A practical approach to the design of maximally permissive liveness-enforcing supervisors for complex resource allocation systems

Ahmed Nazeem and Spyros Reveliotis
School of Industrial & Systems Engineering
Georgia Institute of Technology
anazeem@gatech.edu, spyros@isye.gatech.edu

***Abstract*— The problem of designing and deploying liveness-enforcing supervisors (LES) for sequential resource allocation systems is well-documented and extensively researched in the current literature. Acknowledging the fact that the computation of the maximally permissive LES is an NP-hard problem, most of the present solutions tend to trade off maximal permissiveness for computational tractability and ease of the policy design and implementation. In this work, we demonstrate that the maximally permissive LES can be a viable solution for the resource allocation taking place in many practical applications, by (a) effectively differentiating between the off-line and on-line problem complexity, and (b) controlling the latter through the development of succinct and compact representations of the information that is necessary for the characterization of the maximal permissive LES.**

I. Introduction

The problem of liveness-enforcing supervision – or deadlock avoidance – for sequential resource allocation systems (RAS) has received extensive attention in the literature. In its basic definition, this problem concerns the coordinated allocation of the finite system resources to a set of concurrently executing processes, that are competing for the staged acquisition and release of these resources, so that every process can eventually proceed to its completion. In particular, the applied control policy must avoid the development of circular waiting patterns where a subset of processes are waiting upon each other for the release of the resources that are needed for their further advancement, a situation characterized as "deadlock" in the relevant literature.

The study of the deadlock avoidance problem was initiated in the late 60's and early 70's, and it was primarily motivated by the needs of the resource allocation that was taking place in the context of the computing technologies emerging at that time [9], [8], [4], [10]. Some of the main contributions of that era were (i) the formalization of the concept of deadlock and of the resource allocation dynamics that lead to its formation by means of graph-theoretic concepts and structures, and (ii) the identification of off-line structural conditions and on-line resource allocation policies that would guarantee the deadlock-free operation of the underlying system; the design of the resource allocation processes so that they do not give rise to any circular waiting patterns is an example of the aforementioned structural conditions, while Banker's algorithm [5] is the best known deadlock avoidance policy (DAP) of that era. An additional but later development of that era (late 70's) was the systematic study of the computational complexity of the maximally permissive[1] deadlock avoidance policy for any given RAS and the establishment of its NP-hardness for the majority of RAS behavior [1], [7]. The problem of deadlock avoidance was subsequently revived in the late 80's / early 90's, primarily in the context of the resource allocation taking place in flexibly automated production systems and intelligent transportation systems. The defining characteristics of these new studies were (i) the better specificity and tractability / predictability of the underlying resource allocation processes with respect to their resource allocation requests, and (ii) the employment of the simultaneously emerging qualitative Discrete Event Systems (DES) theory [2] as a powerful and rigorous base for modeling, analyzing and eventually controlling the considered RAS dynamics. The combination of these two effects has led to (a) a more profound understanding of the process of deadlock formation and of the RAS structural attributes that facilitate this process (under various DES-based representations), and also to (b) a multitude of methodologies that can provide effective deadlock avoidance policies for many RAS classes. Most of the developments of this second era of studies on the problem of deadlock avoidance in sequential RAS, as well as the spirit that has driven these developments, can be traced in [14], [16], [12], [11].

The perusal of the available literature on the LES design for sequential RAS cited in the previous paragraph, will reveal that many (actually almost all) of the currently proposed solutions are of a suboptimal nature, since they tend to trade off maximal permissiveness for computational efficiency and ease in the policy development and implementation. In this work, we take a more aggressive attitude to the problem, seeking to synthesize *maximally permissive* LES for the considered RAS. In particular, we seek to develop a methodology that will provide the maximally permissive LES for any given instantiation from the considered RAS class(es), while controlling to any possible extent the on-line computational complexity of the policy implementation. Our intentions are motivated by the following remarks:

- The aforementioned NP-hardness of the maximally

[1]Maximal permissiveness and all other technical concepts appearing in this introductory discussion will be systematically / formally defined in the subsequent sections.

permissive LES for the considered RAS should be interpreted as a "worst-case" result. On the other hand, in decision sciences there are quite a few problems of non-polynomial computational complexity that in practice demonstrate a much more benign "empirical" complexity, accepting exact solutions with reasonable computational effort.

- Furthermore, in many cases, a non-polynomial computational complexity might result from the off-line computational effort that is necessary for acquiring a target solution / policy, but the final result might be adequately compact to enable an efficient / practical on-line implementation.

In the case of the computation of the maximally permissive LES for the considered RAS, the solution can be obtained through the "trimming" of the finite state automaton (FSA) that models the underlying RAS behavior, while assessing state reachability and co-reachability with respect to the RAS empty state. This is a well understood and very straightforward calculation [2]. The only problem is that the aforementioned FSA grows exponentially large with respect to the "size" of the more compact representations expressing the structure of the corresponding RAS. Yet, as we will demonstrate in the following, the severity of this problem is mitigated by the fact that the computation of the trimmed FSA that characterizes the maximally permissive LES is an off-line computation, which makes the increased requirements in terms of time and other computational resources more affordable. On the other hand, any real-time implementation of the maximally permissive LES essentially constitutes a mechanism that assesses whether any reachable RAS state belongs to the aforementioned trimmed automaton or not, a property known as the state "safety". The key thesis of this paper is that in many practical applications of the considered RAS theory, once the aforementioned trimmed FSA has been obtained, it is possible to encode the information necessary to resolve the underlying state safety problem in a "data structure / mechanism" sufficiently compact so that the problem can be effectively addressed within the time and other resource constraints that typically arise in a real-time computation. This result is enabled by:

1) some topological properties of the underlying state space and its partition to safe and unsafe subspaces, that allow the classification of the entire state space while considering explicitly only a (typically very) small subset of the underlying state space;
2) the selection of pertinent data structures that will store the information characterized in step 1 above in a compact manner, and in a way that facilitates the on-line processing of this information.

In the light of the above remarks, the rest of the paper is organized as follows: Section II provides a formal characterization of the RAS class considered in this paper and of the problem of maximally permissive, liveness-enforcing supervision arising in this class. It also provides the aforementioned topological properties of the RAS state space that will enable the subsequent developments of the paper. Section III presents the main results of the paper by detailing the methodological approach pursued in this work. Section IV demonstrates the applicability of the approach by implementing it on an example problem instance borrowed from the literature, and it also discusses our experiences with implementations involving larger and/or more complex RAS configurations. Finally, Section V concludes the paper by summarizing its contributions and outlining some further extensions of theoretical and practical interest.

II. The considered RAS class and the maximally permissive liveness-enforcing supervision problem

The considered RAS For the sake of simplicity and specificity, we present the main results of this paper in the context of the *Conjunctive / Disjunctive (C/D)* class of the RAS taxonomy presented in [14]. We notice, however, that the presented ideas and results are extensible to more complex classes of that taxonomy.

A *Conjunctive / Disjunctive Resource Allocation System (D/C-RAS)* is formally defined by a 4-tuple $\Phi =< \mathcal{R}, C, \mathcal{P}, D >$, where: (i) $\mathcal{R} = \{R_1, \ldots, R_m\}$ is the set of the system *resource types*. (ii) $C : \mathcal{R} \rightarrow Z^+$ – the set of strictly positive integers – is the system *capacity* function, characterizing the number of identical units from each resource type available in the system. Resources are assumed to be *reusable*, i.e., each allocation cycle does not affect their functional status or subsequent availability, and therefore, $C(R_i) \equiv C_i$ constitutes a system *invariant* for each i. (iii) $\mathcal{P} = \{\Pi_1, \ldots, \Pi_n\}$ denotes the set of the system *process types* supported by the considered system configuration. Each process type Π_j is a composite element itself, in particular, $\Pi_j =< \mathcal{S}_j, \mathcal{G}_j >$, where: (a) $\mathcal{S}_j = \{\Xi_{j1}, \ldots, \Xi_{j,l_j}\}$ denotes the set of *processing stages* involved in the definition of process type Π_j, and (b) $\mathcal{G}_j$ is an *acyclic digraph* with its node set, V_j, being bijectively related to the set $\mathcal{S}_j$. Let $V_j^{\nearrow}$ (resp., $V_j^{\searrow}$) denote the set of *source* (resp., *sink*) nodes of $\mathcal{G}_j$. Then, any *path* from some node $v_s \in V_j^{\nearrow}$ to some node $v_f \in V_j^{\searrow}$ defines a *process plan* for process type Π_j. Also, in the following, we shall let $\Xi \equiv \bigcup_{j=1}^{n} \mathcal{S}_j$ and $\xi \equiv |\Xi|$. (iv) $D : \bigcup_{j=1}^{n} \mathcal{S}_j \rightarrow \prod_{i=1}^{m} \{0, \ldots, C_i\}$ is the *resource allocation function* associating every processing stage Ξ_{jk} with the *resource allocation vector* $D(\Xi_{ij})$ required for its execution. At any point in time, the system contains a certain number of (possibly zero) instances of each process type that execute one of the corresponding processing stages. A process instance executing a non-terminal stage $\Xi_{ij} \in V_i \backslash V_i^{\searrow}$, must first be allocated the resource differential $(D(\Xi_{i,j+1}) - D(\Xi_{ij}))^+$ in order to advance to (some of) its next stage(s) $\Xi_{i,j+1}$, and only then it will release the resource units $|(D(\Xi_{i,j+1}) - D(\Xi_{ij}))^-|$, that are not needed anymore. The considered resource allocation protocol further requires that no resource type $R_i \in \mathcal{R}$ is over-allocated with respect to its capacity C_i at any point in time.

The Deterministic FSA abstracting the D/C-RAS dy-

namics The dynamics of the D/C-RAS $\Phi =< \mathcal{R}, C, \mathcal{P}, D >$, described in the previous paragraph, can be further formalized by a *Deterministic Finite State Automaton (DFSA)* [2], $G(\Phi) = (S, E, f, \mathbf{s}_0, S_M)$, that is defined as follows:

1) The *state set* S consists of ξ-dimensional vectors $\mathbf{s}$. The components $\mathbf{s}[q]$, $q = 1, \ldots, \xi$, of $\mathbf{s}$ are in one-to-one correspondence with the RAS processing stages, and they indicate the number of process instances executing the corresponding stage in the considered RAS state. Hence, S consists of all the vectors $\mathbf{s} \in (Z_0^+)^\xi$ that further satisfy

$$\forall i = 1, \ldots, m, \quad \sum_{q=1}^{\xi} \mathbf{s}[q] \cdot D(\Xi_q)[i] \leq C_i \qquad (1)$$

where, according to the adopted notation, $D(\Xi_q)[i]$ denotes the allocation request for resource R_i that is posed by stage Ξ_q.

2) The *event set* E is the union of the disjoint event sets $E^{\nearrow}$, $\bar{E}$ and $E^{\searrow}$, where:
   a) $E^{\nearrow} = \{e_{rp} : r = 0,\ \Xi_p \in \bigcup_{j=1}^{n} V_j^{\nearrow}\}$, i.e., event e_{rp} represents the *loading* of a new process instance that starts from stage Ξ_p.
   b) $\bar{E} = \{e_{rp} : \exists j \in 1, \ldots, n$ s.t. Ξ_p is a successor of Ξ_r in graph $\mathcal{G}_j\}$, i.e., e_{rp} represents the *advancement* of a process instance executing stage Ξ_r to a successor stage Ξ_p.
   c) $E^{\searrow} = \{e_{rp} : \Xi_r \in \bigcup_{j=1}^{n} V_j^{\searrow},\ p = 0\}$, i.e, e_{rp} represents the *unloading* of a finished process instance after executing its last stage Ξ_r.

3) The *state transition function* $f : S \times E \rightarrow S$ is defined by $\mathbf{s}' = f(\mathbf{s}, e_{rp})$, where the components $\mathbf{s}'[q]$ of the resulting state $\mathbf{s}'$ are given by:

$$\mathbf{s}'[q] = \begin{cases} \mathbf{s}[q] - 1 & \text{if} \quad q = r \\ \mathbf{s}[q] + 1 & \text{if} \quad q = p \\ \mathbf{s}[q] & \text{otherwise} \end{cases}$$

   Furthermore, $f(\mathbf{s}, e_{rp})$ is a *partial* function defined only if the resulting state $\mathbf{s}' \in S$.

4) The *initial state* $\mathbf{s}_0 = \mathbf{0}$, which corresponds to the situation when the system is empty of any process instances.

5) The *set of marked states* S_M is the singleton $\{\mathbf{s}_0\}$, and it expresses the requirement for complete process runs.

Let $\hat{f}$ denote the natural extension of the state transition function f to $S \times E^*$; i.e., for any $\mathbf{s} \in S$ and the empty event string ϵ,

$$\hat{f}(\mathbf{s}, \epsilon) = \mathbf{s} \qquad (2)$$

while for any $\mathbf{s} \in S$, $\sigma \in E^*$ and $e \in E$,

$$\hat{f}(\mathbf{s}, \sigma e) = f(\hat{f}(\mathbf{s}, \sigma), e) \qquad (3)$$

In Equation 3 it is implicitly assumed that $\hat{f}(\mathbf{s}, \sigma e)$ is undefined if any of the one-step transitions that are involved in the right-hand-side recursion are undefined.

The behavior of RAS Φ is modeled by the *language* $L(G)$ *generated* by DFSA $G(\Phi)$, i.e., by all strings $\sigma \in E^*$ such that $\hat{f}(\mathbf{s}_0, \sigma)$ is defined. Furthermore, the *reachable subspace* of $G(\Phi)$ is the subset S_r of S defined as follows:

$$S_r \equiv \{\mathbf{s} \in S : \exists \sigma \in L(G) \text{ s.t. } \hat{f}(\mathbf{s}_0, \sigma) = \mathbf{s}\} \qquad (4)$$

We also define the *safe subspace* of $G(\Phi)$, S_s, by:

$$S_s \equiv \{\mathbf{s} \in S : \exists \sigma \in E^* \text{ s.t. } \hat{f}(\mathbf{s}, \sigma) = \mathbf{s}_0\} \qquad (5)$$

In the following, we shall denote the complements of S_r and S_s with respect to S by $S_{\bar{r}}$ and $S_{\bar{s}}$, respectively, and we shall refer to them as the *unreachable* and *unsafe* subspaces. Finally, S_{xy}, $x \in \{r, \bar{r}\}$, $y \in \{s, \bar{s}\}$, will denote the intersection of the corresponding sets S_x and S_y.

The target behavior of $G(\Phi)$ and the structure of the maximally permissive LES The desired (or "target") behavior of RAS Φ is expressed by the *marked language* $L_m(G)$, which is defined by means of the set of marked states S_M, as follows:

$$\begin{aligned} L_m(G) &\equiv \{\sigma \in L(G) : \hat{f}(\mathbf{s}_0, \sigma) \in S_M\} \\ &= \{\sigma \in L(G) : \hat{f}(\mathbf{s}_0, \sigma) = \mathbf{s}_0\} \end{aligned} \qquad (6)$$

Equation 6, when combined with all the previous definitions, further implies that the set of states that are accessible under $L_m(G)$ is exactly equal to S_{rs}. Hence, starting from state $\mathbf{s}_0$, a *maximally permissive liveness-enforcing supervisor (LES)* must allow / enable a system-enabled transition to a next state $\mathbf{s}$ if and only if (*iff*) $\mathbf{s}$ belongs to S_s. This characterization of the maximally permissive LES ensures its uniqueness for any given D/C-RAS instantiation. It also implies that the policy can be effectively implemented through any mechanism that recognizes and rejects the unsafe states that are accessible through one-step transitions from S_{rs}. As we shall see in the following, this last observation can decrease substantially the set of unsafe states that must be explicitly considered in the design of any mechanism that will implement that maximally permissive LES. We conclude this section by discussing an additional property of the considered RAS that will prove very useful in the efficient implementation(s) of the maximally permissive LES sought in this work.

Some monotonicities observed by the state safety and unsafety concepts It should be clear from the previous discussion that the ability of the activated processes in a given D/C-RAS state $\mathbf{s} \in S$ to proceed to completion, depends on the existence of a sequence $< \mathbf{s}^{(0)} \equiv \mathbf{s}, e^{(1)}, \mathbf{s}^{(1)}, e^{(2)}, \mathbf{s}^{(2)}, \ldots, \mathbf{s}^{(n-1)}, e^{(n)}, \mathbf{s}^{(n)} \equiv \mathbf{s}_0 >$, such that at every state $\mathbf{s}^{(i)}$, $i = 0, 1 \ldots, n-1$, the free (or "slack") resource capacities at that state enable the job advancement corresponding to event $e^{(i+1)}$. Furthermore, if such a terminating sequence exists for a given state $\mathbf{s}$, then the event feasibility condition defined by Equation 1 implies that this sequence will also provide a terminating sequence for every other state $\mathbf{s}' \leq \mathbf{s}$, where the inequality is taken component-wise. On the other hand, if state $\mathbf{s}$ possesses no terminating sequences, then it can be safely inferred that no such terminating sequences will exist for any other state $\mathbf{s} \leq \mathbf{s}'$ (since, otherwise, there should also exist a terminating

sequence for $\mathbf{s}$, according to the previous remark). The next proposition provides a formal statement to the above observations; these results are well known in the literature, and therefore, their formal proof is omitted.

Proposition 1: Consider the (partial) ordering relationship "$\leq$" imposed on the state space S of a given D/C-RAS Φ that is defined as follows:

$$\forall \mathbf{s}, \mathbf{s}' \in S, \;\; \mathbf{s} \leq \mathbf{s}' \iff (\forall i = 1, \ldots \xi, \;\; \mathbf{s}[i] \leq \mathbf{s}'[i]) \quad (7)$$

Then,

1) $\mathbf{s} \in S_s \;\wedge\; \mathbf{s}' \leq \mathbf{s} \implies \mathbf{s}' \in S_s$
2) $\mathbf{s} \in S_u \;\wedge\; \mathbf{s} \leq \mathbf{s}' \implies \mathbf{s}' \in S_u$

□

In the light of Proposition 1, next we define the concepts of *maximal safe state* and *minimal unsafe state*, that will play an important role in the subsequent developments:

Definition 1: Let $\mathbf{s} < \mathbf{s}'$ (resp. $\mathbf{s} > \mathbf{s}'$) denote the fact that $\mathbf{s} \leq \mathbf{s}'$ (resp. $\mathbf{s} \geq \mathbf{s}'$) and there is at least a pair of components $\mathbf{s}[i]$, $\mathbf{s}'[i]$ for which the corresponding inequality is strict. Then, given an LC-RAS $\Phi = (\mathcal{R}, \mathcal{C}, \mathcal{P}, D)$,

1) a reachable safe state $\mathbf{s} \in S_{rs}$ is *maximal iff* $\neg\exists$ a reachable safe state $\mathbf{s}' \in S_{rs}$ such that $\mathbf{s}' > \mathbf{s}$;
2) a reachable unsafe state $\mathbf{s} \in S_{r\bar{s}}$ is *minimal iff* $\neg\exists$ a reachable unsafe state $\mathbf{s}' \in S_{r\bar{s}}$ such that $\mathbf{s}' < \mathbf{s}$.

Finally, in the sequel, the set of maximal reachable safe states will be denoted by $\bar{S}_{rs}$, and the set of minimal reachable unsafe states will be denoted by $\bar{S}_{r\bar{s}}$.

III. The proposed approach

Outlining the proposed approach As observed in Section II, the effective implementation of the maximally permissive LES for any given D/C-RAS, Φ, is equivalent to the recognition and the blockage of transitions from the safe to the unsafe region of the underlying state space S. In the following, we shall refer to the reachable unsafe states $\mathbf{s} \in S_{r\bar{s}}$ that are reachable from the safe subspace S_{rs} through a single transition, as *"boundary"* reachable unsafe states, and we shall denote the relevant set by $S^b_{r\bar{s}}$. Then, in principle, an implementation of the maximal LES for any given D/C-RAS Φ can be based on the explicit computation / enumeration and storage of the set $S^b_{r\bar{s}}$; starting from the initial state $\mathbf{s}_0$, any transition $\mathbf{s}' = f(\mathbf{s}, e)$ that is system-enabled according to Equation 1, should be admissible by the maximally permissive LES *iff* $\mathbf{s}' \notin S^b_{r\bar{s}}$. A practical implementation of such a control scheme will require (a) the effective computation of the set $S^b_{r\bar{s}}$ and (b) its storage in such a manner that the test $\mathbf{s}' \notin S^b_{r\bar{s}}$ is tractable within the time budget constraints that are enforced by the "embedded / real-time" nature of the implemented supervisor. The rest of this section discusses how to facilitate these two requirements and render the above control scheme a viable solution for many practical application contexts.

An efficient computation of the set $S^b_{r\bar{s}}$ Given a D/C RAS $\Phi =< \mathcal{R}, C, \mathcal{P}, D >$, the computation of the set $S^b_{r\bar{s}}$ essentially requires (i) the computation of the reachable state space S_r of the corresponding DFSA $G(\Phi)$, (ii) the trimming of this state space with respect to its initial state $\mathbf{s}_0 = \mathbf{0}$, in order to obtain the sets S_{rs} and $S_{r\bar{s}}$, and (iii) the extraction of $S^b_{r\bar{s}}$ from $S_{r\bar{s}}$ by identifying all those states $\mathbf{s} \in S_{r\bar{s}}$ that are accessible from S_{rs} through a single transition. All these three steps are performed off-line, during the controller design process, and therefore, they are more amenable to the complications arising from the expected (very) large sizes of the set S_r and its aforementioned derivatives. Next we report a particular algorithm for the generation and storage of S_r that has been found to be especially efficient in our computational studies. This algorithm provides an enumeration of S_r, by first identifying, as an intermediary step, all the states corresponding to a feasible resource allocation, according to the prevailing resource capacity constraints (c.f., Eq. 1); we shall refer to these RAS states as *"valid"* states, and the corresponding state set will be denoted by S_v. Once S_v has been constructed, a subsequent procedure filters out from it the set of reachable states S_r. Hence, the whole computation is organized naturally into two major procedures: (a) that of generating state set S_v, and (b) that of reducing S_v to S_r.

To describe the first of the aforementioned procedures, let us denote by K_{ij} the maximum number of process instances that can execute concurrently a processing stage Ξ_{ij} without violating the capacity restrictions imposed by the resources involved in the execution of this stage. In the following discussion we shall also use Ξ^k_{ij} to denote the existence of k active process instances at the processing stage Ξ_{ij}, and $\mathbf{s}_{ij}$ to denote the state component corresponding to processing stage Ξ_{ij}. Given a resource allocation state $\mathbf{s}$, we shall say that (the "process load" indicated by) $\Xi^{k'}_{i'j'}$ can be added to state $\mathbf{s}$ *iff* $\mathbf{s}_{i'j'} = 0$ and the state $\mathbf{s}' \equiv \{\forall (i,j) \neq (i',j') : \mathbf{s}'_{ij} = \mathbf{s}_{ij}$ and $\mathbf{s}'_{i'j'} = k\}$ does not violate any resource capacity. The proposed algorithm enumerates the set of valid states, S_v, starting with state $\mathbf{s}_0 \equiv \mathbf{0}$, and subsequently considering for every generated state $\mathbf{s} \in S_v$, the possibility of adding $\Xi^{k'}_{i'j'}$ to it, for all i, j and k'. This enumeration is systematized and facilitated by the following two data structures:

- A composite data structure called $Node_{\mathbf{s}}$, that supports the generation and processing of a single state $\mathbf{s}$ in the overall enumeration process. This data structure consists of the following two components:
  - $\mathbf{s}$: the vector representation of state $\mathbf{s}$.
  - $L_{\mathbf{s}}$: a list containing all the "process loads" Ξ^k_{ij} that can be added to state $\mathbf{s}$.
- The queue, Q, of the "unprocessed" state nodes; a state node $Node_{\mathbf{s}}$ is considered processed, if all the state nodes resulting by the additions indicated in its list L_s, have been constructed and added to Q.

The complete algorithm for generating the valid state space S_v is provided in Figure 1. Next, we highlight the main steps in the algorithm: Since every processing stage Ξ_{ij} can have up to K_{ij} active jobs added to the empty state $\mathbf{s}_0$, the list $L_{\mathbf{s}_0}$, constructed in step 3, contains all the possible states having only one active process stage, and

Input: Representation of a given resource allocation system Φ.
Output: The list of states that constitutes the valid state space S_v.

1) $S_v \leftarrow \emptyset$; $Q \leftarrow \emptyset$
2) Insert $\mathbf{s}_0$ into S_v
3) $L_{\mathbf{s}_0} = \{\Xi_{ij}^k : \forall = i \in \{1,\ldots,n\}, \forall j \in \{1,\ldots,l_i\}, \forall k \in \{1,\cdots,K_{ij}\}\}$
4) Add $Node_{\mathbf{s}_0} \equiv (\mathbf{s}_0, L_{\mathbf{s}_0})$ to Q
5) while($Q \neq \emptyset$)
   a) $Node_{\mathbf{s}} \equiv (\mathbf{s}, L_{\mathbf{s}}) \leftarrow$ Pop Q
   b) for each $\Xi_{pq}^r \in L_{\mathbf{s}}$
      i) $\mathbf{s}^* \leftarrow$ Add$(\mathbf{s}, \Xi_{pq}^r)$
      ii) $L_{\mathbf{s}^*} \leftarrow \{\Xi_{ij}^k : \Xi_{ij}^k$ can be added to state $\mathbf{s}^* \wedge ((i > p) \vee ((i = p) \wedge (j > q)))\}$
      iii) Push $Node_{\mathbf{s}^*} \equiv (\mathbf{s}^*, L_{\mathbf{s}^*})$ to queue Q
      iv) Insert $\mathbf{s}^*$ into S_v
6) Return S_v

Fig. 1. The algorithm constructing the set of valid states S_v.

with this processing stage set to all the possible numbers of active jobs that it can have. On the other hand, the while loop in Step 5 operates as follows: Step 5a extracts a node $(\mathbf{s}, L_{\mathbf{s}})$ stored in queue Q for further processing. For every element Ξ_{pq}^r in list $L_{\mathbf{s}}$, Step 5b(i) constructs a new state $\mathbf{s}^*$ from Ξ_{pq}^r and $\mathbf{s}$ in the manner described above, i.e., $\mathbf{s}^* = \{\forall (i,j) \neq (p,q) : \mathbf{s}_{ij}^* = \mathbf{s}_{ij}$ and $\mathbf{s}_{pq}^* = r\}$. Step 5b(ii), constructs the list $L_{\mathbf{s}^*}$ for the node $Node_{\mathbf{s}^*}$ corresponding to state s^* constructed in step 5b(i). This list contains all Ξ_{ij}^k that satisfy the following properties: First, they can be added to state $\mathbf{s}^*$, which means that $\mathbf{s}_{ij}^* = 0$ but having k jobs at Ξ_{ij} concurrently with all the active jobs at state $\mathbf{s}^*$ will not violate any resource capacities. Second, the index $(\sum_{a=0}^{i-1} l_a + j)$ of processing stage Ξ_{ij} in the state vector is strictly greater than the index $(\sum_{a=0}^{p-1} l_a + q)$ of the processing stage Ξ_{pq} in the state vector, which is true *iff* $(i > p) \vee ((i = p) \wedge (j > q))$. The first condition essentially filters the set of processing stages to detect those which can have active jobs concurrently with the active jobs in $\mathbf{s}^*$. The second condition is necessary in order to avoid the generation of a state more than once. Step 5b(iii) queues the constructed node $Node_{\mathbf{s}^*}$ for further processing. The loop is terminated when all the state nodes entered in queue Q have been processed. It should be clear from the above, that at this point, all the valid states have been generated.

Regarding the (time) complexity of the algorithm of Figure 1, first we notice that Step 5a as well as Steps 5b(i) through 5b(iv) are executed $\mathcal{O}(|S_v|)$ times. On the other hand, the running time of Steps 5a, 5b(iii) and 5b(iv) is $\mathcal{O}(1)$. The running time of Step 5b(i) is $\mathcal{O}(\xi)$. The running time of Step 5b(ii) is $\mathcal{O}(\sum_{i=1}^{n}\sum_{j=1}^{l_i} K_{ij})$. Therefore the overall running time of the algorithm is $\mathcal{O}((\xi + \sum_{i=1}^{n}\sum_{j=1}^{l_i} K_{ij}) \cdot |S_v|)$. Since, practically, $(\xi + \sum_{i=1}^{n}\sum_{j=1}^{l_i} K_{ij}) \ll |S_v|$, we can say that the running time of the algorithm is $\mathcal{O}(|S_v|)$.

Input: The set of valid states S_v.
Output: The set of states that constitutes the reachable subspace S_r.

1) Initialize L with the elements of the input set S_v
2) Sort L in ascending order. /*The empty state $\mathbf{s}_0$will be the first state.*/
3) $\forall i$, $isReachable(i) \leftarrow 0$; $reachableStack \leftarrow \emptyset$
4) push $L(0)$ onto $reachableStack$; $isReachable(\mathbf{s}_0) = 1$
5) while $reachableStack \neq \emptyset$
   a) $\mathbf{s} \leftarrow$ pop $reachableStack$
   b) Identify all the events that can be executed from $\mathbf{s}$, and generate the corresponding list of its sucessor states, N_s
   c) For each state $\mathbf{s}' \in N_s$
      - if $isReachable(\mathbf{s}') = 0$
        - push $\mathbf{s}'$ onto $reachableStack$
        - $isReachable(\mathbf{s}') \leftarrow 1$
6) $S_r = \{\mathbf{s} \in S_v : isReachable(\mathbf{s}) = 1\}$.
7) Return S_r

Fig. 2. The algorithm extracting the set S_r from the set of valid states S_v.

The reader should also notice that whenever a state $\mathbf{s}$ is processed by the above algorithm, it is guaranteed that it will not be considered again. Therefore, we do not need to keep a processed state in the core memory, but we can simply save it (as soon as it is declared "processed") in a file on the hard disk.[2] This remark further implies that the memory consumption of the above algorithm is mainly due to the maintenance of the queue of unprocessed states, Q. But this consumption is quite controllable: whenever Q becomes relatively large, we can write some of the states in a file on the hard disk, remove them from the memory, process the rest of the states, and finally, re-load the saved file into the queue and continue processing these additional states. Working in this way, we have been able to process D/C-RAS Φ with extremely large state spaces.

The second procedure that filters the set of valid states, S_v, to extract the set of reachable states, S_r, is presented in Figure 2. In this procedure, L is a list of states, $reachableStack$ is a stack of states, and $isReachable$ is a binary array whose length equals the length of L, and such that $isReachable(i) = 1$ *iff* $L(i)$ is a reachable state. Then, it should be obvious from the reading of the provided pseudocode, that the procedure essentially implements a "reaching scheme" that marks all the reachable states in the provided set S_v while starting from the initial state $\mathbf{s}_0$.

The (time) complexity of the procedure depicted in Figure 2 can be characterized as follows: Let $\bar{t}$ be the maximum number of transitions that emanate from any given state $\mathbf{s} \in S_r$. It is easy to see that $\bar{t}$ is upper-bounded by $\sum_{i=1}^{n} |V_i^{\nearrow}| + \sum_{i=1}^{n}\sum_{v \in V_i \setminus V_i^{\searrow}} \texttt{outdegree}(v) + \sum_{i=1}^{n} |V_i^{\searrow}|$, according to

[2]Continuous writing on the hard disk is not encouraged though. So, we buffer the processed states and write them to the hard disk in batches.

the notation introduced in Section II, and therefore, it relates polynomially to the parameters defining the size of the underlying RAS. The while loop in Step 5 is executed $\mathcal{O}(|S_r|)$ times. Step 5c is executed $\mathcal{O}(\bar{t})$ times in a single iteration of the while loop. Checking the if-condition inside Step 5c upon any given state $\mathbf{s}'$ takes $\mathcal{O}(log(|S_r|))$ time, using binary search. So, the overall complexity of the above algorithm is $\mathcal{O}(\bar{t}\cdot|S_r|\cdot log(|S_r|))$. Finally, given all the previous analysis, we can see that the overall complexity of the combined execution of the algorithms described in Figures 1 and 2, is $\mathcal{O}(\bar{t}\cdot|S_r|\cdot log(|S_r|)+(\xi+\sum_{i=1}^{n}\sum_{j=1}^{l_i}K_{ij})\cdot|S_v|)$. Practically $|S_v|\approx\mathcal{O}(|S_r|)$, $\bar{t}\ll|S_r|$ and $(\xi+\sum_{i=1}^{n}\sum_{j=1}^{l_i}K_{ij})\ll|S_v|$. Therefore, we can roughly say that the practical complexity of the algorithm is $\mathcal{O}(|S_r|\cdot log(|S_r|)$.

Once S_r has been constructed, its partitioning to S_{rs} and $S_{r\bar{s}}$ can be performed through a reaching scheme similar to that performed by the algorithm of Figure 2, where, however, the search for feasible transitions emanating from each state is in the reverse direction (i.e., across its incoming arcs in the relevant state transition diagram). Due to space limitations, we omit the detailed description of the relevant algorithm, and we only notice that this step can be supported with complexity $\mathcal{O}(\bar{t}_r\cdot|S_r|\cdot log(|S_r|))$, where $\bar{t}_r$ denotes the maximum number of transitions that can lead into any given state $\mathbf{s}\in S_s$. Finally, the extraction of the set $S^b_{r\bar{s}}$ from $S_{r\bar{s}}$ can be performed straightforwardly by an algorithm that, for every state $\mathbf{s}\in S_{r\bar{s}}$, it generates all the states $\mathbf{s}'$ that are accessible from $\mathbf{s}$ through a single reverse transition, and rejects $\mathbf{s}$ *iff* all the generated states $\mathbf{s}'$ belong in $S_{r\bar{s}}$. Foregoing the implementational details of this algorithm, we simply notice that it can be implemented with a computational complexity of $\mathcal{O}(\bar{t}_r\cdot|S_{r\bar{s}}|\cdot log(|S_{r\bar{s}}|))$.

Obtaining a more compressed characterization of the set $S^b_{r\bar{s}}$ The explicit data that is necessary for the complete characterization of the set $S^b_{r\bar{s}}$ can be further compressed, with respect to the explicit enumeration of the set obtained through the computation discussed in the previous paragraph. This compression can be attained on the basis of the following two observations:

Observation 1: Proposition 1 and Definition 1, provided in Section II, imply that we can assess membership into $S^b_{r\bar{s}}$ for any given state $\mathbf{s}\in S_r$, by (i) explicitly storing only the subset of its minimal elements $\overline{S}^b_{r\bar{s}}$, and (ii) checking whether there exists a state $\mathbf{s}'\in\overline{S}^b_{r\bar{s}}$ such that $\mathbf{s}\geq\mathbf{s}'$.

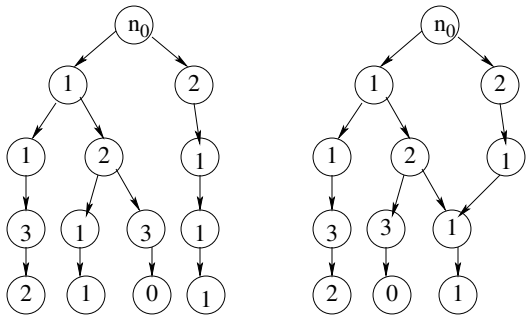
Observation 2: If a certain component q is equal to zero for every state $s\in\overline{S}^b_{r\bar{s}}$, then this component does not contribute any significant information in the state comparisons for the evaluation of the membership discussed in Observation 1 above, and therefore, it can be neglected during the execution of these comparisons. The (state) vector set that is obtained from the elements of $\overline{S}^b_{r\bar{s}}$ after the elimination of their redundant components, will be denoted by $P(\overline{S}^b_{r\bar{s}})$.

Observation 1 enables the further "thinning" of the set of boundary reachable unsafe states $S^b_{r\bar{s}}$, by retaining only its minimal elements, while Observation 2 supports a dimensionality reduction – or "projection" – of the elements of this "thinned" set. From a computational standpoint, both of these steps involve the post-processing of the set $S^b_{r\bar{s}}$ through some very simply and efficient computation.[3] On the other hand, as it will be demonstrated in Section IV, each of these two effects can lead to an extensive (frequently dramatic) reduction of the information that must be explicitly stored and processed for the effective implementation of the proposed control scheme. In fact, for many practical cases, a simple array-based storage of the elements of $P(\overline{S}^b_{r\bar{s}})$ will be quite adequate for effecting the on-line computation that is involved in the implementation of the maximal LES described in the previous paragraphs. However, in the rest of this section, we also discuss an additional data structure that can lead to more efficient storage of the set $P(\overline{S}^b_{r\bar{s}})$ and to more expedient algorithms for the on-line test suggested by Observation 1.

Storage and on-line processing of the set $P(\overline{S}^b_{r\bar{s}})$ through n-ary decision diagrams The (n-ary) decision diagrams proposed in the context of this work for the storage and on-line processing of the set $P(\overline{S}^b_{r\bar{s}})$, is an adaptation of the concept of the binary decision diagram (BDD) that has been used for the efficient storage and manipulation of boolean functions [3]. They can be systematically introduced by first defining the (n-ary) decision tree for the storage of k l-dimensional vectors $\{\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_k\}$: This tree has a dummy root node, n_0, of depth 0, and l layers of nodes with corresponding depths from 1 to l. Each of the layers numbered from 1 to l corresponds to one of the l dimensions of vectors $\mathbf{v}_i$. Starting with node n_0 as the single node of layer 0, the tree nodes at each of the remaining layers are defined recursively as follows: The children of a node n at layer $l(n)\in\{0,\ldots,l-1\}$ correspond to all the possible values of coordinate $l(n)+1$ in the vector subset of $\{\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_k\}$ that is obtained by fixing the first $l(n)$ coordinates at the values specified by the path from the root node n_0 to node n. The coordinate value that corresponds to each node n in layers 1 to l, according to this node generation scheme, is characterized as the "content" of n. Obviously, the nodes generated for layer l according to the previous recursion have no children, and they constitute the tree leaf nodes. Furthermore, it should be clear that in the decision tree constructed according to the aforestated rule, every vector $\mathbf{v}_i$, $i=1,\ldots,k$, is represented by the path to one of the tree leaf nodes.

The decision tree described in the previous paragraph is converted to a decision diagram, by iteratively identifying and eliminating duplicate sub-graphs in the generated structure, while starting from the last layer l. Two subgraphs – or sub-diagrams – originating at given layer $i\in\{1,\ldots,l\}$ are considered duplicate if (i) they are isomorphic and (ii) each isomorphically related pair of nodes has the same content. Figure 3 exemplifies the above definitions by depicting the decision tree and the corresponding decision diagram that store the vector set

[3]Once again, we forego the relevant details due to space limitations.



(a) The decision tree (b) The corresponding decision diagran

Fig. 3. A decision tree and the corresponding decision diagram storing the vector set $\{[1,2,1,1],[2,1,1,1],[1,1,3,2],[1,2,3,0]\}$.

Input: The decision diagram of a vector set V and a vector $\mathbf{v}'$.
Output: A boolean variable indicating whether there is a vector $\mathbf{v} \in V$ such that $\mathbf{v} \leq \mathbf{v}'$.

1) $\bar{l} := \dim(\mathbf{v})$; $EXIT := FALSE$
2) Push $(n_0, 0)$ on $SearchStack$
3) while ($SearchStack \neq \emptyset \wedge \neg EXIT$)
   a) $(n, l) \leftarrow$ pop $SearchStack$
   b) $l := l + 1$
   c) For each child n' of n
      - if ($content(n') \leq \mathbf{v}'[l] \wedge \neg EXIT$)
        - if $l = \bar{l}$ then $EXIT := TRUE$
        - else push (n', l) onto $SearchStack$
4) Return $EXIT$

Fig. 4. An algorithm that takes as input the decision diagram of a vector set V and a vector $\mathbf{v}'$, and checks whether there is a vector $\mathbf{v} \in V$ such that $\mathbf{v} \leq \mathbf{v}'$.

$\{[1,2,1,1],[2,1,1,1],[1,1,3,2],[1,2,3,0]\}$.

Space limitations do not allow a detailed, formal description of the algorithms that construct the decision tree and the corresponding decision diagram for a given vector set V. However, Figure 4 provides an algorithm that takes as input the decision diagram of a vector set V and a vector $\mathbf{v}'$, and it checks whether there is a vector $\mathbf{v} \in V$ such that $\mathbf{v} \leq \mathbf{v}'$. Starting with the root dummy node n_0, this algorithm essentially performs a depth-first search for a path to a leaf node, such that, at every layer $j = 1, \ldots, l$, it engages a node with content no greater than the value of component $\mathbf{v}'[j]$. If such a path is identified, the algorithm returns 'TRUE', (i.e., $\exists \mathbf{v} \in V$ such that $\mathbf{v} \leq \mathbf{v}'$, namely, the vector defined by the node contents of the constructed path). In the opposite case, the algorithm returns 'FALSE'. Obviously, the worst case computational complexity of this algorithm is $\mathcal{O}(\bar{n})$, where $\bar{n}$ denotes the number of nodes in the decision diagram of V.

IV. A CASE STUDY

In this section we consider the application of the methodology developed in the earlier parts of this work, to the synthesis of the maximal permissive LES of the D/C-RAS depicted in Figure 5. As it can be seen in Figure 5, the depicted D/C-RAS has seven resource types, $\{R_1, \ldots, R_7\}$, with the corresponding capacities annotated at the bottom of the figure. It also has three process types, $\{\Pi_1, \Pi_2, \Pi_3\}$, with their corresponding digraphs $\mathcal{G}_i$, $i = 1, 2, 3$, structured as indicated in the figure. Each of the nodes in each digraph $\mathcal{G}_i$ is labelled by the corresponding processing stage Ξ_{ij} of process type Π_i, while the resource allocation request of that stage, $D(\Xi_{ij})$, is indicated below the node (in this example, each processing stage requests only a single unit from a single resource type for its execution).

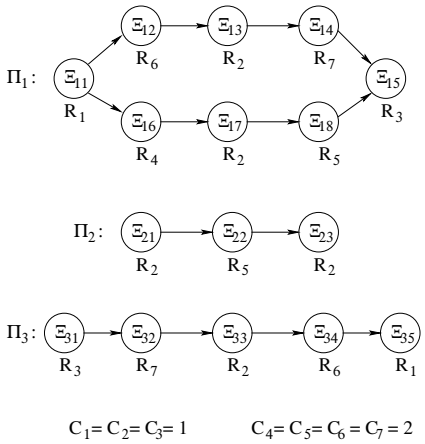


Fig. 5. The D/C-RAS considered in the example of Section IV.

The D/C-RAS depicted in Figure 5 constitutes a representation in the formalism introduced in Section II, of the resource allocation dynamics that take place in a flexible manufacturing system configuration originally introduced in [6]. Since its original introduction in [6], this RAS configuration has functioned as a "benchmark" case for many studies on the problem of deadlock avoidance (e.g., c.f. [11]). Hence, it is currently known that this RAS has a reachable state space S_r consisting of 26750 states, of which 21581 are safe and 5169 are unsafe. The perusal of the literature will also reveal that among the existing approaches to the LES synthesis for sequential RAS, the one that has managed to provide an LES that comes quite close to the maximally permissive while maintaining a compact representation for the resultant supervisor, is that presented in [15]. More specifically, this approach has managed to synthesize an LES that admits 21562 of the 21581 safe states, while imposing 19 linear inequalities on the system state (see also the discussion provided in [11]). On the other hand, the computational effort required by the methodology of [15] is quite intensive, since the aforementioned inequalities are developed through an iterative scheme that involves the regeneration of the reachability space and its re-classification into safe and unsafe sub-spaces after the introduction of each new inequality in the synthesized supervisor.

The method presented in this work manages to provide a compact implementation of the *maximally permissive* LES,

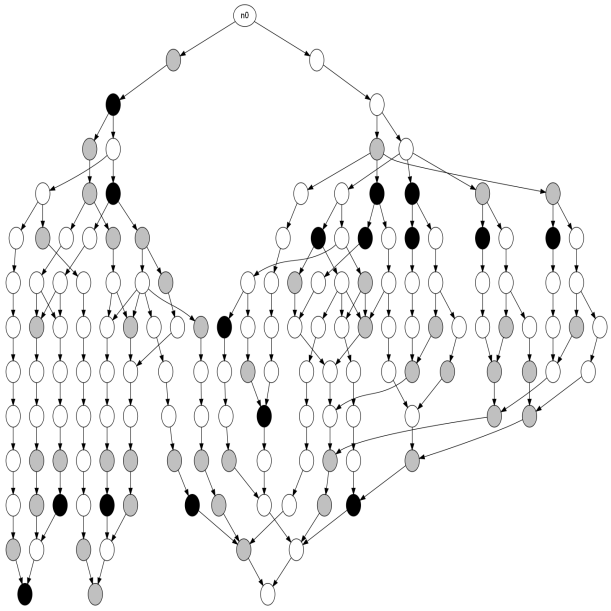


Fig. 6. The decision diagram storing vector set $P(\overline{S}^b_{r\bar{s}})$ for the example of Section IV. The white, gray and black nodes correspond to nodes having "content" values of 0, 1 and 2, respectively.

while generating the underlying reachability space only once. The proposed "thinning" of the set $S_{r\bar{s}}$ to its boundary elements reduces this set from 5169 to 4211 states, and the subsequent extraction of the minimal elements from this last set leaves us with only 34 states! In these 34 states, three components are identically zero,[4] which also leads to a dimensionality reduction of these vectors from 16 to 13 dimensions. The decision diagram that stores the resultant vector set $P(\overline{S}^b_{r\bar{s}})$ has 151 nodes and it is depicted in Figure 6.

We conclude this section by noticing that preliminary experimentation with other D/C-RAS configurations has provided results of a nature similar to those presented above. The proposed method has consistently been able to process state spaces of millions of states, eventually compressing the information necessary for the on-line implementation of the corresponding maximally permissive LES into less than a hundred vectors of small dimensionality.

V. Conclusions

This work has proposed a novel approach for the synthesis of maximally permissive LES for sequential RAS, and it has demonstrated the ability of this method to provide effectively computable and practically implementable solutions for RAS with (very) large state spaces. This capability arises from the development of an efficient customized algorithm for the enumeration of the underlying state space, and from the ability to encode the information that is necessary for on-line implementation of the maximally permissive LES in a very compact manner, by taking advantage of certain topological properties of the underlying state space and employing pertinent data structures. A possible extension of the presented work is through the development of alternative methods for effecting even more compact storage and efficient on-line processing of the information necessary for the recognition of boundary unsafe states, based on classification theory. Another extension is the modification of the method presented in this work in order to facilitate the storage and processing of state vectors that include symbolic information, like the state vectors that have been employed in the past for the compact encoding of the dynamics of AGV and monorail systems.[5] Both of these tasks are part of our ongoing investigations.

[4] In this example, the removed components correspond to the terminal processing stages of each process type Π_i, $i = 1, 2, 3$. The irrelevance of these three components in the determination of state (un-)safety is naturally interpreted by the fact the corresponding process instances will always be able to advance out of the system, and therefore, they will never get entangled in a deadlock.

[5] Certain components of these state vectors encode the vehicle direction of motion; c.f. [13], [14].

Acknowledgement

This work was partially supported by NSF grants CMMI-0619978 and CMMI-0928231.

References

[1] 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.

[2] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems (2nd ed.)*. Springer, NY,NY, 2008.

[3] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

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

[5] E. W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, Netherlands, 1965.

[6] J. Ezpeleta, J. M. Colom, and J. Martinez. A Petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE Trans. on R&A*, 11:173–184, 1995.

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

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

[9] J. W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.

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

[11] Z. Li, M. Zhou, and N. Wu. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. Systems, Man and Cybernetics – Part C: Applications and Reviews*, 38:173–188, 2008.

[12] S. Reveliotis. Algebraic deadlock avoidance policies for sequential resource allocation systems. In M. Lahmar, editor, *Facility Logistics: Approaches and Solutions to Next Generation Challenges*, pages 235–289. Auerbach Publications, 2007.

[13] S. A. Reveliotis. Conflict resolution in AGV systems. *IIE Trans.*, 32(7):647–659, 2000.

[14] S. A. Reveliotis. *Real-time Management of Resource Allocation Systems: A Discrete Event Systems Approach*. Springer, NY, NY, 2005.

[15] M. Uzam and M. Zhou. An iterative synthesis approach to Petri net-based deadlock prevention policy for flexible manufacturing systems. *IEEE Trans. on Systems, Man and Cybernetics – Part A: Systems and Humans*, 37:362–371, 2007.

[16] M. Zhou and M. P. Fanti (editors). *Deadlock Resolution in Computer-Integrated Systems*. Marcel Dekker, Inc., Singapore, 2004.