A practical approach for maximally permissive liveness-enforcing supervision of complex resource allocation systems

Ahmed Nazeem and Spyros Reveliotis

Abstract – **Past works towards the effective deployment of the maximally permissive liveness-enforcing supervisor (LES) for sequential resource allocation systems (RAS) have been stalled by (i) the NP-Hardness of the computation of this policy for the majority of the considered RAS classes, and (ii) the inability of the adopted more compact representations of the underlying RAS dynamics to provide an effective representation of the target policy for all RAS instantiations. This paper proposes a novel approach to the aforementioned problem, that can be perceived as a two-stage process: The first stage computes the maximally permissive LES by employing an automaton-based representation of the RAS behavior and techniques borrowed from the Ramadge & Wonham (R&W) Supervisory Control framework. The second stage seeks the development of a more compact representation for the dichotomy into admissible and inadmissible – or" safe" and "unsafe" – sub-spaces of the RAS state space, that is effected by the LES developed in the first stage. This compact representation is obtained by (i) taking advantage of certain properties of the underlying sub-spaces, and (ii) the employment of pertinent data structures. The resulting approach is also "complete", i.e., it will return an effectively implementable LES for any given RAS instantiation. Numerical experimentation demonstrates the efficacy of the approach and establishes its ability to support the deployment of maximally permissive LES for RAS with very large structure and state spaces.**

Note to Practitioners – **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.**

Index Terms– **Resource Allocation Systems, Liveness Enforcing Supervision, Discrete Event Systems, Nonblocking Supervisory Control, TRIE data structures**

I. Introduction

The problem of liveness-enforcing supervision for sequential resource allocation systems and its current state of art The problem of liveness-enforcing supervision (LES)[1] – 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, in the context of the computing technologies that were emerging at that time [17], [16], [9], [18]. 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 [12] 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[2] deadlock avoidance policy for any given RAS and the establishment of its NP-hardness for the majority of RAS behavior [1], [15].

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, tractability and 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 [24], [31], [5] as a powerful and rigorous base for modeling, analyzing and eventually control-

A. Nazeem and S. Reveliotis are with the School of Industrial & Systems Engineering, Georgia Institute of Technology, email: {anazeem@,spyros@isye.}gatech.edu

The authors were partially supported by NSF grants CMMI-0619978 and CMMI-0928231.

An abridged, preliminary version of this manuscript received the Thorlabs Best Student Paper Award at the 6th IEEE Conference on Automation Science and Engineering – CASE 2010.

[1] In the rest of this document, the abbreviation "LES" will stand either for "liveness-enforcing supervision" or for "liveness-enforcing supervisor", depending on the context.

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

ling the considered RAS dynamics. The combination of these two effects has led to a more profound understanding of the process of deadlock formation and of the RAS structural attributes that affect this process, under various DES-based representations. It has also given rise to a multitude of methodologies that can provide effective deadlock avoidance policies for many RAS classes.

Hence, it is currently known that the aforementioned deadlock avoidance problem can be characterized in the classical Ramadge & Wonham Supervisory Control (R&W SC) framework [24], [31], [5] in a straightforward manner, by (i) expressing the underlying resource allocation dynamics through a Finite State Automaton (FSA) and (ii) requesting the confinement of the RAS behavior to the subspace of this FSA that is defined by its maximal strongly connected component that contains the system state $\mathbf{s}_0$ where the RAS is idle and empty of any jobs. In fact, this characterization of the problem and its solution establishes also a notion of optimality for the considered problem, since the resulting policy prevents the formation of deadlock while retaining the maximum possible behavioral latitude for the underlying RAS.[3]

However, when it comes to the implementation of the control function, the standard approach of R&W SC theory essentially employs the FSA representation of the controlled system behavior under the target policy, and for many practical RAS configurations the explicit storage and on-line parsing of this information is of prohibitive computational cost due to the enormous size of the involved state spaces. Alternatively, one can consider a one-step lookahead control scheme that would seek the "real-time" – or the "on-line" – assessment of the co-accessibility of any given RAS state to the empty state $\mathbf{s}_0$, a property that is otherwise known as the state *"safety"*, in the relevant terminology; but this approach is also computationally challenged, since, for most RAS classes, the assessment of state safety is an NP-complete problem [1], [15], [20]. The research community has tried to circumvent the aforementioned computational challenges either (a) by compromising for sub-optimal – i.e., non-maximally permissive – solutions that are based on polynomially assessed properties of the relevant RAS states (e.g., [3], [30], [19], [13]), or (b) by adopting alternative, more compact representations of the considered RAS dynamics and hoping that the compactness of these alternative representations, combined with further structural properties and insights revealed by them, will also lead, at least in most practical cases, to fairly compact characterizations of the target policy and to more efficient approaches for its derivation.

A modeling framework that seems to hold particular promise for this second line of research, and therefore, has been explored more persistently in the past, is that of Petri nets (PN) [22]. In particular, the attribution of the non-liveness of the RAS-modeling PNs to the formation of some structural objects known as "empty – or, more generally, deadly marked – siphons", has led to the development of a multitude of efforts that seek to characterize the maximally permissive LES by imposing the minimum possible amount of control that will prevent the formation of such deadly marked siphons. However, a significant complication for these approaches arises from the fact that the maximally permissive LES might not admit a PN-based representation [14], [26], and therefore, the practical potential and applicability of these approaches is not fully explored and understood yet. Another prominent approach pursued within the context of the PN modeling framework is that of the "theory of regions" [2] and its derivatives. The key idea behind the theory of regions, as implemented in the considered problem context, is to first compute the maximally permissive LES using the standard R&W SC representations and methods mentioned in the previous paragraphs, and subsequently encode this policy to a PN model. This approach is also limited by the aforementioned potential inability to express the maximally permissive LES as a PN. Furthermore, even in its feasible cases, practical experience has shown that it is very demanding computationally and it results in PN representations of the maximally permissive LES that are much larger than the PN modeling the original RAS. The reader can find an extensive coverage of all these past developments in [33], [29], [25], [21], and the references cited therein.

The proposed approach The above discussion reveals that the primary challenges in deploying the maximally permissive LES for any given RAS configuration are (i) the NP-hardness of the computation of the target policy, for most practical RAS classes, and (iii) the inability of the PN modeling framework to guarantee an effective representation of the maximally permissive LES; we shall refer to this last limitation by saying that the PN modeling framework is an *"incomplete"* representational framework with respect to the maximally permissive LES. In this work we seek to address the limitations stated above, and to develop a methodology that will effectively compute and implement the maximally permissive LES for any given RAS configuration, by taking advantage of the following two observations:

Observation 1: While the aforementioned NP-hardness of the maximally permissive LES for the considered RAS is an undisputable result, it is also pertinent to clearly discriminate between (i) the computational complexity that concerns the *"off-line"* computational effort that is necessary for acquiring the target policy, and (ii) the computational complexity that concerns the *"on-line"* implementation of this policy. In general, one might be able to tolerate a more lengthy computation for the off-line part of the policy development. On the other hand, the computational

[3]We notice, for completeness, that most of the existing supervisory control theory for sequential RAS assumes that the underlying resource allocation function is fully observable and controllable. In particular, it is assumed that there is a central controller that monitors all the events taking place in the underlying RAS and authorizes actions like the initiation of new processes or the advancement of activated processes to their next processing stages, and the allocation of the necessary resources. Such an assumption is in line with the realities of the application domains for which this theory has been developed, e.g., the control of workflow in flexibly automated production systems, the traffic management of guidepath-based traffic systems (like AGV and monorail systems), and, more recently, the internet-based workflow management systems. For an overview of the existing results on the theory of RAS liveness-enforcing supervision under uncontrollable behavior the reader is referred to ([29], Chpt. 4). Also, some discussion on the impact of uncontrollable behavior upon the results presented in this work is provided in the concluding section.

budget for the on-line part of the policy implementation will tend to be quite stringent.

Observation 2: It might be possible to control the computational complexity that is involved with the on-line part of the policy implementation through the pertinent selection of the representation of the target policy. Furthermore, this representation must be chosen in a way that guarantees completeness.

As remarked in the previous paragraphs, the maximally permissive LES for the considered RAS classes 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 [31], [5]. 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, 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, and therefore, the increased requirements in terms of time and other computational resources can be 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 that defines the notion of state "safety" discussed above. *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. a monotonicity property possessed by the partition of the underlying state space to its safe and unsafe subspaces, that allows 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 a pertinent data structure 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.

We also notice, for completeness, than an approach similar to that described in the previous paragraphs has recently been pursued in [23], [7], [6]. All these three works use a set of linear inequalities in order to represent the target policy. Furthermore, computational experimentation presented in the aforementioned papers reveals that, when applicable, this representation can be quite compact, employing only a small number of linear inequalities even for RAS with very large state spaces. In addition, by expressing the target policy as a set of linear inequalities, the results of [23], [7], [6] can be (re-)cast in the PN modeling framework, using, for instance, the methodology presented in [14]. On the other hand, it is also well known that, in general, the safe and unsafe regions of the considered RAS classes might not be linearly separable, and therefore, the representation considered in [23], [7], [6] is not complete.[4] The approach proposed in this paper addresses this deficiency while maintaining a pretty efficient representation of the target policy.

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 monotonicity property of the sought partition 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, and it also reports our experiences with implementations involving larger and/or more complex RAS configurations. Section V concludes the paper by summarizing its contributions and outlining some further extensions of theoretical and practical interest. Finally, some more detailed technical discussion and arguments that support the results of Section III are provided in an appendix. Closing this introductory section, we also notice, for completeness, that a preliminary version of the results appearing in this manuscript was presented at the 6th IEEE Conference on Automation Science and Engineering (CASE 2010).

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

The considered RAS For the sake of simplicity and specificity, we present the main results of this paper in the context of the *Disjunctive / Conjunctive (D/C)* class of the RAS taxonomy presented in [29]. We notice, however, that the presented ideas and results are extensible to more complex classes of that taxonomy.

A *Disjunctive / Conjunctive Resource Allocation System (D/C-RAS)* is formally defined by a 4-tuple $\Phi = < \mathcal{R}, C, \mathcal{P}, D >$,[5] 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 defini-

[4] The inability of linear separation of the safe and the unsafe subspaces arises from the fact that the convex hull of the former might contain elements of the latter; cf. [26] for a concrete example.

[5] The complete definition of a RAS, according to [29], involves an additional component that characterizes the time-based – or *quantitative* – dynamics of the RAS, but this component is not relevant in the modeling and analysis to be pursued in the following developments, and therefore, it is omitted.

tion 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 will it 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 $R_i \in \mathcal{R}$ be over-allocated with respect to its capacity C_i at any point in time. Finally, for the purposes of the complexity analysis pursued in later parts of this work, we define the *size* $|\Phi|$ of RAS Φ by $|\Phi| \equiv |\mathcal{R}| + |\bigcup_{j=1}^{n} \mathcal{S}_j| + \sum_{i=1}^{m} C_i$.

The Deterministic FSA abstracting the D/C-RAS dynamics 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)* [5], $G(\Phi) =< S, E, f, \mathbf{s}_0, S_M >$, 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

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

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.[6]

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) $\bar{E} = \{e_{rp} : \exists j \in 1, \ldots, n \text{ s.t. } \Xi_p \text{ is a successor of } \Xi_r \text{ in graph } \mathcal{G}_j\}$, 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,

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

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

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

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:

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

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

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

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:

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

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

[6] Following standard practice in DES literature (cf., for instance, the relevant definition in page 8 of [5]), in the rest of this document we will frequently use the terms "space" and "subspace" in order to refer to set S and its various subsets considered in this work. We want to emphasize, however, that S and its various subsets involved in this work are *not* vector spaces in the sense that this term is used in linear algebra since they are not closed to vector addition and scalar multiplication.

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.

Some monotonicities observed by the state safety and unsafety concepts We conclude this section by discussing an additional property of the considered RAS that will prove very useful in the efficient implementation of the maximally permissive LES sought in this work. It should be clear from the above 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 of the above observations; these results are well known in the literature, and therefore, their formal proof is omitted.[7]

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:

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

Then,

1. $\mathbf{s} \in S_s \ \wedge \ \mathbf{s}' \leq \mathbf{s} \implies \mathbf{s}' \in S_s$
2. $\mathbf{s} \in S_{\bar{s}} \ \wedge \ \mathbf{s} \leq \mathbf{s}' \implies \mathbf{s}' \in S_{\bar{s}}$

□

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 a D/C-RAS $\Phi =< \mathcal{R}, C, \mathcal{P}, D >$,

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 $\bar{S}_{r\bar{s}}$.

[7]We notice, for completeness, that a formal proof for these results can be obtained, for instance, through the analytical characterization of state safety that is presented in [30], [27].

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 enumeration and storage of the set $S^b_{r\bar{s}}$, and
- a single-step lookahead scheme that, starting from the initial state $\mathbf{s}_0$, enables any transition $\mathbf{s}' = f(\mathbf{s}, e)$ that is system-enabled according to Equation 1, only if $\mathbf{s}' \notin S^b_{r\bar{s}}$.

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. Furthermore, as it was remarked in the introductory section, these steps correspond to standard operations encountered in the R&W SC framework [5], and therefore, in principle, they can be performed by any procedure that has been developed in support of these operations. However, in practice, the applicability of some of these procedures might be challenged by the fact that we want to target RAS configurations with very large state spaces.

Hence, 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. Therefore, 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. The introduction of the intermediate step of generating the state

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($Q! = \emptyset$) do
   a) $Node_{\mathbf{s}} \equiv (\mathbf{s}, L_{\mathbf{s}}) \leftarrow$ Pop Q;
   b) for each $\Xi_{pq}^r \in L_{\mathbf{s}}$ do
      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.

set S_v does not increase substantially the complexity of the involved computation, since, as will be revealed in the subsequent discussion, both sets S_v and S_r are of comparable sizes. On the other hand, performing the overall computation through the proposed sequence enables a more efficient handling and storage of the information characterizing the underlying FSA structure, and also the effective utilization of the auxiliary memory in case that the involved data structures grow so big that they cannot be accommodated in the core memory.

To describe the first of the two procedures listed in the previous paragraph, 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_{ij}^k 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) $\Xi_{i'j'}^{k'}$ 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 $\Xi_{i'j'}^{k'}$ 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_{ij}^k that can be added to state $\mathbf{s}$.
- The queue, Q, that contains all the state nodes that are waiting to be processed.

Processing a node $Node_{\mathbf{s}}$ implies (i) the generation of all the states $\mathbf{s}'$ that result from the addition to $\mathbf{s}$ of the loads included in the list L_s, (ii) the construction of the corresponding nodes $Node_{\mathbf{s}'}$, and (iii) the addition of these nodes to queue Q. The complete algorithm for generating the valid state space S_v is provided in Figure 1, and it is further discussed in Appendix-A. This appendix also establishes that, in most practical applications of this algorithm, its running time will be $\mathcal{O}(|S_v|)$, where $|S_v|$ denotes the cardinality of set S_v.[8]

The reader should also notice that whenever a state $\mathbf{s}$ is processed by the algorithm of Figure 1, it is guaranteed that it will not be considered again. This treatment is essentially different from the treatment applied to the generated states by the standard search-type of algorithms that are used for the direct enumeration of the reachable state space S_r. This last class of algorithms need to constantly check whether any newly reached state has been already generated, and this operation can be computationally demanding. It also necessitates a continuous access to the entire list of the generated states throughout the execution of those algorithms. On the other hand, by not revisiting a processed state, our algorithm does not need to keep such a state in the core memory, and therefore, processed states can simply be saved in a file on the hard disk.[9] 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 that the depicted procedure implements a "reaching scheme" that marks all the reachable states in the provided set S_v, while starting from the initial state $\mathbf{s}_0$. In this reaching scheme, all the information regarding the state reachability is processed and stored through the binary array *isReachable*(), that is indexed by the previously generated listing of S_v, and, therefore, the memory footprint of the presented procedure remains quite efficient. Furthermore, the systematic enumeration

[8]We should emphasize that, while the aforestated result indicates a linear complexity of the algorithm of Figure 1 with respect to $|S_v|$, S_v itself is, in general, exponentially sized with respect to the RAS size $|\Phi|$, and therefore, the algorithm of Figure 1 remains an "expensive" computation. On the other hand, the established complexity of $\mathcal{O}(|S_v|)$ implies that the algorithm of Figure 1 is an efficient algorithm for enumerating the set S_v (among all the algorithms that can support such an explicit enumeration). The practical implications of this efficiency are revealed in the computational results reported in Section IV.

[9]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.

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 lexicographically 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 ($reachableStack! = \emptyset$) do
   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 successor states, N_s;
   c) For each state $\mathbf{s}' \in N_s$ do
      if ($isReachable(\mathbf{s}') == 0$) then
      – 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.

of the state set S_v established by the algorithm of Figure 1, provides also a linear ordering for the elements of this set and an indexing scheme for direct accessing of the elements of the array $isReachable$, upon the provision of the corresponding state $\mathbf{s}$. Therefore, the practical (time) complexity of the algorithm depicted in Figure 2 is $\mathcal{O}(|S_r| \cdot log(|S_r|))$; this result is established in Appendix-B. Finally, combining the results regarding the computational complexities of the algorithms of Figures 1 and 2, and taking into consideration the additional fact that $|S_v| \approx \mathcal{O}(|S_r|)$, we can also infer that the practical complexity of the entire computation of the set S_r, according to the proposed scheme, 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). The relevant details are straightforward and they are omitted for space economy. We notice, however, 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 $S_{r\bar{s}}$ is also straightforward, and can be performed by an algorithm that, for every state $\mathbf{s} \in S_{r\bar{s}}$, 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 $S_{r\bar{s}}$. Such an algorithm 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 data set $S^b_{r\bar{s}}$, obtained through the computation discussed in the previous paragraphs, can be further compressed in a way that supports the informational needs of the look-ahead scheme described in the opening paragraph of this section, on the basis of the following two observations:

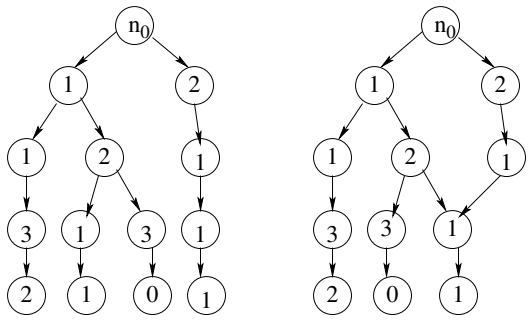
Observation 3: Proposition 1 and Definition 1, provided in Section II, imply that we can assess membership into $S^b_{r\bar{s}}$ for any given state $\mathbf{s} \in S_r$, by (i) explicitly storing only the subset of its minimal elements, $\overline{S}^b_{r\bar{s}}$, and (ii) checking whether there exists a state $\mathbf{s}' \in \overline{S}^b_{r\bar{s}}$ such that $\mathbf{s} \geq \mathbf{s}'$.

Observation 4: 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 3 above, and therefore, 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 3 enables the further "thinning" of the set of boundary reachable unsafe states $S^b_{r\bar{s}}$, by retaining only its minimal elements. Observation 4 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 simple and efficient computation.[10] On the other hand, as 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 (i) to more efficient storage of the set $P(\overline{S}^b_{r\bar{s}})$ and (ii) to more expedient algorithms for the on-line test suggested by Observation 3. The relevant gains are further demonstrated and assessed through the numerical experimentation reported in Section IV.

Storing 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 [8]. 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 depths from 1 to l that correspond to 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, ac-

[10] Once again, we forego the relevant details due to space economies.



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

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

cording 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 leaf nodes of the tree. In the resulting tree structure, every vector $\mathbf{v}_i,\ i=1,\ldots,k$, is represented by the path to one of the leaf nodes. We should also notice that the n-ary decision tree introduced in this paragraph, is a rather standard tool for efficient string storage and retrieval; typically, it is characterized as the *"trie"* (data) structure, and there are many variations of it and a considerable literature investigating their properties (cf. [4] and the references cited therein).[11]

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

Algorithmic construction of n-ary decision diagrams An algorithm for the systematic construction of an n-ary decision diagram for the compact representation of a set of k l-dimensional vectors $V=\{\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_k\}$ is provided in Figure 4. The data structure that is employed by this algorithm in order to represent a node of the derived decision diagram consists of (i) an integer field for storing the node "content", (ii) and two pointer lists that respectively provide the parents and the children of this node in the constructed diagram.[12] As can be seen in Figure 4, the

[11] "trie" is supposed to stand for "reTRIEval". We also note that the literature contains some additional attempts to use the BDD concept and its extensions/ variations not only for data storage, but for the representation and analysis of DES-related dynamics; cf., for instance, the works of [11], [32]. In this work, we place the emphasis primarily on the storage and the retrieval efficiencies that are supported by this type of data structures. The computation of the stored content itself is based on techniques that are motivated by and customized to particular attributes of the considered application, and as discussed in the previous parts of this manuscript, they lead to additional informational compression and storage economies.

[12] Since a decision diagram is an acyclic graph, both of these two concepts make sense for each node. On the other hand, since the

Input: A set of l-dimensional vectors $V=\{\mathbf{v}_1,\ldots,\mathbf{v}_k\}$.
Output: An n-ary decision diagram providing a compact representation for V.

/* Initial Decision Diagram */
1) Construct the two dummy nodes n_0 and n_f;
2) for $i := 1 : k$ do
   a) p:=new Node; $p.val := \mathbf{v}_i[1]$;
   b) Add$(p, n_0.children)$; Add$(n_0, p.parents)$;
   c) for $j := 2 : l$ do
      i) q:=new Node; $q.val := \mathbf{v}_i[j]$;
      ii) $p.children := q$; Add$(p, q.parents)$;
      iii) $p := q$;
   d) $p.children := n_f$; Add$(p, n_f.parents)$;

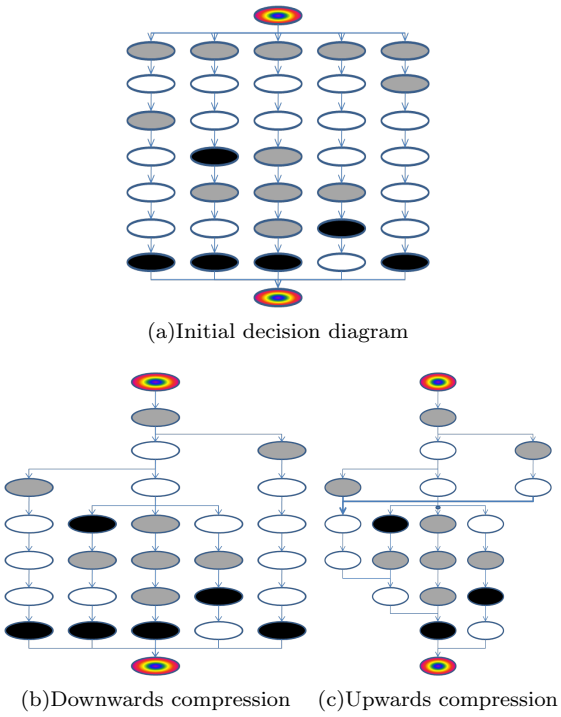
/* Downwards Compression */
3) $Q := NIL$; /* an empty queue of nodes */
4) Enqueue(n_0,Q);
5) while $(Q! = NIL)$ do
   a) $p :=$ Dequeue(Q);
   b) $q := p.children$;
   c) while $(q! = NIL)$ do
      i) $AddtoQ := FALSE$;
      ii) $r := q.next$;
      iii) while $(r! = NIL)$ do
         - if $(q.val == r.val)$ then
           - AddChildren(r, q);
           - Remove1$(r, p.children)$;
           - $AddtoQ := TRUE$;
         - $r++$;
      iv) if $(AddtoQ)$ then
          Enqueue(q,Q);
      v) $q++$;

/* Upwards Compression */
6) $Q := NIL$; /* an empty queue of nodes */
7) Enqueue(n_f,Q);
8) while $(Q! = NIL)$ do
   a) p := Dequeue(Q);
   b) $q := p.parents$;
   c) while $(q! = NIL)$ do
      i) $AddtoQ := FALSE$;
      ii) $r := q.next$;
      iii) while $(r! = NIL)$ do
         - if (Equivalent(q, r)) then
           - AddParents(r, q);
           - Remove2(r);
           - $AddtoQ := TRUE$;
         - $r++$;
      iv) if $(AddtoQ)$ then
          Enqueue(q,Q);
      v) $q++$;

Fig. 4. The algorithm constructing the n-ary decision diagram that provides a compact representation for a given set of vectors V.

TABLE I
THE SET OF VECTORS USED IN THE EXAMPLE OF FIGURE 5

$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$	$\mathbf{v}_4$	$\mathbf{v}_5$
1	1	1	1	1
0	0	0	0	1
1	0	0	0	0
0	2	1	0	0
0	1	1	1	0
0	0	1	2	0
2	2	2	0	2



(a)Initial decision diagram

(b)Downwards compression (c)Upwards compression

Fig. 5. The decision diagrams produced at the end of each of the three major phases of the algorithm of Figure 4 when applied to the vector set of Table I. The node content is encoded as follows: white corresponds to 0, grey to 1, and black to 2. The dummy nodes n_0 and n_f are depicted as multi-colored nodes.

overall computation of the considered algorithm is organized in three major phases: The first phase, consisting of Steps 1–2 in the algorithm, constructs an acyclic digraph with a single source node n_0 and a single sink node n_f, and with its internal nodes encoding the vectors $\mathbf{v}_i \in V$ as distinct, non-overlapping paths from n_0 to n_f. Clearly, the digraph produced at this phase is layered, with l internal layers. Furthermore, each of these internal layers contains $k(\equiv |V|)$ distinct nodes, and each such node has a single parent and a single child in the digraph. Finally, each such internal node is labeled by the numerical value of the component that it represents in the considered vector set V; as already mentioned, this label defines the "content" of the node. The second phase of the algorithm in Figure 4, consisting of Steps 3–5, is an operation of "downwards compression". This part of the computation essentially seeks to identify all the nodes in the diagram generated by phase I that correspond to identical prefixes for the vectors passing through them, and merge them in a single node. Finally, the third phase of the algorithm in Figure 4, consisting of Steps 6–8, is an operation of "upwards compression". This part of the computation seeks to identify nodes in the diagram generated by Phase II that correspond to the same sets of possible suffixes for the state vectors passing through them, and merge them.

A more expansive discussion on the computational logic underlying the various phases of the algorithm of Figure 4, and on the correctness of the algorithm, can be found in Appendix-C. This appendix also establishes that the computational complexity of this algorithm is $O(lk^2c)$, where l, k are respectively the dimensionality and the cardinality of the stored vector set V, while c denotes the maximum number of children in any node of the resulting diagram.

Figure 5 exemplifies the algorithm of Figure 4 by depicting the decision diagrams that result after the execution of each of its three phases on the vector set of Table I. We also notice that while the dummy terminal node n_f is useful for the algorithmic construction of the decision diagram according to the logic described above, it does not play any substantial role during the on-line use of this diagram, and therefore, eventually it can be removed; this is the version of decision diagrams that we shall consider in the sequel.

On-line implementation of the maximally permissive LES through n-ary decision diagrams The employment of the n-ary decision diagram in the context of the single-step lookahead control scheme described at the beginning of this section, is supported through the algorithm provided in Figure 6. More specifically, the algorithm of Figure 6 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'. 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.

Further considerations Closing this discussion on n-ary decision diagrams and their role in the supervisory control problem addressed in this work, we also notice that, in principle, it is possible to enhance the compactness of the corresponding representation of any given vector set V, by permuting the components of its vector elements. However, the identification of an optimal permutation – i.e., a permutation that leads to a decision diagram with the smallest possible number of internal nodes – is an NP-complete problem [10]. In the light of this result, one can seek the development of heuristics that adapt, to the considered application context, ideas and techniques borrowed from the area of combinatorial optimization. This effort should also exploit and integrate further insights regarding

derived structure is a diagram and not a tree, each node can have more than one parent in it.

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

Fig. 6. 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 $\mathbf{v} \leq \mathbf{v}'$.

the dynamics that arise in the merging process of the algorithm in Figure 4. The development of such heuristics and a detailed investigation of their potential is part of our future research.

IV. Experimental Results

In this section we report a number of computational experiments that demonstrate the efficacy of the proposed approach and assess its applicability in the context of the RAS class considered in this work. We begin the presentation of these results by considering the application of our methodology to the synthesis of the maximally permissive LES for the D/C-RAS configuration defined in Table II. The considered D/C-RAS has seven resource types, $\{R_1, \ldots, R_7\}$, each with capacity $C_i = 3$. It also has four process types, $\{\Pi_1, \Pi_2, \Pi_3, \Pi_4\}$, with each process type defined by the linear route provided in Table II. In particular, each of the depicted routes constitutes a sequence of processing stages with each stage engaging a single resource type at the amount indicated by the corresponding coefficient; for instance, the first processing stage of the first process type engages a single unit of resource R_2, the second stage of the same type engages two units of resource R_7, etc. Hence, the depicted D/C-RAS has 20 distinct processing stages, in total, and this number defines the dimensionality of its state vector according to the definitions provided in Section II.

The application of the LES design methodology of Section III revealed that the considered RAS has a reachable state space of 351,604 states, with 135,414 of them being safe states and the remaining 216,190 being unsafe states. Among the boundary unsafe states, 49 of them are minimal. In these 49 minimal unsafe states, 6 out of the 20 state components are always equal to zero, and therefore, the dimensions that need to be explicitly stored are only the remaining 14. The corresponding decision diagram has 180 nodes and it is depicted in Figure 7. The storage of this diagram in a core computer memory requires 399 integer locations, 180 of these locations for storing the nodal content itself, and the remaining 219 locations for storing the pointers that correspond to the arcs of this diagram. On the other hand, the storage of the 49 14-dim vectors of $P(\overline{S}^b_{r\bar{s}})$ in a 2-dim array involves $49 \times 14 = 686$ integer entries. Therefore, the alternative storage mechanism of Figure 7 utilizes only a little more than 58% of the storage space utilized by the array-based storage of $P(\overline{S}^b_{r\bar{s}})$. Finally, we also notice that through a methodology similar to that presented in [23], it can be established that the maximally permissive LES for the considered D/C-RAS cannot be expressed as a set of linear inequalities, and therefore, the construction of the maximally permissive LES for this D/C-RAS configuration is not amenable to the Petri net-based methodologies discussed in the introductory section.

Table III reports the results that were obtained from the application of the proposed method for the deployment of the maximally permissive LES on 10 additional D/C-RAS configurations.[13] More specifically, for each of these configurations, Table III reports: (i) the total number of processing stages (and therefore, the dimensionality of the corresponding state space); (ii) the cardinality of the reachable safe subspace S_{rs}; (iii) the cardinality of the reachable unsafe subspace $S_{r\bar{s}}$; (iv) the cardinality of the set of minimal boundary unsafe states $\overline{S}^b_{r\bar{s}}$; (v) the dimensionality of the projected subspace $P(\overline{S}^b_{r\bar{s}})$ that is obtained after removing the dimensions that are identically equal to zero in $\overline{S}^b_{r\bar{s}}$; (vi) the number of integer entries that would be necessary for the storage of the elements of $P(\overline{S}^b_{r\bar{s}})$ in a 2-dim array – the entries of this column are obtained by multiplying the entries of the previous two columns in the table; (vii) the number of the nodes employed by the corresponding decision diagram; (viii) the total storage capacity, in terms of integer entries, that is required for the storage of the entire structure of the corresponding decision diagram; (ix) the storage compression attained by the n-ary decision diagram – the entries of this column are obtained by taking the ratio of the entries in columns (viii) and (vi) of the depicted table, i.e., the storage compression is measured as the ratio of the storage requirements posed by the n-ary decision diagram to the storage requirements that would be necessary in case of a 2-dim array-based representation of the vector set $P(\overline{S}^b_{r\bar{s}})$; (x) the total computational time, in seconds, that is required for the construction of the decision diagram for each of the listed cases. Figure 8 also depicts the attained storage compression for an even broader data set of 42 D/C-RAS configurations (this data set includes the configurations involved in Table III).

The results of Table III and Figure 8 corroborate that the proposed method is effectively applicable to D/C-RAS with very large state spaces and it can lead to a very compact representation of the corresponding maximally permissive LES. The informational compression and the corresponding storage gains that are attained by the application of the n-ary decision diagrams are substantial - in most of the reported cases the attained compression is more than 80%. Furthermore, the plot of Figure 8 reveals that the attained storage efficiencies become more prominent as the storage

[13] Space limitations do not allow a detailed description of these configurations. This information can be obtained by contacting the authors.

TABLE II
THE D/C-RAS CONSIDERED IN THE EXAMPLE OF SECTION IV

Resource Types:	$\{R_1, R_2, \ldots, R_7\}$
Resource Capacities:	$C_i = 3,\ \forall i \in \{1, 2, \ldots, 7\}$
Process Type 1:	$R_2 \rightarrow 2R_7 \rightarrow 2R_4 \rightarrow 2R_3$
Process Type 2:	$3R_4 \rightarrow 3R_2 \rightarrow 2R_4 \rightarrow 3R_7 \rightarrow 2R_3 \rightarrow R_5 \rightarrow R_4$
Process Type 3:	$R_1 \rightarrow 2R_5 \rightarrow 3R_1 \rightarrow R_7 \rightarrow 2R_6$
Process Type 4:	$2R_2 \rightarrow R_3 \rightarrow R_4 \rightarrow 3R_2$

requirements for the more conventional, array-based representation of the set $P(\overline{S}^b_{r\bar{s}})$ become larger. Finally, it should also be noticed that none of the RAS configurations employed in the presented experiments accepts a characterization of its maximally permissive LES as a set of linear inequalities, and therefore, they are not amenable to the Petri net-based methodologies discussed in the introductory section; this effect was verified through techniques similar to those reported in [23], and it manifests another powerful attribute of the considered approach with respect to other approaches reported in the current literature.

Closing this section on our computational experiments, we also report that these experiments were performed on a 2.66 GHz quad-core Intel Xeon 5430 processor with 6 MB of cache memory and 32 GB RAM; however, each job ran on a single core. The algorithms were encoded in C++, and they were compiled and linked by the GNU g++ compiler under Unix. In all cases, the overall time necessary for the computation of the maximally permissive LES and its representation through the corresponding n-ary decision diagram did not exceed the 600 sec; in fact, as revealed by the computational times reported in Table III, in the majority of the considered cases the required computational time was less than 60 sec.[14]

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. In addition, the proposed method is complete, i.e., it is applicable to any RAS configuration from the considered RAS class. These capabilities arise 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.

A future extension of the presented work concerns the further compression of the data structure that stores the projections of the minimal boundary unsafe states that are employed by our approach. As discussed in Section III, this can be achieved through a pertinent re-arrangement –

[14]We also note that a more detailed breakdown of these computational times reveals that most of the computational effort is expended on (i) the extraction of the set of boundary unsafe states, $S^b_{r\bar{s}}$, from the set of reachable unsafe states, $S_{r\bar{s}}$, and (ii) the identification of the minimal elements in $S^b_{r\bar{s}}$.

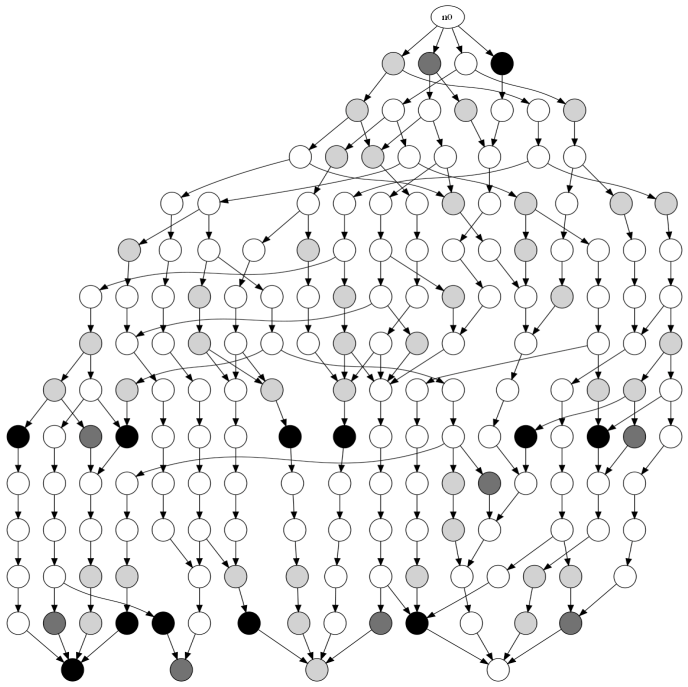


Fig. 7. The acyclic digraph storing the vector set $P(\overline{S}^b_{r\bar{s}})$ for the example D/C-RAS of Table II. The white, light gray, dark gray and black nodes correspond to nodes having respective "content" values of 0, 1, 2 and 3.

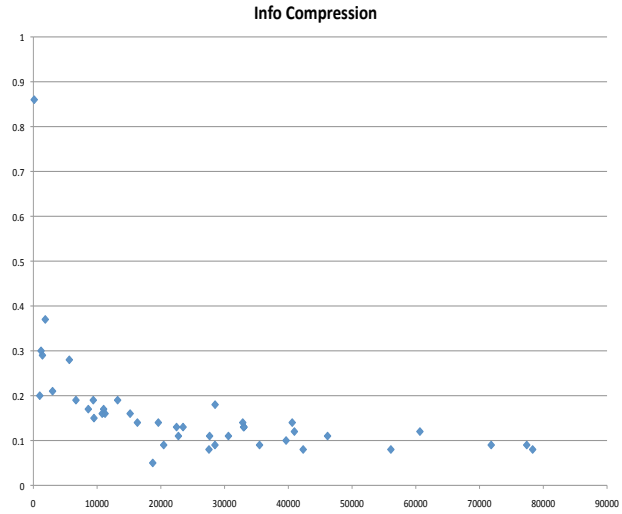


Fig. 8. The information compression in the storage of the set $P(\overline{S}^b_{r\bar{s}})$ attained through the employment of the n-ary decision diagrams proposed in this work. The x-axis reports, for each considered D/C-RAS configuration, the product of the cardinality of $P(\overline{S}^b_{r\bar{s}})$ with the dimensionality of its elements; this product should be perceived as the total number of integer entries that are necessary to store $P(\overline{S}^b_{r\bar{s}})$ in a 2-dim array. The y-axis reports the storage capacity employed by the corresponding n-ary decision diagram as a percentage of the storage needs indicated in the x-axis.

TABLE III
A SAMPLE DATA SET OBTAINED FROM THE PERFORMED EXPERIMENTS

# proc. stages	# safe states	# unsafe states	# min unsafe states	reduced dim	# tab. entries	# dec. nodes	dec. diag. stor. reqs	compression ratio	comp. time (sec)
24	115766	28510	12	11	132	52	114	.86	1
32	163439	192381	131	23	3013	286	643	.21	7
40	42571	69016	188	30	5640	716	1580	.28	1
48	80343	691959	241	39	9399	818	1788	.19	24
52	1622861	2600349	257	37	9509	678	1473	.15	150
64	118470	1253425	612	54	33048	1895	4178	.13	55
70	10956	289962	338	58	19604	1245	2749	.14	12
77	15763	364985	1119	70	78330	2951	6472	.08	17
88	53080	1410311	1046	74	77404	3247	7069	.09	81
99	8425	413822	845	85	71825	3039	6682	.09	20

more formally, a permutation – of the coordinates of the RAS state vector. The resulting optimization problem is of a combinatorial nature and it is NP-Hard. Hence, our future work will seek to provide heuristics for the effective selection of an optimized permutation. An additional line of research will seek the modification of the method presented in this work in order to facilitate the storage and processing of state vectors that include symbolic information; this is, for instance, the case with the state vectors that have been employed in the past for the compact encoding of the dynamics of AGV and monorail systems.[15] In a similar vein, it is interesting to study how the presence of uncontrollable elements in the plant behavior will impact the implementational details of the presented method and the resulting (in-)efficiencies. In general, the presence of uncontrollability in the RAS behavior might cancel the monotonicity of the safety concept discussed in Section II, and therefore, it will have a negative impact on the computational and representational efficiencies established in this paper. The extent that the presented approach remains tractable in the absence of these efficiencies, as well as the identification of structural and behavioral conditions that will (partially) preserve them need to be systematically considered. Finally, another research line will seek the development of more analytical representations for the compact encoding of the dichotomy of the RAS state space in safe and unsafe states. A first set of results along this line, that pertain to RAS sub-classes with binary state spaces, is presented in [23].

APPENDIX

A. A detailed description and complexity analysis of the algorithm of Figure 1

Some remarks that can help the further understanding of the algorithm in Figure 1 are as follows: Since, according to the definitions provided in Section III, every processing stage Ξ_{ij} can have up to K_{ij} active jobs, the list $L_{\mathbf{s}_0}$, constructed in step 3, contains all the possible states that can be obtained from state $\mathbf{s}_0$ by activating only one processing stage, Ξ_{ij}, to some number of jobs in the interval $[1, K_{ij}]$. On the other hand, the while loop in Step 5, that processes the state nodes that are stored in queue Q, is broken down in the following steps: Step 5a extracts a node $(\mathbf{s}, L_{\mathbf{s}})$ stored in queue Q for further processing. For every element Ξ^r_{pq} in list $L_{\mathbf{s}}$, Step 5b(i) constructs a new state $\mathbf{s}^*$ from Ξ^r_{pq} and $\mathbf{s}$ such that $\mathbf{s}^* = \{\forall(i,j) \neq (p,q) : \mathbf{s}^*_{ij} = \mathbf{s}_{ij}$ 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^k_{ij} that satisfy the following conditions: First, load Ξ^k_{ij} can be added to state $\mathbf{s}^*$, i.e., (a) $\mathbf{s}^*_{ij} = 0$ and (b) adding the k jobs at stage Ξ_{ij} to all the active jobs at state $\mathbf{s}^*$ does 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 that 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, typically, $(\xi + \sum_{i=1}^{n}\sum_{j=1}^{l_i} K_{ij}) \ll |S_v|$, we can say that the practical running time of the algorithm is $\mathcal{O}(|S_v|)$.

B. Complexity analysis of the algorithm of Figure 2

The (time) complexity of the algorithm 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 the notation introduced in Section II, and

[15] Certain components of these state vectors encode the vehicle direction of motion; c.f. [28], [29].

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 $\mathcal{O}(\bar{t} \cdot |S_r| \cdot log(|S_r|))$. Since, typically $\bar{t} \ll |S_r|$, we can also say that the practical complexity of the considered algorithm is $\mathcal{O}(|S_r| \cdot log(|S_r|))$.

C. A detailed description and complexity analysis of the algorithm of Figure 4

As remarked in Section III, the algorithm of Figure 4 is a three-phase algorithm, where: (i) the first phase, consisting of Steps 1–2 in the algorithm, constructs an acyclic digraph with a single source node n_0 and a single sink node n_f, and with its internal nodes encoding the vectors $\mathbf{v}_i \in V$ as distinct, non-overlapping paths from n_0 to n_f; (ii) the second phase, consisting of Steps 3–5, is an operation of "downwards compression" that identifies all the nodes in the diagram generated by phase I corresponding to identical prefixes for the vectors passing through them, and merges these nodes in a single node; and (iii) the third phase, consisting of Steps 6–8, is an operation of "upwards compression" that identifies nodes in the diagram generated by phase II that correspond to the same sets of possible suffixes for the vectors passing through them, and merges them. Next we provide a more detailed description of the operations performed by the second and the third phase of the considered algorithm. This discussion will also allow us to ague the technical correctness of the algorithm. We close with a complexity analysis of the algorithm.

In more technical terms, the downwards compression, that is performed in the second phase of the considered algorithm, is attained as follows: The digraph constructed in phase I is traversed on a layer-by-layer basis, starting from the source node n_0, and at each visited node, all of its children with equal labels are merged to a single node. In particular, the function AddChildren(r, q) is meant to add the child of node r to the children of node q, and to redefine q as the parent of this added child.[16] Subsequently, the function Remove1$(r, p.children)$ removes node r from the children of node p and releases the corresponding memory. This traversal continues until a layer is reached where no node merging occurs; at that point, Q will become empty and the algorithm will exit the loop of Step 5, which is the main loop for this phase of the computation. The reader should notice that as a result of the performed nodal merging, some of the internal nodes will have more than one children at the end of phase II, but this merging preserves the single-parent property for the internal nodes. Hence, it is clear that each cluster of merged nodes corresponds to the same vector prefix, as initially stated, and the performed merging does not distort the "information content" of the graph; i.e., the vector set that is defined by all the paths from the source to the sink node remains equal to V.

On the other hand, during its third phase, the considered algorithm traverses again the digraph obtained in phase II on a layer-by-layer basis, but this time, the traversal starts from the sink node n_f. At each visited node, the algorithm checks the list of its parents, to see whether there are equivalent nodes. Two nodes q and r are characterized as "equivalent" if they have the same content and the same lists of children; the relevant testing is performed by function Equivalent(q, r) in the algorithm. A simple induction on the number of traversed layers can show that node equivalence essentially implies the same sets of possible suffixes for all the paths passing through them, and therefore, the nodal merging that is performed by the algorithm is consistent with the phase objective that is stated at the opening paragraph of this sub-section. The merging of the equivalent nodes q and r is performed by the following two functions: Function AddParents(r, q) adds the parent of node r to the parents of node q and also it updates the list of children of this added node by replacing node r in it by node q. Subsequently, the function Remove2(r) removes node r from the parent lists of all its children and releases the memory allocated for the storage of this node.[17] Similarly to the second phase of the algorithm, the graph traversal described above terminates when a layer is encountered where no node merging takes place. Clearly, the information content of the digraph is also preserved under this new phase of node merging. Hence, the algorithm is correct.

An analysis of the computational complexity of the algorithm of Figure 4 can be performed as follows: First, we notice that the first phase of the algorithm runs in time $\mathcal{O}(kl)$ and results in a diagram with $\mathcal{O}(kl)$ nodes.

To analyze the computational complexity of the second phase of the algorithm, for every layer $j = 1, \ldots, l$, let us consider the equivalence relation $\mathcal{V}_j$ that is defined on the vector set V by the equality of the vector prefixes across the layers 1 to $j-1$; i.e., the equivalence classes E_j^i of $\mathcal{V}_j$ consist of all the vectors in V that have their first $j-1$ coordinates equal. For every such equivalence class E_j^i of the partitioning $\mathcal{V}_j$ of V, let us also consider the set consisting of the distinct values of the j-th coordinate for the vectors in E_j^i, and let c_j^i denote the cardinality of that set. Finally, set $c \equiv \max_{j,i}\{c_j^i\}$. From a more intuitive standpoint, c defines the maximum possible number of siblings (i.e., children of the same parent) that can exist in the diagram constructed by phase II of the algorithm. Then, some additional remarks that are important for characterizing the computational complexity of phase II of the algorithm are the following: During the processing of any nodal layer $j \in \{1, \ldots, l\}$ by this phase, the corresponding nodes are partitioned into sibling classes, and from the previous discussion, it follows that the node contents of every such class can take at most c distinct values. This last remark, when combined with the fact that every merged node is removed from the children list of its parent, further implies that the while loop of Step 5c will be executed at most c times for every class. Also, every such execution will go through the while loop of Step 5c(iii) a

[16] The fact that node r has only one child is a consequence of the structure of the diagram that is returned by Phase I of the algorithm and the graph traversal pattern that is applied in Phase II.

[17] Note that since, by the notion of equivalence, q and r have the same children, there is no need to add q in the parent lists of the children of r.

number of times that is no greater than the number of siblings in that class. Hence, the number of executions of the while loop of Step 5c(iii) across all the sibling classes of any single layer of the considered diagram will be $\mathcal{O}(kc)$; and when considered across all layers, this number will be $\mathcal{O}(lkc)$. To obtain the complete characterization of the computational complexity of this phase, it remains to characterize the computational complexity of the operations involved in the merging of the nodes q and r, i.e., the computational complexity of the functions AddChildren(r, q) and Remove1$(r, p.children)$. Obviously, under the pointer-based implementation of the lists employed by the algorithm, the complexity of Remove1$(r, p.children)$ is $\mathcal{O}(1)$. The complexity of AddChildren(r, q) is also $\mathcal{O}(1)$, since as already remarked, node r will have only a single child and this child does not belong to the current children of node q. Hence, the overall complexity of phase II remains $\mathcal{O}(lkc)$.

The computational complexity of the third phase of the algorithm can be obtained through analysis similar to that performed for the complexity of phase II. Some key observations that lead to the characterization of this complexity are as follows: During the execution of this phase, for every nodal pair (r, q) appearing in the function AddParents(r, q), node r has a single parent that is different from the parent of node q and this parent node of r has at most c children (since nodes q and r are equivalent and the diagram obtained by Phase II has a trie structure). Hence, the complexity of function AddParents(r, q) is $\mathcal{O}(c)$. Obviously, the complexity of function Remove2(r) is also $\mathcal{O}(c)$ (since the number of children of a node does not increase in this phase). Finally, the evaluation of the function Equivalence(q, r) is also $\mathcal{O}(c)$ (since equivalent nodes have the same nodal content and the same children lists). Hence, the entire computational complexity of an iteration of the while loop of Step 8c(iii) is $\mathcal{O}(c)$. An analysis similar to that performed for the complexity of phase II can establish that the aforementioned loop will be executed $\mathcal{O}(lk^2)$ times by the algorithm. Hence, the entire computational complexity of this phase is $\mathcal{O}(lk^2c)$. Since this is the phase with the highest computational complexity, its complexity defines also the computational complexity of the entire algorithm.

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] E. Badouel and P. Darondeau. Theory of regions. In W. Reisig and G. Rozenberg, editors, *LNCS 1491 – Advances in Petri Nets: Basic Models*, pages 529–586. Springer-Verlag, 1998.

[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] P. Brass. *Advanced Data Structures*. Cambridge University Press, NY,NY, 2008.

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

[6] Y. F. Chen and Z. W. Li. Design of a maximally permissive liveness-enforcing supervisor with compressed supervisory structure for flexible manufacturing systems. *Automatica*, 47:1028–1034, 2011.

[7] Y. F. Chen, Z. W. Li, M. Khalgui, and O. Moshabi. Design of maximally permissive liveness-enforcing petri net supervisor for flexible manufacturing systems. *IEEE Trans. on Automation Science and Engineering*, 8:374–393, 2011.

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

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

[10] P. Commer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24:428–440, 1977.

[11] J. M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. A. Wacrenier. Data decision diagrams for Petri net analysis. In *Applications and Theory of Petri Nets 2002*, pages 101–120, 2002.

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

[13] J. Ezpeleta, F. Tricas, F. Garcia-Valles, and J. M. Colom. A Banker's solution for deadlock avoidance in FMS with flexible routing and multi-resource states. *IEEE Trans. on R&A*, 18:621–625, 2002.

[14] A. Giua, F. DiCesare, and M. Silva. Generalized mutual exclusion constraints on nets with uncontrollable transitions. In *Proceedings of the 1992 IEEE Intl. Conference on Systems, Man and Cybernetics*, pages 974–979. IEEE, 1992.

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

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

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

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

[19] M. Lawley, S. Reveliotis, and P. Ferreira. A correct and scalable deadlock avoidance policy for flexible manufacturing systems. *IEEE Trans. on Robotics & Automation*, 14:796–809, 1998.

[20] M. A. Lawley and S. A. Reveliotis. Deadlock avoidance for sequential resource allocation systems: hard and easy cases. *Intl. Jrnl of FMS*, 13:385–404, 2001.

[21] 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.

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

[23] A. Nazeem, S. Reveliotis, Y. Wang, and S. Lafortune. Designing compact and maximally permissive deadlock avoidance policies for complex resource allocation systems through classification theory: the linear case. *IEEE Trans. on Automatic Control*, (to appear).

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

[25] 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.

[26] S. Reveliotis, E. Roszkowska, and J. Y. Choi. Generalized algebraic deadlock avoidance policies for sequential resource allocation systems. *IEEE Trans. on Automatic Control*, 52:2345–2350, 2007.

[27] S. A. Reveliotis. *Structural Analysis & Control of Flexible Manufacturing Systems with a Performance Perspective*. PhD thesis, University of Illinois, Urbana, IL, 1996.

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

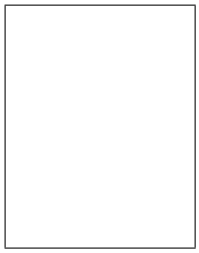
[29] S. A. Reveliotis. *Real-time Management of Resource Allocation Systems: A Discrete Event Systems Approach*. Springer, NY, NY, 2005.

[30] S. A. Reveliotis and P. M. Ferreira. Deadlock avoidance policies for automated manufacturing cells. *IEEE Trans. on Robotics & Automation*, 12:845–857, 1996.

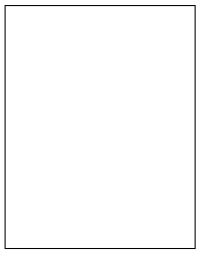
[31] W. M. Wonham. Supervisory control of discrete event systems. Technical Report ECE 1636F / 1637S 2009-10, Electrical & Computer Eng., University of Toronto, 2010.

[32] Z. H. Zhang and W. M. Wonham. STCT: An efficient algorithm for supervisory control design. In B. Caillaud, P. Darondeau, L. Lavango, and X. Xie, editors, *Synthesis and Control of Discrete Event Systems*, pages 77–100. Kluwer, 2002.

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



Ahmed Nazeem holds an M.Sc. degree in Industrial Engineering from Georgia Institute of Technology, where he is currently pursuing his Ph.D. in Industrial Engineering. His research interest is in the area of discrete event systems theory and applications. In 2010, Ahmed was the recipient of the Thorlabs Best Student Paper Award for the IEEE Conference on Automation Science and Engineering.



Spyros Reveliotis is a Professor in the School of Industrial & Systems Engineering, at the Georgia Institute of Technology. He holds a Diploma in Electrical Engineering from the National Technical University of Athens, Greece, an M.Sc. degree in Computer Systems Engineering from Northeastern University, Boston, and a Ph.D. degree in Industrial Engineering from the University of Illinois at Urbana-Champaign. Dr. Reveliotis' research interests are in the area of Discrete Event Systems theory and its applications. He is a Senior member of IEEE and a member of INFORMS.

Dr. Reveliotis currently serves as Associate Editor for the IEEE Trans. on Automatic Control, as Department Editor for IIE Transactions, and he is also a Senior Editor in the Conference Editorial Board for the IEEE Intl. Conference on Robotics & Automation. He has also been an Associate Editor for IEEE Trans. on Robotics and Automation and for IEEE Trans. on Automation Science and Engineering, while in 2009 he was the Program Chair for the IEEE Conference on Automation Science and Engineering. Dr. Reveliotis has been the recipient of a number of awards, including the 1998 IEEE Intl. Conf. on Robotics & Automation Kayamori Best Paper Award.