An electronic supplement to the manuscript "Maximal linear deadlock avoidance policies for complex resource allocation systems: characterization, computation and approximation"

Michael Ibrahim, Spyros Reveliotis and Ahmed Nazeem

***Abstract*—This document constitutes an "electronic supplement" to the manuscript entitled "Maximal linear deadlock avoidance policies for complex resource allocation systems: characterization, computation and approximation", that is co-authored by the same authors, providing some supporting, yet important, material for the algorithmic developments that are presented in that original document.**

I. Introduction

As indicated by the title, this document is an "electronic supplement" to the manuscript of [1] that is co-authored by the same authors, and it complements the material that is presented in [1] in an essential manner. In more specific terms, the primary roles of this document can be stated as follows: (i) It provides a complete formal proof for Proposition 4 of [1]. (ii) It describes in detail the main data structures and the primary procedures that are used in the implementation of the algorithms that are presented in that document. The provided information on these items is essential for an effective implementation of the algorithms that are presented in [1]. (iii) It also presents a systematic complexity analysis for all the algorithmic procedures that are covered herein, and it further uses this analysis in order to characterize the worst-case complexity of the two primary algorithms that were presented in [1].

The rest of this document is organized into four major sections, with each section addressing distinct themes in the developments that are presented in [1]. Also, the propositions, algorithms and tables that are presented in this document, are numbered in a way that establishes a "continuity" with the respective numbering schemes of [1].

II. A complete proof for Proposition 4 of [1]

This section provides a complete proof for Proposition 4 of [1]. We start by restating this proposition for the readers' convenience.

Proposition 4: The sets $S_a(\Delta(\bar{S}_r))$ corresponding to the policies $\Delta(\bar{S}_r)$ that are generated by Algorithm 1 of [1] through Equation 21 of that document, satisfy the "monotonicity" condition of Definition 4 in [1].

M. Ibrahim is with the Department of Computer Engineering, Cairo University, Egypt; email: michael.nawar@eng.cu.edu.eg. S. Reveliotis is with the School of Industrial & Systems Engineering, Georgia Institute of Technology; email: {spyros@isye}.gatech.edu. A. Nazeem is with Facebook; email: nazeem.35@gmail.com This work has been partially supported by NSF grant ECCS-1707695.

Proof: As remarked in [1], we shall establish the result of Proposition 4 through a double induction, where the outer induction will run on the sets that enter list $EXPLORE$, and the inner induction will be with respect to the state sets that are pruned during the corresponding iterations that take place in Algorithm 2 of [1] (i.e., in the 'WHILE'-loop of Lines 5–7 in that algorithm). Also, in the rest of the proof, we shall write S_a instead $S_a(\Delta(\bar{S}_r))$, in order to simplify the corresponding notation.

As the base case for the outer part of the pursued induction, we notice that the set S_a that is induced by the first state set to enter list $EXPLORE$, during the initialization phase of Algorithm 1 of [1], is the set $S_{rs} \cup S_{\bar{r}s} = S_s$, which satisfies the "monotonicity" condition of Definition 4 of [1] by Proposition 1 of that document.

Next, suppose that the "monotonicity" property is possessed by the first k entries to list $EXPLORE$, and consider the set S_a that corresponds to entry $(k+1)$ to this list. We shall show that this set possesses the desired "monotonicity" property by showing that, for every state $\mathbf{s}$ pruned from this set by function $PRUNE$, all states $\mathbf{s}''$ s.t. $\mathbf{s}'' > \mathbf{s}$ will be pruned, as well.

We establish this result, using the second induction. For the base case of this induction, we establish the aforementioned result for those states that are removed by the first execution of the "While" loop of Lines 5–7 in $PRUNE$. Hence, let $\mathbf{s}'$ denote the maximal state that was removed from the "parent" set S'_a during the generation of the considered set S_a, and $\mathbf{s}$ denote a state that was removed from the newly generated set S_a during the first iteration of the "while" loop in Algorithm 2 of [1].

From the logic that drives this pruning process, it follows that the set $\Gamma(\mathbf{s}) \cap (E \setminus E^{\nearrow})$ is a singleton, containing an event e that leads to the removed maximal state $\mathbf{s}'$. Furthermore, event e cannot be an unloading event, since, then, we shall have $\mathbf{s} > \mathbf{s}'$, and $\mathbf{s}'$ is not maximal. Since event e is not a loading event either, it follows that the pruned state $\mathbf{s}$ must contain the same set of process instances with state $\mathbf{s}'$.

Next, we use the above facts to show that any state that dominates state $\mathbf{s}$ in S'_a will also be pruned from S_a by the pruning function of Algorithm 2 of [1]. Hence, consider a state $\mathbf{s}'' \in S'_a$ with $\mathbf{s}'' > \mathbf{s}$. According to the D/C-RAS dynamics, the presence of the extra processes in state $\mathbf{s}''$ can only hinder the further progress of the common processes of the state $\mathbf{s}''$ with the state $\mathbf{s}$. Hence, the only available move in $\mathbf{s}''$ for this last set of processes, is the event e that led from $\mathbf{s}$ to $\mathbf{s}'$. If this event can be executed in $\mathbf{s}''$ while some of the extra processes are still present in the system, then, for the resulting

state $\mathbf{s}''''$, it will hold $\mathbf{s}''' > \mathbf{s}'$, which contradicts the presumed maximality of $\mathbf{s}'$ in S'_a. Hence, the extra processes in state $\mathbf{s}''$ must be cleared before the remaining processes in that state can move. But then, any process-completing path for state $\mathbf{s}''$ must still go through state $\mathbf{s}$, and therefore, state $\mathbf{s}''$ must be pruned as well.

For the inductive step of the inner induction, suppose that the inductive hypothesis holds for the first k executions of the "while" loop of function $PRUNE$, and consider a state $\mathbf{s}$ that is pruned during the $(k+1)$ iteration of this loop. If this state is maximal in S'_a, then, its removal from S_a cannot impair the "monotonic" structure of this set. If, on the other hand, there exists $\mathbf{s}' \in S'_a$ s.t. $\mathbf{s}' > \mathbf{s}$, then, we discern two cases for the advancement of the extra processes in $\mathbf{s}'$: If the advancement of these processes is interleaved with the advancement of some processes that are also present in state $\mathbf{s}$, then, any such interleaving will result in a state $\mathbf{s}''$ that dominates the corresponding state $\mathbf{s}'''$ that would result from the advancement of the same processes in $\mathbf{s}$. But state $\mathbf{s}'''$ is a state that must have been removed in the previous iterations of the considered "while" loop in the $PRUNE$ function, and, according to the inductive hypothesis, state $\mathbf{s}''$ has been pruned as well. If, on the other hand, the extra processes in state $\mathbf{s}'$ must complete before any further advancement of the common processes in $\mathbf{s}'$ and $\mathbf{s}$, then, the completion of these extra processes would result in state $\mathbf{s}$, which is pruned according to the working hypothesis. Hence, state $\mathbf{s}'$ must also be pruned. □

III. The main data structures employed in the algorithms of [1] and their maintenance

A. The class RAS

In our implementation of Algorithms 1 and 3 of the main manuscript, we organized all the information processed by these algorithms in a class that was named RAS and consisted of two parts:

- A "static" part that is shared by all the instances of this class, and provides an effective and efficient representation of the state transition diagram (STD) of the underlying RAS Φ. This part is computed only once at the beginning of the algorithm execution.
- A "dynamic" part that is distinct for each object that instantiates this class, and provides an effective and efficient representation of the tentative policies Δ that are generated by the conducted search process.

Both parts of class RAS are detailed in Table IV. When considering the algorithm semantics that are introduced in Sections IV.B and IV.D of [1], it should be clear that each instance of class RAS essentially collects all the information that is stored and processed at a single node of the search processes that are conducted by Algorithms 1 and 3 in that document. Next, we discuss how the different data structures appearing in Table IV are built and maintained by these algorithms every time that they generate an instance from class RAS.

B. Representing and evaluating the maximally permissive DAP Δ^* through the class RAS

Given a certain D/C-RAS Φ, Algorithms 1 and 3 of [1] will compute the static part of the class RAS at the initializing phase of their execution. The computation of the values for the first four data structures in this part is performed through standard reachability analysis [2] conducted on the state space of Φ. During the same phase, the hashmap DominatedBy is also easily constructed by means of the list ReachableStates. The corresponding procedure will scan the list ReachableStates and, for each state $\mathbf{s}$ that is stored in this list, it will (i) generate the states $\mathbf{s}_i = \mathbf{s} + \mathbf{e}_i,\ i = 1, \ldots, \xi$, where $\mathbf{e}_i$ denotes the corresponding unit vector with the single unit element at its i-th component, and (ii) will register each of these generated states $\mathbf{s}_i$ in the hashmap DominatedBy if they constitute reachable states; this last check can be done efficiently through the hashset ReachableStateSet.

The complexity of the entire construction of the static part of the class RAS, that was described in the previous paragraph, is $O(|S_r| \cdot (\bar{\Gamma} + \xi))$, where, according to the notation of [1], (i) $|S_r|$ is the cardinality of the reachable subspace S_r of the considered RAS Φ, (ii) $\bar{\Gamma} = \max_{\mathbf{s} \in S_r}\{|\Gamma(\mathbf{s})|\}$, with $\Gamma(\mathbf{s})$ denoting the set of feasible events at state $\mathbf{s}$, and (iii) ξ is the dimensionality of the RAS states $\mathbf{s}$. The term $(|S_r| \cdot \Gamma)$ in the above expression concerns the complexity of the construction of the first three lists of the static part of the class RAS, while the term $(|S_r| \cdot \xi)$ characterizes the complexity of the construction of the hashmap DominatedBy. Furthermore, since $\Gamma(\mathbf{s}) \leq \xi,\ \forall \mathbf{s} \in S_r$, we can eventually think of this complexity as $O(|S_r| \cdot \xi)$.

The list IsAdmissible belonging to the first instance of the class RAS that is built by Algorithms 1 and 3, is obtained through standard co-reachability analysis [2] on the STD of the

TABLE IV: The main data structures that are employed by class RAS for the representation of the underlying RAS dynamics and the various policies Δ evaluated by Algorithms 1 and 3 of [1].

Class RAS: Static Part
ReachableStates: A dynamic list containing all reachable states of the considered RAS. NextStates: A list of hashsets mapping each reachable state to the set of states that are immediately reachable from it. PrevStates: A list of hashsets mapping each reachable state to the set of reachable states that are immediately backward reachable from it. ReachableStatesSet: A hashset that enables testing of membership in list ReachableStates, for any state $\mathbf{s}$, in $O(1)$. DominatedBy: A hashmap mapping each reachable state to a hashset of the reachable states that dominate it by one extra process instance; this data structure is used to facilitate the computation of maximal safe states and minimal unsafe states.
Class RAS: Dynamic Part (provides effective representation to a tentative policy Δ represented by state set $\bar{S}_r$)
IsAdmissible: A list of Boolean entries used to mark each reachable state as admissible or not by the considered policy Δ. MaxAdmissible: A hashset of the maximal admissible states under the considered policy Δ. This is set $\bar{S}_r$ in the semantics of Algorithms 1and 3. MinBoundaryInadmissible: A hashset of the minimal boundary inadmissible states under the considered policy Δ. This is set $\bar{S}_r^b$ in the semantics of Algorithms 1and 3. InseparableMinBoundaryInadmissible: A hashset of the minimal boundary inadmissible states that are not linearly separable from the maximal admissible states under the considered policy Δ.

Algorithm 4 Calculates Maximal Admissible States

Input: `IsAdmissible`
Output: `MaxAdmissible`
1: **for all** $\mathbf{s}$ in `IsAdmissible` **do**
2: TEST := TRUE;
3: **for all** $\mathbf{s}'$ in `DominatedBy`[$\mathbf{s}$] **do**
4: **if** `IsAdmissible`[$\mathbf{s}'$] **then**
5: TEST := FALSE; GOTO step 8;
6: **end if**
7: **end for**
8: **if** TEST **then**
9: `MaxAdmissible`.add($\mathbf{s}$);
10: **end if**
11: **end for**
12: **return** `MaxAdmissible`;

underlying RAS Φ with respect to its initial (and also marked) state $\mathbf{s}_0$. This computation will return the set S_{rs} as the set of states that is admissible by the DAP Δ^*, which is the policy assessed by Algorithms 1 and 3 at this initial phase, and the corresponding computational cost is also $O(|S_r| \cdot \bar{\Gamma})$.

The obtained list `IsAdmissible` is subsequently processed through Algorithm 4 in order to obtain the hashset `MaxAdmissible` that collects the maximal elements of this list. By making use of the pre-constructed hashmap `DominatedBy`, Algorithm 4 computes the sought set `MaxAdmissible` with complexity $O(|S_{rs}| \cdot \xi)$, where (i) $|S_{rs}|$ is the cardinality of the input set S_{rs} to Algorithm 4, and, as already noticed, (ii) ξ is the dimensionality of the stored state vectors $\mathbf{s}$ of Φ (according to the definition of the hashmap `DominatedBy`, the dimensionality ξ of the considered state vectors determines the execution of the inner loop of Algorithm 4).

For the first instantiation of the class `RAS` in Algorithms 1 and 3, the hashset `MinBoundaryInadmissible` is obtained by feeding the set of inadmissible states to Algorithm 5; this last set can be obtained straightforwardly from the content of the list `IsAdmissible`, with complexity $O(|S_r|)$. The first part (Lines 1–12) of Algorithm 5 identifies the set of the boundary inadmissible states, $S^b_{r\bar{s}}$, by checking their one-step reachability from some admissible state. As in the case of the computation of the list `IsAdmissible`, the complexity of this first part of Algorithm 5 is $O(|S_{r\bar{s}}| \cdot \bar{\Gamma})$. The second part (Lines 13–23) of Algorithm 5 uses the `DominatedBy` hashmap in order to identify the minimal elements of the set $S^b_{r\bar{s}}$, that is computed in the first part of the algorithm. The complexity of this particular computation is $O(|S^b_{r\bar{s}}| \cdot \xi)$.

For a well-formed RAS Φ, the maximally permissive DAP Δ^* is known to be complete, by the definition of this policy; hence, there is no need for testing this particular DAP for completeness.

On the other hand, the linearity of the DAP Δ^* can be assessed by using the computed lists `MaxAdmissible` and `MinBoundaryInadmissible` in order to perform the corresponding test that is defined by the LP of Equations 17-19 in [1]. This test will either recognize the DAP Δ^* as linear, or, in the opposite case, it will return the states $\mathbf{u} \in \bar{S}^b_{r\bar{s}} \cap conv(S_{rs})$, i.e., the content of the `InseparableMinBoundaryInadmissible` hash-

Algorithm 5 Calculates Minimal Boundary Inadmissible States

Input: STATES
Output: `MinBoundaryInadmissible`
1: `MinBoundaryInadmissible` := STATES;
2: **for all** $\mathbf{s}$ in STATES **do**
3: PrevAdmissible := FALSE;
4: **for all** $\mathbf{s}'$ in `PrevStates`[$\mathbf{s}$] **do**
5: **if** `IsAdmissible`[$\mathbf{s}'$] **then**
6: PrevAdmissible := TRUE;
7: **end if**
8: **end for**
9: **if** $\neg$ PrevAdmissible **then**
10: `MinBoundaryInadmissible`.remove($\mathbf{s}$);
11: **end if**
12: **end for**
13: `nonMinBoundaryInadmissible`:= $NULL$;
14: **for all** $\mathbf{s}$ in `MinBoundaryInadmissible` **do**
15: **for all** $\mathbf{s}'$ in `DominatedBy`[$\mathbf{s}$] **do**
16: `nonMinBoundaryInadmissible`.add($\mathbf{s}'$);
17: **end for**
18: **end for**
19: **for all** $\mathbf{s}$ in `nonMinBoundaryInadmissible` **do**
20: **if** `MinBoundaryInadmissible`.contains($\mathbf{s}$) **then**
21: `MinBoundaryInadmissible`.remove($\mathbf{s}$);
22: **end if**
23: **end for**
24: **return** `MinBoundaryInadmissible`;

set, which is the last component of the developed object `RAS` for the representation of the policy Δ^* (c.f. Table IV).

The computational complexity of the linearity test that is described in the previous paragraph can be characterized as follows: First, we notice that the considered test will solve $|\bar{S}^b_{r\bar{s}}|$ LPs, with each LP involving $\xi+1$ variables and $|\bar{S}_{rs}|+1$ "technological" constraints.[1] Furthermore, it is known that the worst-case complexity of any LP is polynomial with respect to the number of variables and "technological" constraints involved [3]. Hence, the considered linearity test has polynomial worst-case computational complexity with respect to $|S_r|$.

But another remark in [3] (c.f. pgs 193-194) that is even more relevant to this discussion, is that the *empirical* computational complexity of the Simplex algorithm[2] when applied on an LP formulation with n variables and m "technological" constraints, is $O(m^a \cdot n \cdot d^b)$, where (i) d denotes the density of the $m \times n$ matrix A that defines the left hand-side of the LP "technological" constraints in the matrix-based representation of these constraints, (ii) $a \in (1.25, 2.5)$, and (iii) $b \cong 0.33$. When translated in the context of the DAP linearity test that is used in this work, the above result of [3] implies an *empirical* computational complexity for this test of no more than $O(|\bar{S}^b_{r\bar{s}}| \cdot |\bar{S}_{rs}|^a \cdot \xi)$, where $a \in (1.25, 2.5)$.

We should also point out that the empirical complexity of the employed linearity test that is derived in the previous

[1] In the corresponding terminology, these are all the constraints that define the LP except for the sign restrictions that are imposed upon its variables.

[2] which is still the primary LP algorithm that is used in most commercial solvers, including the CPLEX package that is used in this work

Algorithm 6 Constructs a Child Node from its Parent Node and Updates State Admissibility for the Child Node

Input: parent RAS instance, pruned state $\mathbf{s}$
Output: child RAS instance

```
1: child := parent
2: child.IsAdmissible[s] := FALSE;
3: Q := {s}; E := {s};
4: while Q ≠ NULL do
5:    u := Q.pop();
6:    for all x in PrevStates[u] do
7:       TEST := FALSE;
8:       for all y in NextStates[x] do
9:          if child.IsAdmissible[y] then
10:            TEST := TRUE;
11:         end if
12:      end for
13:      if ¬ (TEST ∨ E.contains(x)) then
14:         child.IsAdmissible[x] := FALSE;
15:         Q.add(x); E.add(x);
16:      end if
17:   end for
18: end while
19: return child;
```

paragraph, is drastically controlled by the employment of the sets $\bar{S}_{rs}$ and $\bar{S}^b_{r\bar{s}}$, that contain, respectively, only the maximal and the minimal elements of their "parent" sets S_{rs} and $S^b_{r\bar{s}}$; as it can be seen in the tables that report our experimental results in Section V of [1], the cardinality of the sets $\bar{S}_{rs}$ and $\bar{S}^b_{r\bar{s}}$ is smaller than the cardinality of the respective sets S_{rs} and $S^b_{r\bar{s}}$ by many orders of magnitude, and it grows much more slowly than the cardinality of the corresponding state spaces.

Furthermore, another element of the linearity test of [1] that contributes very substantially to the reduction of the empirical computational complexity of this test, is the fact that the LP formulation of the Equations 17-19 that is employed in this test, is actually a feasibility test for the corresponding constraints; hence, the expected number of iterations that will be performed by the Simplex algorithm when applied on these LPs, is very minimal. Translated in the context of the formula $O(|\bar{S}^b_{r\bar{s}}|\cdot|\bar{S}_{rs}|^a\cdot\xi)$, the last remark implies that the parameter a appearing in this formula will take some of its lowest possible values.

C. Generating, representing and evaluating, through the class RAS*, the additional policies Δ generated by Algorithms 1 and 3 of [1]*

In this subsection, we discuss how we have organized the computation of the dynamic part of the class RAS for the additional policies Δ that will be generated by the search process to be conducted by Algorithms 1 and 3 of [1] when $\Delta^* \notin \tilde{\mathcal{L}}(\Phi)$. This computation seeks to take advantage of (i) the "locality" of the pruning process that is effected by Algorithm 2 in [1], and (ii) the information that is available in the instance of the class RAS representing the "parent" node that has spawned off the considered policy Δ.

The considered computation starts with the execution of Algorithm 6. This algorithm works as follows: (i) First, it constructs the RAS instance of the newly generated node (to be called the "child" node in the following) by replicating the RAS instance of its parent node. (ii) Next, it performs the pruning of the state $\mathbf{s}$ that generates this new node, by eliminating state $\mathbf{s}$ from the IsAdmissible list of this node. (iii) And, finally, it proceeds to perform the remaining state-pruning that is dictated by Algorithm 2. This additional pruning utilizes the information on (a) the state admissibility that is provided in the IsAdmissible list of the parent node[3] and (b) the structure of the underlying automaton Φ that is encoded in the static part of the RAS data structure. In particular, for any already pruned state $\mathbf{u}$, Algorithm 6 identifies additional states $\mathbf{x}$ that need to be pruned, by looking into the PrevStates hashset of $\mathbf{u}$ for members of this list that do not have any (remaining) admissible states in their NextStates hashset. Any such state $\mathbf{x}$ is removed from the IsAdmissible list, it is entered into list Q for further processing according to the logic that was described above, and it is also entered into list E in order to be recognized as an already identified inadmissible state. The complexity of this entire pruning operation is no more than $O(|S_r|\cdot\bar{\Gamma})$.

Once the "child" IsAdmissible list has been properly updated, it is run through Algorithm 4 that will compute its maximal elements, i.e., the MaxAdmissible hashset of the "child" node, through a computation similar to that discussed for the case of the DAP Δ^*. On the other hand, in order to compute the MinBoundaryInadmissible hashset of this node, Algorithms 1 and 3 of [1] first combine (i) the MinBoundaryInadmissible hashset of the parent node, and (ii) the set of states that were pruned during the execution of Algorithm 6, into a new hashset that is called PotentialMinBoundaryInadmissible, and this last state set is subsequently fed into Algorithm 5. Obviously, the computational complexity of all the aforementioned operations is similar to the complexity of the corresponding operations that take place in the processing of the policy Δ^*.

The completeness of the policy Δ that is represented by the constructed RAS object, can be assessed via Equation 11 of [1] and the information that is provided in the MaxAdmissible list, as explained in the last part of the description of Algorithm 1 in [1]. The complexity of the corresponding test is $O(|\Gamma(\mathbf{s}_0)|\cdot|\bar{S}_r|)$, where $|\bar{S}_r|$ is the length of the MaxAdmissible list for the considered policy Δ.

Finally, the assessment of the linearity of the policy Δ that is induced by the derived IsAdmissible list, and the computation of the corresponding InseparableMinBoundaryInadmissible hashset, are performed through the linearity test of Section IV.B in [1], as described in the previous parts of this document.

It should be evident from all the discussion that has been provided in this section, that the data structures defined in Table IV, and the accompanying Algorithms 4–6, manage, indeed, to support a localized and incremental computation of all those elements that define each policy Δ considered by the Algorithms 1 and 3 of [1]. Furthermore, by taking full advantage of the "monotonicity" property of the target

[3]This information has been inherited by the "child" node through the aforementioned replication of the corresponding RAS object.

policies that has been established in [1], the representations and the operations that have been described in this section, enable the assessment of the completeness and the linearity of the considered DAPs in a way that establishes the tractability of these tests. Hence, the data structures and the procedures that have been presented in this section, are, indeed, essential elements for the proposed implementations of the Algorithms 1 and 3 in [1].

Furthermore, the provided discussion on the complexity of the procedures that implement the various operations that are considered in this section, substantiates the claim that was made in Section IV.C of [1] that all these operations present a polynomial complexity with respect to $|S_r|$, i.e., the size of the reachable state space of the underlying RAS Φ. In fact, with the exception of the linearity test for the considered DAPs Δ, all the remaining operations are of linear worst-case computational complexity with respect to $|S_r|$. These remarks also identify the employed linearity test as the most expensive operation that will be performed during a single iteration of the Algorithms 1 and 3 of [1]. And when considering the remarks on the complexity of the linearity test that were provided in the closing part of Section III-B, it follows that the execution of just a single iteration by Algorithms 1 and 3 of [1] will be a quite tractable task, in terms of, both, its time and its memory requirements, even in the case of D/C-RAS configurations that possess very large state spaces by the standards of the current literature.

Hence, any memory or time "explosion" that might be observed during the execution of the Algorithms 1 and 3 of [1] on any given D/C-RAS instance with a size that is comparable to the RAS sizes that have been addressed by the current literature, will be primarily due to an "explosion" of the space that must be covered by the search process that is conducted by these two algorithms. In Sections IV.C and IV.D of [1], we also explained that the size of these "search spaces" is determined by (i) the depth of the DAGs that represent these "search spaces", and (ii) the extent of the "branching" that takes place at the nodes of these DAGs. Furthemore, the closing part of Section V in [1], provided some strong empirical support to all these claims.

IV. The detailed implementation of Algorithm 1 of [1] in the experiments discussed in [1]

In this section, we present the detailed implementation of the search process conducted by Algorithm 1 of [1], that was used in the experiments that are discussed in Section V of that document. Furthermore, the last part of the section presents a detailed example that traces the execution of the presented implementation of Algorithm 1 on a small conjunctive RAS, and provides the reader with a more concrete experience of the various concepts and algorithmic procedures that are presented in [1] and this electronic supplement.

As already remarked in [1], Algorithm 1 is not a very scalable proposition, for the reasons that were explained in that document, and further reiterated in the closing part of the previous section. Nevertheless, the execution of this algorithm on some smaller D/C-RAS configurations allows us to get a more concrete feeling of the structure of the target policy sets $\bar{\mathcal{L}}(\Phi)$ and its member DAPs, and it also defines some benchmarks for assessing the performance of other algorithms of a more heuristic nature, like Algorithm 3 of [1]. The experiments presented in Section V of [1] exploit and highlight these possibilities.

A. Refining the "branching" logic of Algorithm 1 of [1]

In the original statement of Algorithm 1 in [1], the algorithm is supposed to branch on all the elements of the set $\bar{S}_r$, i.e., on each maximal admissible state $\mathbf{s}_i$ by the policy Δ that corresponds to the current search node (c.f. Line 10 of the pseudocode that is provided for this algorithm in [1]).

However, in the eventual implementation of this algorithm in our experiments, we have confined the nodal branching only to a certain subset of the original set $\bar{S}_r$, focusing on those elements of $\bar{S}_r$ that can substantially improve the prospects of obtaining a linear policy Δ' from the corresponding state pruning. The rationale for the selection of the target subset of $\bar{S}_r$ that is eventually used for the proposed nodal branching, is defined by the results of the two propositions presented in the rest of this subsection.

Proposition 5: Let $\bar{S}'_r = \{\mathbf{s} \in \bar{S}_r : \mathbf{s}$ is an extreme point[4] of $conv(S_r)\}$. Then, the sets $(\bar{S}_r, \bar{S}^b_{\bar{r}})$ are linearly separable if and only if the sets $(\bar{S}'_r, \bar{S}^b_{\bar{r}})$ are linearly separable.

Proof: If the sets $(\bar{S}_r, \bar{S}^b_{\bar{r}})$ are linearly separable, then the sets $(\bar{S}'_r, \bar{S}^b_{\bar{r}})$ are linearly separable since $\bar{S}'_r$ is a subset of $\bar{S}_r$. On the other hand, if the sets $(\bar{S}_r, \bar{S}^b_{\bar{r}})$ are not linearly separable, then, according to the developments of [1], $\bar{S}^b_{\bar{r}} \cap conv(S_r) \neq \emptyset$. But the set $conv(S_r)$ is determined by the set $\bar{S}'_r$ [4], and therefore, the sets $(\bar{S}'_r, \bar{S}^b_{\bar{r}})$ are also not linearly separable. □

In order to calculate the set $\bar{S}'_r$, we iterate over all states $\mathbf{s}_i \in \bar{S}_r$, and for each of these states we solve the following linear program that checks whether this state can be expressed as a convex combination of the other states in $\bar{S}_r$:

$$\min_{\mathbf{x} \geq \mathbf{0}} x_i \tag{1}$$

s.t.

$$\sum_{k:\mathbf{s}_k \in \bar{S}_r} x_k \mathbf{s}_k \quad \geq \quad \mathbf{s}_i \tag{2}$$

$$\sum_{k:\mathbf{s}_k \in \bar{S}_r} x_k \quad = \quad 1 \tag{3}$$

If the optimal solution of the above LP is $x_i = 1$, we can conclude that the state $\mathbf{s}_i$ is an extreme point of $conv(S_r)$, and we add the state $\mathbf{s}_i$ to the set $\bar{S}'_r$.

Furthermore, working as in the case of the DAP-linearity test that was discussed in the previous section, we can show that the empirical complexity of computing the set $\bar{S}'_r$ from the given set $\bar{S}_r$ is $O(|\bar{S}_r|^2)$.

The constructed set $\bar{S}'_r \subseteq \bar{S}_r$ that is obtained through the above process, can be further thinned out through the procedure that is stated in Algorithm 7.

Algorithm 7 identifies subsets of the set $\bar{S}'_r$ that contain in their convex hull some unsafe state $\mathbf{u} \in$ `InseparableMinBoundaryInadmissible`, or maybe a point which dominates some unsafe state $\mathbf{u} \in$ `InseparableMinBoundaryInadmissible`. All these

[4]We remind the reader that a weighted sum $\sum_{i=1}^{n} w_i \mathbf{v}_i$ of a finite set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$, with $w_i \in [0, 1]$ and $\sum_{i=1}^{n} w_i = 1.0$, is characterized as a *convex combination* of these vectors. Also, an *extreme point* of a convex polytope is any point of this polytope that cannot be expressed as a convex combination of the remaining points of the polytope.

Algorithm 7 Calculates the state set $\bar{S}_r''$ that is used for the branching that takes place at each internal node of the search process conducted by Algorithm 1 of [1].

Input: $(\bar{S}_r', \bar{S}_{\bar{r}}^b)$
Output: $\bar{S}_r''$
1: $\bar{S}_r'' := \emptyset$;
2: $I := \emptyset$;
3: **for all** $\mathbf{u} \in \bar{S}_{\bar{r}}^b$ **do**
4: Solve the LP

$$\min_{\mathbf{x} \geq \mathbf{0}} 0$$

s.t.

$$\begin{aligned} \sum_{k:\mathbf{s}_k \in \bar{S}_r'} x_k \cdot \mathbf{s}_k &\geq \mathbf{u} \\ \sum_{k:\mathbf{s}_k \in \bar{S}_r'} x_k &= 1 \\ x_k &= 0, \quad \forall k \in I \end{aligned}$$

5: **if** the above LP is feasible **then**
6: **for all** $\mathbf{s}_k \in \bar{S}_r'$ **do**
7: **if** $x_k > 0$ **then**
8: $\bar{S}_r'' := \bar{S}_r'' \cup \{\mathbf{s}_k\}$;
9: $I := I \cup \{k\}$;
10: **end if**
11: **end for**
12: GOTO step 3;
13: **end if**
14: **end for**
15: **return** $\bar{S}_r''$;

subsets are compiled in the set $\bar{S}_r''$. The significance of the thus computed set $\bar{S}_r''$, is established by the following proposition.

Proposition 6: Let the set $\bar{S}_r''$ be the subset of $\bar{S}_r'$ that is obtained by Algorithm 7. Then, the sets $(\bar{S}_r, \bar{S}_{\bar{r}}^b)$ are linearly separable if and only if the sets $(\bar{S}_r'', \bar{S}_{\bar{r}}^b)$ are linearly separable.

Proof: If the sets $(\bar{S}_r, \bar{S}_{\bar{r}}^b)$ are linearly separable, the sets $(\bar{S}_r'', \bar{S}_{\bar{r}}^b)$ are linearly separable as well, since, in this case, the set $\bar{S}_r''$ returned by Algorithm 7 will be the empty set. On the other hand, if the sets $(\bar{S}_r, \bar{S}_{\bar{r}}^b)$ are not linearly separable, then the sets $(\bar{S}_r'', \bar{S}_{\bar{r}}^b)$ are not linearly separable either, because by the construction of the set $\bar{S}_r''$ in Algorithm 7, this set contains at least one subset of $\bar{S}_r$ that has a minimal boundary inadmissible state $\mathbf{u}$ within or below its convex hull. □

The empirical complexity of Algorithm 7 is determined by the solution of the involved LPs, and it is no more than $O(|\bar{S}_{\bar{r}}^b| \cdot |\bar{S}_r'|)$. Remarks similar to those provided in Section III for the empirical complexity of the linearity test that is discussed in that section, apply to this case, as well. In particular, it is important to notice that the LPs that are solved by Algorithm 7 are feasibility tests of the corresponding constraints, and therefore, they are expected to run very fast.

In view of Propositions 5 and 6, in our implementation of Algorithm 1 of [1], the branching that takes place at each node of the underlying search tree, is based on the set $\bar{S}_r''$ that corresponds to this node. This set will be considerably "thinner" than the original set $\bar{S}_r$.

Finally, an additional remark that is important for the overall organization of Algorithm 1 of [1] under the "branching" logic that was presented in this subsection, is that the computation of the `InseparableMinBoundaryInadmissible` list for the considered DAP Δ is a "by-product" of the computation that is performed by Algorithm 7. Therefore, the computation of the set $\bar{S}_r''$ that has been presented in this section, subsumes the resolution of the linearity of Δ; if this set turns out to be empty, then, the considered DAP Δ is linear.

B. The management of the list $EXPLORE$

A last issue regarding the detailing of Algorithm 1 of [1] for a more tractable and streamlined execution, concerns the queueing discipline to be followed in the management of the list $EXPLORE$ that is maintained by this algorithm. In our eventual implementation of the algorithm, this list was managed as a stack, since any other discipline resulted in an explosion of the list contents to the point that it was impossible to maintain this list in the core memory. This choice further implies that the underlying search process was conducted according to a "depth-first" scheme.

Furthermore, an additional heuristic that was very helpful in identifying high-quality policies early on in the conducted search, pertains to the storing of the elements of the set $\bar{S}_r''$ in the maintained stack. According to this heuristic, the elements of the set $\bar{S}_r''$ should enter the list $EXPLORE$ in a way that those elements with the largest weights in the corresponding linear combinations that introduced them in the set $\bar{S}_r''$ (c.f. Algorithm 7), will be retrieved from this list first.

C. Applying the presented algorithm on an example RAS

In this example, we consider a conjunctive RAS Φ that supports three process types Π_1, Π_2 and Π_3. RAS Φ has six resource types – i.e., $\mathcal{R} = \{R_1, \ldots, R_6\}$ – with corresponding capacities $C_i = 4$, $i = 1, \ldots, 6$. The three process types Π_1, Π_2 and Π_3, that are supported by RAS Φ, possess a strictly sequential structure, and the corresponding resource allocation function $\mathcal{A}$ is described by the following vector-sequences:

$$\Pi_1 = \left\langle \begin{pmatrix} 2 \\ 0 \\ 0 \\ 0 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \\ 0 \\ 4 \\ 4 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 2 \\ 0 \\ 4 \\ 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 2 \\ 0 \\ 4 \\ 2 \\ 1 \end{pmatrix} \right\rangle$$

$$\Pi_2 = \left\langle \begin{pmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 2 \\ 0 \\ 4 \end{pmatrix}, \begin{pmatrix} 4 \\ 1 \\ 0 \\ 2 \\ 0 \\ 4 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 2 \\ 0 \\ 4 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ 4 \\ 2 \\ 2 \\ 4 \end{pmatrix} \right\rangle$$

$$\Pi_3 = \left\langle \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 0 \\ 3 \\ 1 \\ 0 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ 4 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 0 \\ 4 \\ 0 \\ 0 \end{pmatrix} \right\rangle$$

TABLE V: The safe states for the example RAS of Section IV-C

State	State Vector	State	State Vector	State	State Vector
$\mathbf{s}_0$	(0,0,0,0,0,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{14}$	(0,0,0,0,1,1,0,0,0,0,0,0,0,0)	$\mathbf{s}_{28}$	(0,0,1,0,0,0,0,0,0,0,1,0,0,0)
$\mathbf{s}_1$	(1,0,0,0,0,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{15}$	(0,0,0,0,0,0,1,0,0,0,0,0,0,0)	$\mathbf{s}_{29}$	(0,0,0,0,1,0,0,1,0,0,0,0,0,0)
$\mathbf{s}_2$	(0,0,0,0,1,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{16}$	(0,0,0,0,0,0,0,0,0,1,1,0,0,0)	$\mathbf{s}_{30}$	(0,0,0,0,0,0,0,0,0,1,0,0,0,0)
$\mathbf{s}_3$	(0,0,0,0,0,0,0,0,0,1,0,0,0,0)	$\mathbf{s}_{17}$	(0,0,0,0,0,0,0,0,0,0,0,1,0,0)	$\mathbf{s}_{31}$	(0,0,0,0,0,0,0,0,0,1,0,0,1,0)
$\mathbf{s}_4$	(0,1,0,0,0,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{18}$	(0,0,0,1,0,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{32}$	(0,0,0,0,0,0,0,0,0,0,0,0,0,1)
$\mathbf{s}_5$	(1,0,0,0,0,0,0,0,0,1,0,0,0,0)	$\mathbf{s}_{19}$	(0,0,1,0,0,0,0,0,0,1,0,0,0,0)	$\mathbf{s}_{33}$	(0,0,0,1,0,0,0,0,0,2,0,0,0,0)
$\mathbf{s}_6$	(0,0,0,0,2,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{20}$	(0,1,0,0,0,0,0,0,0,2,0,0,0,0)	$\mathbf{s}_{34}$	(0,0,0,1,0,0,0,0,0,0,0,1,0,0)
$\mathbf{s}_7$	(0,0,0,0,0,1,0,0,0,0,0,0,0,0)	$\mathbf{s}_{21}$	(0,1,0,0,0,0,0,0,0,0,1,0,0,0)	$\mathbf{s}_{35}$	(0,0,0,0,1,0,0,0,1,0,0,0,0,0)
$\mathbf{s}_8$	(0,0,0,0,0,0,0,0,0,2,0,0,0,0)	$\mathbf{s}_{22}$	(0,0,0,0,1,0,1,0,0,0,0,0,0,0)	$\mathbf{s}_{36}$	(0,0,0,0,0,0,0,0,0,0,1,0,0,1)
$\mathbf{s}_9$	(0,0,0,0,0,0,0,0,0,0,1,0,0,0)	$\mathbf{s}_{23}$	(0,0,0,0,0,0,0,1,0,0,0,0,0,0)	$\mathbf{s}_{37}$	(0,0,0,0,0,0,0,0,0,2,0,0,0,1)
$\mathbf{s}_{10}$	(0,0,1,0,0,0,0,0,0,0,0,0,0,0)	$\mathbf{s}_{24}$	(0,0,0,0,0,0,0,0,0,1,0,1,0,0)	$\mathbf{s}_{38}$	(0,0,0,0,0,0,0,0,0,0,1,0,0,1)
$\mathbf{s}_{11}$	(0,1,0,0,0,0,0,0,0,1,0,0,0,0)	$\mathbf{s}_{25}$	(0,0,0,0,0,0,0,0,0,0,0,1,0)	$\mathbf{s}_{39}$	(0,0,0,0,0,0,0,0,0,1,1,0,0,1)
$\mathbf{s}_{12}$	(1,0,0,0,0,0,0,0,0,2,0,0,0,0)	$\mathbf{s}_{26}$	(0,0,0,1,0,0,0,0,0,1,0,0,0,0)	$\mathbf{s}_{40}$	(0,0,0,0,0,0,0,0,0,0,0,1,0,1)
$\mathbf{s}_{13}$	(1,0,0,0,0,0,0,0,0,0,1,0,0,0)	$\mathbf{s}_{27}$	(0,0,1,0,0,0,0,0,0,2,0,0,0,0)	$\mathbf{s}_{41}$	(0,0,0,0,0,0,0,0,0,1,0,1,0,1)

Hence, the state $\mathbf{s}$ of the considered RAS Φ is a vector with fourteen components: the first four components of vector $\mathbf{s}$ report the number of parts in each processing stage of the first process type; the next five components of vector $\mathbf{s}$ report the number of parts in each processing stage of the second process type; and the final five components of vector $\mathbf{s}$ report the number of parts in each processing stage of the third process type.[5] Also, state $\mathbf{s}_0 = (0 \dots 0)^T$ denotes the initial empty state.

The state space S of the considered RAS Φ consists of 98 states, with 42 of them being safe. These safe states are listed in Table V. Furthermore, there are 17 maximal safe states, with $\bar{S}_r = \{\mathbf{s}_6, \mathbf{s}_{12}, \mathbf{s}_{13}, \mathbf{s}_{14}, \mathbf{s}_{20}, \ \mathbf{s}_{21}, \mathbf{s}_{22}, \mathbf{s}_{27}, \mathbf{s}_{28}, \mathbf{s}_{29}, \mathbf{s}_{31}, \mathbf{s}_{33}, \mathbf{s}_{34}, \mathbf{s}_{35}, \mathbf{s}_{37}, \mathbf{s}_{39}, \mathbf{s}_{41}\}$, and a single boundary unsafe state, $\mathbf{u}$, that is not separable from the maximal safe states; in particular, $\mathbf{u} = (0,0,0,0,1,0,0,0,0,1,0,0,0,0)^T$, i.e., it has one active process instance at the first processing stage of the second process type and a second active process instance at the first processing stage of the third process type. The presence of the aforementioned state $\mathbf{u}$ in the convex hull of S_{rs} implies that, for the considered RAS Φ, the maximally permissive DAP Δ^* is not linear.

In order to apply the branching scheme that was presented in the Section IV-A of this document to the first node of the underlying search process, we first compute the set $\bar{S}'_r$ of the maximal safe states that are extreme points of the convex hull of the set of reachable and safe states S_{rs}. It turns out that all of the 17 maximal safe states are extreme points of this convex hull, i.e., $\bar{S}'_r = \bar{S}_{rs}$. But the subsequent application of Algorithm 7 on the set $\bar{S}'_r$ returns the set $\bar{S}''_r = \{\mathbf{s}_6, \mathbf{s}_{12}\}$, and therefore, the branching process at the considered node will focus on these two states only.

Running Algorithm 6 for the child node that results from the pruning of state $\mathbf{s}_6$, we find out that the pruning of this state from the admissible space of the parent node does not require the pruning of any further nodes in order to obtain a correct policy Δ for the child node. This finding further implies that the obtained policy Δ is complete. Also, the set of the maximal reachable admissible states for this child node is $\bar{S}_r \setminus \{\mathbf{s}_6\}$, where $\bar{S}_r$ is the set of the maximal reachable admissible states for the parent node that was reported above. Finally, using Algorithm 5 as explained in Section III-C of this document, we also find that the set of minimal boundary inadmissible states for this child node is the set $\bar{S}^b_{\bar{r}} \cup \{s_6\}$, where $\bar{S}^b_{\bar{r}}$ denotes the set of minimal boundary inadmissible states of the parent node.

At this point, the algorithm proceeds to assess the linearity of the policy Δ corresponding to this child node, making use of (i) the computed sets of maximally admissible states and minimal boundary inadmissible states, and (ii) Algorithm 7. It turns out that this policy Δ is linear, indeed, and therefore, it is entered into list $STORE$.

Next, the algorithm shifts its attention to the node that is obtained from the root node of underlying search process by pruning the maximal safe state $\mathbf{s}_{12}$. This node is processed similarly to the previous child node, but the resulting policy Δ' is not linear. Hence, the algorithm branches further on this node, and at the end of the algorithm execution, we find out that the two maximal linear policies for the considered RAS Φ are (a) the policy Δ that was identified in the first child node discussed above, and also (b) the policy Δ'' that admits all states of Table V except for the states in $\{\mathbf{s}_8, \mathbf{s}_{12}, \mathbf{s}_{20}, \mathbf{s}_{27}, \mathbf{s}_{33}, \mathbf{s}_{37}\}$.

V. Worst-case complexity analysis of Algorithms 1 and 3 of [1]

In this section, we use the complexity analysis for the various procedures and algorithms that were presented in the earlier parts of this document, in order to develop a characterization of the worst-case complexity of Algorithms 1 and 3 of [1], which constitute the primary algorithms developed in this work. This analysis substantiates and supports the corresponding claims and remarks that are made in [1] regarding the computational complexity of these two algorithms and the factors that affect this complexity.

From an organizational standpoint, the section consists of three major subsections: The first subsection characterizes the growth of the size of the state space S of any given D/C-RAS Φ with respect to the various elements that define RAS Φ according to Definition 1 of [1]. The second subsection provides

[5] In fact, for the purposes of deadlock avoidance that is the focus of this paper, the state component corresponding to the last processing stage of each process type could have been dropped from the processed (state) vectors $\mathbf{s}$ without compromising the correctness of the derived policies; c.f. the corresponding discussion in [5]. But we have opted to keep these processing stages in the presented developments in order to not complicate any further the corresponding exposition of this material.

a characterization of the worst-case computational complexity of Algorithm 1 in [1], and the third subsection characterizes the worst-case computational complexity of Algorithm 3 in [1]. The last subsection also provides some remarks of a more practical nature that are meant as further interpretation and appreciation of the more theoretical developments that are presented in this section.

A. Characterizing the growth of $|S|$ with respect to the defining elements of the corresponding RAS Φ

In this subsection, we provide a characterization of the growth of the cardinality of the state space S of a D/C-RAS Φ with respect to the defining elements of this RAS according to Definition 1 in [1], under the further assumption that the resource allocation vectors $D(\theta_{j,k}), j = 1, \ldots, n,\ k = 1, \ldots, l_j$, are *unit* vectors of the corresponding vector spaces; i.e., in the D/C-RAS subclass that is considered in this subsection, each processing stage requires a single unit from a single resource type for its support. This D/C-RAS subclass is characterized as "Single-Unit (SU)" in the corresponding taxonomy of [6], and its employment in the analysis that is pursued in this subsection, is justified by the fact that the corresponding restriction of the resource requirement that is posed by any single processing stage, increases the concurrency that can take place in the considered RAS Φ and, therefore, the size of the corresponding state space S – c.f. Equation (1) in [1].[6]

A complete characterization of $|S|$ for an SU-RAS Φ has been provided in [7]. The corresponding analysis exploits the fact that the state space S of any given SU-RAS Φ can be perceived as the product of the subspaces $S_i,\ i = 1, \ldots, m$, that characterize that allocation of the capacity C_i of resource type R_i to the various processing stages that are supported by this resource. More specifically, let $\Theta(R_i),\ i = 1, \ldots, m$, denote the set of processing stages supported by resource R_i. Then, at any given state $\mathbf{s}$, each capacity unit of resource type R_i may be allocated to a process instance j_j executing one of the processing stages in $\Theta(R_i)$ or be part of the slack capacity of resource R_i at state $\mathbf{s}$. This remark further implies that the elements of the aforementioned subspace S_i characterizing the allocation of the capacity C_i of resource type R_i are defined by all the possible ways that the C_i capacity units of resource R_i can be split into $|\Theta(R_i)| + 1$ subsets; each of the first $|\Theta(R_i)|$ subsets corresponds to capacity allocated to process instances executing the corresponding processing stage, and the last subset corresponds to the slack capacity of R_i in the corresponding state $\mathbf{s}$. Hence, using standard results from elementary combinatorics theory [8], we have that

$$\forall i = 1, \ldots, m,\ |S_i| = \frac{(C_i + |\Theta(R_i)|)!}{C_i! \cdot |\Theta(R_i)|!} \tag{4}$$

[6]For a more complete appreciation of the developments that are provided in this subsection, and the proposed focus on the SU-RAS model, we want also to point out that for a very large subclass of the SU-RAS model, maximally permissive deadlock avoidance can be attained with polynomial complexity; the reader is referred to Section 3.3 of [6] for the corresponding results. Yet, we can still focus the subsequent developments on those SU-RAS that do not meet the criteria that define the aforementioned SU-RAS subclass. Even more importantly, the qualitative insights that are provided by the analysis that is pursued in this subsection, carry over to the more general class of D/C-RAS that is not (easily) amenable to a closed-form characterization of the size of the involved state spaces.

and, finally,

$$|S| = |\times_{i=1}^{m} S_i| = \Pi_{i=1}^{m} |S_i| = \Pi_{i=1}^{m} \frac{(C_i + |\Theta(R_i)|)!}{C_i! \cdot |\Theta(R_i)|!} \tag{5}$$

Equation 5 implies that $|S|$ is a super-polynomial function of the corresponding RAS size $|\Phi|$. Furthermore, we have that

$$\begin{aligned} \forall i = 1, \ldots, m,\ |S_i| &= \frac{(C_i + |\Theta(R_i)|)!}{C_i! \cdot |\Theta(R_i)|!} \\ &= \frac{(C_i + 1)(C_i + 2) \ldots (C_i + |\Theta(R_i)|)}{1 \cdot 2 \cdot \ldots \cdot |\Theta(R_i)|} \\ &= (C_i + 1)(\frac{C_i}{2} + 1) \ldots (\frac{C_i}{|\Theta(R_i)|} + 1) \\ &\geq (\frac{C_i}{|\Theta(R_i)|} + 1)^{|\Theta(R_i)|} \end{aligned} \tag{6}$$

and for $|\Theta(R_i)| \to \infty$,

$$(\frac{C_i}{|\Theta(R_i)|} + 1)^{|\Theta(R_i)|} \ \to\ e^{C_i} \tag{7}$$

From Equations 5 – 7, we also get that, for $|\Theta(R_i)| \to \infty$,

$$|S| \ \geq\ e^{\sum_{i=1}^{m} C_i} \tag{8}$$

Finally, the cardinalities of the various subsets of S that are processed more explicitly by Algorithms 1 and 3 in [1], can be perceived as substantial percentages of $|S|$, and therefore, the above analysis carries over to these values as well. For the same reason, in the complexity characterizations that are provided in the rest of this section, the cardinalities of all these subsets will be replaced by the cardinality of the entire state space S.

B. Worst-case complexity analysis of Algorithm 1 in [1]

As explained in [1], Algorithm 1 essentially conducts a search process over the subsets of S_{rs} that possess some additional properties, according to a "branch & bound (b&b)" scheme. Hence, the involved computational effort is determined by (i) the necessary effort to process any single node in the DAG that represents this b&b scheme, during a single iteration of Algorithm 1, and (ii) the number of iterations that will be performed by the algorithm throughout its entire execution.

A "worst-case" characterization of these two quantities can be obtained as follows:

I) The relevant discussion that was provided in Section III of this document, has established that the most difficult task in the processing that takes place during a main iteration of Algorithm 1, is the assessment of the linearity of the policy $\Delta(\bar{S}_r)$ that is induced by the assessed set $\bar{S}_r$. Furthermore, in Section IV-A of this document, it was shown that this task can be performed with complexity $O(|S|^2)$.

II) The number of the subsets of S_{rs} that may be considered during the search conducted by Algorithm 1, is $O(2^{|S_{rs}|})$. Furthermore, the material of Section IV-B of this document implies that any given subset $\mathcal{S}$ of S_{rs} (represented by the corresponding set $\bar{S}_r$ in the lists that are maintained by this algorithm) may be generated more than once. An upper bound of the number of revisits of this set by Algorithm 1 is provided by the number of the minimal elements of the set $S_{rs} \setminus \mathcal{S}$. But

according to our previous remarks regarding the cardinality of these sets, this last number is $O(|S|)$. Hence, the total number of iterations of Algorithm 1 is $O(|S| \cdot 2^{|S|})$.

Finally, combining the results of parts I) and II) above, we get a characterization of the worst-case computational complexity of Algorithm 1 as $O(|S|^3 \cdot 2^{|S|})$. Clearly, in this result, the dominant factor is the exponential $2^{|S|}$ that expresses the intention of Algorithm 1 to conduct a complete search process over the power-set of the set S_{rs}. Furthermore, as discussed in Section IV-B, this particular feature of Algorithm 1 results also in a very high space complexity for this algorithm, which is manifested by the very fast growth of the list $EXPLORE$ that is maintained by the algorithm.

C. Worst-case complexity analysis of Algorithm 3 in [1]

As explained in [1], Algorithm 3 is an effort to cope with the complexity problems of Algorithm 1 that were highlighted in the last paragraph of the previous subsection, while maintaining near-optimality of the derived policies. The mechanism employed towards this objective is the restriction of the computation of Algorithm 3 on a single path of the DAG representing the underlying search space that leads to a linear DAP; this path is incrementally defined by the algorithm in a manner that is expected to result in a near-optimal policy.

Once the above defining mechanism for Algorithm 3 is understood, then, an analysis along lines similar to the lines that were pursued for the corresponding complexity analysis of Algorithm 1, can establish a worst-case computational complexity for this new algorithm of $O(|S|^{1+a} \cdot \xi)$; in this result, (i) $a \in (1.25, 2.5)$, and (ii) we have used the characterization of the empirical computational complexity of the Simplex algorithm of [3], that was discussed in Section III-B.

Concluding the discussion of this subsection, we also want to make the following remarks:

First of all, while the above characterization of the complexity of Algorithm 3 of [1] is a low degree polynomial with respect to the involved parameters $|S|$ and ξ, we must keep in mind that, as shown in Section V-A, $|S|$ is a fast-growing super-polynomial function of the basic elements that define the RAS Φ and its size, according to Definition 1 in [1].

Recognizing this restricting effect, the considered work has tried to control the fast growth of $|S|$ by employing more parsimonious representations of the various subsets of S that are processed by the presented algorithms, especially during their iterative modes. While they cannot suppress the exponential growth of the processed subsets in terms of $|\Phi|$, the employed representations can still reduce the volume of information that is explicitly processed at every iteration by many orders of magnitude. And as revealed by the numerical experiments that are reported in Section V of [1], this reduction can lead to reasonable run times and a short memory "footprint" even for quite sizeable D/C-RAS configurations.

Our experience from the numerical experiments that are reported in Section V of [1] also indicates that a short memory "footprint" is crucial for an expedient execution of Algorithm 3, since it determines the ability of this algorithm to execute without resorting to secondary memory; any arising need to use this extra memory can slow down dramatically the algorithm.

An additional element that enables the execution of Algorithm 3 even on some pretty large D/C-RAS configurations, as demonstrated in the computational experiments that are reported in Section V in [1], is the "worst-case" nature of the complexity characterizations that are provided above. As remarked in [1] itself, the actual execution time of Algorithms 1 and 3 will be determined by the number of the iterations that must be executed by these algorithms before they complete their computation, and this last number can be considerably lower than the values that are suggested by the corresponding expressions that have been developed in this worst-case analysis. Also, the eventual determination of this number, for any given D/C-RAS configuration Φ, will depend on elements like the detailed positioning of the boundary unsafe states in the convex hull of the safe states, which, unfortunately, are not easily traceable in the data that define the considered RAS Φ itself.

Nevertheless, the complexity analysis that has been pursued in this document, and the corresponding formulae that have been developed from this analysis, still reveal the most critical factors that are behind the super-polynomial complexity of Algorithms 1 and 3 and their supporting procedures. This fact is also highlighted by the corresponding material of [1]. In particular, Algorithm 3 itself is the product of the key insights that resulted from the complexity analysis of Algorithm 1, while the entire discussion on the results of the numerical experiments that are presented in Section V of [1], is driven by, and corroborates, the results that were developed in this section.

REFERENCES

[1] M. Ibrahim, S. Reveliotis, and A. Nazeem, "Maximal linear deadlock avoidance policies for complex resource allocation systems: characterization, computation and approximation," ISyE, Georgia Tech, Tech. Rep. (submitted for publication), 2020.

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

[3] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows (2nd ed.)*. NY, NY: John Wiley & Sons, 1990.

[4] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*. NY, NY: Wiley-Interscience, 1998.

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

[6] S. Reveliotis, "Logical Control of Complex Resource Allocation Systems," *NOW Series on Foundations and Trends in Systems and Control*, vol. 4, pp. 1–224, 2017.

[7] S. A. Reveliotis, "Structural analysis & control of flexible manufacturing systems with a performance perspective," Ph.D. dissertation, University of Illinois, Urbana, IL, 1996.

[8] R. Durrett, *The Essentials of Probability*. Belmont, CA: Duxbury Press, 1994.