exist. We refer to these segments as the *unshared subzones* of job route JR_j. The interleaving segments of shared resources are called the *shared subzones* of job route JR_j. A pair of consecutive shared and unshared subzones in a job route constitute a *production zone*. In this way, a job route is naturally decomposed to a sequence of production zones, i.e. $JR_j =< z_j^1, z_j^2, \dots, z_j^{l(j)} >,\ j = 1,\dots,r$. It should be noticed that production zone z_j^1 ($z_j^{l(j)}$)

[7] Notice that a resource is shared even if it is only used by two or more processing steps of the same job route; this feature is necessary if BKDAA is to be correct for AMC's with *cyclic* routes.

might have its shared (unshared) subzone, s_j^1 ($u_j^{l(j)}$), missing. Finally, given resource $R_k \in \mathcal{R}^s$ and processing step (j,q), $j = 1,\ldots,r$, $q = 1,\ldots,l(j)$, supported by R_k, let $\mathcal{T}(R_k,(j,q))$ denote the subsequence of resources which supports the steps in the shared subzone of step (j,q) that follow step (j,q).

BKDAA Given the previous definitions, BKDAA is expressed by the following two constraints, written in the notation of Section 4:

$$\sum_{i=1}^{J_0} \sum_{k:R_k \in z_j^n} \delta_{ikt} \le \begin{cases} \sum_{k:R_k \in u_j^n} C_k & \text{if } u_j^n \text{ exists} \\ \infty & \text{o.w.} \end{cases} \qquad \forall t,j,n \tag{9}$$

$$\delta_{i,\mathtt{sr}(i,k,p_i),(t+1)} + \delta_{ikt} \le \min_{R_n \in \mathcal{T}(\mathtt{sr}(i,k,p_i),p_i+1);\ n \neq k} \{C_n - \sum_{j=1}^{J_0} \delta_{jnt}\} + 1 \quad \forall i,k,t:\ \mathtt{sr}(i,k,p_i) \in \mathcal{R}^s \tag{10}$$

Quoting [3], "rule DAA1 (i.e. constraint 9) allows a token (job) to enter a new zone in the production sequence only when the capacity in the unshared subzone of the zone exceeds the number of tokens (jobs) currently in the zone. Rule DAA2 (i.e. constraint 10) assures that if a shared resource is being requested by the job, all of the shared resources in the remainder of the zone are available at that time".

The correctness of the policy is established by the following theorems:

Theorem 3 *If there exists a set of variables $\{\delta_{ik\tau}\}$ satisfying Equations 2, 3, and 7, together with Equations 9 for step $t = \tau$, then there exists a set of variables $\{\delta_{ik(\tau+1)}\}$ satisfying the same set of equations for step $t = \tau + 1$. Furthermore, the combined set of variables $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ satisfies Equations 5, 6 and 10.*

Proof: As in the proof of Theorem 1, this theorem is proved by showing that if there exists a set of variables $\{\delta_{ik\tau}\}$ satisfying Equations 2, 3, 7 and 9, at least one job can proceed in the cell, according to the job advancement scheme defined in Section 2. In this way, Equations 3, 7 are satisfied automatically (by construction) by the variable set $\{\delta_{ik(\tau+1)}\}$, while the combined set $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ satisfies Equation 5. It remains to be shown that Equations 2 and 9 will be satisfied by $\{\delta_{ik(\tau+1)}\}$, and Equations 6 and 10 are satisfied by the combined set $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$.

First, notice that if job j_i, with job route JR_q, is currently allocated a unit of resource R_k in the unshared subzone u_q^n and it requires advancement to resource $R_l \in u_q^n$, then this job can proceed if there is available capacity on R_l. This is the case since the value of the summation on the left side of Equation 9 is not altered by this movement, while Equation 10 constrains only motion in shared subzones. For this reason, in the following we assume that all jobs presently located in unshared subzones of their routes have been advanced as far as possible, so that any free capacity in the unshared subzone is available on its initial resource(s).

We consider two basic cases:

<u>Case 1: $\exists j_i,\ U_{i\tau} \in \mathcal{R}^s$</u>

In words, there exist jobs allocated to shared resources. Consider job j_i^* that was advanced last to a resource $R_k \in \mathcal{R}^s$. Let the production zone currently containing job j_i^* be z_i^*.

Then, if the next resource required by j_i^* is $R_l \in u_i^*$, Equation 9, combined with the previous assumption about the distribution of free capacity in unshared zones, guarantees that the job is able to proceed. Specifically, Equation 9 guarantees that there is free capacity in the unshared subzone u_i^*, and due to the previously made assumption, (at least part of) this free capacity is located on resource R_l. As a result, Equation 2 will be satisfied by $\{\delta_{ik(\tau+1)}\}$, and Equation 6 will be satisfied by $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$. Furthermore, since proceeding job j_i^* does not change its production zone, Equation 9 is still satisfied by $\{\delta_{ik(\tau+1)}\}$. Finally, since $R_l \in \mathcal{R}^u$, constraint 10 is ineffective (i.e. automatically satisfied by $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$).

On the other hand, if $R_l \in s_i^*$, then DAA2 guarantees that there exists at least one unit of free capacity on it (remember, j_i^* is defined to be the job last advanced to a resource in $\mathcal{R}^s$). This implies that by advancing j_i^*, the resulting set of variables $\{\delta_{ik(\tau+1)}\}$ satisfies Equation 2, while the combined set $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ satisfies Equation 6. Equation 9 is satisfied as above. Equation 10 is satisfied by $\{\delta_{ik\tau}, \delta_{ik(\tau+1)}\}$ because job j_i^* was the last job to be advanced to a resource in $\mathcal{R}^s$. Specifically, for every resource $R_q \in \mathcal{R}^s$ with $R_q \neq R_k$, $\sum_i \delta_{iqt}$ has remained the same or decreased since the last movement of job j_i^*, when Equation 10 was holding for

$\mathcal{T}(R_k, (\mathtt{rt}(U^*_{i,\tau}), \mathtt{st}(U^*_{i,\tau})))$.

Case 2: $\forall j_i,\ U_{i\tau} \in \mathcal{R}^u$

In words, all jobs are currently allocated to unshared resources. As per the initial remark about advancement in unshared subzones, let us assume that all jobs in the system have advanced as far as possible in their unshared subzones, so that only advancement into shared resources is possible. Since all resources in $\mathcal{R}^s$ are empty, it immediately follows that such an advancement cannot violate Equations 2, 6 and 10. It remains to be shown that there exists a job, the advancement of which does not violate constraint 9.

To see that this is the case, consider the following: Given a job route $JR_j,\ j = 1, \ldots, r$, its production zones $< z_j^1, \ldots, z_j^{l(j)} >$ are determined by its unshared subzones $u_j^q,\ q = 1, \ldots, l(j)$. By definition, resources in $\bigcup_q u_j^q$ are used exclusively by jobs with job route JR_j. Since we assume that $\mathcal{R}^s$ is empty of jobs, switching of production zones by a job j_i with job route JR_j is constrained due to Equation 9, only by the distribution in the AMC of jobs executing the same route JR_j. Furthermore, for jobs in the last unshared subzone $u_j^{l(j)}$ of JR_j, Equation 9 is effectively inactive. Thus, if there exist any jobs in $u_j^{l(j)}$, due to our initial assumption, there must be at least one job allocated one unit of the last resource in $u_j^{l(j)}$, and it can proceed to its last shared subzone – possibly consisting of the dummy shared resource, only. Otherwise, we can identify the unshared subzone u_j^n for which subzones $u_j^q,\ q = n+1, \ldots, l(j)$, are empty. Then, jobs allocated one unit of the last resource in u_j^n can proceed without violating constraint 9. $\diamond$

The next theorem is analogous to Theorem 2, and it can be proved in exactly the same way, by invoking Theorem 3.

Theorem 4 *BKDAA is* correct.

5.3 The "Essential Difference" of RUN and BKDAA

By focusing on different aspects of the AMC behavior in order to develop the defining control rule, RUN and BKDAA generally recognize the safety of different subsets of the safe subspace $\mathcal{S}_s$. We formalize this observation in the following proposition.

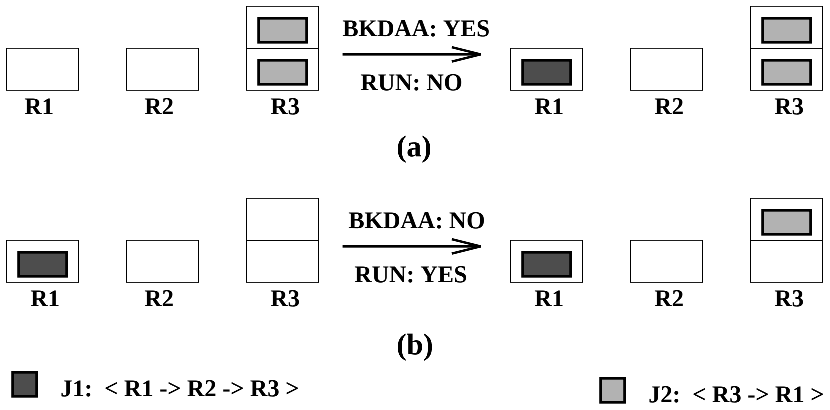

Figure 2: The essential difference of RUN and BKDAA

Proposition 4 *RUN and BKDAA are* essentially different*, i.e., there exist AMC configurations, for which*

$$\mathcal{S}_a(RUN) \oplus \mathcal{S}_a(BKDAA) \equiv (\mathcal{S}_a(RUN) \cup \mathcal{S}_a(BKDAA)) \backslash (\mathcal{S}_a(RUN) \cap \mathcal{S}_a(BKDAA)) \neq \emptyset \quad (11)$$

Proof: The validity of Equation 11 is established by means of the construction provided in Figure 2. The AMC depicted in this figure consists of resources R_1, R_2 and R_3 with capacities 1, 1 and 2, respectively. In its current configuration the cell supports two job routes, JR_1 and JR_2, with the related routing sequences annotated in the figure. Detailed analysis of case (a) depicted in Figure 2, establishes that the transition attempted (i.e., loading an instance of job route JR_1) not admissible by RUN, since the number of jobs in the neighborhood of resource R_3 exceeds its capacity. On the other hand, BKDAA "recognizes" the fact that the newly loaded job, can in fact be eventually buffered in the unshared resource R_2, unclogging the system of shared resources. Hence it admits the transition considered. The situation is reversed in case (b). Here, loading of a new instance of job route JR_2 is rejected by BKDAA, since there is no free path for it to its next unshared segment (i.e., to its completion and unloading from the system, in this particular case). The step is accepted by RUN, though, since no neighborhood capacity constraint is violated; in essence, RUN makes more pertinent use in this case of the fact that resource R_3 has increased capacity.

In summary, Figure 2 depicts an AMC configuration which gives rise to a transition (a) admitted by BKDAA and not admitted by RUN, and to a transition (b) which is admitted by RUN and not admitted by BKDAA. It must further be noticed that since, both, RUN and BKDAA have been proved to be correct, admittance of a transition by either one, guarantees the safety of the resulting state. These two remarks are sufficient to establish the validity of Proposition 4. $\diamond$

The significance of the essential difference of these two policies can be realized by noticing that *the disjunction of two correct deadlock avoidance policies is a correct policy*. To see this, observe that the possibility of advancement of at least one job in any state s^i admissible by the policy disjunction, is guaranteed by the policy(-ies) which admits the state s^i (cf. Prop. 3). Furthermore, the policy resulting from the disjunction of two essentially different policies admits a larger subspace of the safe state space of the system, and thus, it is closer to the optimal policy $\mathcal{P}^*$. Thus, disjunction of two essential different policies can lead to increased flexibility and higher effective production capacity and AMC performance than in the case where only one of the two policies is applied.

6 Conclusions

In this last section, the contributions of this paper are summarized, and possible extensions of this work are proposed.

The starting point for this work was the observation that the AMC operational environment is susceptible to deadlocks, which, given the high level of structuring and the monitoring capabilities of the AMC, should be resolved through avoidance techniques. However, this is a computationally involved task, due to the NP-complete nature of the state-safety decision problem. At the same time, our position is that any viable solution to the problem should possess the properties of correctness and scalability. In our efforts to design such solutions, an operational model for the AMC which is based on the FSA computational model was developed. The problem of state-safety was reformulated as a set of binary equations in the context of that model. This formulation provides an analytical tool / framework for formally proving the correctness of a certain class of avoidance policies, i.e. those which can be expressed as a set of "static" constraints on the system resource allocation. The functionality of our framework was demonstrated by using it to prove the correctness of the RUN avoidance policy, developed in the authors' research and first presented in [10], and to provide an alternative proof for BKDAA, introduced in [3]. In addition, we have shown that by looking at different operational aspects of the AMC, RUN and BKDAA, in general, recognize the safeness of different subspaces of the initial $\mathcal{S}_{rs}$ space.

This result of the essential difference of RUN and BKDAA has motivated the search for additional policies of the above class, with some successful preliminary results. Completion of the development of these new policies is part of our future work. Our future plans also include the comparative study of these policies w.r.t. AMC flexibility. The extension of this framework to cover other RAS classes is being undertaken. For example, the situation where multiple units of the same resource are allocated to a process at any step is a straightforward generalization of the material presented. The case where multiple units of different resources are simultaneously required for a job is less intuitive. Finally, the integration of the correctness and performance analyses by means of the proposed FSA model is another issue to be resolved.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] T. Araki, Y. Sugiyama, and T. Kasami. Complexity of the deadlock avoidance problem. In *2nd IBM Symp. on Mathematical Foundations of Computer Science*, pages 229–257. IBM, 1977.

[3] Z. A. Banaszak and B. H. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. *IEEE Trans. on Robotics and Automation*, 6:724–734, 1990.

[4] C. G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis.* Aksen Associates Inc. Publ., Boston, MA, 1993.

[5] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3:67–78, 1971.

[6] National Research Council. Information technology and manufacturing. Technical report, National Academy Press, 1993.

[7] P. F. Drucker. The emerging theory of manufacturing. *Harvard Business Review*, May-June:94–102, 1990.

[8] L. Ferrarini. An incremental approach to logic controller design with petri nets. *IEEE Trans. on Systems Man and Cybernetics*, 22:461–473, 1992.

[9] L. Ferrarini, M. Narduzzi, and M. Tassan-Solet. A new approach to modular liveness analysis conceived for large logic controllers' design. *IEEE Trans. on Robotics and Automation*, 10:169–184, 1994.

[10] E. H. Gaarder. Deadlock avoidance in flexible manufacturing systems. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1993.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., New York, NY, 1979.

[12] A. Giua and F. DiCesare. Petri net structural analysis for supervisory control. *IEEE Trans. on Robotics and Automation*, 10:169–184, 1994.

[13] E. M. Gold. Deadlock prediction: Easy and difficult cases. *SIAM Journal of Computing*, 7:320–336, 1978.

[14] A. N. Habermann. Prevention of system deadlocks. *Comm. ACM*, 12:373–377, 1969.

[15] R. D. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4:179–196, 1972.

[16] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[17] F. S. Hsieh and S. C. Chang. Dispatching-driven deadlock avoidance controller synthesis for flexible manufacturing systems. *IEEE Trans. on Robotics and Automation*, 10:196–209, 1994.

[18] M. A. Lawley. Computational modeling, structural analysis and configuration flexibility for flexible manufacturing system controllers. Technical Report UILU-ENG 94-4015 – Ph. D. Thesis Proposal, Univesity of Illinois at Urbana-Champaign, 1994.

[19] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, 1989.

[20] J. L. Peterson. *Operating System Concepts*. Addison-Wesley, 1981.

[21] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[22] A. Shoshani and E. G. Coffman. Sequencing tasks in multiprocess systems to avoid deadlocks. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory*, pages 225–233, 1970.

[23] N. Viswanadham, Y. Narahari, and T. L. Johnson. Deadlock avoidance in flexible manufacturing systems using petri net models. *IEEE Trans. on Robotics and Automation*, 6:713–722, 1990.

[24] R. A. Wysk, N. S. Yang, and S. Joshi. Detection of deadlocks in flexible manufacturing cells. *IEEE Trans. on Robotics and Automation*, 7:853–859, 1991.

[25] R. A. Wysk, N. S. Yang, and S. Joshi. Resolution of deadlocks in flexible manufacturing systems: Avoidance and recovery approaches. *Journal of Manufacturing Systems*, 13:128–138, 1994.

Spiridon A. Reveliotis received his B.S. in Electrical Engineering from National Technical University of Athens, Greece, in 1989, and his M.S. in Computer Systems Engineering / Engineering Software Design from Northeastern University, Boston, in 1992. Currently he is with the Mechanical & Industrial Engineering Department of the University of Illinois at Urbana-Champaign, pursuing a Ph.D. in the area of supervisory control of Flexible Manufacturing Systems. His academic interests are related to systems modeling and control by means of classical and modern control methodologies, and the development of functional frameworks that can accommodate these methodologies, on the basis of the particular application features. Mr. Reveliotis is a member of the IEEE, SME, and Phi-Kappa-Phi societies.

Placid M. Ferreira is an Associate Professor at the Mechanical & Industrial Engineering Department of the University of Illinois at Urbana-Champaign. He received his Ph.D. in Industrial Engineering from Purdue University, in 1987. Dr. Ferreira is an Associate Editor of the IIE Transactions. His research interests include Computer Aided Design, flexible automation, and the supervisory control of Flexible Manufacturing Systems.