Efficient enumeration of minimal unsafe states in complex resource allocation systems

Ahmed Nazeem and Spyros Reveliotis

Abstract – **An earlier work of ours has proposed a novel approach for the deployment of the maximally permissive deadlock avoidance policy for complex resource allocation systems (RAS), that is based on the identification and the efficient storage of a critical subset of states of the underlying RAS state space; the availability of this information enables an expedient one-step-lookahead scheme for the identification and blockage of transitions that will take the system behavior outside its safe region. This paper complements the aforementioned results by introducing a novel algorithm that provides those critical states while avoiding the complete enumeration of the RAS state space.**

Note to Practitioners – **The establishment of deadlock-free resource allocation is a prominent problem that underlies the behavioral dynamics of many contemporary applications of automation science and engineering. Ideally, one would like to establish the aforementioned deadlock freedom while imposing the minimum possible restriction to the underlying resource allocation function. The results presented in this paper facilitate the deployment of such minimally restrictive solutions to the deadlock avoidance problem for applications of practical size and behavioral complexity.**

Keywords – **Deadlock Avoidance, Sequential Resource Allocation Systems, Minimal Deadlocks, Minimal Unsafe States**

I. Introduction

The problem of deadlock avoidance in sequential, complex resource allocation systems is well-established in the Discrete Event Systems (DES) literature. Some quite comprehensive expositions of the past results on this problem can be found in [15], [17], [6]. The work presented in this paper is part of an ongoing, more recent endeavor to deploy effective and computationally efficient realizations of the maximally permissive deadlock avoidance policy (DAP) for various classes of resource allocation systems (RAS), in spite of the fact that the computation of this policy can be an NP-Hard task in the context of the considered RAS classes [15]. Specific examples of this endeavor are the works presented in [3], [5], [10], [8], [9], [13]. The key idea underlying all those approaches is (i) to distinguish the computation that is involved in the deployment of the target policy into an "off-line" part, that concerns all the preliminary steps that are necessary for the policy specification, and the "on-line" part that involves the computation that takes place during the real-time implementation of the policy, and (ii) to isolate the high computational complexity in the off-line part of the performed computation. More specifically, in the aforementioned approaches, the on-line part of the policy implementation is streamlined through the selection of a pertinent, efficient representation for the policy-defining logic. In general, these efficient representations can be constructed by treating the considered DAP as a "classifier" that dichotomizes the RAS state space into the admissible and inadmissible subspaces defined by the policy; in the relevant terminology, these two subspaces are respectively referred to as the "safe" and "unsafe" subspaces. Hence, the "off-line" part of the aforementioned computation eventually boils down to (a) the enumeration of the safe and unsafe subspaces and (b) the encoding of this classification in a pertinently designed "mechanism" that will function as the "on-line classifier". The construction of the sought classifiers is further facilitated by an important "monotonicity" property of the "safety" and "unsafety" concepts. This property implies that the set of "unsafe" states is structurally similar to an "upwards-closed" set [2], and enables the effective representation of the aforementioned dichotomy of the RAS state space by focusing only on some extreme elements of the two subspaces, namely, the minimal unsafe states, and, in certain cases, the maximal safe states.

In the context of the aforementioned developments, of particular relevance to this work is the approach presented in [9]. Under that approach, the (on-line) safety assessment of the encountered RAS states is effected by the explicit storage of the minimal unsafe states that are on the boundary between the safe and unsafe subspaces, and the comparison of the assessed states against the elements of the stored data-set: according to the aforementioned monotonicity property of state-unsafety, an assessed state will be unsafe if and only if (*iff*) it dominates (componentwise) some element in the stored data-set. From a computational standpoint, this classification scheme is further facilitated by the employment of some variation of the "TRIE" data structure [4], that enables substantial gains for the storage of the aforementioned state set and

A. Nazeem is with United Airlines, email: ahmed.nazeem@united.com. S. Reveliotis is with the School of Industrial & Systems Engineering, Georgia Institute of Technology, email: spyros@isye.gatech.edu

The authors were partially supported by NSF grant CMMI-0928231.

for the performance of the delineated state-safety test.

Yet, a substantial computational "bottleneck" in all the aforementioned developments results from the fact that they pre-suppose the availability of the enumerations of the RAS safe and unsafe subspaces. Currently, these enumerations are typically produced through the "trimming" of the finite state automaton (FSA) that models the RAS behavior, according to techniques that are borrowed from DES theory [1]. But it is well-known that the aforementioned FSA is of super-polynomial size with respect to (w.r.t.) the size of the more parsimonious representations of the underlying RAS structure, and, in fact, it can grow prohibitively large for the purposes of the aforementioned computations, even for moderate RAS sizes. On the other hand, it should be evident from the above description of the considered methodology that what is really necessary for the construction of the sought classifiers is only a subset of the entire state space. This state subset is smaller than the size of the entire state space, usually by many orders of magnitude [10], [8], [9], [13].

Motivated by the above remarks, this paper proposes a new algorithm that can enumerate the minimal unsafe states for the RAS classes that are considered in the aforementioned works, while foregoing the complete enumeration of the state space of the underlying FSA. In the process of developing this algorithm, the presented work also derives new results and insights regarding the notion of minimality as it pertains to unsafe and deadlock states. The key idea for the proposed algorithm stems from the remark that, in the considered RAS dynamics, unsafety is defined by unavoidable absorption into the system deadlocks. Hence, the unsafe states of interest can be retrieved by a localized computation that starts from the RAS deadlocks and "backtraces" the RAS dynamics until it hits the boundary between the safe and unsafe subspaces. In particular, our interest in minimal unsafe states implies that we can focus this backtracing only to minimal deadlocks. Hence, the proposed algorithm decomposes into a two-stage computation, with the first stage identifying the minimal deadlock states, and the second stage performing the aforementioned backtracing process in order to identify the broader set of minimal unsafe states. Together with the results of [9] that were described in the previous paragraphs, the presented algorithm provides a powerful method for the deployment of the maximally permissive DAP even for RAS with extremely large state spaces, and it effectively "shifts the boundary" regarding the applicability of the aforementioned methods.

In the light of the above discussion, 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 deadlock avoidance that arises in this class. Section III presents and analyzes the algorithm utilized for enumerating the minimal deadlock states. Section IV presents and analyzes the algorithm that enumerates the remaining minimal unsafe states. Section V reports a series of computational experiments that demonstrate the extensive computational gains obtained by the proposed algorithm. Finally, Section VI concludes the paper by summarizing its contributions and outlining some possible extensions of the work. We also notice that a preliminary version of these results has appeared in [11].

II. The considered RAS class and the corresponding deadlock avoidance problem

The considered RAS class We begin the technical discussion of the paper developments, by providing a formal characterization of the RAS class to be considered in this work. An instance Φ from this class is defined as a 4-tuple $\langle \mathcal{R}, C, \mathcal{P}, \mathcal{A} \rangle$[1] where: (i) $\mathcal{R} = \{R_1, \ldots, R_m\}$ is the set of the system *resources*. (ii) $C : \mathcal{R} \to \mathbb{Z}^+$ – i.e., the set of strictly positive integers– is the system *capacity* function, with $C(R_i) \equiv C_i$ characterizing the number of identical units from resource type R_i that are available in the system. Resources are considered to be *reusable*, i.e., they are engaged by the various processes according to an allocation/de-allocation cycle, and each such cycle does not affect their functional status or subsequent availability. (iii) $\mathcal{P} = \{J_1, \ldots, J_n\}$ is the set of the system *process types* supported by the considered system configuration. Each process type J_j is a composite element itself; in particular, $J_j = < \mathcal{S}_j, \mathcal{G}_j >$, where: (a) $\mathcal{S}_j = \{\Xi_{j1}, \ldots, \Xi_{j,l(j)}\}$ is the set of *processing stages* involved in the definition of process type J_j, and (b) $\mathcal{G}_j$ is a connected acyclic digraph $(\mathcal{V}_j, \mathcal{E}_j)$ that defines the sequential logic of process type J_j, $j = 1, \ldots, n$. More specifically, the node set $\mathcal{V}_j$ of graph $\mathcal{G}_j$ is in one-to-one correspondence with the processing stage set, $\mathcal{S}_j$, and furthermore, there are two subsets $\mathcal{V}_j^{\nearrow}$ and $\mathcal{V}_j^{\searrow}$ of $\mathcal{V}_j$ respectively defining the sets of initiating and terminating processing stages for process type J_j. The connectivity of digraph $\mathcal{G}_j$ is such that every node $v \in \mathcal{V}_j$ is accessible from the node set $\mathcal{V}_j^{\nearrow}$ and co-accessible to the node set $\mathcal{V}_j^{\searrow}$. Finally, any directed path of $\mathcal{G}_j$ leading from a node of $\mathcal{V}_j^{\nearrow}$ to a node of $\mathcal{V}_j^{\searrow}$ constitutes a complete execution sequence – or a "route" – for process type J_j. (iv) $\mathcal{A} : \bigcup_{j=1}^{n} \mathcal{S}_j \to \prod_{i=1}^{m} \{0, \ldots, C_i\}$ is the *resource allocation function*, which associates every processing stage Ξ_{jk} with a *resource allocation request* $\mathcal{A}(j,k) \equiv \mathcal{A}_{jk}$. More specifically, each $\mathcal{A}_{jk}$ is an m-dimensional vector, with its i-th component indicating the number of resource units of resource type R_i necessary to support the execution of stage Ξ_{jk}. Furthermore, it is assumed that $\mathcal{A}_{jk} \neq \mathbf{0}$, i.e., every processing stage requires at least one resource unit for its execution. Finally, according to the applying

[1]The complete definition of a RAS, according to [15], 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. We also notice that the RAS class defined in this section corresponds to the class of Conjunctive / Disjunctive RAS in [15]. This class allows for arbitrary resource allocation at the various process stages and process routing flexibility, but process instances must preserve their atomic nature throughout their execution, i.e., this class does not allow for merging or splitting operations.

resource allocation protocol, a process instance executing a processing stage Ξ_{jk} will be able to advance to a successor processing stage $\Xi_{j,k+1}$, only after it is allocated the resource differential $(A_{j,k+1}-A_{jk})^+$. [2] And it is only upon this advancement that the process will release the resource units $|(A_{j,k+1}-A_{jk})^-|$, that are not needed anymore.

In order to facilitate the subsequent developments, we also introduce the following additional notation: In the sequel, we shall set $\xi \equiv \sum_{j=1}^{n}|\mathcal{S}_j|$; i.e., ξ denotes the number of distinct processing stages supported by the considered RAS, across the entire set of its process types. Furthermore, in some of the subsequent developments, the various processing stages $\Xi_{jk},\ j=1,\ldots,n,\ k=1,\ldots,l(j)$, will be considered in the context of a total ordering imposed on the set $\bigcup_{j=1}^{n}\mathcal{S}_j$; in that case, the processing stages themselves and their corresponding attributes will be indexed by a single index q that runs over the set $\{1,\ldots,\xi\}$ and indicates the position of the processing stage in the considered total order. Given an edge $e\in\mathcal{G}_j$ linking Ξ_{jk} to $\Xi_{j,k+1}$, we define $e.src\equiv\Xi_{jk}$ and $e.dst\equiv\Xi_{j,k+1}$; i.e., $e.src$ and $e.dst$ denote respectively the source and the destination nodes of edge e. The number of edges in process graph $\mathcal{G}_j$ that emanate from its node that corresponds to stage Ξ_{jk} will be denoted $\mathcal{D}(\Xi_{jk})$. Also, in the following, we shall use the notation $\mathcal{G}$ to refer to the "union" of process graphs $\mathcal{G}_j,\ j=1,\ldots,n$, i.e., $\mathcal{G}\equiv(\mathcal{V},\mathcal{E})$, with $\mathcal{V}=\cup_{j=1}^{n}\mathcal{V}_j$ and $\mathcal{E}=\cup_{j=1}^{n}\mathcal{E}_j$. Finally, $\eta_{kl},\ l=1,\ldots,C_k$, will denote the number of processing stages that require the allocation of l units from resource type R_k.

Modeling the RAS dynamics as a Finite State Automaton The dynamics of the RAS $\Phi=\langle\mathcal{R},C,\mathcal{P},\mathcal{A}\rangle$, introduced in the previous paragraph, can be formally described by a *Deterministic Finite State Automaton (DFSA)* ([1]), $G(\Phi)=(S,E,f,\mathbf{s}_0,S_M)$, 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 RAS state modeled by $\mathbf{s}$. Hence, S consists of all the vectors $\mathbf{s}\in(\mathbb{Z}_0^+)^{\xi}$ that further satisfy

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

where $\mathcal{A}(\Xi_q)[i]$ denotes the allocation request for resource R_i that is posed by stage Ξ_q.

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

3. The *state transition function* $f:S\times E\to S$ 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* $\mathbf{s}_0$ is set equal to $\mathbf{0}$.

5. The *set of marked states* S_M is the singleton $\{\mathbf{s}_0\}$, indicating the request for complete process runs.

The target behavior of $G(\Phi)$ and the maximally permissive DAP In the following, the set of states $S_r\subseteq S$ that are accessible from state s_0 through a sequence of feasible transitions will be referred to as the *reachable subspace* of Φ. We shall also denote by $S_s\subseteq S$ the set of states that are co-accessible to s_0; i.e., S_s contains those states from which s_0 is reachable through a sequence of feasible transitions. In addition, we define $S_{\bar{r}}\equiv S\setminus S_r$, $S_{\bar{s}}\equiv S\setminus S_s$ and $S_{xy}\equiv S_x\cap S_y,\ x=r,\bar{r},\ y=s,\bar{s}$. As mentioned in the introductory section, in the deadlock avoidance literature, the sets S_{rs} and $S_{r\bar{s}}$ are respectively characterized as the reachable safe and unsafe subspaces; following standard practice, in the sequel, sometimes we shall drop the characterization "reachable" if it is implied by the context.

The RAS unsafety characterized in the previous paragraph results from the formation of RAS *deadlocks*, i.e., RAS states where a subset of the running processes are entangled in a circular waiting pattern for resources that are held by other processes in this set, blocking, thus, the advancement of each other in a permanent manner. The RAS unsafe states are essentially those RAS states from which the formation of deadlock is unavoidable. In the following, the set of deadlock states will be denoted by S_d, while S_{rd} will denote the set of reachable deadlock states. Finally, it is clear from the above that $S_d\subseteq S_{\bar{s}}$ and $S_{rd}\subseteq S_{r\bar{s}}$.

A *maximally permissive deadlock avoidance policy (DAP)* for RAS Φ is a supervisory control policy that restricts the system operation within the subspace S_{rs}, guaranteeing, thus, that every initiated process can complete successfully. This definition further implies that the maximally permissive DAP is unique and it can be implemented by an one-step-lookahead mechanism that recognizes and prevents transitions to unsafe states.

Some monotonicities observed by the state unsafety concept Next we review some additional structure possessed by the set $S_{\bar{s}}$, that was introduced in [13], and enables the effective representation of the maximally permissive DAP through the explicit storage of a very

[2] We remind the reader that $a^+\equiv\max\{a,0\}$ and $a^-\equiv\min\{a,0\}$.

small subset of unsafe states of the underlying state space. It should be clear from the above that the ability of the activated processes in a given 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 observation; these results are well known in the literature, and therefore, their formal proof is omitted.[3]

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

$$\forall \mathbf{s}, \mathbf{s}' \in S, \quad \mathbf{s} \leq \mathbf{s}' \iff (\forall q = 1, \ldots \xi, \quad \mathbf{s}[q] \leq \mathbf{s}'[q]) \quad (2)$$

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 following, we shall use the notation '$\mathbf{s} < \mathbf{s}'$' to denote that the condition of Eq. 2 holds as strict inequality for at least one component $q \in \{1, \ldots, \xi\}$. In the light of Proposition 1, we define the set of *minimal reachable unsafe states* $\bar{S}_{r\bar{s}} \equiv \{\mathbf{s} \in S_{r\bar{s}} \mid \nexists \mathbf{s}' \in S_{r\bar{s}} \text{ s.t. } \mathbf{s}' \leq \mathbf{s}\}$. Similarly, we define the set of *minimal reachable deadlocks* $\bar{S}_{rd} \equiv \{\mathbf{s} \in S_{rd} \mid \nexists \mathbf{s}' \in S_{rd} \text{ s.t. } \mathbf{s}' \leq \mathbf{s}\}$. Proposition 1 implies that, for the considered RAS class, the maximally permissive DAP can be implemented as follows: First, we compute and store $\bar{S}_{r\bar{s}}$ in an appropriate data structure. Then, during the online stage, given a contemplated transition to a state $\mathbf{s}$, we assess the unsafety of $\mathbf{s}$ by searching the stored data for a state $\mathbf{u} \in \bar{S}_{r\bar{s}}$ such that $\mathbf{s} \geq \mathbf{u}$; if such a state $\mathbf{u}$ is found, state $\mathbf{s}$ is unsafe; hence, the contemplated transition is blocked by the supervisory controller. Otherwise, $\mathbf{s}$ is safe and the contemplated transition is allowed. As explained in the introductory discussion, this implementation of the maximally permissive DAP has already been proposed in [9], where the "TRIE" data structure has been used for the efficient storage of $\bar{S}_{r\bar{s}}$. In the next sections, we propose an efficient algorithm for the initial enumeration / extraction of the set $\bar{S}_{r\bar{s}}$ from the underlying RAS dynamics.

[3]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 [16], [14].

TABLE I
THE RAS CONSIDERED IN THE EXAMPLE

Resource Types:	$\{R_1, \ldots, R_8\}$
Resource Capacities:	$C_i = 1, \forall i \in \{1, \ldots, 7\}$, $C_8 = 2$
Process Type 1:	$\Xi_{11}(R_1) \rightarrow \Xi_{12}(R_2) \rightarrow \Xi_{13}(R_3)$
Process Type 2:	$\Xi_{21}(R_3) \rightarrow \Xi_{22}(R_2, R_5) \rightarrow \Xi_{23}(R_4)$
Process Type 3:	$\Xi_{31}(R_4) \rightarrow \Xi_{32}(R_2) \rightarrow \Xi_{33}(R_1)$
Process Type 4:	$\Xi_{41}(R_5) \rightarrow \Xi_{42}(R_1) \rightarrow$ $\Xi_{43}(R_6)$ or $\Xi_{44}(R_7) \rightarrow \Xi_{45}(R_8)$
Process Type 5:	$\Xi_{51}(R_8) \rightarrow \Xi_{52}(R_1)$

Example We shall use the RAS configuration depicted in Table I as a running example in the rest of the paper to demonstrate the application of the steps of the introduced algorithm. The considered RAS has eight resource types, $\{R_1, \ldots, R_8\}$, all with single unit capacities, except for resource R_8, which has a capacity of two units. The considered RAS also has five process types, $\{J_1, \ldots, J_5\}$. The reader should notice that process J_4 presents routing flexibility. More specifically, a job at the second processing stage Ξ_{42} can advance to stage Ξ_{43} (acquiring one unit of R_6), or to stage Ξ_{44} (acquiring one unit of R_7). On the other hand, all the other processes have simple linear structures. We also notice that all the processing stages have single-type resource allocation, except for stage Ξ_{22} which requests the allocation of both R_2 and R_5. For representational economy, in the subsequent discussion a state will be represented by the multi-set of the processing stages with non-zero process content in it. Applying the FSA-based analysis introduced in this section to the considered RAS, reveals that $\bar{S}_{rd} = \{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_4\}$ and that $\bar{S}_{r\bar{s}} = \{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_4, \mathbf{u}_5, \mathbf{u}_6\}$, where the states $\mathbf{u}_1, \ldots, \mathbf{u}_6$ are depicted in Table IV. Space limitations prohibit the depiction of the entire state transition diagram (STD) of the underlying FSA, but we report that it includes 1008 reachable states, of which 638 are safe and the remaining unsafe. In the next two sections, we shall demonstrate how the introduced algorithms construct the two state sets $\bar{S}_{rd}$ and $\bar{S}_{r\bar{s}}$ while visiting only a very small subset of the aforementioned STD.

III. ENUMERATING $\bar{S}_{rd}$

As pointed out in the introductory section, the first step towards the enumeration of the minimal unsafe states is the enumeration of the minimal deadlocks. This is the content of this section. First, we define some terms and notation that will be used throughout the rest of the paper. Next, we proceed to describe the detailed flow of the proposed algorithm.

A. Preamble

Let $\mathbf{s}.R_i$, $i = 1, \ldots, m$, denote the total number of units from resource R_i that are allocated at state $\mathbf{s}$. Then, given an edge $e \in \mathcal{G}$, we shall say that e is "*blocked*" at state $\mathbf{s}$ *iff* $\mathbf{s}[e.src] > 0$, and $\exists R_k \in \mathcal{R}$ s.t. $\mathbf{s}.R_k + \mathcal{A}_{e.dst}[k] - \mathcal{A}_{e.src}[k] > C_k$. We shall say that edge $e \in \mathcal{G}$ is "*enabled*" at state $\mathbf{s}$ *iff*

$\mathbf{s}[e.src] > 0$ and e is not blocked. Similarly, a processing stage q is blocked at state $\mathbf{s}$ *iff* all its outgoing edges are blocked, whereas it is enabled *iff* at least one of its outgoing edges is enabled. The set of enabled edges at $\mathbf{s}$ will be denoted by $g(\mathbf{s})$.

Definition 1: Given a set of states X, define the state λ_X by $\lambda_X[q] \equiv max_{\mathbf{x}\in X}\mathbf{x}[q],\ q = 1\ldots\xi$. In the sequel, we shall refer to λ_X as the *"combination"* of the states of X.

By its definition, a minimal deadlock state $\mathbf{s}_d$ is a state at which all the processing stages with non-zero process content are blocked. Let $\{q_1,\ldots,q_t\}$ refer to this set of processing stages. Then, we have the following lemma:

Lemma 1: A minimal deadlock state $\mathbf{s}_d$ with active processing stages $\{q_1,\ldots,q_t\}$ can be expressed as the combination of a set of minimal states $\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}$ such that q_i is blocked at $\mathbf{x}_i$.

Proof: Consider the vectors $\mathbf{x}_i,\ i = 1,\ldots,t$, that are obtained by starting from the deadlock state $\mathbf{s}_d$ and iteratively removing processes from this state, one at a time, until no further process can be removed without unblocking stage q_i. Then, clearly, each state $\mathbf{x}_i$ is a minimal state at which processing stage q_i is blocked. Furthermore, by the construction of $\{\mathbf{x}_i, i = 1,\ldots,t\}$, $\lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}} \leq \mathbf{s}_d$, and $\lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}}$ is itself a deadlock state. But then, the minimality of $\mathbf{s}_d$ implies that $\lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}} = \mathbf{s}_d$. □

Let $\{e_{i1},\ldots,e_{i\mathcal{D}(q_i)}\}$ refer to the set of edges emanating from node q_i in graph $\mathcal{G}_i$, where $\mathcal{D}(q_i)$, as defined in Section II, is the number of edges emanating from the node q_i in the graph. Then, by an argument similar to that in the proof of Lemma 1, we can perceive each state $\mathbf{x}_i$ appearing in the statement of Lemma 1 as a combination of a set of minimal states $\{\mathbf{x}_{i1},\ldots,\mathbf{x}_{i\mathcal{D}(q_i)}\}$ such that e_{ij} is blocked at state $\mathbf{x}_{ij}$, i.e., $\mathbf{x}_i = \lambda_{\{\mathbf{x}_{i1},\ldots,\mathbf{x}_{i\mathcal{D}(q_i)}\}}$. Each $\mathbf{x}_{ij}$ is a state that has active processes at stage $e_{ij}.src$, and for some resource type R_k s.t. $\mathcal{A}_{e_{ij}.dst}[k] - \mathcal{A}_{e_{ij}.src}[k] > 0$, $\mathbf{x}_{ij}.R_k > C_k - \mathcal{A}_{e_{ij}.dst}[k] + \mathcal{A}_{e_{ij}.src}[k] \equiv l$. Hence, the minimal states that block e_{ij} through R_k can be obtained by enumerating all the minimal states that allocate $l+1,\ldots,C_k$ units of R_k (i.e., the minimal states $\mathbf{s}$ for which $\mathbf{s}.R_k \in \{l+1,\ldots,C_k\}$).

B. Outline of the proposed algorithm

The proposed algorithm for the enumeration of the minimal deadlocks is motivated by the analysis presented in the pervious subsection, and it can be outlined as follows:

1) For each resource type R_k, and for each occupancy level l, $1 \leq l \leq C_k$, compute the set of minimal states that allocate l units of R_k; call it $MinStR[k][l]$.
2) Use the results obtained in Step 1 in order to compute, for each edge e, the set of minimal states at which e is blocked; call it $BlockEd[e]$.
3) Use the results obtained in Step 2 in order to compute, for each processing stage q, the set of minimal states at which q is blocked; call it $BlockPs[q]$.
4) Finally, enumerate the set of minimal deadlocks through the following recursive scheme that, for each processing stage q and each minimal state $\mathbf{s} \in BlockPs[q]$, does the following: It sets $\mathbf{p}_1 := \mathbf{s}$, and then searches for an enabled processing stage q' at $\mathbf{p}_1$. Next, it branches for each minimal state $\mathbf{x}$ at which q' is blocked (i.e., $\mathbf{x} \in BlockPs[q']$), combining such a state with $\mathbf{p}_1$ (i.e., it computes the combination $\lambda_{\{\mathbf{p}_1,\mathbf{x}\}}$). Let $\mathbf{p}_2$ be a feasible state generated at one of those branches; i.e., $\mathbf{p}_2 = \lambda_{\{\mathbf{p}_1,\mathbf{x}'\}}$ for some $\mathbf{x}' \in BlockPs[q']$, and furthermore, $\mathbf{p}_2 \in S$. State $\mathbf{p}_2$ is processed in a similar manner with state $\mathbf{p}_1$ above, and the branching continues across all the generated paths of the resulting search graph until a deadlock state is reached on each path.

The rest of this section discusses further the various steps in the above outline. A complete algorithmic implementation of all these steps together with a detailed complexity analysis of these algorithms can be found in [7]. Closing this introductory discussion on the presented algorithms, we also notice that a process instance executing a terminal processing stage can immediately exit the system upon completion; hence terminal processing stages do not have any active process instances at any minimal deadlock state. Therefore, these stages will be ignored in the subsequent constructions.

C. Computing $MinStR[k]$

A minimal state $\mathbf{s}$ that allocates l units of resource type R_k may be either a unit vector state with $\mathbf{s}[q] = 1$ for some component $q \in \{1,\ldots,\xi\}$, $\mathbf{s}[q'] = 0,\ \forall q \neq q'$, and $\mathcal{A}_q[k] = l$, or a vector equal to $\mathbf{s}_1 + \mathbf{s}_2$ where $\mathbf{s}_1$ is a minimal state using j units of R_k and $\mathbf{s}_2$ is a minimal state using $l-j$ units of R_k. Based on the above remark, $MinStR[k][l]$ is initialized with the η_{kl} unit vector states corresponding to the stages that request l units of R_k. In particular, $MinStR[k][1]$ will contain only these η_{k1} unit vector states. Proceeding inductively for $l > 1$, and assuming that, $\forall j \leq \lfloor l/2 \rfloor$, $MinStR[k][j]$ has been already computed, we add each state in $MinStR[k][j]$ to each state in $MinStR[k][l-j]$, and insert the resultant states into $MinStR[k][l]$, provided that they satisfy the feasibility conditions of Eq. 1.

Example (cont.) Consider the resource type R_2. Only stages Ξ_{12}, Ξ_{22}, and Ξ_{32} occupy one unit of R_2. Hence, $MinStR[2][1]$ contains only the states $\{\Xi_{12}\}$,$\{\Xi_{22}\}$, and $\{\Xi_{32}\}$. On the other hand, consider the resource type R_8. Only stage Ξ_{51} occupies one unit of R_8. Hence $MinStR[8][1]$ contains only the state $\{\Xi_{51}\}$. Moreover, to obtain a minimal state that occupies two units of R_8, we can add any two members of $MinStR[8][1]$. Since the state $\{\Xi_{51}\}$ is the only member in $MinStR[8][1]$, adding it to itself results in the state $\{2\Xi_{51}\}$ which is the only member of $MinStR[8][2]$. The complete array $MinStR$ computed for the resource types in this example is depicted in Table II.

TABLE II
THE ARRAY $MinStR$ FOR THE EXAMPLE

$MinStR[1][1]$	$\{\Xi_{11}\}$, $\{\Xi_{42}\}$
$MinStR[2][1]$	$\{\Xi_{12}\}$,$\{\Xi_{22}\}$,$\{\Xi_{32}\}$
$MinStR[3][1]$	$\{\Xi_{21}\}$
$MinStR[4][1]$	$\{\Xi_{31}\}$
$MinStR[5][1]$	$\{\Xi_{41}\}$, $\{\Xi_{22}\}$
$MinStR[6][1]$	$\{\Xi_{43}\}$
$MinStR[7][1]$	$\{\Xi_{44}\}$
$MinStR[8][1]$	$\{\Xi_{51}\}$
$MinStR[8][2]$	$\{2\Xi_{51}\}$

TABLE III
THE ARRAY $BlockEd$ FOR THE EXAMPLE

$BlockEd[\Xi_{11} \rightarrow \Xi_{12}]$	$\mathbf{es}_1 = \{\Xi_{11}, \Xi_{12}\}$, $\mathbf{es}_2 = \{\Xi_{11}, \Xi_{22}\}$, $\mathbf{es}_3 = \{\Xi_{11}, \Xi_{32}\}$
$BlockEd[\Xi_{12} \rightarrow \Xi_{13}]$	$\mathbf{es}_4 = \{\Xi_{12}, \Xi_{21}\}$
$BlockEd[\Xi_{21} \rightarrow \Xi_{22}]$	$\mathbf{es}_5 = \{\Xi_{21}, \Xi_{12}\}$, $\mathbf{es}_6 = \{\Xi_{21}, \Xi_{22}\}$, $\mathbf{es}_7 = \{\Xi_{21}, \Xi_{32}\}$, $\mathbf{es}_8 = \{\Xi_{21}, \Xi_{41}\}$
$BlockEd[\Xi_{22} \rightarrow \Xi_{23}]$	$\mathbf{es}_9 = \{\Xi_{22}, \Xi_{31}\}$
$BlockEd[\Xi_{31} \rightarrow \Xi_{32}]$	$\mathbf{es}_{10} = \{\Xi_{31}, \Xi_{12}\}$, $\mathbf{es}_{11} = \{\Xi_{31}, \Xi_{22}\}$, $\mathbf{es}_{12} = \{\Xi_{31}, \Xi_{32}\}$
$BlockEd[\Xi_{32} \rightarrow \Xi_{33}]$	$\mathbf{es}_{13} = \{\Xi_{32}, \Xi_{11}\}$, $\mathbf{es}_{14} = \{\Xi_{32}, \Xi_{42}\}$
$BlockEd[\Xi_{41} \rightarrow \Xi_{42}]$	$\mathbf{es}_{15} = \{\Xi_{41}, \Xi_{11}\}$, $\mathbf{es}_{16} = \{\Xi_{41}, \Xi_{42}\}$
$BlockEd[\Xi_{42} \rightarrow \Xi_{43}]$	$\mathbf{es}_{17} = \{\Xi_{42}, \Xi_{43}\}$
$BlockEd[\Xi_{42} \rightarrow \Xi_{44}]$	$\mathbf{es}_{18} = \{\Xi_{42}, \Xi_{44}\}$
$BlockEd[\Xi_{43} \rightarrow \Xi_{45}]$	$\mathbf{es}_{19} = \{\Xi_{43}, 2\Xi_{51}\}$
$BlockEd[\Xi_{44} \rightarrow \Xi_{45}]$	$\mathbf{es}_{20} = \{\Xi_{44}, 2\Xi_{51}\}$
$BlockEd[\Xi_{51} \rightarrow \Xi_{52}]$	$\mathbf{es}_{21} = \{\Xi_{51}, \Xi_{11}\}$, $\mathbf{es}_{22} = \{\Xi_{51}, \Xi_{42}\}$

D. Computing $BlockEd[e]$

According to the remarks that were provided in Subsection III-A, the computation of this data structure can be organized as follows: For each resource R_k s.t. $\mathcal{A}_{e.dst}[k] - \mathcal{A}_{e.src}[k] > 0$, and for each occupancy level l s.t. $l > C_k - \mathcal{A}_{e.dst}[k] + \mathcal{A}_{e.src}[k]$, we insert all the states from $MinStR[k][l]$ into $BlockEd[e]$ after adding one process at $e.src$, if needed; in this last case, the resulting state must also be checked for feasibility. Finally, the non-minimal states are removed from $BlockEd[e]$.

Example (cont.) Consider the edge $\Xi_{11} \rightarrow \Xi_{12}$. Advancement across this edge requires the allocation of the single unit of resource R_2. Hence, only states where the unit of R_2 is allocated, can be used to block the edge; i.e., the states that can block this edge are those in $MinStR[2][1]$. Thus, $BlockEd[\Xi_{11} \rightarrow \Xi_{12}]$ contains the states $\mathbf{es}_1 = \{\Xi_{11}, \Xi_{12}\}$, $\mathbf{es}_2 = \{\Xi_{11}, \Xi_{22}\}$, and $\mathbf{es}_3 = \{\Xi_{11}, \Xi_{32}\}$. On the other hand, consider the edge $\Xi_{44} \rightarrow \Xi_{45}$. Advancement across this edge requires the allocation of one unit from resource R_8. Since the capacity of R_8 equals two, the states that can be used to block this edge are those in $MinStR[8][2]$. Thus, $BlockEd[\Xi_{44} \rightarrow \Xi_{45}]$ contains the state $\{\Xi_{44}, 2\Xi_{51}\}$. The complete array $BlockEd$ computed for this example is depicted in Table III.

E. Computing $BlockPs[q]$

Let $\{e^q_1, \ldots, e^q_{\mathcal{D}(q)}\}$ be the set of edges emanating from q. Then, $BlockPs[q]$ is computed by taking all the feasible combinations of states from $BlockEd[e^q_1] \times BlockEd[e^q_2] \times \ldots BlockEd[e^q_{\mathcal{D}(q)}]$, while eliminating those combinations that result in non-minimal elements.

Example (cont.) Consider the processing stage Ξ_{42}. It has two outgoing edges, $\Xi_{42} \rightarrow \Xi_{43}$ and $\Xi_{42} \rightarrow \Xi_{44}$. It can be seen from Table III that $\mathbf{es}_{17}$ (resp., $\mathbf{es}_{18}$) is the only member of $BlockEd[\Xi_{42} \rightarrow \Xi_{43}]$ (resp., $BlockEd[\Xi_{42} \rightarrow \Xi_{44}]$). Hence, the combination operation (c.f. Definition 1) is applied to states $\mathbf{es}_{17}$ and $\mathbf{es}_{18}$ to generate the state $\mathbf{ps}_1 \equiv \{\Xi_{42}, \Xi_{43}, \Xi_{44}\}$, which is a minimal state at which Ξ_{42} is blocked. Hence $BlockPs[\Xi_{42}] = \{\Xi_{42}, \Xi_{43}, \Xi_{44}\}$. Except for Ξ_{42}, each processing stage has only one outgoing edge. Hence $\forall\ \Xi_{jk} \neq \Xi_{42},\ BlockPs[\Xi_{jk}] = BlockEd[\Xi_{jk} \rightarrow \Xi_{j,k+1}]$.

F. Enumerating the minimal deadlock states

The complete algorithm for enumerating the minimal reachable deadlock states is depicted in Procedure 1. Lines 2-10 involve the computation of the lists $MinStR$, $BlockEd$, and $BlockPs$. For each processing stage q, all the minimal deadlock states at which q has non-zero processes are enumerated by Lines 11-28. In particular, for a given processing stage q, we start by inserting into the list $workingQueue$ each minimal state at which q is blocked. In the "While" loop of Lines 14-26, we extract (dequeue) every state $\mathbf{p}$ from this queue and examine $\mathbf{p}$ for enabled processing stages. If $\mathbf{p}$ has no enabled processing stages, the function $getAnEnabledProcStg$ at Line 16 returns a value of 0; hence, it is inferred that $\mathbf{p}$ is a deadlock state at which q has non-zero processes, and it is inserted into the hash table $deadlockHT$ (c.f. Line 18). Otherwise, $getAnEnabledProcStg$ returns an enabled processing stage q^*. In this case, Lines 20-24 generate every feasible combination of state $\mathbf{p}$ with the minimal states blocking q^* and add them to $workingQueue$. $workingQueue$ becomes empty when all the deadlock states at which q has non-zero processes have been enumerated across all the paths of the generated search graph. Finally, Line 29 removes the non-minimal and unreachable deadlock states from $deadlockHT$.

Example (cont.) Consider the iteration of Procedure 1 that starts from stage Ξ_{42} and state $\mathbf{ps}_1 = \{\Xi_{42}, \Xi_{43}, \Xi_{44}\}$ ($\mathbf{ps}_1 \in BlockPs[\Xi_{42}]$). To block Ξ_{43}, we combine $\mathbf{ps}_1$ with state $\mathbf{es}_{19} \in BlockPs[\Xi_{43}]$, resulting in state $\mathbf{u}_1 \equiv \{\Xi_{42}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$ which is a state at which all the active processing stages are blocked; hence it is a deadlock state. Continuing the application of the algorithm yields the minimal deadlock states $\mathbf{u}_2 \equiv \{\Xi_{11}, \Xi_{32}\}$, $\mathbf{u}_3 \equiv \{\Xi_{12}, \Xi_{21}\}$, $\mathbf{u}_4 \equiv \{\Xi_{22}, \Xi_{31}\}$. Hence, $\bar{S}_{rd} = \{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_4\}$, as reported at the end of Section II. □

The next theorem establishes the correctness of Procedure 1.

Theorem 1: Procedure 1 enumerates all the minimal reachable deadlock states of its input RAS Φ.

Proof: Let $\mathbf{s}_d$ be an arbitrary minimal deadlock state, and

Procedure 1 EnumMinReachDeadlocks(Φ)

Input: A RAS instance Φ
Output: the list *deadlockHT* containing all the reachable minimal deadlocks of Φ

```
1: deadlockHT ← ∅
2: for k = 1 : m do
3:    Compute MinStR[k]
4: end for
5: for all e ∈ E do
6:    Compute BlockEd[e]
7: end for
8: for q = 1 : ξ do
9:    Compute BlockPs[q]
10: end for
11: for q = 1 : ξ do
12:    for s ∈ BlockPs[q] do
13:       workingQueue ← s
14:       while workingQueue ≠ ∅ do
15:          p ← dequeue(workingQueue)
16:          q* ← getAnEnabledProcStg(p)
17:          if q* = 0 then
18:             Insert p into deadlockHT
19:          else
20:             for all x ∈ BlockPs[q*] do
21:                if Feasible(λ_{x,p}) then
22:                   Insert λ_{x,p} into workingQueue
23:                end if
24:             end for
25:          end if
26:       end while
27:    end for
28: end for
29: Remove non-minimal states and unreachable states
    from deadlockHT
30: return deadlockHT
```

$\{q_1, \ldots, q_t\}$ be the set of processing stages that have active processes at $\mathbf{s}_d$. Then, according to Lemma 1, there exists a set of states $\{\mathbf{x}_1, \ldots, \mathbf{x}_t\}$ such that $\mathbf{x}_j \in BlockPs[q_j]$, $\mathbf{x}_j \leq \mathbf{s}_d$, and $\mathbf{s}_d = \lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}}$. $\mathbf{x}_1$ will be picked by Line 12. W.l.o.g., assume that $q_2 = getAnEnabledProcStg(\mathbf{x}_1)$. Then, Line 22 implies that the state $\mathbf{p}_2 = \lambda_{\{\mathbf{x}_1,\mathbf{x}_2\}}$ is inserted into *workingQueue*; hence, it will be eventually extracted at Line 15. Repeating the same argument, assume that $q_3 = getAnEnabledProcStg(\mathbf{p}_2)$; then, we will have the state $\mathbf{p}_3 = \lambda_{\{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3\}}$ inserted into *workingQueue*. Let $t' \leq t$ be the last processing stage in this tracing sequence. Thus, $\mathbf{p}_{t'} = \lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_{t'}\}}$ is a deadlock state. But, $\mathbf{p}_{t'} = \lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_{t'}\}} \leq \lambda_{\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}} = \mathbf{s}_d$. Therefore, by the minimality of $\mathbf{s}_d$, it must be that $\mathbf{s}_d = \mathbf{p}_{t'}$. Hence $\mathbf{s}_d$ is enumerated. □

Complexity Considerations As already mentioned, a complete complexity analysis of Procedure 1 and of its supporting subroutines can be found in [7], where it is shown that the overall computational complexity of Procedure 1 can be characterized as $\mathcal{O}(\xi \cdot m^{2\cdot|\mathcal{E}|} \cdot \eta^{2\cdot C\cdot|\mathcal{E}|} + |S_r|)$. In this expression, m denotes the number of the resources types in the input RAS Φ, $C \equiv max_{k=1}^{m} C_k$, and $\eta \equiv max_{k=1}^{m} \max_{l=1}^{C_k} \eta_{kl}$.[4] The term $|S_r|$ appears in the above expression due to the state reachability analysis that is performed in Line 29 of Procedure 1. However, while this term provides a worst-case bound for the corresponding operation, the practical run-time for this operation is typically very small, due to the small process content of the assessed states. Hence, based on the above discussion, we can conclude that the algorithm complexity is most sensitive to the capacity of the resource types and to the number of the distinct event types that take place in the underlying RAS.

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

In this section, we provide an algorithm that enumerates the subspace $\bar{S}_{r\bar{s}}$ without enumerating the entire reachable state space. We proceed as follows: First, we introduce all the necessary definitions for the description of the algorithm. Next, we introduce the algorithm itself. Finally, we prove the correctness of the algorithm.

A. Preamble

Given a minimal deadlock-free unsafe state $\mathbf{u}$, we notice the following: (i) No unloading event is enabled at $\mathbf{u}$, since otherwise $\mathbf{u}$ would not be minimal. (ii) The unsafety of $\mathbf{u}$ is a consequence of its current process content and it does not require the loading of any new processes in order to manifest itself. (iii) The advancement of any unblocked process at $\mathbf{u}$ leads to another unsafe state; however, this new unsafe state can be minimal or non-minimal. The following definition characterizes further the dynamics that result from the advancement of unblocked processes in a minimal unsafe state.

Definition 2: Given a minimal unsafe state $\mathbf{u}$ such that $g(\mathbf{u}) = \{e_1, \ldots, e_K\}$, let $\mathbf{h}_1, \ldots, \mathbf{h}_K$ be the respective states that result from executing events $e_1, \ldots, e_K$ at $\mathbf{u}$. Then, $\forall i = 1 : K$, $nextMin(\mathbf{u}, e_i) \equiv \{\mathbf{z}_{i1}, \ldots, \mathbf{z}_{iw(i)}\}$ where $\forall j = 1 : w(i)$, $\mathbf{z}_{ij} \leq \mathbf{h}_i$ is a minimal unsafe state. We also set $nextMin(\mathbf{u}) \equiv \bigcup_{i=1}^{K} nextMin(\mathbf{u}, e_i)$. Finally, we denote by $\mathbf{s}_{ij}$ the result of backtracing e_i at $\mathbf{z}_{ij}$.

It is easy to see that if $\mathbf{h}_i$, in the above definition, is a minimal unsafe state, then $w(i) = 1$, $\mathbf{z}_{i1} = \mathbf{h}_i$, $\mathbf{s}_{i1} = \mathbf{u}$. Otherwise, to show that $\mathbf{s}_{ij}$ is well-defined, it suffices to show that: (i) $\mathbf{z}_{ij}[e_i.dst] = \mathbf{h}_i[e_i.dst]$, and (ii) state $\mathbf{s}_{ij}$ is a feasible state according to Eq. 1. To establish item (i), first notice that $e_i.dst$ is the unique entry for which $\mathbf{h}_i$ is greater than $\mathbf{u}$. Hence, if item (i) was not true, then $\mathbf{z}_{ij} < \mathbf{u}$, a result that violates the minimality of $\mathbf{u}$. On the other hand, item (ii) is established by the fact that $\mathbf{z}_{ij} < \mathbf{h}_i$. It can also be seen that if $\mathbf{z}_{ij} < \mathbf{h}_i$, then $\mathbf{s}_{ij} < \mathbf{u}$. Combined with the minimality of $\mathbf{u}$ as an unsafe state, this

[4] We also remind the reader that ξ denotes the total number of stages and $|\mathcal{E}|$ denotes the number of distinct event types in RAS Φ.

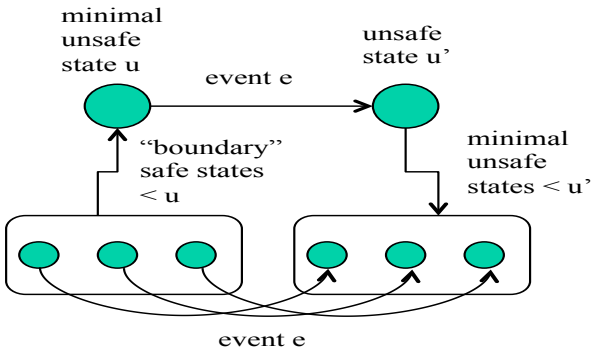

Fig. 1. A schematic diagram of the RAS transitional structure that is leveraged by the proposed algorithm for the enumeration of $\bar{S}_{r\bar{s}}$.

last result implies that $\mathbf{s}_{ij}$ is a safe state in this case. The structure revealed by Definition 2 and the above discussion is depicted schematically in Figure 1.

Example (cont.) Consider state $\mathbf{u}_5 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{31}\}$. It can be easily checked that this state is minimal unsafe but deadlock-free. The set of enabled edges in $\mathbf{u}_5$ is $g(\mathbf{u}_5) = \{\Xi_{11} \rightarrow \Xi_{12}, \Xi_{21} \rightarrow \Xi_{22}, \Xi_{31} \rightarrow \Xi_{32}\}$, and the resulting states are respectively $\mathbf{h}_1 \equiv \{\Xi_{12}, \Xi_{21}, \Xi_{31}\}$, $\mathbf{h}_2 \equiv \{\Xi_{11}, \Xi_{22}, \Xi_{31}\}$, and $\mathbf{h}_3 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{32}\}$. It can be easily seen that $\mathbf{u}_3 < \mathbf{h}_1$, $\mathbf{u}_4 < \mathbf{h}_2$, and $\mathbf{u}_2 < \mathbf{h}_3$, where the definition of states $\mathbf{u}_2$, $\mathbf{u}_3$ and $\mathbf{u}_4$ is provided in Table IV. Hence, $\mathbf{z}_{11} = \mathbf{u}_3$, $\mathbf{z}_{21} = \mathbf{u}_4$, and $\mathbf{z}_{31} = \mathbf{u}_2$. Backtracing the edge $\Xi_{11} \rightarrow \Xi_{12}$ from $\mathbf{z}_{11}$ yields the safe state $\mathbf{s}_{11} \equiv \{\Xi_{11}, \Xi_{21}\}$. Similarly, backtracing the corresponding edges from $\mathbf{z}_{21}$ and $\mathbf{z}_{31}$ yields respectively the safe states $\mathbf{s}_{21} \equiv \{\Xi_{21}, \Xi_{31}\}$ and $\mathbf{s}_{31} \equiv \{\Xi_{11}, \Xi_{31}\}$. □

As explained in the introductory section, the algorithm proposed in this work seeks to enumerate all the minimal reachable unsafe states starting from the minimal reachable deadlocks, and tracing backwards the dynamics that are described in Definition 2. This reconstructive process can be described as follows: Let us first characterize a safe state $\mathbf{a}$ as a "*boundary safe*" state *iff* it is one-transition away from reaching some unsafe state. During the course of its execution, the proposed algorithm generates, both, unsafe and safe states. The generated safe states are all boundary safe states, and they are used as "stepping stones" to reach further parts of the unsafe state space. More specifically, the proposed algorithm employs three different mechanisms to generate states in its exploration process: (i) backtracing from an unsafe state; (ii) combining two boundary safe states according to the logic of Definition 1 (i.e., taking the maximum number of processes at each processing stage); and (iii) adding some processes to a boundary safe state to make it unsafe. The first two mechanisms can return, both, safe and unsafe states, whereas the last mechanism returns only unsafe states. In the case of the first two mechanisms, once a state $\mathbf{a}$ has been generated, its potential unsafety will be identified by running upon it a search-type algorithm that assesses the state co-reachability w.r.t. the target state $\mathbf{s}_0$. If $\mathbf{a}$ is found to be unsafe, it is also tested for non-minimality w.r.t. the previously generated unsafe states; if it is minimal, it is backtraced to generate its immediate predecessors, and then it is saved. On the other hand, if the generated state $\mathbf{a}$ is safe, then, it is endowed by an additional attribute that is computed upon its generation; this attribute will be denoted by $\tau_{\mathbf{a}}$ and it constitutes a set of edges that emanate from state $\mathbf{a}$ and are known to lead to unsafe states. The set of the unsafe states that are reached from $\mathbf{a}$ through the edges in $\tau_{\mathbf{a}}$ will be denoted by $U(\mathbf{a})$. The detailed logic for the computation of the set $\tau_{\mathbf{a}}$ depends on the particular mechanism that generated safe state $\mathbf{a}$, and it can be described as follows:

- If state $\mathbf{a}$ was generated by tracing back upon edge e from unsafe state $\mathbf{u}$, then, the algorithm sets $\tau_{\mathbf{a}} = \{e\}$. Clearly, firing e at $\mathbf{a}$ leads to unsafety.
- If $\mathbf{a}$ was generated by combining two previously generated boundary safe states $\mathbf{a}_1$ and $\mathbf{a}_2$ (i.e., $\mathbf{a} = \lambda_{\{\mathbf{a}_1, \mathbf{a}_2\}}$), then, $\tau_{\mathbf{a}} = (\tau_{\mathbf{a}_1} \cup \tau_{\mathbf{a}_2}) \cap g(\mathbf{a})$. Indeed, it is easy to see that firing any enabled transition among $\tau_{\mathbf{a}_1} \cup \tau_{\mathbf{a}_2}$ at state $\lambda_{\{\mathbf{a}_1, \mathbf{a}_2\}}$ will lead to a state that dominates a state in $U(\mathbf{a}_1) \cup U(\mathbf{a}_2)$; hence to an unsafe state.

Furthermore, it is possible that a boundary safe state $\mathbf{a}$ will be generated more than once in the execution of the proposed algorithm. In fact, it might happen that $\mathbf{a'}_1 = \mathbf{a'}_2$, but $\tau_{\mathbf{a'}_1} \neq \tau_{\mathbf{a'}_2}$. This will happen if $\mathbf{a'}_1$ and $\mathbf{a'}_2$ are generated by different mechanisms, by backtracing from different unsafe states, or by combining different pairs of boundary safe states. Assuming w.l.o.g. that state $\mathbf{a'}_1$ was generated first in the course of the algorithm execution, state $\mathbf{a'}_2$ will be discarded upon its generation, but $\tau_{\mathbf{a'}_1}$ will be updated to $\tau_{\mathbf{a'}_1} := \tau_{\mathbf{a'}_1} \cup \tau_{\mathbf{a'}_2}$.

Example (cont.) Consider the minimal deadlock state $\mathbf{u}_1 \equiv \{\Xi_{42}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$. The jobs at Ξ_{51} can not be backtraced because Ξ_{51} is an initiating stage. Also, the jobs at Ξ_{43} and Ξ_{44} can not be backtraced because there is no slack capacity for resource type R_1, which is held by Ξ_{42}. On the other hand, backtracing the job at Ξ_{42} across the edge $\Xi_{41} \rightarrow \Xi_{42}$ yields the safe state $\mathbf{a}_1 \equiv \{\Xi_{41}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$, where $\tau_{\mathbf{a}_1} \equiv \{\Xi_{41} \rightarrow \Xi_{42}\}$.

The second column in Table IV depicts the states that result while backtracing from each of the minimal deadlocks, $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$ and $\mathbf{u}_4$, that were identified in the previous section. The boldfaced processing stages in the listed states $\mathbf{a}_i$, $i = 1, \ldots, 4$, indicate the source nodes on the backtraced edges that provided these states. Also, it is easy to check that all states $\mathbf{a}_i$, $i = 1, \ldots, 4$, that were obtained from this backtracing, are safe. □

The rationale for basing the overall search process for minimal unsafe states upon the three state-generation mechanisms that were described above, can be explained as follows: The first mechanism is the primary backtracing mechanism employed by the proposed algorithm, and its role is self-explanatory. On the other hand, in order to explain the role of the second and the third mechanisms,

TABLE IV
THE RESULTS OF ALGORITHM 3 FOR THE CONSIDERED EXAMPLE

$\mathbf{u}$	Backtrace($\mathbf{u}$)
$\mathbf{u}_1 = \{\Xi_{42}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$	$\mathbf{a}_1 = \{\boldsymbol{\Xi_{41}}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$
$\mathbf{u}_2 = \{\Xi_{11}, \Xi_{32}\}$	$\mathbf{a}_2 = \{\Xi_{11}, \boldsymbol{\Xi_{31}}\}$
$\mathbf{u}_3 = \{\Xi_{12}, \Xi_{21}\}$	$\mathbf{a}_3 = \{\boldsymbol{\Xi_{11}}, \Xi_{21}\}$
$\mathbf{u}_4 = \{\Xi_{22}, \Xi_{31}\}$	$\mathbf{a}_4 = \{\boldsymbol{\Xi_{21}}, \Xi_{31}\}$
$\mathbf{u}_5 = \{\Xi_{11}, \Xi_{21}, \Xi_{31}\}$	$\emptyset$
$\mathbf{u}_6 = \{\Xi_{11}, \Xi_{21}, \Xi_{41}\}$	$\emptyset$
$\mathbf{u}_7 = \{\Xi_{41}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}, \Xi_{11}, \Xi_{21}\}$	$\emptyset$

we remind the reader that Definition 2 implies that a boundary safe state $\mathbf{a}$ might be dominated by another minimal unsafe state leading to unsafe states that dominate some state(s) in $U(\mathbf{a})$ (c.f. also Figure 1); these two state-generation mechanisms enable the proposed algorithm to reach these additional minimal unsafe states. More specifically, by applying the second mechanism on any pair of boundary safe states $\mathbf{a}_1$ and $\mathbf{a}_2$, we obtain the state $\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}$ that dominates both $\mathbf{a}_1$ and $\mathbf{a}_2$ w.r.t. the partial state order that is established by " $\leq$ ". This domination further implies that $g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}) \subseteq g(\mathbf{a}_1) \cup g(\mathbf{a}_2)$. If $\tau_{\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}} \equiv \{\tau_{\mathbf{a}_1} \cup \tau_{\mathbf{a}_2}\} \cap g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}) \neq \emptyset$, the aforementioned domination also implies that $g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}})$ contains transitions to states that dominate unsafe states and, therefore, they are themselves unsafe. Hence, the constructed state $\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}$ is either an unsafe state, or if it is safe, it remains boundary. In the former case, the mechanism has succeeded in its objective of reaching a new part of the unsafe region, as described above. In the second case, the mechanism provides another boundary safe state that can be used for the generation of new unsafe states through the second and the third mechanism. Finally, when using the third mechanism, we seek to add some processes to a boundary safe state $\mathbf{a}$, in order to obtain a state $\mathbf{y}$ such that $g(\mathbf{y}) \subseteq \tau_{\mathbf{a}}$. Thus, any enabled transition at $\mathbf{y}$ leads to a state that dominates a state in $U(\mathbf{a})$; hence to an unsafe state. Therefore, $\mathbf{y}$ is also unsafe.

The following definitions provide a more formal characterization for the second and the third mechanisms.

Definition 3: Consider a pair of boundary safe states $\mathbf{a}_1$ and $\mathbf{a}_2$. The pair $(\mathbf{a}_1, \mathbf{a}_2)$ is "*combinable*" *iff* (i) $\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}$ satisfies Equation 1, and (ii) $\tau_{\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}} \equiv \{\tau_{\mathbf{a}_1} \cup \tau_{\mathbf{a}_2}\} \cap g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}) \neq \emptyset$.

Example (cont.) Consider the pair of boundary safe states $\mathbf{a}_2$ and $\mathbf{a}_3$ in Table IV. Assessing their combinability, we can see that $\lambda_{\{\mathbf{a}_2,\mathbf{a}_3\}} = \{\Xi_{11}, \Xi_{21}, \Xi_{31}\}$, with the enabled edges $g(\lambda_{\{\mathbf{a}_2,\mathbf{a}_3\}}) = \{\Xi_{11} \to \Xi_{12}, \Xi_{21} \to \Xi_{22}, \Xi_{31} \to \Xi_{32}\}$ and $\tau_{\lambda_{\{\mathbf{a}_2,\mathbf{a}_3\}}} = \{\Xi_{11} \to \Xi_{12}, \Xi_{31} \to \Xi_{32}\}$. Hence $\mathbf{a}_2$ and $\mathbf{a}_3$ are combinable. Moreover, as discussed in a previous step of this example, the state $\mathbf{u}_5 \equiv \lambda_{\{\mathbf{a}_2,\mathbf{a}_3\}}$ is an unsafe state.

Next, consider the pair of boundary safe states $\mathbf{a}_1$ and $\mathbf{a}_2$. $\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}} = \{\Xi_{11}, \Xi_{31}, \Xi_{41}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$, with the enabled edges $g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}) = \{\Xi_{11} \to \Xi_{12}, \Xi_{31} \to \Xi_{32}\}$ and $\tau_{\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}} = \{\Xi_{31} \to \Xi_{32}\}$. Hence $\mathbf{a}_1$ and $\mathbf{a}_2$ are combinable. Let $\mathbf{a}_5 \equiv \lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}$. Among the edges in $g(\mathbf{a}_5)$ edge $\Xi_{31} \to \Xi_{32}$ belongs to $\tau_{\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}}$, and therefore, it leads to unsafety. On the other hand, it can be checked that there is a transition sequence that starts with the edge $\Xi_{11} \to \Xi_{12}$ and leads from state $\mathbf{a}_5$ to the empty state $\mathbf{s}_0$. Hence, the constructed state $\mathbf{a}_5$ is a boundary safe state. □

Definition 4: Given a boundary safe state $\mathbf{a}$, define the set of states $Confine(\mathbf{a}, \tau_{\mathbf{a}})$ as follows: $\mathbf{x}' \in Confine(\mathbf{a}, \tau_{\mathbf{a}})$ iff (i) $\mathbf{x}' > \mathbf{a}$, (ii) $g(\mathbf{x}') \subseteq \tau_{\mathbf{a}}$, (iii) $\nexists \mathbf{y} < \mathbf{x}'$ that satisfies (i) and (ii).

Condition (iii) in Definition 4 eliminates non-minimal unsafe states. The next proposition shows that any state in $Confine(\mathbf{a}, \tau_{\mathbf{a}})$ is an unsafe state.

Proposition 2: If $\mathbf{x}' \in Confine(\mathbf{a}, \tau_{\mathbf{a}})$, then $\mathbf{x}'$ is an unsafe state.

Proof: Consider a transition $t_1 \in g(\mathbf{x}')$. Definition 4 implies that $t_1 \in \tau_{\mathbf{a}}$. Hence, firing t_1 at $\mathbf{a}$ leads to an unsafe state $\mathbf{u}_1$. By Definition 4 again, $\mathbf{x}' > \mathbf{a}$. Therefore, firing t_1 at $\mathbf{x}'$ leads to a state that dominates $\mathbf{u}_1$, and therefore, to an unsafe state. Since t_1 was chosen arbitrarily among the transitions of $g(\mathbf{x}')$, it follows that all the enabled transitions at $\mathbf{x}'$ lead to unsafety. Hence, $\mathbf{x}'$ is an unsafe state. □

Example (cont.) Consider the boundary safe state $\mathbf{a}_3$ in Table IV. $\tau_{\mathbf{a}_3} = \{\Xi_{11} \to \Xi_{12}\}$ and $g(\mathbf{a}_3) = \{\Xi_{11} \to \Xi_{12}, \Xi_{21} \to \Xi_{22}\}$. Hence, to apply $Confine(\mathbf{a}_3, \tau_{\mathbf{a}_3})$, we need to block the edge $\Xi_{21} \to \Xi_{22}$. This can be done using any of the states in $BlockEd[\Xi_{21} \to \Xi_{22}]$. Combining each of the states $\mathbf{es}_5$, $\mathbf{es}_6$, $\mathbf{es}_7$, and $\mathbf{es}_8$ (c.f. Table III) with $\mathbf{a}_3$ results respectively in the states $\mathbf{y}_1 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{12}\}$, $\mathbf{y}_2 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{22}\}$, $\mathbf{y}_3 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{32}\}$, and $\mathbf{y}_4 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{41}\}$. It can be seen that $\mathbf{y}_1 > \mathbf{u}_3$ and that $\mathbf{y}_3 > \mathbf{u}_2$, where states $\mathbf{u}_2$ and $\mathbf{u}_3$ are defined in Table IV. On the other hand, state $\mathbf{y}_2$ enables edge $\Xi_{22} \to \Xi_{23}$. Thus, to block this edge, $\mathbf{y}_2$ is combined with $\mathbf{es}_9 \in BlockEd[\Xi_{22} \to \Xi_{23}]$, resulting in the state $\mathbf{y}_5 \equiv \{\Xi_{11}, \Xi_{21}, \Xi_{22}, \Xi_{31}\}$. Again, it can be seen that $\mathbf{y}_5 > \mathbf{u}_4$, where $\mathbf{u}_4$ is the corresponding state in Table IV. Hence, it follows from the above discussion that $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$, and $\mathbf{y}_5$ do not belong to $Confine(\mathbf{a}_3, \tau_{\mathbf{a}_3})$. On the other hand, at state $\mathbf{y}_4$, $g(\mathbf{y}_4) = \{\Xi_{11} \to \Xi_{12}\} = \tau_{\mathbf{a}_3}$, and $\mathbf{y}_4$ does not dominate any unsafe state. Hence $\mathbf{u}_6 = \mathbf{y}_4$ is a minimal unsafe state, and $Confine(\mathbf{a}_3, \tau_{\mathbf{a}_3}) = \{\mathbf{u}_6\}$.

B. The proposed algorithm

We start the presentation of the proposed algorithm for the enumeration of $\bar{S}_{r\bar{s}}$ by introducing two supporting subroutines for this algorithm. The first one is the procedure Insert_Non_Min(U, Q_1, Q_2) and it is used for removing the non-minimal unsafe states that are generated during the course of the execution of the algorithm. More specifically, procedure Insert_Non_Min(U, Q_1, Q_2) is invoked to insert the states in U into Q_1, while removing any non-minimal state vectors from $Q_1 \cup Q_2$. A state $\mathbf{u} \in U$ is inserted into Q_1 *iff* the set $Q_1 \cup Q_2$ does not contain any

Procedure 2 Confine($\mathbf{a}$, $\tau_{\mathbf{a}}$)

Input: $\mathbf{a}$, $\tau_{\mathbf{a}}$
Output: U^*

1: $workingQueue \leftarrow \mathbf{a}$;
2: **while** $workingQueue \neq \emptyset$ **do**
3: $\quad \mathbf{p} \leftarrow dequeue(workingQueue)$
4: $\quad e^* \leftarrow getAnEnabledEdge(\mathbf{p}, \tau_{\mathbf{a}})$
5: $\quad$ **if** $e^* = 0$ **then**
6: $\quad\quad$ Insert $\mathbf{p}$ into U^*
7: $\quad$ **else**
8: $\quad\quad$ **for all** $\mathbf{x} \in BlockEd[e^*]$ **do**
9: $\quad\quad\quad$ **if** Feasible($\lambda_{\{\mathbf{x},\mathbf{p}\}}$) **then**
10: $\quad\quad\quad\quad$ Insert $\lambda_{\{\mathbf{x},\mathbf{p}\}}$ into $workingQueue$
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **end for**
13: $\quad$ **end if**
14: **end while**
15: **return** U^*

state dominated by $\mathbf{u}$. Furthermore, if $\mathbf{u}$ is dominated by a state $\mathbf{x} \in Q_1 \cup Q_2$, $\mathbf{x}$ is removed from $Q_1 \cup Q_2$.

The second subroutine supports the $Confine$ operation that was introduced in Definition 4. The algorithmic steps of the $Confine$ operation are depicted in Procedure 2, and the logic that is implemented by this procedure is very similar to that of Lines 12-27 in Procedure 1. But instead of seeking to block enabled processing stages, the $Confine$ procedure seeks to block the enabled edges that do not belong to $\tau_{\mathbf{a}}$. In particular, the function $getAnEnabledEdge$ at Line 4 returns an enabled edge at state $\mathbf{p}$ that does not belong to $\tau_{\mathbf{a}}$. If no such edge exists, the function returns a value of 0, and thus, $\mathbf{p}$ is added to U^* at Line 6. On the other hand, if an enabled edge e^* is found, then the code of Lines 8-11 generates and processes every feasible combination of state $\mathbf{p}$ with the minimal states blocking e^*.

Now, we present the main content of the section, which is the algorithm used to enumerate the minimal reachable unsafe states. The complete logic of this algorithm is detailed in Algorithm 3. Algorithm 3 employs the queue Q to store unprocessed unsafe states, the list $\dot{U}$ to store processed unsafe states, and the hash table $\dot{A}$ to store boundary safe states. The algorithm starts by enumerating all the minimal reachable deadlock states using Procedure 1, and adds the returned states to Q. For each state $\mathbf{u}$ in Q, $\mathbf{u}$ is traced back by one transition in Line 6. Then, in Line 7, the states generated in Line 6 are partitioned into the sets $Safe\_Prev$ and $Unsafe\_Prev$ (i.e., the safe and unsafe state subsets of $Prev(\mathbf{u})$), using standard reachability analysis w.r.t. the target state $\mathbf{s}_0$. In Line 8, the elements of $Unsafe\_Prev$ are inserted into Q to be processed later. On the other hand, the function $Combine$ in Line 12 returns $\lambda_{\mathbf{a},\dot{\mathbf{a}}}$ if $\mathbf{a}$ and $\dot{\mathbf{a}}$ are combinable according to Definition 3. Otherwise, it returns $\emptyset$. Hence, in Lines 9-17, for each state $\mathbf{a} \in Safe\_Prev$, the $Combine$ function is applied with every state $\dot{\mathbf{a}} \in \dot{A}$, and the result is inserted in Z_a. In Line 14, Z_a is partitioned using standard reachability analysis into its subset of safe states, $Safe(Z_a)$, and its subset of unsafe states, $Unsafe(Z_a)$. As explained in the opening part of this section, the states of $Safe(Z_a)$ are boundary safe states by construction; hence, they are inserted into $\dot{A}$ at Line 15. Whenever a boundary state $\mathbf{a'}$ is inserted into $\dot{A}$, we check first if $\exists \dot{\mathbf{a}} \in \dot{A}$ s.t. $\mathbf{a'} = \dot{\mathbf{a}}$; in this case $\tau_{\dot{\mathbf{a}}}$ is updated to $\tau_{\dot{\mathbf{a}}} \cup \tau_{\mathbf{a'}}$, and $\mathbf{a'}$ is discarded. On the other hand, the states of $Unsafe(Z_a)$ are unsafe; hence they are inserted into Q. If Q is empty, then we apply the $Confine$ operation described in Procedure 2 to every state in $\dot{A}$ that has not been subjected to this operation yet.

Algorithm 3

Input: A RAS instance Φ
Output: the list $\dot{U}$ containing all the reachable minimal unsafe states of Φ

1: $\dot{U}, \dot{A} \leftarrow \emptyset$; $k \leftarrow 0$;
2: $Q \leftarrow EnumMinReachDeadlocks(\Phi)$
3: **while** $Q \neq \emptyset$ or $k < |\dot{A}|$ **do**
4: $\quad$ **if** $Q \neq \emptyset$ **then**
5: $\quad\quad \mathbf{u} \leftarrow$ dequeue(Q);
6: $\quad\quad Prev(\mathbf{u}) \leftarrow$ Backtrace($\mathbf{u}$);
7: $\quad\quad (Safe\_Prev,\ Unsafe\_Prev) \leftarrow$ Classify($Prev(\mathbf{u})$)
8: $\quad\quad$ Insert_Non_Min($Unsafe\_Prev$, Q, $\dot{U}$)
9: $\quad\quad$ **for** each $\mathbf{a} \in Safe\_Prev$ **do**
10: $\quad\quad\quad Z_a \leftarrow \emptyset$
11: $\quad\quad\quad$ **for all** $\dot{\mathbf{a}} \in \dot{A}$ **do**
12: $\quad\quad\quad\quad Z_a \leftarrow Z_a \cup Combine(\dot{\mathbf{a}}, \mathbf{a})$
13: $\quad\quad\quad$ **end for**
14: $\quad\quad\quad (Safe(Z_a), Unsafe(Z_a)) \leftarrow Classify(Z_a)$
15: $\quad\quad\quad$ Insert $\mathbf{a}$, $Safe(Z_a)$ into $\dot{A}$
16: $\quad\quad\quad$ Insert_Non_Min($Unsafe(Z_a)$, Q, $\dot{U}$)
17: $\quad\quad$ **end for**
18: $\quad\quad$ Insert_Non_Min($\mathbf{u}$, $\dot{U}$, Q)
19: $\quad$ **else**
20: $\quad\quad$ **while** $k < |\dot{A}|$ **do**
21: $\quad\quad\quad \mathbf{a}_k \leftarrow \dot{A}[k++]$;
22: $\quad\quad\quad U^* \leftarrow Confine(\mathbf{a}_k,\ \tau_{\mathbf{a}_k})$
23: $\quad\quad\quad$ Insert_Non_Min(U^*, Q, $\dot{U}$)
24: $\quad\quad$ **end while**
25: $\quad$ **end if**
26: **end while**
27: Remove_Unreachable($\dot{U}$)
28: **return** $\dot{U}$

Complexity Considerations In [7], it is shown that the overall complexity of Algorithm 3 can be characterized as $\mathcal{O}(m^{|\mathcal{E}|+1} \cdot \eta^{C \cdot |\mathcal{E}|} \cdot \xi \cdot |\mathcal{E}| \cdot 2^{\alpha \cdot |\mathcal{E}|} + |S_r|)$. The quantity α that appears in this expression denotes the total number of unsafe states that are added to list Q throughout the entire execution of the algorithm; the remaining quantities appearing in this expression are defined in the same way as in the complexity characterization of Procedure 1.

Furthermore, the role of $|S_r|$ in the above expression is moderated by remarks similar to those that apply in

the complexity analysis of Procedure 1. More specifically, the (co-)reachability analysis that is performed in Lines 7, 14 and 27, is implemented using depth-first search supported with hash tables to mark visited states, and the empirical complexity of the corresponding computation is much smaller than that suggested by the term $|S_r|$. In fact, we can forego the co-reachability analysis of Lines 7 and 14. This variant of the proposed algorithm will assume that each state $\mathbf{x}$ that results from backtracing (Line 6) or *Combine* (Lines 10-13) is a safe state, unless $g(\mathbf{x}) = \tau_{\mathbf{x}}$, in which case all the enabled events ($g(\mathbf{x})$) lead to unsafety. To describe the algorithm modifications in this alternative approach, let $\mathbf{x}_1$ be a state that results from backtracing $\mathbf{u}$, and let $\mathbf{x_2}$ be a state that results from the *Combine* operation. If $g(\mathbf{x_1}) = \tau_{\mathbf{x_1}}$, then $\mathbf{x}_1$ is added to $Unsafe\_Prev$. Otherwise, it is added to $Safe\_Prev$, i.e., it is assumed to be safe. Similarly, if $g(\mathbf{x_2}) = \tau_{\mathbf{x_2}}$, then $\mathbf{x}_2$ is added to $Unsafe(Z_a)$. Otherwise, it is added to $Safe(Z_a)$. It was shown in [11] that this alternative approach enumerates correctly all the minimal reachable unsafe states.

In our experimental work, we tried both approaches, and it turned out that the first one – i.e., the one presented in the main statement of Algorithm 3 – is much more computationally efficient. This can be explained by the following two reasons: (i) As already mentioned, the states that are evaluated for co-reachability, are either minimal unsafe states or few steps away from minimal unsafe states. Thus, the examined states are characterized by a low process content, and the applied depth-first search method is computationally very efficient. (ii) At the same time, as revealed in the sequel, the empirical complexity of Algorithm 3 is highly dependent on $|\dot{A}|$. In the second approach that is outlined above, the list $\dot{A}$ grows much larger because it contains both safe and unsafe states and their combinations, whereas in the first approach, $\dot{A}$ contains only safe states.

From the above discussion, it can be concluded that the computational complexity of Algorithm 3 is particularly sensitive to the capacities of the resource types, the number of the distinct event types, and the number of enumerated unsafe states. In addition, as remarked in the previous paragraph, the computational experiments presented in Section V will reveal that, statistically, the algorithm running time is mostly correlated with the size of list $\dot{A}$.

Example (cont.) As it was previously mentioned, the application of Procedure 1 on the considered RAS results in the minimal deadlock states $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$, and $\mathbf{u}_4$. Table IV depicts the unsafe states generated by Algorithm 3,and the results of backtracing these states. The algorithm starts by inserting the four identified minimal deadlocks, $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$, and $\mathbf{u}_4$, into Q. Backtracing these states results in adding the states $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$, and $\mathbf{a}_4$ to $\dot{A}$. As illustrated in an earlier part of this example, $Combine(\mathbf{a}_1, \mathbf{a}_2)$ results in adding $\mathbf{a}_5 = \{\Xi_{11}, \Xi_{31}, \Xi_{41}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$ to $Safe(Z_a)$ and consequently to $\dot{A}$. Similarly, $Combine(\mathbf{a}_1, \mathbf{a}_4)$ results in adding $\mathbf{a}_6 = \{\Xi_{21}, \Xi_{31}, \Xi_{41}, \Xi_{43}, \Xi_{44}, 2\Xi_{51}\}$ to $\dot{A}$. On the other hand, $Combine(\mathbf{a}_1, \mathbf{a}_3)$ results in adding the unsafe state $\mathbf{u}_7$ to $Unsafe(Z_a)$ and consequently to Q. Similarly, $Combine(\mathbf{a}_2, \mathbf{a}_3)$ results in adding the unsafe state $\mathbf{u}_5$ to Q. Combining $\mathbf{a}_4$ with each of $\mathbf{a}_2$ and $\mathbf{a}_3$ yields $\mathbf{u}_5$ again. Hence, $\mathbf{u}_5$ can be constructed by applying the *Combine* operation to any pair of $\{\mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4\}$. On the other hand, combining each of $\mathbf{a}_5$ and $\mathbf{a}_6$ with the other members of $\dot{A}$ does not yield any minimal unsafe states or boundary safe states. Finally, $\mathbf{u}_5$ and $\mathbf{u}_7$ can not be traced back. Thus, after the sixth iteration, $Q = \emptyset$ and $\dot{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_6\}$. As illustrated in an earlier part of this example, applying the $Confine$ operation to $\mathbf{a}_3$ adds the state $\mathbf{u}_6$ to Q. It can be seen that $\mathbf{u}_6 < \mathbf{u}_7$. Thus, $\mathbf{u}_7$ is a non-minimal unsafe state and it is removed from $\dot{U}$ by the procedure Insert_Non_Min. On the other hand, the application of the $Confine$ operation to the other elements of $\dot{A}$ does not generate any minimal unsafe state. Since the newly generated state $\mathbf{u}_6$ can not be traced back, the algorithm terminates.

Before we conclude the example, we also consider the alternative approach that skips the co-reachability analysis of the generated states. After the generation of $\mathbf{u}_5$ by $Combine(\mathbf{a}_2, \mathbf{a}_3)$, $\mathbf{u}_5$ is inserted into $\dot{A}$ because $g(\lambda_{\{\mathbf{a}_2,\mathbf{a}_3\}}) \not\subseteq \tau_{\lambda_{\{\mathbf{a}_2,\mathbf{a}_3\}}}$. Also, $\tau_{\mathbf{u}_5} = \{\Xi_{11} \to \Xi_{12}, \Xi_{31} \to \Xi_{32}\}$. In the next iteration, $\mathbf{u}_5$ is generated three more times as a result of $Combine(\mathbf{a}_4, \mathbf{a}_2)$, $Combine(\mathbf{a}_4, \mathbf{a}_3)$ and $Combine(\mathbf{a}_4, \mathbf{u}_5)$. But only at $\mathbf{u}_5 = \lambda_{\{\mathbf{a}_4,\mathbf{u}_5\}}$ we get $g(\lambda_{\{\mathbf{a}_4,\mathbf{u}_5\}}) = \tau_{\lambda_{\{\mathbf{a}_4,\mathbf{u}_5\}}}$; at that point, $\mathbf{u}_5$ is eventually classified as an unsafe state. Thus, it took the alternative approach one more iteration, one more element in $\dot{A}$, and three more *Combine* operations to realize that $\mathbf{u}_5$ is unsafe.

C. Proving the correctness of Algorithm 3

In this part we establish the correctness of Algorithm 3. More specifically, first we show that Algorithm 3 terminates in a finite number of steps, and subsequently, we establish that upon its termination, the algorithm will have enumerated correctly all the minimal unsafe states of its input RAS Φ.

Proposition 3: When applied on any well-defined instance Φ from the RAS class considered in this work, Algorithm 3 will terminate in a finite number of steps.

Proof: We first notice that the algorithm terminates when there are no more unsafe states to be traced back nor any states to be confined. As illustrated in the previous subsection, a state $\mathbf{s}$ is inserted in $L = \dot{U} \cup Q$ via the procedure Insert_Non_Min, only if it does not dominate an element of L. Furthermore, a state $\mathbf{s}'$ that is deleted from L due to such dominance considerations, will never enter L again. But then, the finiteness of the computation of Algorithm 3 results from the above remarks and the finiteness of the underlying state space. □

To establish that the algorithm enumerates correctly all the minimal reachable unsafe states, we start by observing that Theorem 1 establishes that Procedure 1 enumerates correctly all the minimal deadlock states. Thus, our goal is to show that Algorithm 3 enumerates also all the deadlock-free minimal unsafe states. The main idea is to show that a minimal unsafe state is missed only if a "next" minimal unsafe state – i.e., a member of the set $nextMin$ introduced in Definition 2 – is also missed by the algorithm. Then, the algorithm correctness can be based upon the elimination of the possibility of having cyclical dependencies in the relationship that is defined by the $nextMin$ operator. In particular, it will be shown that every chain of dependencies among the minimal unsafe states that is defined by operator $nextMin$, will end up in a minimal deadlock, and the sought result will be obtained from Theorem 1.

Proposition 4: A deadlock-free minimal unsafe state $\mathbf{u}$ is missed by Algorithm 3 only if an entire set of states $nextMin(\mathbf{u}, e_i)$ is missed by the algorithm.

Proof: Suppose that the algorithm generates at least one element from each set $nextMin(\mathbf{u}, e_i)$, and w.l.o.g further assume that $\{\mathbf{h}_1, \ldots, \mathbf{h}_k\}$ are minimal unsafe states and $\{\mathbf{h}_{k+1}, \ldots, \mathbf{h}_K\}$ are non-minimal unsafe states, $0 \leq k \leq K$. By the working assumption, we can consider that the algorithm generates the states $\{\mathbf{h}_1, \ldots, \mathbf{h}_k, \mathbf{z}_{k+1,1}, \ldots, \mathbf{z}_{K,1}\}$, where the notation of Definition 2 has been applied. The set $Y = \{\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(k)}, \mathbf{s}_{k+1,1}, \ldots, \mathbf{s}_{K,1}\}$ collects the corresponding predecessors of the states in the set $\{\mathbf{h}_1, \ldots, \mathbf{h}_k, \mathbf{z}_{k+1,1}, \ldots, \mathbf{z}_{K,1}\}$, that are obtained by tracing back from each of the elements of the set $\{\mathbf{h}_1, \ldots, \mathbf{h}_k, \mathbf{z}_{k+1,1}, \ldots, \mathbf{z}_{K,1}\}$ respectively across the edges $e_1, \ldots, e_K$. It can be seen that $\mathbf{u}^{(1)} = \ldots = \mathbf{u}^{(k)} = \mathbf{u}$, $\tau_{\mathbf{u}^{(i)}} = \{e_i\}$, $i \leq k$, and $\tau_{\mathbf{s}_{i1}} = \{e_i\}$, $i > k$. If $k \geq 1$, then $\mathbf{u}$ is the predecessor of $\mathbf{h}_1$ that is obtained by tracing back e_1; hence $\mathbf{u}$ is generated by Lines 6-7 of Algorithm 3, and we are done. On the other hand, if $k = 0$, then $\forall i = 1:K$, $\mathbf{s}_{i1}$ is a boundary safe state obtained by tracing back e_i from the unsafe state $\mathbf{z}_{i1}$. Definition 2 and its accompanying discussion reveal that $\mathbf{s}_{i1} < \mathbf{u}$, $\forall i$. Hence, $\mathbf{a_2} = \lambda_{\{\mathbf{s}_{11}, \mathbf{s}_{21}\}} \leq \mathbf{u}$. Therefore, we can infer that e_1 and e_2 are enabled at $\mathbf{a}_2$. Hence, $\mathbf{s}_{11}$ and $\mathbf{s}_{21}$ are combinable, and $\tau_{\mathbf{a_2}} = (\tau_{\mathbf{s}_{i1}} \cup \tau_{\mathbf{s}_{i2}}) \cap g(\mathbf{a}_2) = \{e_1, e_2\}$. If $\mathbf{a}_2$ is unsafe, then, by the minimality of $\mathbf{u}$, $\mathbf{a}_2 = \mathbf{u}$. Hence, $\mathbf{u}$ is added to $Unsafe(Z_a)$ at Line 14, and consequently to Q at Line 16. Otherwise, $\mathbf{a}_2$ is added to $Safe(Z_a)$ at Line 14, and consequently to $\dot{A}$ at Line 15. In a subsequent iteration, Line 12 will find that $\mathbf{s}_{31}$ and $\mathbf{a}_2$ are also combinable, yielding $\mathbf{a}_3 = \lambda_{\{\mathbf{s}_{31}, \mathbf{a}_2\}}$. The same argument is repeated until either $\mathbf{u}$ or $\mathbf{a}_K \equiv \lambda_{\{\mathbf{s}_{11}, \ldots, \mathbf{s}_{K1}\}} < \mathbf{u}$ is generated. It also holds that $\tau_{\mathbf{a}_K} = \{e_1, \ldots, e_K\}$. Hence, we can see that, if $\mathbf{u}$ has not been generated during the $K-1$ "combine" iterations that generate $\mathbf{a}_K$, by the definition of the $Confine$ operation, it will be contained in the set of states returned by $Confine(\mathbf{a}_K, \tau_{\mathbf{a}_K})$, and it is thereby generated by Line 22. □

The next lemma eliminates the possibility of having cyclical dependencies in the relationship among minimal unsafe states that is defined by the $nextMin$ operator.

Lemma 2: Consider a sequence of minimal unsafe states $\{\mathbf{u}_k\}_{k=1}^{l}$ such that $\mathbf{u}_k \in nextMin(\mathbf{u}_{k-1})$, $k = 2:l$. Then $\mathbf{u}_1 \notin nextMin(\mathbf{u}_l)$.

Proof: Let $e_1 = (\Xi_{ij}, \Xi_{ij^*})$ be the transition edge between $\mathbf{u}_1$ and its next state in the considered sequence. Let $\vec{\Xi}_{ij}$ denote the processing stages in $\mathcal{G}_i$ that are co-reachable to Ξ_{ij}; by definition, $\Xi_{ij} \in \vec{\Xi}_{ij}$. The acyclicity of $\mathcal{G}_i$ implies that $\sum_{\Xi_{ij'} \in \vec{\Xi}_{ij}} \mathbf{u}_2(\Xi_{ij'}) \leq \sum_{\Xi_{ij'} \in \vec{\Xi}_{ij}} \mathbf{u}_1(\Xi_{ij'}) - 1$. Due to the absence of loading events from the considered sequence and the acyclicity of $\mathcal{G}_i$, there does not exist a state $\mathbf{u}_k$, $k \geq 2$, in this sequence where $\sum_{\Xi_{ij'} \in \vec{\Xi}_{ij}} \mathbf{u}_k(\Xi_{ij'})$ is restored to its original level at $\mathbf{u}_1$. Therefore, $\mathbf{u}_1 \notin nextMin(\mathbf{u}_l)$. □

Now we are ready to present the main result regarding the correctness of Algorithm 3.

Theorem 2: Algorithm 3 generates all the minimal reachable unsafe states of its input RAS Φ.

Proof: Assume that the minimal reachable unsafe state $\mathbf{u}_1$ is missed by the algorithm. Proposition 4 ensures that this is due to missing another minimal unsafe state $\mathbf{u}_2 \in nextMin(\mathbf{u}_1)$. Repeating the same argument with $\mathbf{u}_2$, we get that the algorithm missed the minimal unsafe states $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_l$, where $\mathbf{u}_k \in nextMin(\mathbf{u}_{k-1})$. But Lemma 2 eliminates the possibility of having a cycle in this path and l is bounded from above by $|\bar{S}_{\bar{s}}|$. Hence, $\mathbf{u}_l$ is a missed minimal deadlock state. $\mathbf{u}_l$ is also reachable, by the specification of the sequence $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_l$, and the fact that, in the considered RAS class, $\mathbf{s} \in S_r \wedge \mathbf{s}' \leq \mathbf{s} \Longrightarrow \mathbf{s}' \in S_r$. But when combined with Lines 2 and 18 of Algorithm 3, the above results contradict Theorem 1.

V. Computational results

In this section we report the results from a series of computational experiments, in which we applied Algorithm 3 upon a number of randomly generated instantiations of the RAS class that was defined in Section II. Moreover, we compare the performance of this algorithm with the algorithm of [9] that relies on the exhaustive enumeration of the underlying state space. Each of the generated instances was further specified by:

- The number of resource types in the system; the range of this parameter was between 3 and 16.
- The capacities of the resource types in the system; the range of this parameter was between 1 to 4.
- The number of process types in the system; the range of this parameter was between 3 and 5.
- The structure of the process graphs $\mathcal{G}_i$ and of the resource request vectors that were supported by the resource allocation function $\mathcal{A}$. In terms of the first attribute, the generated instances contain RAS where all their processes presented a simple linear structure, and RAS where some of their processes possessed

routing flexibility (Disjunctive RAS). In terms of the second attribute mentioned above, the generated instances contain RAS with single-type resource allocation and instances with conjunctive resource allocation (Conjunctive RAS).

- The number of processing stages in each process; the range of this parameter was between 3 and 16. Furthermore, in order to remain consistent with the RAS structure defined in Section II, no processing stage has a zero resource-allocation vector.

The employed RAS generator was encoded and compiled in C++. For each generated RAS instance, Φ, we enumerated the reachable minimal unsafe states by applying, both, (i) the algorithm of [9] and (ii) Algorithm 3. We imposed a hard limit of 48 hours for the solution of these instances. All our computational 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. Both Algorithm 3 and the algorithm of [9] were encoded in C++, compiled and linked by the GNU g++ compiler under Unix. [5]

Table V reports a representative sample of the results that we obtained in our experiments. The first section of the table is for RAS instances with simple linear structure and single-type resource allocation. The second section is for RAS instances with simple linear structure and conjunctive resource allocation. Finally, the last section of the table is for RAS instances with routing flexibility and conjunctive resource allocation. Columns 1 – 3 in Table V report, respectively, the cardinalities of the set of reachable states, the set of the reachable unsafe states, and the set of reachable minimal unsafe states. Column 4 (γ) reports the number of minimal unsafe states generated by Algorithm 3 without performing the reachability evaluation of the generated states (Line 27). Column 5 (α) reports the total number of unsafe states added to Q throughout the course of the execution of Algorithm 3. Column 6 reports the cardinality of the list $\dot{A}$ at the end of the execution of Algorithm 3. Column 7 (t_o) reports the amount of computing time (in seconds) that was required to compute the minimal unsafe states through the algorithm of [9]. Finally, Columns 8 (t_n) and 9 (t_r) report, respectively, the amount of time (in seconds) spanned by Lines 1-26 and Line 27 of Algorithm 3. The rows that have some unreported entries correspond to RAS instances for which the algorithm of [9] did not conclude within 48 hours. The reported cases are ordered in increasing magnitude of the corresponding t_o.

As mentioned in the previous paragraph, the data provided in Table V are representative of a more extensive sample that was collected in our experiments. The perusal of these data reveals very clearly the computational efficacy of Algorithm 3 in computing the reachable minimal unsafe states, especially when compared to the algorithm of [9]. The presented data also indicate that (i) the computational complexity of Algorithm 3 is mostly dependent on the cardinality of the set $\dot{A}$, and that (ii) Algorithm 3 demonstrates more computational efficacy compared to the algorithm of [9] for RAS configurations with routing flexibility; this last effect can be explained by the relative scarcity of unsafe states in RAS with routing flexibility. Finally, the last column of Table V highlights the fact that, in spite of the high theoretical worst-case complexity of the computation in Line 26 of Algorithm 3 (assessing state reachability), the practical complexity of this computation is very benign.

[5]The implementation of the algorithms and the RAS configurations used for these experiments can be obtained by contacting the authors.

VI. Conclusion

This work has presented a novel algorithm for the enumeration of the set of reachable minimal unsafe states for the complex RAS that were described in Section II. A defining feature of this algorithm is that it avoids the complete enumeration of the underlying state space. Furthermore, a series of computational experiments has manifested the superiority of this algorithm w.r.t. the existing approaches that rely on a complete state space enumeration.

A companion research endeavor to this work has sought to adapt the presented algorithm to some RAS classes whose state space is infinite. The dynamics of these classes cannot be modeled by a Finite State Automaton, and therefore, their behavioral analysis is not amenable to the elementary, enumerative techniques that can provide the basic characterizations for deadlock and deadlock avoidance along the lines discussed in Section II. Some relevant results are reported in [12].

References

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

[2] H. C. Chen and C. L. Chen. On minimal elements of upward-closed sets. *Theoretical Computer Science*, 410:2442–2452, 2009.

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

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

[5] R. Cordone and L. Piroddi. Monitor optimzation in Petri net control. In *Proceedings of the 7th IEEE Conf. on Automation Science and Engineering*, pages 413–418. IEEE, 2011.

[6] 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.

[7] A. Nazeem. *Designing parsimonious representations of the maximally permissive deadlock avoidance policy for complex resource allocation systems through classification theory.* PhD thesis, Georgia Tech, Atlanta, GA, 2012.

[8] A. Nazeem and S. Reveliotis. Designing maximally permissive deadlock avoidance policies for sequential resource allocation systems through classification theory. In *Proceedings of the 7th IEEE Conf. on Automation Science and Engineering*, pages 405–412. IEEE, 2011.

TABLE V
A sample of our computational results regarding the efficacy of Algorithm 3

$\|S_r\|$	$S_{r\bar{s}}$	$\bar{S}_{r\bar{s}}$	γ	α	$\hat{A}$	t_o	t_n	t_r
635572	349546	1654	2807	5912	8548	1225	69	0
1463878	458127	284	540	868	2124	1418	6	0
404542	237591	3475	6419	12848	9210	2002	73	0
1508301	573055	686	1076	3017	5792	2633	35	0
799071	401974	7568	11753	20736	15534	4759	199	0
1743534	887064	2840	4130	18516	33020	13270	555	0
1659342	987461	10468	21410	45139	39937	16690	1447	1
1962454	1098441	7171	11002	24859	42639	21001	989	0
4488904	2748034	658	2088	8384	23095	29765	259	0
3436211	1789990	36874	68317	135762	91164	105074	5863	1
14158338	4977105	35022	54327	116337	136038	229759	31933	10
		30110	59864	169071	234629		36305	4
		33744	72935	193572	274553		44222	5
		22157	43718	129210	190378		23382	7
		20891	39738	165495	299266		51295	4
		16910	29819	82621	138687		14851	3
646746	175449	779	849	1815	908	167	2	0
1767552	406300	559	660	1050	4729	541	20	0
915716	412871	849	1103	6065	23837	1551	509	0
738720	644955	4162	4614	10752	8646	2251	327	0
2939463	1328411	3655	5928	10619	11354	5060	193	0
2430581	867647	14185	16547	29722	38765	6774	1112	0
1962454	1098441	7171	11002	24859	42639	12087	1073	0
1712672	977484	7214	10617	32861	54891	13027	1782	0
3554952	2366757	2533	3659	17855	19856	21082	614	0
6051299	2300151	5069	6837	24373	65602	34650	1865	0
24430444	7268845	9783	11307	32410	90665	114614	5339	1
		3145	6998	20843	27747		550	0
		16910	29819	82621	138687		12649	3
		33744	72935	193572	274553		45550	5
		4046	4393	6592	7585		92	0
		428	585	2957	31425		1098	0
571536	0	0	0	0	0	19	0	0
2171880	13992	28	28	32	126	48	0	0
1229688	64968	241	271	279	373	105	0	0
2693250	76536	79	79	87	37	130	0	0
3416000	280000	1	1	8	1	158	0	0
2448000	410112	9	9	9	50	580	0	0
1663534	230889	2665	5857	6966	5030	801	173	0
2340408	511277	1522	2620	3376	3760	4455	37	0
7885856	605977	2323	2628	2710	4882	4871	19	0
		17394	24192	27412	25199		1001	4
		381	516	691	2729		77	0
		1215	1245	1294	3490		15	0
		24	31	31	77		747	0
		6635	8543	13382	42510		665	2
		792	1975	2000	2920		6	4
		2468	6035	7444	12777		88	19

[9] A. Nazeem and S. Reveliotis. A practical approach for maximally permissive liveness-enforcing supervision of complex resource allocation systems. *IEEE Trans. on Automation Science and Engineering*, 8:766–779, 2011.

[10] A. Nazeem and S. Reveliotis. Designing maximally permissive deadlock avoidance policies for sequential resource allocation systems through classification theory: the non-linear case. *IEEE Trans. on Automatic Control*, 57:1670–1684, 2012.

[11] A. Nazeem and S. Reveliotis. An efficient algorithm for the enumeration of the minimal unsafe states in complex resource allocation systems. In *Proceedings of the 8th IEEE Conf. on Automation Science and Engineering*. IEEE, 2012.

[12] A. Nazeem and S. Reveliotis. Maximally permissive deadlock avoidance for resource allocation systems with R/W-locks. In *Proceedings of WODES 2012*. IFAC, 2012.

[13] A. Nazeem, S. Reveliotis, Y. Wang, and S. Lafortune. Designing maximally permissive deadlock avoidance policies for sequential resource allocation systems through classification theory: the linear case. *IEEE Trans. on Automatic Control*, 56:1818–1833, 2011.

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

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

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

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