Discrete Event Dynamic Systems: Theory and Applications manuscript No.
(will be inserted by the editor)

Maximally Permissive Deadlock Avoidance for Resource Allocation Systems with R/W-Locks

Ahmed Nazeem · **Spyros Reveliotis.**

Received: 01-24-2013 / Accepted: XX-XX-XXXX

Abstract This paper extends the existing theory on maximally permissive liveness-enforcing supervision of resource allocation systems (RAS) so that it can handle RAS with reader / writer (R/W-) locks. A key challenge that is posed by this new RAS class stems from the fact that the underlying state space is not necessarily finite. We effectively address this obstacle by taking advantage of special structure that exists in the set of inadmissible states and enables a finite representation of this set through its minimal elements.

Keywords Reader/Writer Locks · Deadlock · Parallel Programming · Supervisory Control · Discrete Event Systems · Right-Closed Sets

1 Introduction

The problem of deadlock avoidance – or, liveness-enforcing supervision – for sequential resource allocation systems (RAS) has been studied extensively in the Discrete Event Systems (DES) literature. Some comprehensive expositions of the relevant results can be found in Reveliotis (2005); Zhou and Fanti (2004), and Li et al (2008). Many of these results were initially motivated by the supervisory control (SC) needs of flexibly automated production systems (e.g., Viswanadham et al (1990); Banaszak and Krogh (1990); Ezpeleta et al (1995); Reveliotis and Ferreira (1996); Fanti et al (1997)), but the theory was subsequently applied to additional applications, like the traffic management of automated transport systems (e.g., Reveliotis (2000); Wu and Zhou (2007); Roszkowska and Reveliotis (2008); Reveliotis and Roszkowska (2011)) and the management of the resource allocation in automated workflow systems (e.g., Park (2004)). More recently, the aforementioned theory has been employed and extended in order to address deadlock-related issues that arise in the context of multi-threaded programs, a programming paradigm that becomes more and more prevalent due

This work was partially supported by NSF grant CMMI-MES-0928231.

Ahmed Nazeem
United Airlines, Chicago, IL, USA
E-mail: ahmed.nazeem@united.com

Spyros Reveliotis
School of Industrial & Systems Engineering, Georgia Institute of Technology
E-mail: spyros@isye.gatech.edu

to the multi-core architectures that are promoted by the computer industry; the corresponding developments are collectively known as the "Gadara project".[1] More specifically, in the context of the Gadara project, a series of recent papers (Nazeem et al (2011); Wang et al (2009, 2010); Liao et al (2013b,a,c)) have studied the problem of deadlock avoidance in multi-threaded programs where the concurrently running threads coordinate their execution in their critical section through mutual exclusion locks (mutexes). From the standpoint of RAS theory, mutexes can be abstracted as reusable resources of *unit* capacity, and the program threads contesting for these resources, while trying to access their critical section, define the RAS processes. The resulting RAS models have enabled the development of SC policies that establish deadlock-free allocation of the aforementioned mutexes to the contesting processes in a way that ensures maximal permissiveness.[2] In Nazeem and Reveliotis (2011, 2012), the original results were extended to also handle the allocation of semaphores, which constitute resource types of multi-unit capacity.

In this work, we further extend the aforementioned results in order to address the problem of deadlock avoidance for multi-threaded programs involving reader / writer (R/W) locks, besides mutexes and semaphores. Reader-writer synchronization, as introduced by Courtois et al (1971), relaxes the constraint of mutual exclusion to permit more than one process to inspect a shared resource concurrently, as long as none of them changes its value. Thus, multiple threads can read from the shared resource simultaneously but a thread can write to the shared resource only if no other thread is writing to, or reading from, this resource. We shall characterize this effect by saying that when perceived as resources, R/W-locks work in two modes: (i) a "writing" mode where the resource capacity is equal to one, and (ii) a "reading" mode where the capacity is infinite. We shall refer to a RAS that contains R/W resource types as a R/W-RAS.

The novel attributes exhibited by the R/W-locks have implications for the behavior of the R/W-RAS, differentiating them significantly from the previously studied RAS, and complicating their analysis and their management. More specifically, due to the infinite capacity of the R/W-lock when operating in the reading mode, it might not be possible to model R/W-RAS with Finite State Automata or bounded Petri nets. As a result, their behavioral analysis is not immediately amenable to the elementary, enumerative techniques that can provide the basic characterization for deadlock and deadlock avoidance in the case of the previously studied RAS. In fact, the logical behavior of R/W-RAS cannot be modeled even by unbounded Petri nets (when staying within the basic definition of this formalism). Indeed, a process seeking the allocation of R/W-locks must consider not only the availability of their capacity, but also their current operational mode; in particular, a writing stage can be performed only if there are no active readers allocated the corresponding lock. In view of the infinite capacity of R/W-locks in their reading mode, the latter test can be supported only through the introduction of inhibitor arcs in the underlying PN model (Peterson (1981)). Thus, the past PN-based structural approaches for the synthesis of a deadlock avoidance policy (e.g. Ezpeleta et al (1995); Huang et al (2006)) are not transferrable to the new RAS model.

[1] A comprehensive exposition of the Gadara project, including its goals and its current achievements, can be found at: `http://gadara.eecs.umich.edu`. We should also notice, for completeness, that the very first studies on the problem of deadlock avoidance took place in the 1960's / early 1970's in the context of the computing technologies of that era (e.g., Dijkstra (1965); Coffman et al (1971); Holt (1972)). But the connection of deadlock avoidance to DES theory took place primarily through the works mentioned above.

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

Yet, in the rest of this work, we establish that the maximally permissive deadlock avoidance policy (DAP) is effectively computable for R/W-RAS, in spite of all the aforementioned challenges. The key enabler of this result is the fact that the set of inadmissible – or unsafe – states in the considered SC problem possesses a monotonicity property that endows it with qualities similar to those of an upward-closed set (Chen and Chen (2009); Valk and Jantzen (1985)). In particular, it is well known that an upward-closed set of nonnegative integer vectors, even of infinite cardinality, will admit an effective, finite representation by the subset of its minimal elements, as long as this subset is effectively enumerable (Valk and Jantzen (1985)).

We have exploited such a parsimonious representation of the set of unsafe states by means of its minimal elements, in order to develop an efficient implementation of the maximally permissive DAP, even in the context of the more conventional RAS that we have studied in our past work (Nazeem and Reveliotis (2011, 2014)). The basic idea underlying this implementation is (i) to first identify all the minimal unsafe states, and (ii) subsequently to store them in a pertinent advanced data structure, like the "TRIE" data structure and the (n-ary) decision diagrams (Commer and Sethi (1977)), which will enable the efficient identification and blockage of transitions that lead to problematic behavior; more specifically, in view of the monotonicity property of safety and the induced upward-closed structure of the unsafe subspace, a tentative transition must be blocked if it leads to a state that dominates some element in the stored set of minimal unsafe states. In this work, we show that such an implementation of the maximally permissive DAP is still plausible for RAS with R/W-locks, since the set of minimal unsafe states remains effectively enumerable, in spite of the infinite cardinality of the underlying state space and the intricacies in the system behavior that are introduced by the new dynamics of the R/W-locks.

More specifically, and in view of all the above discussion, the main technical results and the corresponding contributions of this paper can be summarized as follows: (i) We introduce the class of R/W-RAS and we provide a formal characterization of its behavior by means of a deterministic state automaton (Cassandras and Lafortune (2008)). (ii) We formally define the problem of maximally permissive deadlock avoidance for this new RAS and we establish the aforementioned upward-closure of the corresponding set of forbidden (i.e., unsafe) states. (iii) Finally, we adapt the results of Nazeem and Reveliotis (2014) towards the development of an algorithm for the enumeration of all the minimal unsafe states in the new RAS class. Once the set of minimal states is available, (n-ary) decision diagrams can be employed for the implementation of the maximally permissive DAP, in the spirit of Nazeem and Reveliotis (2011) that was discussed in the previous paragraph. However, we must also notice that the aforementioned developments are organized in two stages: In the first stage, we consider a R/W-RAS model encoding a process behavior that is more restricted than the finally targeted behavior of the program threads in the current multi-core computer architectures. The initial study of this more restricted RAS model highlights more clearly the key elements that enable the aforementioned implementation of the maximally permissive DAP through the identification and storage of the minimal unsafe states, and identifies the common elements between the developments presented in this work and the developments presented in Nazeem and Reveliotis (2014). In the second stage, we extend the original R/W-RAS model so that it enables the complete sequential logic for the process behavior that is recognized by the Gadara RAS (Liao et al (2013b); Nazeem et al (2011)).[3] Using some fundamental properties of the Gadara RAS that were established in Liao et al (2013b), we show that the methodology developed in Stage 1 extends to the more complex

[3] And, of course, it extends the original Gadara RAS model with the novel element of R/W-locks.

RAS behavior that is addressed in this second stage, with some minimal adjustments in its computational logic.

The rest of the paper is organized as follows: Section 2 introduces the initial version of the R/W-RAS model to be considered in this paper, providing a formal characterization of this model, the automaton-based representation of its behavioral dynamics, and the definition of the problem of maximally permissive deadlock avoidance arising in this system. Section 3 shows that, for the considered RAS, the set of minimal unsafe states is finite, and outlines some possible methods for its effective enumeration. Sections 4 and 5 detail the particular algorithm that we propose for enumerating the minimal unsafe states of the R/W-RAS of Section 2, and Section 6 reports on a set of computational experiments that assess and demonstrate the efficacy of the proposed approach. Section 7 discusses the extension of the R/W-RAS model of Section 2, and the necessary modifications for the algorithm that is presented in Section 5, that will ensure the compliance of the paper developments to the Gadara modeling framework and the eventual applicability of these developments in the context of multi-threaded programming. Finally, Section 8 concludes the paper and outlines some directions for future work.

2 R/W-RAS and the corresponding deadlock avoidance problem

In this section, we introduce the more restricted version of the R/W-RAS, that, as discussed in the Introduction, will be used as a stepping stone towards the final modeling of the dynamics of the lock allocation that takes place in multi-threaded software. More specifically, the version of the R/W-RAS that is defined in this section will help us (i) establish the finiteness of the representation of the underlying unsafe subspace through the minimal unsafe states, and (ii) develop the necessary algorithms for the enumeration of this state set. The necessary extensions to cover the complete process behavior considered in this work are addressed in Section 7.

The considered R/W-RAS class: An instance Φ from the R/W-RAS class considered in this section is defined as a 5-tuple $\langle \mathcal{R}, \mathcal{RW}, C, \mathcal{P}, \mathcal{A} \rangle$ where: (i) $\mathcal{R} = \{R_1, \ldots, R_m\}$ is the set of the *(conventional) resources*. (ii) $\mathcal{RW} = \{RW_1, \ldots, RW_h\}$ is the set of the *reader/writer (R/W) resources*. In the sequel, we shall denote the total number of resource types, $m+h$, by μ. (iii) $C : \mathcal{R} \rightarrow \mathbb{Z}^+ -$, 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. (iv) $\mathcal{P} = \{J_1, \ldots, J_n\}$ 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 an acyclic[4] 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

[4] The acyclicity requirement for digraphs $\mathcal{G}_j$ will be removed in Section 7.

$\mathcal{G}_j$ 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. (v) $\mathcal{A} : \bigcup_{j=1}^{n} \mathcal{S}_j \rightarrow \prod_{i=1}^{m}\{0,\ldots,C_i\} \times \prod_{i=1}^{h}\{0,1,2\}$ is the *resource allocation function*, which associates every processing stage Ξ_{jk} with a *resource allocation request* $\mathcal{A}(\Xi_{jk}) \equiv \mathcal{A}_{jk}$. More specifically, each $\mathcal{A}_{jk}$ is a μ-dimensional vector such that:

- $\forall i = 1 : m,\ \mathcal{A}_{jk}[i]$ indicates the number of resource units of resource type R_i necessary to support the execution of stage Ξ_{jk}.
- $\forall i = 1 : h,\ \mathcal{A}_{jk}[m+i]$ equals 1 *iff* Ξ_{jk} acquires RW_i in the reading mode, and $\mathcal{A}_{jk}[m+i]$ equals 2 *iff* Ξ_{jk} acquires RW_i in the writing mode. Otherwise $\mathcal{A}_{jk}[m+i]$ equals 0.

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. According to the applied resource allocation protocol, a process instance executing processing stage Ξ_{jk} will be able to advance to a successor processing stage $\Xi_{j,k'}$, only after it is allocated the resource differential $(A_{j,k'}[i] - A_{jk}[i])^+, \forall i \in \{1,\ldots,\mu\}$.[5] And it is only upon this advancement that the process will release the resource units $|(A_{j,k'}[i] - A_{jk}[i])^-|, \forall i \in \{1,\ldots,\mu\}$. Some further qualifications are necessary in order to specify completely the considered resource allocation protocol w.r.t. the allocation of the R/W resource types. In particular: (i) A process is allowed access to resource RW_i in the reading mode only if no other process is currently accessing RW_i in the writing mode. (ii) A process is allowed to access RW_i in the writing mode only if no other process is accessing RW_i in either the reading or the writing modes. A process is also allowed to change its mode of accessing a resource RW_i as long as the two aforementioned rules are respected. Hence, if $A_{jk}[m+i] = 1$ and $A_{j,k'}[m+i] = 2$, then the process will be able to advance from stage Ξ_{jk} to stage $\Xi_{j,k'}$ only if no other process is concurrently accessing RW_i in the reading mode. On the other hand, if $A_{jk}[m+i] = 2$ and $A_{j,k'}[m+i] = 1$, then the process can advance immediately from stage Ξ_{jk} to stage $\Xi_{j,k'}$, changing the mode of acquisition of RW_i from writing to reading, when the allocation protocol constraints pertinent to the other resource types are satisfied.

For notational convenience, in the following 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 R/W-RAS, across the entire set of its process types. Furthermore, in some of the subsequent developments, the various processing stages Ξ_{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 $\Xi_{j,k'}$, we define $e.src \equiv \Xi_{jk}$ and $e.dst \equiv \Xi_{j,k'}$; 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 $\mathcal{E} = \cup_{j=1}^{n} \mathcal{E}_j$. Also, $\eta_{kl},\ k = 1,\ldots,m,\ l = 1,\ldots,C_k$, will denote the number of processing stages that require the allocation of l units from resource type R_k, whereas $\eta_{m+k,1}$ (resp., $\eta_{m+k,2}$), $k = 1,\ldots,h$ will denote the number of processing stages accessing resource RW_k in the reading (resp., writing) mode. Additionally, we define $\eta_k \equiv max_{l=1}^{C_k} \eta_{kl},\ k = 1,\ldots,m$. Finally, in the sequel, unless qualified otherwise, any reference to a *resource* will imply any member of the set $\mathcal{R} \cup \mathcal{RW}$.

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

Modeling the dynamics of the R/W-RAS as a State Automaton: The dynamics of the R/W-RAS $\Phi = \langle \mathcal{R}, \mathcal{RW}, C, \mathcal{P}, \mathcal{A} \rangle$, introduced in the previous paragraph, can be formally described by a *Deterministic State Automaton (DSA)* (Cassandras and Lafortune (2008)), $G(\Phi) = (S, E, f, \mathbf{s}_0, S_M)$, 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 R/W-RAS processing stages, and they indicate the number of process instances executing the corresponding stage in the R/W-RAS state modeled by $\mathbf{s}$. Hence, S consists of all the vectors $\mathbf{s} \in (\mathbb{Z}_0^+)^\xi$ that further satisfy

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

$$\forall i := 1 \ldots h, \ (\exists q' : \ \mathbf{s}[q'] > 0 \ \wedge (\mathcal{A}(\Xi_{q'})[m+i]) = 2) \\ \Longrightarrow \sum_{q=1}^{\xi} \mathbf{s}[q] \cdot (\mathcal{A}(\Xi_q)[m+i]) = 2 \tag{2}$$

Constraint 1 expresses the capacity constraints that must be observed w.r.t. the conventional resources. Constraint 2 enforces the exclusive acquisition of a R/W resource in the writing mode.

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} 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., e_{rp} 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, e_{rp} represents the *unloading* of a finished process instance after executing its last stage Ξ_r.

3. The *state transition function* $f : S \times E \to S$ 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.

It is important to notice that if a processing stage involves only access of R/W resources in the reading mode, and no further allocation of any conventional resources, then an arbitrary number of processes might exist simultaneously in this stage, a fact that implies that the state space of the automaton might be infinite.

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 $S_{xy} \equiv S_x \cap S_y$, $x = r, \bar{r}$, $y = s, \bar{s}$. 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 R/W-RAS unsafety characterized in the previous paragraph results from the formation of R/W-RAS *deadlocks*, i.e., R/W-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 R/W-RAS unsafe states are those R/W-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 R/W-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 can be implemented by an one-step-lookahead mechanism that recognizes and prevents transitions to unsafe states. On the other hand, due to the potentially infinite nature of the state space of R/W-RAS, the computation of the maximally permissive DAP cannot be performed through the basic enumerative techniques that are amenable for the more conventional RAS structures studied in the past. Therefore, the technique proposed in this paper is of paramount importance for the effective deployment of the maximally permissive deadlock avoidance in the considered RAS context.

Some monotonicities observed by the state unsafety concept: Next we review some additional structure possessed by the set $S_{\bar{s}}$, that enables the effective representation of the maximally permissive DAP through the explicit storage of a finite (and typically very 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 R/W-RAS state $\mathbf{s} \in S$ to proceed to completion, depends on the existence of a sequence $< \mathbf{s}^{(0)} \equiv \mathbf{s}, e^{(1)}, \mathbf{s}^{(1)}, e^{(2)}, \mathbf{s}^{(2)}, \ldots, \mathbf{s}^{(N-1)}, e^{(N)}, \mathbf{s}^{(N)} \equiv \mathbf{s}_0 >$, such that at every state $\mathbf{s}^{(i)}$, $i = 0, 1 \ldots, N-1$, the free (or "slack") resource capacities at that state enable the job advancement corresponding to event $e^{(i+1)}$. Furthermore, if such a terminating sequence exists for a given state $\mathbf{s}$, then the event feasibility condition defined by Equations 1–2 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.[6]

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

$$\forall \mathbf{s}, \mathbf{s}' \in S, \;\; \mathbf{s} \leq \mathbf{s}' \iff (\forall q = 1, \ldots \xi, \;\; \mathbf{s}[q] \leq \mathbf{s}'[q]) \tag{3}$$

Then,

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

□

[6] 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 Reveliotis and Ferreira (1996); Reveliotis (1996).

In the following, we shall use the notation '$\mathbf{s} < \mathbf{s}'$' to denote that the condition of Eq. 3 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 R/W-RAS class, the maximally permissive DAP can be implemented as follows: First we find and store $\bar{S}_{r\bar{s}}$ in an appropriate data structure. Then, during the online stage, given 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 imminent event is blocked by the supervisory controller. Otherwise, it is safe and the imminent event is allowed. As remarked in the introductory section, an implementation of the maximally permissive DAP through the aforementioned scheme in the context of more conventional RAS classes, using (n-ary) decision diagrams, can be found in Nazeem and Reveliotis (2011). That work contains also an experimental section that demonstrates and assesses the representational and computational efficiencies that are established by the employment of (n-ary) decision diagrams and the "TRIE" data structure for a more compact and more structured storage of the derived set of the minimal unsafe states.

3 On the finiteness and computation of $\bar{S}_{r\bar{s}}$

It is easy to see that $\bar{S}_{r\bar{s}}$ is a set of incomparable vectors w.r.t. the partial order '$\leq$' defined in Equation 3. That is, $\forall \mathbf{x},\mathbf{x}' \in \bar{S}_{r\bar{s}},\ \mathbf{x} \not\geq \mathbf{x}' \ \wedge\ \mathbf{x} \not\leq \mathbf{x}'$. Therefore, by Dickson's Lemma (c.f. Dickson (1913), Lemma 2A), $\bar{S}_{r\bar{s}}$ is a finite set.

A set U over ξ-dimensional vectors of natural numbers is called "*upward-closed*" (or "*right-closed*") if $\forall \mathbf{x} \in U,\ \mathbf{y} \geq \mathbf{x} \Longrightarrow \mathbf{y} \in U$. Proposition 1 implies that the set $S_{r\bar{s}}$ resembles the structure of upward-closed sets; however, because its elements must satisfy the resource feasibility constraints (Eqs. 1-2) and they must also be accessible from $\mathbf{s}_0$, $S_{r\bar{s}}$ is not upward-closed, in a strict sense. To circumvent the technical difficulties arising from this fact, we define the set $U(S_{r\bar{s}}) \equiv S_{r\bar{s}} \cup \{\mathbf{y} \in \mathbb{N}^{\xi} \mid \exists \mathbf{x} \in S_{r\bar{s}} \text{ s.t. } \mathbf{y} > \mathbf{x}\}$. It can be easily seen that $U(S_{r\bar{s}})$ is upward-closed, and that it shares the same set of minimal elements with $S_{r\bar{s}}$.

Let $\mathbb{N}_{\omega} \equiv \mathbb{N} \cup \{\omega\}$, where the element ω denotes an arbitrarily large number; in particular, $\forall n \in \mathbb{N},\ \max\{n,\omega\} = \omega$ and $\min\{n,\omega\} = n$. Also, for any $\mathbf{x} \in \mathbb{N}_{\omega}^{\xi}$, define $reg(\mathbf{x}) \equiv \{\mathbf{x}' \in \mathbb{N}^{\xi} : \mathbf{x}' \leq \mathbf{x}\}$. In Valk and Jantzen (1985), it is established that the minimal elements of a right-closed set U are effectively computable *iff* the decision problem '$(reg(\mathbf{x}) \cap U \neq \emptyset)$?' is decidable for every $\mathbf{x} \in \mathbb{N}_{\omega}^{\xi}$. Also, that work provides an algorithm for the effective enumeration of the set of minimal elements of a right-closed set U, when the test '$(reg(\mathbf{x}) \cap U \neq \emptyset)$?' is effectively computable (c.f. Theorem 2.14 in that work). In the light of that result, a potential approach to effectively construct $\bar{S}_{r\bar{s}}$, is by trying first to develop an algorithm for the resolution of the test '$(reg(\mathbf{x}) \cap U(S_{r\bar{s}}) \neq \emptyset)$?' for every $\mathbf{x} \in \mathbb{N}_{\omega}^{\xi}$, and subsequently apply the algorithm of Valk and Jantzen (1985). Furthermore, it is easy to see that as long as $\mathbf{x}$ remains in $\mathbb{N}^{\xi}$, the question '$(reg(\mathbf{x}) \cap U(S_{r\bar{s}}) \neq \emptyset)$?' can be effectively resolved by constructing the subspace of S that is contained in $reg(\mathbf{x})$ and assessing the safety of every state in that subspace. The main challenge regarding the resolution of the above test is in the case that the considered vector $\mathbf{x}$ contains ω elements, especially in the state coordinates that can be arbitrarily large. These more complicated cases can be potentially resolved by developing an upper bound for the number of process instances that can execute simultaneously any of the RAS stages in a minimal unsafe state. In fact, the availability of such a bound B can enable

an even more straightforward algorithm for the enumeration of $\bar{S}_{r\bar{s}}$ than the aforementioned algorithm in Valk and Jantzen (1985): One could just construct the partial state space contained in $reg([B,B,\ldots,B]^T)$ and identify the set of minimal unsafe states in that subspace; this set would constitute the entire $\bar{S}_{r\bar{s}}$.

However, in this work we pursue a different approach that takes advantage of additional structure that is present in the set of unsafe states $S_{r\bar{s}}$. More specifically, the proposed methodology is motivated and enabled by the fact that, in the considered R/W-RAS, 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 R/W-RAS deadlocks and "backtraces" the R/W-RAS dynamics until it hits the boundary between safe and unsafe subspaces. In particular, our interest in minimal unsafe states implies that we can focus this backtracing only to minimal deadlocks. The resulting algorithm decomposes naturally into a two-stage computation, with the first stage identifying all minimal deadlocks, and the second stage performing the aforementioned backtracing process in order to identify the broader set of minimal unsafe states. The next two sections detail each of these two stages in the context of the R/W-RAS introduced in Section 2. The presented results customize and extend to the considered problem setting some similar results developed in Nazeem and Reveliotis (2014) for the efficient enumeration of minimal unsafe states in more conventional RAS structures. On the other hand, Section 7 extends the aforementioned approach to R/W-RAS encompassing the more complex process behavior that is modeled by the Gadara RAS, a pertinent abstraction capturing the dynamics of lock allocation in multi-threaded programming.

4 Enumerating $\bar{S}_{rd}$

As pointed out in the previous section, the first step in the proposed 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 flow of the proposed algorithm.

4.1 Preamble

Let $\mathbf{s}.R_i$ denote the total number of units from the conventional resource type R_i that are allocated at state $\mathbf{s}$. Furthermore, for a R/W resource RW_i, we set $\mathbf{s}.RW_i = 1$ (resp., 2) *iff* RW_i is accessed in state $\mathbf{s}$ in the reading (resp., writing) mode by a single process, and $\mathbf{s}.RW_i = 3$ *iff* RW_i is accessed in the reading mode by multiple processes; otherwise, $\mathbf{s}.RW_i = 0$. Then, we can characterize the blocking dynamics of the considered RAS through the following definitions:

Definition 1 *Given an edge $e \in \mathcal{G}$, e is "*blocked*" at state* $\mathbf{s}$ iff $\mathbf{s}[e.src] > 0$*, and one of the following four conditions is true:*

1. $\exists R_k \in \mathcal{R}$ *s.t.* $\mathbf{s}.R_k + \mathcal{A}_{e.dst}[k] - \mathcal{A}_{e.src}[k] > C_k$*, or*
2. $\exists RW_k \in \mathcal{RW}$ *s.t.* $\mathcal{A}_{e.src}[m+k] = 0$*,* $\mathcal{A}_{e.dst}[m+k] = 2$*, and* $\mathbf{s}.RW_k > 0$*, or*
3. $\exists RW_k \in \mathcal{RW}$ *s.t.* $\mathcal{A}_{e.src}[m+k] = 0$*,* $\mathcal{A}_{e.dst}[m+k] = 1$*, and* $\mathbf{s}.RW_k = 2$*, or*
4. $\exists RW_k \in \mathcal{RW}$ *s.t.* $\mathcal{A}_{e.src}[m+k] = 1$*,* $\mathcal{A}_{e.dst}[m+k] = 2$*, and* $\mathbf{s}.RW_k = 3$*.*

*Edge e is "*enabled*" at state* $\mathbf{s}$ iff $\mathbf{s}[e.src] > 0$ *and e is not blocked. The set of enabled edges at* $\mathbf{s}$ *will be denoted by* $g(\mathbf{s})$*.*

In plain terms, an edge e is blocked at state $\mathbf{s}$ by a conventional resource if its slack capacity is less than the required number of additional units needed for $e.dst$. Furthermore, e is blocked by an R/W resource type RW_k if (a) $e.src$ does not access RW_k, RW_k is accessed in any of its modes at $\mathbf{s}$, and $e.dst$ requests RW_k in the writing mode, or (b) $e.src$ does not access RW_k, RW_k is accessed in the writing mode at $\mathbf{s}$, and $e.dst$ requests RW_k in the reading mode, or (c) $e.src$ accesses RW_k in the reading mode, RW_k is accessed by multiple processes in the reading mode at $\mathbf{s}$, and $e.dst$ requests RW_k in the writing mode.

Definition 2 *A processing stage q is blocked at state* $\mathbf{s}$ iff *all its outgoing edges are blocked. Stage q is enabled at state* $\mathbf{s}$ iff *at least one of its outgoing edges is enabled.*

The next definition introduces a more technical concept which is at the center of all the technical developments that are presented in the rest of this work.

Definition 3 *Given a set of states X, we define the state λ_X by $\lambda_X[q] \equiv max_{\mathbf{x}\in X}\mathbf{x}[q]$, $q = 1,\ldots,\xi$. State λ_X will be characterized as the* "combination" *of the states in X.*

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

Proposition 2 *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 $\mathcal{G}_i$. Then, by an argument similar to that in the proof of Proposition 2, we can perceive $\mathbf{x}_i$ 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 satisfies one of the four conditions in Definition 1; i.e.,

1. for some conventional resource type R_k s.t. $\mathcal{A}_{e_{ij}.dst}[k] - \mathcal{A}_{e_{ij}.src}[k] > 0$, $\mathbf{x}_{ij}.R_k > C_k - \mathcal{A}_{e_{ij}.dst}[k] + \mathcal{A}_{e_{ij}.src}[k] \equiv l$, or
2. for some R/W resource type RW_k s.t. $\mathcal{A}_{e_{ij}.src}[m+k] = 0 \wedge \mathcal{A}_{e_{ij}.dst}[m+k] = 2$, $\mathbf{x}_{ij}.RW_k = 1 \vee \mathbf{x}_{ij}.RW_k = 2$, or
3. for some R/W resource type RW_k s.t. $\mathcal{A}_{e_{ij}.src}[m+k] = 0 \wedge \mathcal{A}_{e_{ij}.dst}[m+k] = 1$, $\mathbf{x}_{ij}.RW_k = 2$, or
4. for some R/W resource type RW_k s.t. $\mathcal{A}_{e_{ij}.src}[m+k] = 1 \wedge \mathcal{A}_{e_{ij}.dst}[m+k] = 2$, $\mathbf{x}_{ij}.RW_k = 3$.

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, and the minimal states that block e_{ij} through RW_k can be obtained by enumerating all the minimal states at which $\mathbf{s}.RW_k = 1,\ 2,\ 3$. The reader should also notice that a minimal state at which $\mathbf{s}.RW_k = 1(resp.,2)$ is a single unit vector with one process instance at a processing stage that accesses RW_k in the reading (resp., writing) mode. On the other hand, a minimal state at which $\mathbf{s}.RW_k = 3$ is a state that has exactly two processes executing a pair of processing stages that accesses RW_k in the reading mode.

4.2 Outline of the proposed algorithm

The proposed algorithm for the enumeration of the set of minimal reachable deadlocks, $\bar{S}_{rd}$, is motivated by the analysis presented in the previous subsection, and it can be described as follows:

1. For each conventional resource type R_k, and for each occupancy level $l,\ 1 \leq l \leq C_k$, compute the set of minimal states $MinStR[k][l]$ that allocate l units of R_k.
2. For each R/W resource type RW_k, compute the sets of minimal states $MinStR[m+k][1]$ and $MinStR[m+k][2]$ at which RW_k is acquired respectively in the reading and the writing modes.
3. For each edge e, compute the set of minimal states $BlockEd[e]$ at which e is blocked.
4. For each processing stage q, compute the set of minimal states $BlockPs[q]$ at which q is blocked.
5. 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., $\mathbf{p}_2 = \lambda_{\{\mathbf{p}_1,\mathbf{x}'\}},\ \mathbf{x}' \in BlockPs[q']$. 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 details further the above algorithm. Steps 1 and 4 are exactly the same in their algorithmic details with their counterparts in Nazeem and Reveliotis (2014) (c.f. Procedures 1 and 3 in Nazeem and Reveliotis (2014)). Hence, while a high level description is provided in this manuscript for these two steps, the reader is referred to Nazeem and Reveliotis (2014) for further details. Also, before delving into the more detailed description of the algorithm, we would like to highlight the fact 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 unsafe state. Therefore, these stages are explicitly ignored by the proposed algorithm.

Example 1: We shall use the R/W-RAS configuration depicted in Table 1 as a running example to demonstrate the application of the steps of the above introduced algorithm. The considered R/W-RAS has three conventional resource types, R_1, R_2 and R_3, and one R/W resource type, RW_1. Resources R_1 and R_3 have a capacity of one unit, whereas the capacity of R_2 is equal to two. The considered R/W-RAS has two process types, J_1 and J_2. For each process type $J_i,\ i = 1,2$, Table 1 provides the structure of the corresponding graph $\mathcal{G}_i$ as well as the resource allocation request that is posed by each processing stage Ξ_{ij} that appears in those graphs. The reader should notice that process J_1 presents routing flexibility. More specifically, a job at the first processing stage Ξ_{11} can advance to stage Ξ_{12} (acquiring one unit of R_2), or to stage Ξ_{13} (acquiring one unit of R_3). On the other hand, J_2 has a simple linear structure. 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.

4.3 Computing $MinStR[k][l]$ for R_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

Table 1: The R/W-RAS considered in Example 1

Resource Types:	$\{R_1, R_2, R_3\}$ $\{RW_1\}$
Resource Capacities:	$C_1 = C_3 = 1, C_2 = 2$
Process Type 1:	$\Xi_{11}(R_1) \rightarrow \Xi_{12}(R_2)$ or $\Xi_{13}(R_3) \rightarrow \Xi_{14}(read(RW_1))$
Process Type 2:	$\Xi_{21}(write(RW_1)) \rightarrow \Xi_{22}(R_1)$

Table 2: The array $MinStR$ for Example 1

$MinStR[1][1]$	$\{\Xi_{11}\}$
$MinStR[2][1]$	$\{\Xi_{12}\}$
$MinStR[2][2]$	$\{2\Xi_{12}\}$
$MinStR[3][1]$	$\{\Xi_{13}\}$
$MinStR[4][1]$	$\emptyset$
$MinStR[4][2]$	$\{\Xi_{21}\}$

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 this 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 Eqs. 1-2.

Example 1 (cont.): Consider the resource type R_2. Ξ_{12} is the only processing stage that occupies one unit of R_2. Hence, $MinStR[2][1]$ contains only the state $\{\Xi_{12}\}$. On the other hand, to obtain a minimal state that occupies two units of R_2, we can add any two members of $MinStR[2][1]$. Since the state $\{\Xi_{12}\}$ is the only member in $MinStR[2][1]$, adding it to itself results in the state $\{2\Xi_{12}\}$ which is the only member of $MinStR[2][2]$. The complete array $MinStR$ computed for the conventional resource types in this example is depicted in the first four rows of Table 2.

4.4 Computing $MinStR[m+k][1]$ and $MinStR[m+k][2]$ for RW_k

As already pointed out, a minimal state $\mathbf{s}$ at which RW_k is accessed by a single process in the reading (resp., writing) mode can only be a unit vector state $\mathbf{s}[q] = 1$ for some component $q \in \{1, \ldots, \xi\}$, $\mathbf{s}[q'] = 0$, $\forall q \neq q'$, such that $\mathcal{A}_q[m+k] = 1$ (resp., $\mathcal{A}_q[m+k] = 2$). Therefore, $MinStR[m+k][1]$ (resp., $MinStR[m+k][2]$) shall contain only the $\eta_{m+k,1}$ (resp., $\eta_{m+k,2}$) unit vector states corresponding to the stages that access RW_k in the reading (resp., writing) mode.

Example 1 (cont.): Consider the resource type RW_1. Ξ_{14} is the only processing stage that acquires RW_1 in the reading mode. But since Ξ_{14} is a terminal stage, $MinStR[4][1]$ is empty. On the other hand, Ξ_{21} is the only processing stage that acquires RW_1 in the writing mode. Hence, the state $\{\Xi_{21}\}$ is the only member of $MinStR[4][2]$.

Table 3: The array *BlockEd* for Example 1

$BlockEd[(\Xi_{11},\Xi_{12})]$	$\mathbf{es}_1 = \{\Xi_{11}, 2\Xi_{12}\}$
$BlockEd[(\Xi_{11},\Xi_{13})]$	$\mathbf{es}_2 = \{\Xi_{11},\Xi_{13}\}$
$BlockEd[(\Xi_{12},\Xi_{14})]$	$\mathbf{es}_3 = \{\Xi_{12},\Xi_{21}\}$
$BlockEd[(\Xi_{13},\Xi_{14})]$	$\mathbf{es}_4 = \{\Xi_{13},\Xi_{21}\}$
$BlockEd[(\Xi_{21},\Xi_{22})]$	$\mathbf{es}_5 = \{\Xi_{21},\Xi_{11}\}$

Table 4: The array *BlockPs* for Example 1

$BlockPs[\Xi_{11}]$	$\mathbf{ps}_1 = \{\Xi_{11}, 2\Xi_{12}, \Xi_{13}\}$
$BlockPs[\Xi_{12}]$	$\mathbf{ps}_2 = \{\Xi_{12},\Xi_{21}\}$
$BlockPs[\Xi_{13}]$	$\mathbf{ps}_3 = \{\Xi_{13},\Xi_{21}\}$
$BlockPs[\Xi_{21}]$	$\mathbf{ps}_4 = \{\Xi_{21},\Xi_{11}\}$

4.5 Computing $BlockEd[e]$

According to the remarks that were provided in Subsection 4.1, the computation of this data structure can be organized as follows: For each conventional 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. On the other hand, for each R/W resource RW_k s.t. $\mathcal{A}_{e.src}[m+k] = 0 \wedge \mathcal{A}_{e.dst}[m+k] = 1$, we insert all the states from $MinStR[m+k][2]$ into $BlockEd[e]$, whereas for each R/W resource RW_k s.t. $\mathcal{A}_{e.src}[m+k] = 0 \wedge \mathcal{A}_{e.dst}[m+k] = 2$, we insert all the states from $MinStR[m+k][1] \cup MinStR[m+k][2]$ into $BlockEd[e]$; we also add one process at *e.src* in both cases. Similarly, for each R/W resource RW_k s.t. $\mathcal{A}_{e.src}[m+k] = 1$, $\mathcal{A}_{e.dst}[m+k] = 2$, we insert all the states from $MinStR[m+k][1]$ after adding one process at *e.src*. If a process is added at *e.src*, the resulting states must also be checked for feasibility. Finally, the non-minimal states are removed from $BlockEd[e]$. The complete algorithm supporting this computation is depicted in Procedure 1.

Example 1 (cont.): Consider the edge (Ξ_{11},Ξ_{12}). Advancement across this edge requires only the allocation of one unit from resource R_2. According to Line 5 in Procedure 1, only states in $MinStR[2][2]$ can be used to block the edge. Thus, $BlockEd[(\Xi_{11},\Xi_{12})]$ contains only the state $\{\Xi_{11}, 2\Xi_{12}\}$. The complete array *BlockEd* computed for this example is depicted in Table 3.

4.6 Computing $BlockPs[q]$

Let $\{e_1^q, \ldots, e_{\mathcal{D}(q)}^q\}$ be the set of edges emanating from q, where $\mathcal{D}(q)$, as defined in Section 2, is the number of edges in process graph $\mathcal{G}$ that emanate from its node that corresponds to stage q. Then, $BlockPs[q]$ is computed by taking all the feasible combinations of states from $BlockEd[e_1^q] \times BlockEd[e_2^q] \times \ldots BlockEd[e_{\mathcal{D}(q)}^q]$, while eliminating those combinations that result in non-minimal elements.

Procedure 1 CompBlockEd(e)

Input: an edge e from the "union" process graph $\mathcal{G}$
Output: $BlockEd[e]$

```
1: for k := 1 → μ do
2:     startIdx ← 1, endIdx ← 0
3:     if k ≤ m then
4:         if A_{e.dst}[k] − A_{e.src}[k] > 0 then
5:             startIdx ← C_k − A_{e.dst}[k] + A_{e.src}[k] + 1; endIdx ← C_k;
6:         end if
7:     else
8:         if A_{e.dst}[k] = 2 ∧ A_{e.src}[k] = 0 then
9:             startIdx ← 1; endIdx ← 2
10:        else if A_{e.dst}[k] = 2 ∧ A_{e.src}[k] = 1 then
11:            startIdx ← 1; endIdx ← 1
12:        else if A_{e.dst}[k] = 1 ∧ A_{e.src}[k] = 0 then
13:            startIdx ← 2; endIdx ← 2;
14:        end if
15:    end if
16:    for l = startIdx → endIdx do
17:        for s ∈ MinStR[k][l] do
18:            s' ← s
19:            if s'[e.src] = 0 then
20:                s'[e.src] ← 1
21:                if Feasible(s') then
22:                    Insert s' into BlockEd[e]
23:                end if
24:            else
25:                Insert s' into BlockEd[e]
26:            end if
27:        end for
28:    end for
29: end for
30: Remove non-minimal states from BlockEd[e]
31: return BlockEd[e]
```

Example 1 (cont.): Consider the processing stage Ξ_{11}. It has two outgoing edges, (Ξ_{11}, Ξ_{12}) and (Ξ_{11}, Ξ_{13}). It can be seen from Table 3 that es_1 is the only member of $BlockEd[(\Xi_{11}, \Xi_{12})]$, and that es_2 is the only member of $BlockEd[(\Xi_{11}, \Xi_{13})]$. Hence, the combination operation (c.f. Definition 3) is applied to states es_1 and es_2 to generate the state $\{\Xi_{11}, 2\Xi_{12}, \Xi_{13}\}$, which is a minimal state at which Ξ_{11} is blocked. Hence $BlockPs[\Xi_{11}] = \{\Xi_{11}, 2\Xi_{12}, \Xi_{13}\}$. The complete array $BlockPs$ computed for this example is depicted in Table 4.

4.7 Enumerating the minimal reachable deadlocks

The complete algorithm for enumerating the minimal reachable deadlock states is depicted in Procedure 2. 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 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

Procedure 2 EnumMinReachDeadlocks(Φ)

Input: A R/W-RAS instance Φ
Output: the list $deadlockHT$ containing all the reachable minimal deadlocks of Φ

```
 1: deadlockHT ← ∅
 2: for k = 1 : μ do
 3:     MinStR[k] ← Compute MinStR[k]
 4: end for
 5: for all e ∈ E do
 6:     BlockEd[e] ← Compute BlockEd[e];
 7: end for
 8: for q = 1 : ξ do
 9:     BlockPs[q] ← 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
```

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 $deadlockHT$. The reachability of any given state $\mathbf{s}$ in the computed set $deadlockHT$ can be resolved by reducing this decision problem to the assessment of the co-reachability of $\mathbf{s}$ in the automaton $\hat{G}(\Phi)$ that is obtained by reversing the transitions of the automaton $G(\Phi)$ while maintaining the same set of states, S, as well as the definitions the initial and the marked states.

Example 1 (cont.): Consider the iteration of Procedure 2 that starts from stage Ξ_{11} and state $\mathbf{ps}_1 = \{\Xi_{11}, 2\Xi_{12}, \Xi_{13}\}$. To block Ξ_{12}, we combine state $\mathbf{ps}_2$ with state $\mathbf{ps}_1$ (c.f. Table 4) resulting in state $\{\Xi_{11}, 2\Xi_{21}, \Xi_{13}, \Xi_{21}\}$ 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 does not yield any other minimal deadlock state. □

The next theorem establishes the correctness of Procedure 2.

Theorem 1 *Procedure 2 enumerates all the minimal reachable deadlock states of its input R/W-RAS* Φ.

Proof: Let $\mathbf{s}_d$ be an arbitrary minimal deadlock state, and $\{q_1,\ldots,q_t\}$ be the set of processing stages that have active processes at $\mathbf{s}_d$. Then, according to Lemma 2, 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. Without loss of generality (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 *working Queue*. 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: A complete complexity analysis of Procedure 2 and of its supporting subroutines is provided in Nazeem (2012). It is shown that the overall computational complexity of Procedure 2 can be characterized as $O(\xi \cdot \mu^{2\cdot|\mathcal{E}|} \cdot \eta^{2\cdot C\cdot|\mathcal{E}|} + K)$, where μ denotes the total number of the conventional and R/W resources types in the input R/W-RAS Φ, $\eta \equiv \max_{k=1}^{\mu} \eta_k$, and $C \equiv max_{k=1}^{m} C_k$. On the other hand, the term K that appears in the above expression denotes the effort that must be expended for the execution of Step 29 of the depicted algorithm. In general, this effort will depend on the cardinality of the constructed set *deadlockHT* and on the number of active process instances that appear in these elements. The minimality of the constructed deadlocks in *deadlockHT* implies that both of these numbers will tend to stay at fairly small values. In fact, empirical evidence suggests that the practical run time of Step 29 is very small when compared with computational cost of some other stages of the algorithm.[7] Hence, we can conclude that the algorithm complexity is most sensitive to the capacity of the conventional resource types and to the number of the distinct event types that take place in the underlying R/W-RAS.

5 Enumerating $\bar{S}_{r\bar{s}}$

In this section, we present the algorithm that enumerates the entire set of minimal unsafe states, $\bar{S}_{r\bar{s}}$, by backtracing from the set of minimal deadlocks, $\bar{S}_{rd}$, for the R/W-RAS of Section 2. As it will be revealed in the following discussion, once the applying blocking mechanisms have been detailed according to the analysis provided in Section 4, the remaining logic of the considered algorithm is determined by (i) the definition of state unsafety as unavoidable absorption to some deadlock state, and (ii) some further properties that this definition implies for the topology of the underlying state space w.r.t. the classification induced by the notion of (minimal) unsafety. However, the definition of unsafety and the aforementioned properties that are involved in the algorithm development are common for the class of the R/W-RAS considered in this section and the more conventional RAS classes considered in Nazeem and Reveliotis (2014). Therefore, the developments of this section parallel closely those of Section IV in Nazeem and Reveliotis (2014), and the reader is referred to that work for formal correctness and complexity analyses of the presented algorithm. On

[7] This claim is substantiated by the computational experiments that are presented in Section 6. Also, we notice that it is possible to skip the elimination of the reachable unsafe states in the construction of the list *deadlockHT*, without compromising the correctness of the resulting implementation of the maximally permissive DAP that was discussed in Section 2. However, the presence of the unreachable deadlock states in *deadlockHT* would have an adversarial impact on the complexity of the computation of the set $\bar{S}_{r\bar{s}}$ that is discussed in Section 5, that is much more severe than the computational cost of their removal from that list.

the other hand, in Nazeem and Reveliotis (2014), the convergence of the presented algorithm was established on the finiteness of the underlying state space. Therefore, the algorithm convergence is revisited in this work. To smoothen the flow of the exposition and for self-containment, some of the content of Nazeem and Reveliotis (2014) is recapitulated here. Hence, we proceed as follows: First, we introduce all the necessary definitions for the description of the algorithm. Next, we introduce the algorithm itself and demonstrate its application in the context of the considered R/W-RAS, by means of two R/W-RAS configurations. Finally, we prove the convergence of the algorithm.

5.1 Preamble

Given a minimal deadlock-free unsafe state $\mathbf{u}$ from the R/W-RAS class introduced in Section 2, 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 4 *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 Eqs. 1-2. 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 last result implies that $\mathbf{s}_{ij}$ is a safe state in this case. The structure revealed by Definition 4 and the above discussion is depicted schematically in Figure 1.

As explained in Section 3, 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 4. 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 3 (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

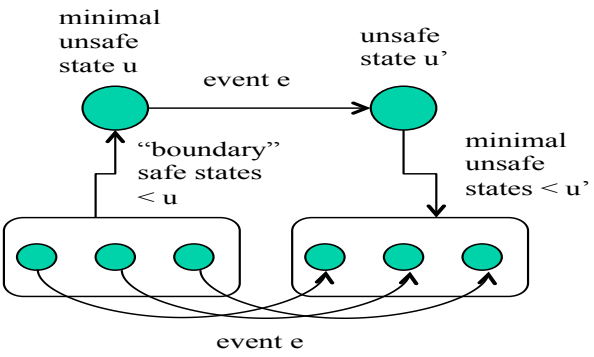

Fig. 1: A schematic diagram of the transitional structure that is leveraged by the proposed algorithm for the enumeration of $\bar{S}_{r\bar{s}}$.

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 algorithm 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. Assume w.l.o.g. that $\mathbf{a}'_1$ was generated first in the course of the algorithm execution. Then, $\mathbf{a}'_2$ will be discarded upon its generation, but $\tau_{\mathbf{a}'_1}$ will be updated to $\tau_{\mathbf{a}'_1} := \tau_{\mathbf{a}'_1} \cup \tau_{\mathbf{a}'_2}$.

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 therefore, its role is self-explanatory. On the other hand, in order to explain the role of the second and the third mechanisms, we remind the reader that Definition 4 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 5 *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 1-2, and (ii)* $\tau_{\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}} \equiv \{\tau_{\mathbf{a}_1} \cup \tau_{\mathbf{a}_2}\} \cap g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}) \neq \emptyset$.

Definition 6 *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 6 eliminates non-minimal unsafe states. The next proposition shows that any state in $Confine(\mathbf{a},\ \tau_{\mathbf{a}})$ is an unsafe state.

Proposition 3 *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 6 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 6 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. □

5.2 The proposed algorithm and its analysis

The complete logic for the enumeration of minimal reachable unsafe states 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 2, 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 5. 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

Algorithm 3

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

```
1: U̇, Ȧ ← ∅; k ← 0;
2: Q ← EnumMinReachDeadlocks(Φ)
3: while Q ≠ ∅ or k < |Ȧ| do
4:     if Q ≠ ∅ then
5:         u ← dequeue(Q);
6:         Prev(u) ← Backtrace(u);
7:         (Safe_Prev, Unsafe_Prev) ← Classify(Prev(u))
8:         Insert_Non_Min(Unsafe_Prev, Q, U̇)
9:         for each a ∈ Safe_Prev do
10:            Z_a ← ∅
11:            for all ȧ ∈ Ȧ do
12:                Z_a ← Z_a ∪ Combine(ȧ, a)
13:            end for
14:            (Safe(Z_a), Unsafe(Z_a)) ← Classify(Z_a)
15:            Insert a, Safe(Z_a) into Ȧ
16:            Insert_Non_Min( Unsafe(Z_a), Q, U̇)
17:        end for
18:        Insert_Non_Min(u, U̇, Q)
19:    else
20:        while k < |Ȧ| do
21:            a_k ← Ȧ[k++];
22:            U* ← Confine(a_k, τ_{a_k})
23:            Insert_Non_Min(U*, Q, U̇)
24:        end while
25:    end if
26: end while
27: Remove_Unreachable(U̇)
28: return U̇
```

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 to every state in $\dot{A}$ that has not been subjected to this operation yet. The mechanism of the *Confine* operation is very similar to that of Lines 12-27 in Procedure 2, but instead of seeking to block enabled processing stages, the *Confine* procedure seeks to block the enabled edges that do not belong to $\tau_{\mathbf{a}}$. On the other hand, the subroutine Insert_Non_Min(U, Q_1, Q_2) invoked in Lines 16, 18 and 23 removes the non-minimal unsafe states that are generated during the course of the execution of the algorithm. In particular, Insert_Non_Min(U, Q_1, Q_2) is invoked to insert the minimal states in U into Q_1, while removing any non-minimal state vectors from $Q_1 \cup Q_2$. In other words, a state $\mathbf{u} \in U$ is inserted into Q_1 *iff* the set $Q_1 \cup Q_2$ does not contain any 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 $Q_1 \cup Q_2$.

Complexity Considerations: In Nazeem (2012) it is shown that the overall complexity of Algorithm 3 can be characterized as $O(\mu^{|\mathcal{E}|+1} \cdot \eta^{C \cdot |\mathcal{E}|} \cdot \xi \cdot |\mathcal{E}| \cdot 2^{\alpha \cdot |\mathcal{E}|} + K')$. 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. η and C are defined as in the complexity analysis of Procedure 2. $|\mathcal{E}|$ is the total number of edges in the process graph $\mathcal{G}$. $O(K')$ is an upper bound to the running time required for the co-reachability analysis for all the

Table 5: The R/W-RAS considered in Example 2

Resource Types:	$\{R_1, R_2\}, \{RW_1, RW_2\}$
Resource Capacities:	$C_1 = C_2$
Process Type 1:	$\Xi_{11}(R_1) \rightarrow \Xi_{12}(R_2) \rightarrow \Xi_{13}(read(RW_1))$
Process Type 2:	$\Xi_{21}(write(RW_1)) \rightarrow \Xi_{22}(R_2) \rightarrow \Xi_{23}(read(RW_2))$
Process Type 3:	$\Xi_{31}(write(RW_2)) \rightarrow \Xi_{32}(R_2) \rightarrow \Xi_{33}(R_1)$

states generated throughout the algorithm in Lines 7 and 14, in addition to the reachability analysis performed at Line 27 for the states in $\dot{U}$. The reachability and the co-reachability analysis are implemented using depth-first search supported with hash tables to mark visited states, and, as in the case of Procedure 2, the empirical complexity of the corresponding computation is very benign.

From the above discussion, it can be concluded that the computational complexity of Algorithm 3 is particularly sensitive to the capacities of the conventional resource types, the number of the distinct event types, and the number of enumerated unsafe states. In addition, the computational experiments presented in Section 6 will reveal that, statistically, the algorithm running time is mostly correlated with the size of the constructed list $\dot{A}$.

The correctness of Algorithm 3 is shown in the last part of this section. Next, we present two examples that demonstrate the respective application of the *Combine* and the $Confine$ operations.

Example 2: To illustrate the application of the *Combine* operation, consider the R/W-RAS configuration depicted in Table 5. It has two conventional resource types, R_1 and R_2, all with a single unit capacity, and two R/W resource types, RW_1 and RW_2. It also has three process types, J_1, J_2 and J_3, all with simple linear structure. Applying Procedure 2 results in the minimal deadlock states $\mathbf{u}_1 = \{\Xi_{11}, \Xi_{32}\}$, $\mathbf{u}_2 = \{\Xi_{12}, \Xi_{21}\}$, and $\mathbf{u}_3 = \{\Xi_{22}, \Xi_{31}\}$. Table 6 depicts the minimal unsafe states obtained by applying Algorithm 3, and the result of backtracing from each of them. The underlined processing stages indicate the source nodes of the backtraced edges in $\tau_{\mathbf{a}}$. As a more concrete example, consider state $\mathbf{u}_1 = \{\Xi_{11}, \Xi_{32}\}$. Ξ_{11} cannot be traced back because this is an initiating stage. Backtracing on $(\Xi_{31} \rightarrow \Xi_{32})$ yields state $\mathbf{a}_1 = \{\Xi_{11}, \underline{\Xi}_{31}\}$, which is a safe state. The algorithm starts by inserting $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$ into Q. First, $\mathbf{u}_1$ is backtraced, adding state $\mathbf{a}_1$ to $\dot{A}$. Next, $\mathbf{u}_2$ is backtraced, generating the boundary safe state $\mathbf{a}_2$. Assessing the combinability of $\mathbf{a}_1$ and $\mathbf{a}_2$, we can see that $\lambda_{\{\mathbf{a_1},\mathbf{a_2}\}} = \{\Xi_{11}, \Xi_{21}, \Xi_{31}\}$, with the enabled edges $g(\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}) = \{\Xi_{11} \rightarrow \Xi_{12}, \Xi_{21} \rightarrow \Xi_{22}, \Xi_{31} \rightarrow \Xi_{32}\}$ and $\tau_{\lambda_{\{\mathbf{a}_1,\mathbf{a}_2\}}} = \{\Xi_{11} \rightarrow \Xi_{12}, \Xi_{31} \rightarrow \Xi_{32}\}$. Hence $\mathbf{a_1}$ and $\mathbf{a_2}$ are combinable. Moreover, $\lambda_{\{\mathbf{a_1},\mathbf{a_2}\}}$ is an unsafe state, call it $\mathbf{u}_4$. Hence, $\mathbf{u}_4$ is added to $Unsafe(Z_a)$, and consequently into Q. The same process is repeated with $\mathbf{u}_3$, but combining $\mathbf{a}_3$ with each of $\mathbf{a}_1$ and $\mathbf{a}_2$ yields $\mathbf{u}_4$ again. Hence $\mathbf{u_4}$ can be constructed by applying the *Combine* operation to any pair of $\{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$. On the other hand, $\mathbf{u}_4$ can not be traced back because all its processing stages are initiating stages. Thus, after the fourth iteration, $Q = \emptyset$ and $\dot{A} = \{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$. Applying the $Confine$ operation to the elements of $\dot{A}$ does not generate any minimal unsafe state. Hence, the algorithm terminates. □

Example 3: To illustrate the application of the $Confine$ operation, consider the R/W-RAS configuration depicted in Table 7. The considered R/W-RAS has three conventional resource types, R_1, R_2 and R_3, all with a single unit capacity, and the R/W resource type RW_1. It also has three process types, J_1, J_2 and J_3, all with simple linear structure and single-type resource allocation, except for processing stage Ξ_{22} which requests the allocation of,

Table 6: Tracing back from the minimal unsafe states for Example 2

$\mathbf{u}$	$Safe\_Prev$	$Unsafe\_Prev$
$\mathbf{u}_1 = \{\Xi_{11}, \Xi_{32}\}$	$\mathbf{a}_1 = \{\Xi_{11}, \underline{\Xi}_{31}\}$	$\emptyset$
$\mathbf{u}_2 = \{\Xi_{12}, \Xi_{21}\}$	$\mathbf{a}_2 = \{\underline{\Xi}_{11}, \Xi_{21}\}$	$\emptyset$
$\mathbf{u}_3 = \{\Xi_{22}, \Xi_{31}\}$	$\mathbf{a}_3 = \{\underline{\Xi}_{21}, \Xi_{31}\}$	$\emptyset$
$\mathbf{u}_4 = \{\Xi_{11}, \Xi_{21}, \Xi_{31}\}$	$\emptyset$	$\emptyset$

Table 7: The RAS considered in Example 3

Resource Types:	$\{R_1, R_2, R_3\}$, RW_1
Resource Capacities:	$C_1 = C_2 = C_3 = 1$
Process Type 1:	$\Xi_{11}(R_1) \rightarrow \Xi_{12}(R_2) \rightarrow \Xi_{13}(read(RW_1))$
Process Type 2:	$\Xi_{21}(write(RW_1)) \rightarrow \Xi_{22}(\{R_2,\ R_3\})$
Process Type 3:	$\Xi_{31}(R_3) \rightarrow \Xi_{32}(R_1)$

both, R_2 and R_3. The application of Procedure 2 yields the minimal deadlock state $\mathbf{u}_1 = \{\Xi_{12}, \Xi_{21}\}$. Backtracing from $\mathbf{u}_1$ yields $\mathbf{a}_1 = \{\underline{\Xi}_{11}, \Xi_{21}\}$, which is a safe state: $\tau_{\mathbf{a}_1} = (\Xi_{11} \rightarrow \Xi_{12})$, whereas $g(\mathbf{a}_1) = \{\Xi_{11} \rightarrow \Xi_{12}, \Xi_{21} \rightarrow \Xi_{22}\}$. Applying the $Confine$ operation to $\mathbf{a}_1$ to block the edge $(\Xi_{21} \rightarrow \Xi_{22})$ yields the second minimal unsafe state $\mathbf{u}_2 = \{\Xi_{11}, \Xi_{21}, \Xi_{31}\}$. □

Proving the correctness of Algorithm 3: To establish the correctness of Algorithm 3, we need to show that (i) Algorithm 3 terminates in a finite number of steps, and that (ii) upon its termination, the algorithm will have enumerated correctly all the minimal unsafe states of its input R/W-RAS Φ.

To establish that the algorithm enumerates correctly all the minimal reachable unsafe states, we start by observing that Theorem 1 establishes that Procedure 2 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 4 – 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. For the R/W-RAS of Section 2, the aforementioned absence of cyclic dependencies can be argued from the acyclic structure of the process digraphs $\mathcal{G}_i,\ i \in \{1, \ldots, n\}$. Thus, 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. A formal proof that relies on the above idea can be found in Nazeem and Reveliotis (2014)-Section IV.

On the other hand, the next proposition proves the proper termination of the algorithm.

Proposition 4 *Algorithm 3 terminates in a finite number of steps.*

Proof: First, we remind the reader that an unsafe state is discarded if it dominates some other state in $\dot{U} \cup Q$. Second, we notice that the algorithm will diverge only if it keeps adding states to Q forever. To trace the states added to Q, let L_1 and L_2 be two sets of states such that when a state $\mathbf{s}$ is added to Q, if $\mathbf{s}$ is incomparable with all the states in L_1, $\mathbf{s}$ is added to L_1. Otherwise, i.e., if $\mathbf{s}$ is dominated by some state in L_1, $\mathbf{s}$ is added to L_2. The third case, i.e., $\mathbf{s}$ dominates some state in L_1 is ruled out by the opening remark in this proof. Thus L_1 is a set of incomparable vectors, and thus, by Dickson's Lemma, L_1 can never grow to be

Table 8: A sample of our computational results regarding the efficacy of Algorithm 3

$\lvert\xi\rvert$	$\lvert S_{r\bar{s}}\rvert$	γ	α	$\lvert A\rvert$	dec. diag. stor. reqs.	t_n	t_r
42	8427	13159	31838	58750	34223	2531	0
42	3733	8260	21014	27433	20170	566	0
42	15715	28712	75836	134231	67964	14152	2
27	1887	2165	5480	5254	3939	39	0
27	11510	13566	30651	32488	17964	1619	0
27	3370	5122	9090	12123	7656	205	0
27	1714	1901	2845	4885	2372	22	0
42	14796	20638	25177	33996	19040	1379	2
36	5684	6473	10205	33831	7794	512	1
36	2092	3412	7342	9789	7403	86	0
39	3446	8116	13601	8998	20428	108	0
39	4813	8601	17606	15505	26538	313	0
45	13773	26780	73735	128306	63612	19318	1
45	22609	45119	118370	171071	68660	31434	9
45	9610	22202	43249	40825	58482	1592	0
42	6585	14967	30030	27562	30060	740	0
42	5806	9276	22043	40055	27752	866	0
48	18692	44411	91397	53362	76599	5962	1
39	9288	17802	43676	55320	41346	4174	1
48	32861	69247	129651	94059	141651	11179	1
42	37006	57230	129443	150807	138205	22030	3
42	17445	33188	98009	192840	76121	35909	3

an infinite set. But, if L_1 is a finite set, then L_2 will also be a finite set (because the number of states dominated by each state in L_1 is finite). Thus the algorithm terminates in a finite number of steps. □

6 Computational results

In this section we report a set of experiments that demonstrate the applicability and assess the efficacy of Algorithm 3, by applying this algorithm upon a number of randomly generated instantiations of the R/W-RAS class that was defined in Section 2. Each of the generated instances was further specified by:

- The number of conventional resource types in the system; the range of this parameter was between 3 and 16.
- The number of the R/W resource types in the system; the range of this parameter was between 1 and 4.
- The capacities of the conventional resource types in the system; the range of this parameter was between 1 and 4.
- The number of process types in the system; the range of this parameter was between 3 and 5.
- 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 R/W-RAS structure defined in Section 2, no processing stage has a zero resource-allocation vector.

Both the employed R/W-RAS generator and Algorithm 3 were encoded and compiled in C++. 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.

Table 8 reports the results that were obtained in our experiments. Column 1 in Table 8 reports the total number of processing stages ξ. Column 2 reports the cardinality of the set of minimal reachable unsafe states. Column 3 (γ) reports the number of minimal unsafe states generated by Algorithm 3 without performing the reachability evaluation of the generated states (Line 28). Column 4 (α) reports the total number of unsafe states added to Q throughout the course of the execution of Algorithm 3. Column 5 reports the cardinality of the list $\dot{A}$ at the end of the execution of Algorithm 3. Column 6 reports the total storage capacity, in terms of integer entries, that is required for the storage of the decision diagram that encodes $\bar{S}_{r\bar{s}}$. Finally, Columns 7 (t_n) and 8 (t_r) report, respectively, the amount of time (in seconds) spanned by Lines 1-26 and Line 27 of Algorithm 3.

It is clear from Table 8 that Algorithm 3 facilitates the effective computability and the development of a parsimonious representation of the maximally permissive DAP for R/W-RAS of extensive size and behavioral complexity. Also, as pointed out in earlier parts of this manuscript, the data presented in Table 8 further indicates that (i) the computational complexity of the algorithm is mostly dependent on the cardinality of the set $\dot{A}$, and that (ii) the empirical complexity of the computation in Line 27 of Algorithm 3 is very benign.

7 Extension of the presented results to encompass the complete process behavior that is exhibited by the Gadara RAS

The RAS model defined in Section 2 allows its constituent processes to possess routing flexibility; each process J_j, $j = 1,\ldots,n$, can execute through any of the paths connecting some source node $v \in \mathcal{V}_j^{\nearrow}$ to a node $v' \in \mathcal{V}_j^{\searrow}$ of the corresponding graph $\mathcal{G}_j$. On the other hand, by requesting that the graphs $\mathcal{G}_j$ are acyclic, that model precludes the presence of cyclic behavior in the defining logic of these processes. However, RAS models that seek to capture the behavior of the various threads of multi-threaded computer programs executing in their critical region, may need to express behavior that results from conditional loops like the "while" or the "until" loops that are encountered in these programs. This, in turn, requires the provision for potential cyclic behavior in the specification of the process-defining graphs $\mathcal{G}_j$. In this section, we extend the RAS model of Section 2, and the results that were developed in the previous parts of this work, so that they encompass potential cyclic behavior by the constituent processes. In particular, under the modeling extensions to be introduced in this section, the dynamics for the process routing to be supported by the resultant RAS class will be equivalent to the dynamics for the process routing that are supported by the Gadara RAS (Liao et al (2013b)); as remarked in the introductory section, the latter is an established abstraction in the current literature for modeling the dynamics of (mutex) lock allocation in multi-threaded programs.

The proposed modification for the RAS model of Section 2 can be described as follows: In the new RAS model, the process-defining graphs $\mathcal{G}_j$ still encode the sequential logic for the corresponding process types J_j as paths leading from some node $v \in \mathcal{V}_j^{\nearrow}$ to a node $v' \in \mathcal{V}_j^{\searrow}$, but these paths need not be acyclic anymore. Furthermore, following the rationale that underlies the specification of the Gadara RAS (Liao et al (2013b)), we separate the process routing dynamics from the dynamics that pertain to resource allocation. To formalize this idea, consider an edge (v,v') in some graph $\mathcal{G}_j$, $j = 1,\ldots,n$, and let $\Xi(v)$, $\Xi(v')$ denote the corresponding processing stages. Then, we stipulate the following condition for the extended R/W-RAS model that is considered in this section:

Condition 1 *Consider an instance Φ from the R/W-RAS class defined in this section, and an edge (v,v') in the corresponding "union" graph $\mathcal{G} = (\cup_{j=1}^{n}\mathcal{V}_j, \cup_{j=1}^{n}\mathcal{E}_j)$. If the out-degree*[8] *of node v in $\mathcal{G}$ is greater than one, then, $\mathcal{A}(\Xi(v)) = \mathcal{A}(\Xi(v'))$.*

The practical implication of Condition 1 is that, in the considered RAS, branching decisions by the running process instances can be effected without altering the resources allocated to these processes. A dual interpretation of this observation is that, in the considered RAS, the aforementioned branching decisions are not impacted by the underlying resource allocation function. These two remarks substantiate the aforementioned separation of the process routing dynamics from the dynamics that pertain to resource allocation. We further insist that this separation is respected by the applied DAP. Formally, we stipulate the following:

Condition 2 *Consider an instance Φ from the R/W-RAS class defined in this section, and an edge (v,v') in the corresponding "union" graph $\mathcal{G} = (\cup_{j=1}^{n}\mathcal{V}_j, \cup_{j=1}^{n}\mathcal{E}_j)$. If the out-degree of node v in $\mathcal{G}$ is greater than one, then, the state transitions that correspond to edge (v,v') in the state automaton $G(\Phi)$ are* uncontrollable *transitions.*

Conditions 1 and 2 render the process routing dynamics in the considered RAS equivalent to the process routing dynamics in the Gadara RAS. Then, one can establish the following result:[9]

Proposition 5 *Consider an instance Φ from the R/W-RAS class that is considered in this section. Then, for every unsafe state $\mathbf{u} \in S_{r\bar{s}}$, there exists a deadlock state $\mathbf{d} \in S_{rd}$ that is accessible from $\mathbf{u}$.*

Proof: Let $\mathbf{u}$ be some arbitrary unsafe state in the subspace $S_{r\bar{s}}$ of the considered RAS Φ, and also let $\mathcal{J}(\mathbf{u})$ denote the set of process instances that are active in state $\mathbf{u}$. For each process instance $j_j \in \mathcal{J}(\mathbf{u})$, also let $\Xi(j_j)$ denote the processing stage executed by j_j in $\mathbf{u}$, and $J(j_j)$ denote the corresponding process type; obviously, $J(j_j) \equiv J_k$ for some $k \in 1,\ldots,n$. Finally, for each process instance $j_j \in \mathcal{J}(\mathbf{u})$, pick a shortest path in the corresponding graph $\mathcal{G}_k$, that defines the process type J_k, leading from the node corresponding to $\Xi(j_j)$ to the set of terminal nodes of $\mathcal{G}_k$, $\mathcal{V}_k^{\searrow}$; let these paths be denoted by $p(j_j)$. For any R/W-RAS Φ with well-defined process types, there will exist at least one shortest path $p(j_j)$ for every $j_j \in \mathcal{J}(\mathbf{u})$, and these paths will exhibit no cyclic behavior (since, otherwise, an even shorter path can be obtained by removing the appearing cycles).

Next, consider any transition sequence σ in the automaton $G(\Phi)$ that emanates from state $\mathbf{u}$ and seeks to advance the active process instances $j_j \in \mathcal{J}(\mathbf{u})$ to their completion while following the pre-selected paths $p(j_j)$. The acyclic nature of the paths $p(j_j)$ implies that σ is finite; let $\hat{\mathbf{u}}(\sigma)$ denote the reached final state. The unsafety of state $\mathbf{u}$ further implies that $\hat{\mathbf{u}}(\sigma) \neq \mathbf{s}_0$. Let $\mathcal{J}(\hat{\mathbf{u}}(\sigma)) \subseteq \mathcal{J}(\mathbf{u})$ denote the process instances in $\mathbf{u}$ that have not completed in state $\hat{\mathbf{u}}(\sigma)$. Obviously, the inability of the process instances $j_j \in \mathcal{J}(\hat{\mathbf{u}}(\sigma))$ to advance any further towards the corresponding terminal stages in the paths $p(j_j)$ is due to the lack of some resource(s) needed for their next processing stage on $p(j_j)$; i.e., all these processes are deadlocked when restricted on the corresponding paths $p(j_j)$. But, then, Condition 1 also implies that the processing stages of the considered process instances j_j in state $\hat{\mathbf{u}}(\sigma)$

[8] We remind the reader that the out-degree of a node v in a digraph $\mathcal{G}$ is equal to the number of edges that emanate from v.

[9] This result is similar to a result that is established in the "$\Longleftarrow$" part of the proof for Theorem 1 in Liao et al (2013b). Here we state and prove the result in the context of the representational formalisms for the R/W-RAS and their behavioral dynamics that are employed in this work.

do not involve any branching. Hence, state $\hat{\mathbf{u}}(\sigma)$ belongs in S_{rd} and constitutes (one of) the deadlock state(s) $\mathbf{d}$ that is asserted by the considered proposition. □

Proposition 5 ensures that the fundamental idea underlying the methodology that is proposed in this work still applies in the dynamics of the considered RAS model: one can still try to retrieve all the unsafe states in S_r by backtracing from the reachable (minimal) deadlocks. However, the implementation of this idea in the dynamics of the modified R/W-RAS that is considered in this section, must also address any potential complications that might result from the uncontrollable dynamics that are introduced by Condition 2. The first issue to observe along these lines is that the introduced uncontrollability does not compromise the monotonicity of the (un-)safety property that is implied by Proposition 1. Indeed, consider an instance Φ from the considered R/W-RAS class, and a reachable unsafe state $\mathbf{u}$ of $G(\Phi)$. Under the presence of uncontrollable behavior, it is generally possible that the addition of a process instance j_j to state $\mathbf{u}$ might lead to a state $\mathbf{u}'$ that is safe. More specifically, this will happen if the unsafety of state $\mathbf{u}$ is due to some uncontrollable state transition that would take the system to its unsafe subspace $S_{\bar{s}}$, and the addition of process instance j_j to $\mathbf{u}$ blocks this transition by allocating to j_j some of the resources that are necessary for the execution of this transition. However, Conditions 1 and 2 imply that such a blocking effect is not possible in the considered RAS model, and therefore, the monotonicity of (un-)safety is preserved.[10]

Yet, while the argument that was provided in the previous paragraph establishes the preservation of the monotonicity of the (un-)safety property in the considered RAS, it also reveals a need for revising the concept of the state unsafety in the new RAS dynamics; the following definition provides this revision in an inductive manner:

Definition 7 *In the R/W-RAS class considered in this section, the revised set of reachable unsafe states, $S'_{r\bar{s}}$, is inductively defined as follows:*

1. *$S'_{r\bar{s}}$ contains the set $\mathcal{S}_{r\bar{s}}$ defined in Section 2, i.e., all the states $\mathbf{u} \in S_r$ that are not co-accessible to state $\mathbf{s}_0$.*
2. *Furthermore, $S'_{r\bar{s}}$ contains any state $\mathbf{u} \in S_r$ that has (i) an uncontrollable transition to $S'_{r\bar{s}}$ or (ii) all its transitions leading to $S'_{r\bar{s}}$.*

In the following, we shall denote the minimal elements of $S'_{r\bar{s}}$ by $\bar{S}'_{r\bar{s}}$. The new definition of the state unsafety necessitates some modification of the logic of Algorithm 3 in order to characterize correctly the (un-)safety of the various states that are generated during its execution. In particular, every time that the algorithm backtraces from some unsafe state $\mathbf{u}$ upon an uncontrollable transition, the generated state $\mathbf{u}'$ must be immediately recognized as an unsafe state. This modification affects the state classification that is performed in Line 7 of Algorithm 3. On the other hand, the same modification further implies that all the remaining states, that are processed through Lines 9-25 of the algorithm, are connected to the recognized unsafe region through controllable transitions only; hence, the logic of the "combine" and "confine" operations that are performed in those lines will remain unchanged.

Example 4: Consider the R/W-RAS configuration depicted in Figure 2. It has two conventional resource types, R_1 and R_2, both with single-unit capacities, and the R/W resource type RW_1. It also has two process types, J_1 and J_2. Process type J_1 contains the loop $\Xi_{11} \rightarrow \Xi_{12} \rightarrow \Xi_{14} \rightarrow \Xi_{11}$. Furthermore, Condition 2 implies that the branching transitions $\Xi_{12} \rightarrow \Xi_{13}$ and $\Xi_{12} \rightarrow \Xi_{14}$ (depicted by dashed lines) are uncontrollable. It can also be checked that $\mathcal{A}(\Xi_{12}) = \mathcal{A}(\Xi_{13}) = \mathcal{A}(\Xi_{14})$, i.e., the branching transitions do not alter the

[10] It is interesting to notice that the preservation of the monotonicity of (un-)safety in the face of the underlying uncontrollable behavior has been accepted rather silently in the previous works on the Gadara RAS.

Table 9: Tracing back from the minimal unsafe states for Example 4

$\mathbf{s}$	*Safe_Prev*	*Unsafe_Prev*
$\mathbf{s}_{10} = \{\Xi_{11}, \Xi_{14}\}$	$\emptyset$	$\mathbf{s}_5 = \{\Xi_{11}, \underline{\Xi}_{12}\}$
$\mathbf{s}_{12} = \{\Xi_{21}, \Xi_{13}\}$	$\emptyset$	$\mathbf{s}_6 = \{\Xi_{21}, \underline{\Xi}_{12}\}$
$\mathbf{s}_5 = \{\Xi_{11}, \Xi_{12}\}$	$\emptyset$	$\emptyset$
$\mathbf{s}_6 = \{\Xi_{21}, \Xi_{12}\}$	$\emptyset$	$\mathbf{s}_4 = \{\Xi_{21}, \underline{\Xi}_{11}\}$
$\mathbf{s}_4 = \{\Xi_{21}, \Xi_{11}\}$	$\emptyset$	$\mathbf{s}_{13} = \{\Xi_{21}, \underline{\Xi}_{14}\}$
$\mathbf{s}_{13} = \{\Xi_{21}, \Xi_{14}\}$	$\emptyset$	$\mathbf{s}_6 = \{\Xi_{21}, \underline{\Xi}_{12}\}$

resource allocation to the corresponding process instances, and thus, Condition 1 is also satisfied. We further notice that the R/W resource RW_1 is requested by the last processing stage of J_1 in the reading mode, and by the first processing stage of J_2 in the writing mode. Finally, processing stage Ξ_{22} requests the allocation of both R_1 and R_2.

As indicated in Section 4, terminal processing stages do not participate in any deadlock formation. Hence, they need not be explicitly considered in the subsequent analysis. In the context of the considered example, the implicit representation of the terminal processing stages in the modeling of the underlying RAS dynamics enables the representation of these dynamics by a *finite* state automaton (FSA), since the only stage that can support an infinite number of active process instances is the terminal processing stage Ξ_{15}. The state transition diagram (STD) of this FSA is depicted in Figure 3. Using co-reachability analysis on the STD of Figure 3, we can see that the set of states $S_{r\bar{s}}$, from which state $\mathbf{s}_0$ is not accessible, equals to $\{\mathbf{s}_4, \mathbf{s}_6, \mathbf{s}_{10}, \mathbf{s}_{11}, \mathbf{s}_{12}, \mathbf{s}_{13}, \mathbf{s}_{14}, \mathbf{s}_{15}\}$; this set of states are marked in gray in Figure 3. On the other hand, the highlighted state $\mathbf{s}_5$ has the following path to sate $\mathbf{s}_0$: $\mathbf{s}_5 \rightarrow \mathbf{s}_9 \rightarrow \mathbf{s}_1 \rightarrow \mathbf{s}_3 \rightarrow \mathbf{s}_7 \rightarrow \mathbf{s}_0$. But state $\mathbf{s}_5$ enables the uncontrollable transition $\Xi_{12} \rightarrow \Xi_{14}$ that leads to state $\mathbf{s}_{10}$, a member of $S_{r\bar{s}}$. Hence $\mathbf{s}_5$ is an unsafe state, and $S'_{r\bar{s}} = S_{r\bar{s}} \cup \{\mathbf{s}_5\}$. It can be easily verified that $\bar{S}_{r\bar{s}} = \{\mathbf{s}_4, \mathbf{s}_6, \mathbf{s}_{10}, \mathbf{s}_{12}, \mathbf{s}_{13}\}$, and that $\bar{S}'_{r\bar{s}} = \bar{S}_{r\bar{s}} \cup \{\mathbf{s}_5\}$. Finally, the set of minimal reachable deadlocks is $\bar{S}_{rd} = \{\mathbf{s}_{10}, \mathbf{s}_{12}\}$.

Next, we demonstrate the application of Algorithm 3 on the considered R/W-RAS. Applying Lines 1-10 of Procedure 2, we get that $BlockPs[\Xi_{11}] = \{\{\Xi_{11}, \Xi_{13}\}, \{\Xi_{11}, \Xi_{14}\}\}$, $BlockPs[\Xi_{12}] = \emptyset$, $BlockPs[\Xi_{13}] = \{\{\Xi_{13}, \Xi_{21}\}\}$, $BlockPs[\Xi_{14}] = \{\{\Xi_{14}, \Xi_{11}\}\}$, $BlockPs[\Xi_{21}] = \{\{\Xi_{21}, \Xi_{11}\}, \{\Xi_{21}, \Xi_{13}\}, \{\Xi_{21}, \Xi_{14}\}\}$. Subsequently, applying the rest of Procedure 2, we get the minimal deadlock states $\bar{S}_{rd} = \{\mathbf{s}_{10}, \mathbf{s}_{12}\}$.

Table 9 depicts the minimal unsafe states obtained by applying Algorithm 3, and the result of backtracing from each of them. The underlined processing stages indicate the source nodes of the backtraced edges in the corresponding states; double underlining further implies backtracing on uncontrollable transitions. As a more concrete example, backtracing on the uncontrollable transition $(\Xi_{12} \rightarrow \Xi_{14})$ from the minimal unsafe state $\mathbf{s}_{10}$ yields the minimal unsafe state $\mathbf{s}_5$. In this example, backtracing from the unsafe states does not yield any safe states. Hence, Lines 9-23 in Algorithm 3 are not executed for this example. The algorithm yields the set of states $\{\mathbf{s}_4, \mathbf{s}_5, \mathbf{s}_6, \mathbf{s}_{10}, \mathbf{s}_{12}, \mathbf{s}_{13}\}$, which is identical to the set $\bar{S}'_{r\bar{s}}$ obtained using co-reachability analysis. □

We close the developments of this section by formally stating and proving the correctness of Algorithm 3 when applied to the enumeration of the reachable minimal unsafe states of the R/W-RAS that are considered in this section. The finite convergence of the algorithm can be established exactly in the same way as the finite convergence of its counterpart presented in Section 5.

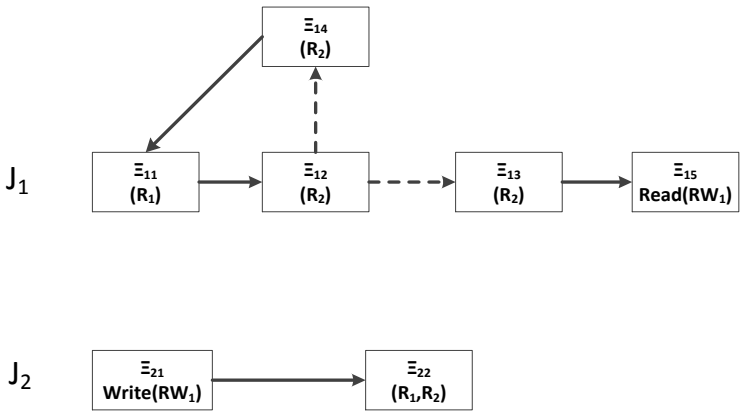

Fig. 2: The R/W-RAS considered in Example 4. Dashed lines correspond to uncontrollable transitions.

Theorem 2 *Under the unsafety interpretation of Definition 7, Algorithm 3 will enumerate correctly all the minimal reachable unsafe states of any instance* Φ *from the R/W-RAS considered in this section.*

Proof: Consider the set of minimal unsafe states $\bar{S}'_{r\bar{s}}$ of an instance Φ from the considered R/W-RAS, and the transition structure that is induced upon this set by the sets $nextMin(\mathbf{u}, e_i)$ that were introduced in Definition 4; that is, in the considered transition system, there is a transition $(\mathbf{u}, \mathbf{u}')$ with $\mathbf{u}, \mathbf{u}' \in \bar{S}'_{r\bar{s}}$, iff $\mathbf{u}' \in nextMin(\mathbf{u}, e_i)$ for some enabled event $e_i \in g(\mathbf{u})$. Next, we also consider the state transition diagram (STD) of this transition system and the communication structure that is present in this diagram.[11] Proposition 5 implies that all the terminal nodes in the corresponding condensation graph are defined by the minimal deadlocks in $\bar{S}'_{r\bar{s}}$. Furthermore, since the model and the algorithm modifications presented in the earlier parts of this section do not affect the notion of the minimal deadlock for the considered RAS, Theorem 1 implies that all the minimal deadlocks in $\bar{S}'_{r\bar{s}}$ will be correctly identified by Algorithm 3 through the execution of Line 2. Next, we use an inductive argument, that employs a partial order of the nodes in the aforementioned STD condensation, in order to establish that Algorithm 3 will detect all the communication classes of $\bar{S}'_{r\bar{s}}$ and all the states that are included in each communication class.

More specifically, let us consider the imposition of a partial order on the nodes of the condensation of the considered STD, based on a "leveling" scheme that is defined as follows: Level 0 collects all the terminal nodes of this condensation. Level 1 collects all the nodes in the STD condensation with emanating edges leading only to Level-0 nodes. Level 2 collects those nodes of the STD condensation with emanating edges leading to nodes in Levels

[11] We remind the reader that two nodes v, v' in a digraph $\mathcal{G} = (V, E)$ are communicating if there are directed paths in $\mathcal{G}$ that lead from each of these two nodes to the other one. Nodal communication defines an equivalence relationship on the node set V of $\mathcal{G}$ and the corresponding equivalence classes are known as the communication classes of $\mathcal{G}$. The *condensation* of $\mathcal{G}$ that is induced by this relationship, is the digraph $\hat{\mathcal{G}}$ that is obtained by collapsing each communication class to a single (macro-)node while retaining all edges that connect nodes in different communication classes. By its construction, $\hat{\mathcal{G}}$ is an acyclic digraph.

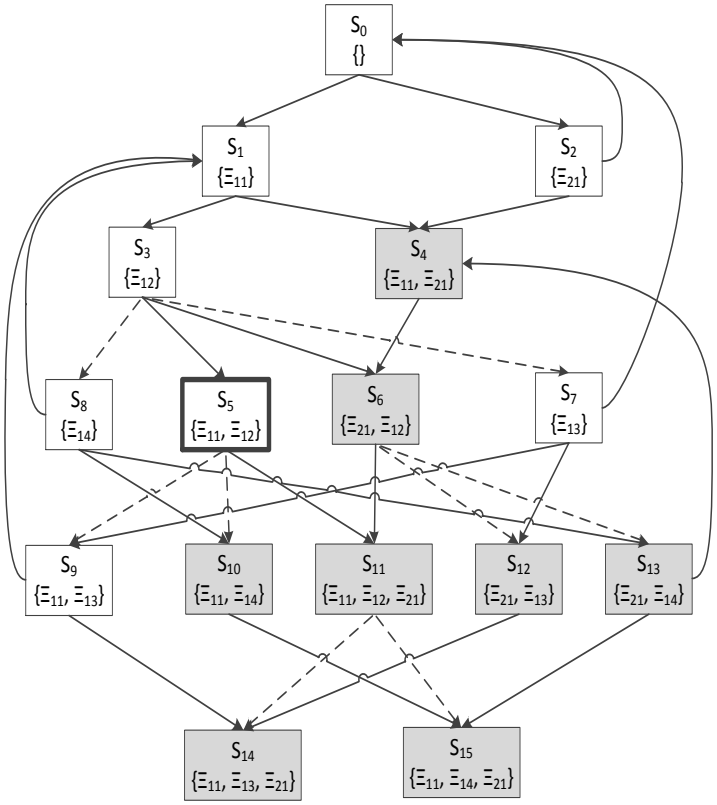

Fig. 3: The finite state automaton modeling the (simplified) dynamics of the R/W-RAS instantiation considered in Example 4. Dashed lines correspond to uncontrollable transitions. The gray states are the members of $S_{r\bar{s}}$, whereas the highlighted state $\mathbf{s}_5$ is the only member of the revised set of reachable unsafe states $S'_{r\bar{s}}$ that does not belong to $S_{r\bar{s}}$.

0 and 1 (we notice, however, that each Level-2 node must have at least one emanating edge leading to a Level-1 node, since, otherwise, it would be a Level-1 node). Generalizing the above pattern, Level i of the considered partial order, collects the nodes of the STD condensation with emanating edges to nodes in Levels $0,1,\ldots,i-1$, and there must be at least one edge leading to a Level-$(i-1)$ node. The aforementioned induction runs on this leveling structure. The base case, concerning the Level-0 nodes, was already addressed in the previous paragraph. Next, suppose that Algorithm 3 has already identified all the states in all the communicating classes that are contained in Levels $0,\ldots,i$ of the considered leveling scheme. We shall show that the algorithm must also discover all the states in any communicating class that corresponds to a Level-$(i+1)$ node in the condensed STD.

First, we assume that the considered communication class is a singleton, i.e., it consists of a single minimal state $\mathbf{u} \in \bar{S}'_{r\bar{s}}$. Then, all the edges emanating from $\mathbf{u}$ must be leading to minimal states in communicating classes belonging in Levels $0,1,\ldots,i$, and therefore,

already detected by the algorithm (according to the inductive hypothesis). But then, an argument very similar to that for the establishment of Proposition 4 in Nazeem and Reveliotis (2014) will establish that the considered algorithm will identify state $\mathbf{u}$ as well.[12]

Next we address the case where the considered communication class contains two or more states from $\bar{S}'_{r\bar{s}}$. For this case, the following remarks are in order: First we notice that the communicating relationship of all the states that belong to the considered class, together with the fact that the events e_i that define the underlying transition mechanism do not involve the loading of any new process instances, imply that all these states will have the same process content. This realization further implies that once the considered algorithm identifies any state in the considered class, then it is guaranteed to identify any other state in it through the basic backtracing step that is performed in Lines 6-8. It remains to characterize a state in the considered communication class that is guaranteed to be traceable by Algorithm 3. The communicating structure of the considered state class, when combined with the underlying dynamics that govern the process routing and the transition mechanism of the considered STD, imply that there will be some state $\mathbf{u}$ of this class with a state transition $(\mathbf{u},\mathbf{u}')$ to some lower level class in the STD condensation, and this transition will correspond to a process-branching event e'. We claim that the set $nextMin(\mathbf{u},e')$ that results from the aforementioned transition is the singleton $\{\mathbf{u}'\}$, and therefore, the detection of state $\mathbf{u}'$ by Algorithm 3, according to the induction hypothesis, guarantees also the detection of state $\mathbf{u}$, through the basic backtracing step that is performed in Lines 6-8. We establish the above claim, by referring to the topological structure that is depicted in Figure 1. Suppose that $nextMin(\mathbf{u},e') \neq \{\mathbf{u}'\}$, and pick a state $\mathbf{z}'_i$ in $nextMin(\mathbf{u},e')$. Also, let $\mathbf{s}'_i$ denote the state that results by backtracing from $\mathbf{z}'_i$ upon event e'. The discussion in the paragraph that follows Definition 4, in Section 5, guarantees the feasibility of this backtracing step, and it also requires the safety of state $\mathbf{s}'_i$. But state $\mathbf{s}'_i$ is connected to the unsafe state $\mathbf{z}'_i$ through an uncontrollable transition, and therefore, it must be an unsafe state; this contradiction establishes the validity of our claim, and also concludes the argumentation of the second case considered the inductive step.

The proof completes by noticing that the two cases that were addressed in the previous paragraphs cover all the communication classes that are contained in Level $(i+1)$ of the condensed STD. □

8 Conclusion

This work has extended the definition of the RAS abstraction to encompass the dynamics of R/W-locks. It also proposed an algorithm for the effective enumeration of the set of minimal unsafe states for any given R/W-RAS, and established the significance of this capability for the effective implementation of the maximally permissive DAP in the context of this new RAS class. The presented results address the behavioral scope of the Gadara RAS, which is a well established abstraction for modeling the dynamics w.r.t. the lock allocation that takes place in multi-threaded programming.

Future work will seek to extend the presented results to R/W-RAS with even more complex process behavior. It will also consider the plausibility of employing symbolic computation for the support of the computation that is involved in the proposed algorithms, and the potential gains that might result from such an endeavor.

[12] For the sake of brevity, we refer to Nazeem and Reveliotis (2014) for the relevant details.

References

Banaszak ZA, Krogh BH (1990) Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. IEEE Trans on Robotics and Automation 6:724–734

Cassandras CG, Lafortune S (2008) Introduction to Discrete Event Systems (2nd ed.). Springer, NY, NY

Chen HC, Chen CL (2009) On minimal elements of upward-closed sets. Theoretical Computer Science 410:2442–2452

Coffman EG, Elphick MJ, Shoshani A (1971) System deadlocks. Computing Surveys 3:67–78

Commer P, Sethi R (1977) The complexity of TRIE index construction. Journal of the ACM 24:428–440

Courtois P, Heymans F, Parnas D (1971) Concurrent control with "readers" and "writers". Communications of the ACM 14(10):667–668

Dickson L (1913) Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. American Journal of Mathematics 35(4):413–422

Dijkstra EW (1965) Cooperating sequential processes. Tech. rep., Technological University, Eindhoven, Netherlands

Ezpeleta J, Colom JM, Martinez J (1995) A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Trans on Robotics and Automation 11:173–184

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

Holt RD (1972) Some deadlock properties of computer systems. ACM Computing Surveys 4:179–196

Huang Y, Jeng M, Xie X, Chung D (2006) Siphon-based deadlock prevention policy for flexible manufacturing systems. IEEE Trans on Systems, Man and Cybernetics, Part A: Systems and Humans 36(6):1248–1256

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

Liao H, Lafortune S, Reveliotis S, Wang Y, Mahlke S (2013a) Optimal liveness-enforcing control of a class of Petri nets arising in multithreaded software. IEEE Trans Autom Control 58:1123–1138

Liao H, Wang Y, Cho HK, Stanley J, Kelly T, Lafortune S, Mahlke S, Reveliotis S (2013b) Concurrency bugs in multi-threaded software: Modeling and analysis using Petri nets. Discrete Event Systems: Theory and Applications 23:157–195

Liao H, Wang Y, Stanley J, Lafortune S, Reveliotis S, Kelly T, Mahlke S (2013c) Eliminating concurrency bugs in multithreaded software: a new approach based on discrete-event control. IEEE Trans Control System Technology 21:2067–2082

Nazeem A (2012) Designing parsimonious representations of the maximally permissive deadlock avoidance policy for complex resource allocation systems through classification theory. PhD thesis, Georgia Institute of Technology, Atlanta, GA

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

Nazeem A, Reveliotis S (2012) Designing maximally permissive deadlock avoidance policies for sequential resource allocation systems through classification theory: the non-

linear case. IEEE Trans on Automatic Control 57(7):1670–1684
Nazeem A, Reveliotis S (2014) An efficient algorithm for the enumeration of the minimal unsafe states in complex resource allocation systems. IEEE Trans on Automation Science and Engineering 11:111–124
Nazeem A, Reveliotis S, Wang Y, Lafortune S (2011) 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
Park J (2004) A deadlock and livelock free protocol for decentralized internet resource coallocation. IEEE Trans on Systems, Man and Cybernetics, Part A 34:123–131
Peterson JL (1981) Operating System Concepts. Addison-Wesley, Boston, MA
Reveliotis S, Roszkowska E (2011) Conflict resolution in free-ranging multi-vehicle systems: a resource allocation paradigm. IEEE Trans on Robotics 27:283–296
Reveliotis SA (1996) Structural analysis & control of flexible manufacturing systems with a performance perspective. PhD thesis, University of Illinois, Urbana, IL
Reveliotis SA (2000) Conflict resolution in AGV systems. IIE Trans 32(7):647–659
Reveliotis SA (2005) Real-time Management of Resource Allocation Systems: A Discrete Event Systems Approach. Springer, NY, NY
Reveliotis SA, Ferreira PM (1996) Deadlock avoidance policies for automated manufacturing cells. IEEE Trans on Robotics & Automation 12:845–857
Roszkowska E, Reveliotis S (2008) On the liveness of guidepath-based, zoned-controlled, dynamically routed, closed traffic systems. IEEE Trans on Automatic Control 53:1689–1695
Valk R, Jantzen M (1985) The residue of vector sets with applications to decidability problems in Petri nets. Acta Informatica 21(6):643–674
Viswanadham N, Narahari Y, Johnson TL (1990) Deadlock avoidance in flexible manufacturing systems using Petri net models. IEEE Trans on Robotics and Automation 6:713–722
Wang Y, Liao H, Nazeem A, Reveliotis S, Kelly T, Mahlke S, Lafortune S (2009) Maximally permissive deadlock avoidance for multithreaded computer programs. In: Proceedings of the 5th IEEE Conf. on Automation Science and Engineering, pp 37–41
Wang Y, Cho H, Liao H, Nazeem A, Kelly T, Lafortune S, Mahlke S, Reveliotis S (2010) Supervisory control of software execution for failure avoidance: Experience from the Gadara project. In: Proceedings of the 10th Intl. Workshop on Discrete Event Systems
Wu N, Zhou M (2007) Deadlock and blocking-free shortest routing of bi-directional automated guided vehicles. IEEE Trans on Mechatronics 12:63–72
Zhou M, Fanti MP (2004) Deadlock Resolution in Computer-Integrated Systems. Marcel Dekker, Inc., Singapore