Technische Universiteit
Eindhoven
University of Technology

Partial-Order Reduction for Synthesis and Performance Analysis of Supervisory Controllers

Bram van der Sanden*§, Marc Geilen*, Michel Reniers*, Twan Basten*§
*Eindhoven University of Technology, Eindhoven, The Netherlands
§ESI (TNO), Eindhoven, The Netherlands
Email: bram.vandersanden@tno.nl, m.c.w.geilen@tue.nl,
m.a.reniers@tue.nl, a.a.basten@tue.nl

ES Reports
ISSN 1574-9517

ESR-2019-02
11 November 2019

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Partial-Order Reduction for Synthesis and Performance Analysis of Supervisory Controllers

Bram van der Sanden*§, Marc Geilen*, Michel Reniers*, Twan Basten*§
*Eindhoven University of Technology, Eindhoven, The Netherlands
§ESI (TNO), Eindhoven, The Netherlands
Email: bram.vandersanden@tno.nl, m.c.w.geilen@tue.nl, m.a.reniers@tue.nl, a.a.basten@tue.nl

***Abstract*—A key challenge in the synthesis and analysis of supervisory controllers is the impact of state-space explosion caused by concurrency. The main bottleneck is often the memory needed to store the composition of plant and requirement automata and the resulting supervisor. Partial-order reduction is a well-established technique in the field of model checking that alleviates this issue. It does so by exploiting redundancy in the model with respect to the properties of interest. In the context of controller synthesis these properties are nonblockingness, controllability, and least-restrictiveness. From the performance point of view, we consider throughput and latency. We propose an on-the-fly partial-order reduction on the input model that preserves both synthesis and performance properties in the synthesized supervisory controller. This improves scalability of both synthesis and performance analysis. Experiments show the effectiveness of the method on a set of realistic manufacturing system models.**

I. Introduction

Supervisory controller synthesis [1] is a method to automatically synthesize a supervisor that restricts the behavior of a system, described by a plant, to a given requirement that describes the allowed behaviors of the plant. Standard synthesis first computes the composition of all plant and requirement automata, and subsequently prunes the state space to ensure properties like controllability and nonblockingness (explained below) of the resulting supervisor [1], [2]. A disadvantage of this synthesis is its limited scalability, caused by the memory complexity of $\mathcal{O}(|Q_P|^2 \cdot |\Sigma|)$ [1], [3], where Q_P is the set of plant states and Σ the set of events. The memory needed to store the full state space often becomes a bottleneck [4]. The size of the supervisor also directly impacts the efficiency of performance analysis, performed on the state space of the supervisor augmented with timing information.

An approach to improve scalability of the synthesis and performance analysis is partial-order reduction (POR) [5], [6]. The idea of POR is to exploit redundancy in the network of automata to obtain a reduced composition, while preserving the properties of interest. In each state, a subset of redundant enabled transitions is removed, which, in turn, reduces the number of reachable states. Synthesis can then be performed on this smaller model leading to a smaller supervisor, and performance analysis can then be performed on a smaller timed state space.

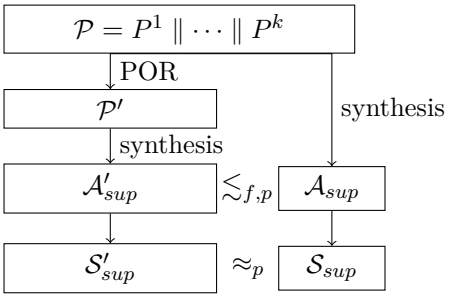


Fig. 1: Partial-order-reduction approach.

In the reduction we want to preserve both functional and performance aspects. The functional properties of interest are controllability, nonblockingness, and least-restrictiveness. The supervisor needs to be *controllable* with respect to the plant, which means that it should not disable uncontrollable events. Furthermore, the composition of the supervisor with the plant, the *controlled system*, should be *nonblocking*, which means that from every reachable state in the controlled system, a *marked state* can be reached that typically indicates the completion of an operation. In addition, it is required that the supervisor is *least restrictive*, which means that it restricts the system as little as possible while still being controllable and nonblocking. The performance aspects of interest are throughput and latency. *Throughput* describes the system performance in the long run, for instance the number of products produced by the system per hour. *Latency* describes the temporal distance between certain events, for instance the time between the start and end of processing a product.

This paper extends [7], where we introduced a POR technique that considered only the preservation of performance aspects. In this paper, we additionally consider the preservation of functional aspects. Furthermore, we have a more extensive experimental evaluation. Fig. 1 shows the proposed approach. It is convenient to treat all automata in the same way. Therefore, we assume that all requirements are translated into plant automata using a *plantify* transformation [8]; for every uncontrollable event that is not enabled in a state, a transition to a new blocking state is added. The considered input model $\mathcal{P}$ then consists of a composition $P^1 \parallel \cdots \parallel P^k$ of plant automata. Synthesis can be applied on this model directly to obtain a supervisor $\mathcal{A}_{sup}$ (an automaton). Alternatively, using POR, a reduced

plant $\mathcal{P}'$ is computed on which synthesis can be applied to obtain a reduced supervisor $\mathcal{A}'_{sup}$. The conditions on the reduction to $\mathcal{P}'$ guarantee that the functional and performance properties of $\mathcal{A}_{sup}$ are preserved in $\mathcal{A}'_{sup}$, denoted $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$ (defined precisely in Def. 32). This relation ensures that $\mathcal{A}'_{sup}$ is nonblocking if $\mathcal{A}_{sup}$ is nonblocking, $\mathcal{A}'_{sup}$ is controllable with respect to $\mathcal{P}$ if $\mathcal{A}_{sup}$ is controllable with respect to $\mathcal{P}$, and $\mathcal{A}'_{sup}$ is least-restrictive to $\mathcal{P}$ under an adapted notion of least-restrictiveness (Def. 10) that considers redundancy. The relation also guarantees that the throughput and latency values are preserved in the corresponding timed state spaces $\mathcal{S}_{sup}$ and $\mathcal{S}_{sup'}$, denoted by $\mathcal{S}_{sup} \approx_p \mathcal{S}_{sup'}$ (Def. 14).

We follow [9] by taking system *activities* as the events in our models. An activity captures a functionally deterministic part of the system behavior, consisting of low-level actions operating on system resources and (acyclic) precedences between those actions. An activity may, for example, correspond to moving a robot arm from one specified position to another specified position, consisting of several actions on one or more motor resources in some specified order. Actions of different activities may not interfere with each other except through resource claims and releases. Activities can then be treated as atomic events, abstracting from the execution orders of activity-internal concurrent actions. As shown in [9], this improves scalability of controller synthesis.

There are various ways to capture the timing behavior of supervisory controllers. Some well-known approaches are real-valued clocks as used in timed automata [10], discrete-valued clocks as used in tick-based models [11], and (max,+) algebra [9], [12], [13]. We use (max,+) algebra (see for instance [14]), which fits naturally with the notion of activities. A (max,+) timing matrix expresses the relation between the availability times of the system resources and the release times of the resources after executing an activity. Such a (max,+) timing model enables efficient performance analysis [9]. Supervisor synthesis on automata with activities can be done without considering their timing because activities can be treated as atomic events. Given the supervisor and the timing matrices of the activities, a timed (max,+) state space can be computed that provides the necessary timing information to evaluate system throughput and latency. Our POR technique improves scalability of both the supervisor synthesis and the performance analysis based on activity models with (max,+) timing semantics.

The paper is structured as follows. Section II introduces the framework that we use to capture a (max,+) timed system. Section III defines the functional and performance aspects that need to be preserved in the reduction. Section IV describes the conditions that are needed on a (max,+) state-space reduction to preserve the performance aspects. Section V lifts these conditions to the level of automata, and adds conditions to also preserve the functional aspects. Section VI then introduces an on-the-fly reduction that uses local conditions to compute a reduced automaton directly from a composition of automata. The experimental evaluation described in Section VII shows the effectiveness of the reduction technique. Related work is described in Section VIII and Section IX concludes. Proofs are omitted for space reasons. They can be found in [15][1].

II. Modeling

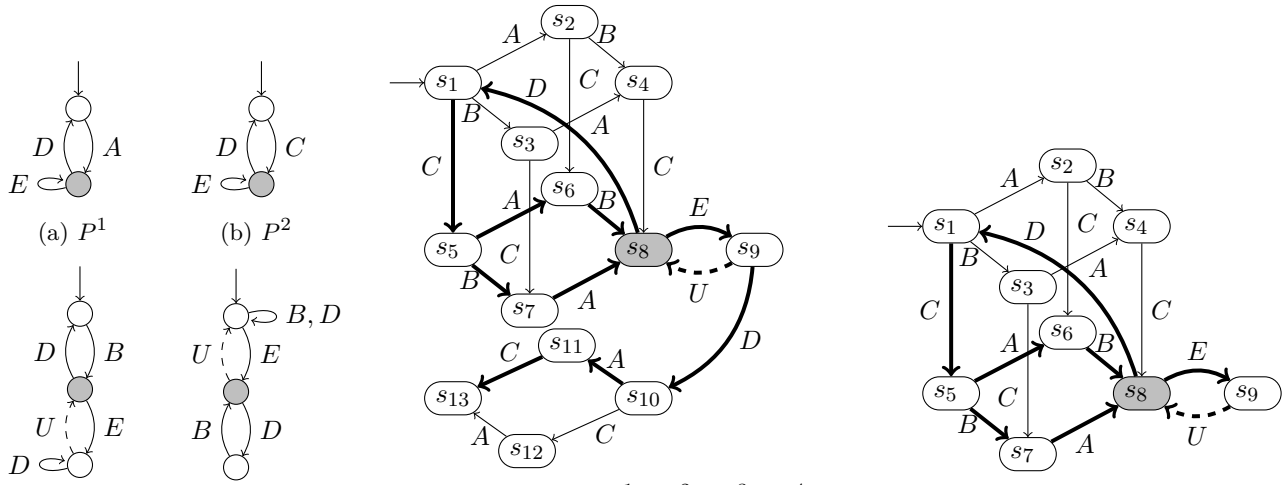
Consider a running example with activities A, B, C, D, E, and U. To capture all possible activity orderings in the system, we use (max,+) automata. A (max,+) automaton is a conventional finite-state automaton, where the timing semantics of each activity in the automaton is described by a (max,+) matrix, as explained below. The (max,+) automata of the running example are shown in Fig. 2. We use a representation of (max,+) automata that generalizes the definition of [16] with rewards. It corresponds to the refinement discussed in Section VI in [16], in which a regular language constraining the possible sequences of events is combined with a (max,+) automaton defining the timing constraints of the events using the Hadamard product. In our representation the automata encode the possible sequences of activities (the regular language of events) and the timing constraints are encoded using the classical matrix representation of (max,+) automata.

Definition 1 ((max,+) automaton (adapted from [16])). *A* (max,+) automaton $\mathcal{A}$ *is a tuple* $\langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ *where* S *is a finite set of states,* $\hat{s} \in S$ *is the initial state,* $S^m \subseteq S$ *is the set of* marked states*,* $\mathcal{A}ct$ *is a nonempty set of activities, function* $reward : \mathcal{A}ct \to \mathbb{R}^{\geq 0}$ *quantifies the progress per activity, function* M *maps each activity to its associated (max,+) matrix, and* $T \subseteq S \times \mathcal{A}ct \times S$ *is the transition relation. We assume that* $\mathcal{A}$ *is deterministic; for any* $s, s', s'' \in S$ *and* $A \in \mathcal{A}ct$*,* $\langle s, A, s'\rangle \in T$ *and* $\langle s, A, s''\rangle \in T$ *imply* $s' = s''$*.*

Both plants and supervisors are described as (max,+) automata. Marked states are used to define a notion of progress of the plant, used in supervisor synthesis, as illustrated below.

Let $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ be a (max,+) automaton. Let set $\mathcal{A}ct^*$ contain all finite *strings* over $\mathcal{A}ct$, including the empty string ε. The transition relation is written in infix notation $s_1 \xrightarrow{A} s_2$ if $\langle s_1, A, s_2\rangle \in T$, and is extended to strings in $\mathcal{A}ct^*$ in the relation $\to^*$ by letting $s \xrightarrow{\varepsilon}{}^* s$ for all $s \in S$, and for all $\alpha \in \mathcal{A}ct^*$, $A \in \mathcal{A}ct$, $s, s' \in S$, $s \xrightarrow{\alpha A}{}^* s'$ if $s \xrightarrow{\alpha}{}^* s''$ and $s'' \xrightarrow{A} s'$ for some $s'' \in S$. We write $s \to^* s'$ if $s \xrightarrow{\alpha}{}^* s'$ for some $\alpha \in \mathcal{A}ct^*$. Set $enabled(s) = \{A \in \mathcal{A}ct \mid \exists s' : s \xrightarrow{A} s'\}$ contains all activities that are *enabled* in s. State s is a *deadlock* state if $enabled(s) = \emptyset$. Since $\mathcal{A}$ is deterministic, for any activity $A \in enabled(s)$, there is a unique A-successor of s, denoted by $A(s)$. For activity sequence $A_1 \dots A_n$, the resulting state is defined inductively as $\varepsilon(s) = s$ and $(A_1 \dots A_n A_{n+1})(s) = A_{n+1}((A_1 \dots A_n)(s))$ for $n \geq 0$ if $A_{n+1} \in enabled((A_1 \dots A_n)(s))$ and $(A_1 \dots A_n)(s)$ is defined. Given this definition, $A(s)$ denotes the state reached after executing activity A.

[1] and are included in the appendix for this submission



(a) P^1 (b) P^2 (c) P^3 (d) P^4

(e) Composition $\mathcal{P} = P^1 \parallel P^2 \parallel P^3 \parallel P^4$. The reduced composition $\mathcal{P}'$ is shown with thick transitions.

(f) Supervisor $\mathcal{A}_{sup} = sup\mathcal{CN}(\mathcal{P})$. The reduced supervisor $\mathcal{A}'_{sup} = sup\mathcal{CN}(\mathcal{P}')$ is shown with thick transitions.

Fig. 2: Running example: plants P^1, P^2, P^3, and P^4, and the composition $\mathcal{P}$. Transitions of uncontrollable activities are denoted with dashed arrows. Marked states are indicated in gray.

$$\underset{\boldsymbol{M}_A}{\begin{bmatrix} 4 & 5 & -\infty \\ -\infty & 3 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix}} \quad \underset{\boldsymbol{M}_B}{\begin{bmatrix} 1 & 3 & -\infty \\ 1 & 3 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix}} \quad \underset{\boldsymbol{M}_C}{\begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 4 \end{bmatrix}}$$

$$\underset{\boldsymbol{M}_D}{\begin{bmatrix} 2 & -\infty & 3 \\ -\infty & 0 & -\infty \\ 2 & -\infty & 3 \end{bmatrix}} \quad \underset{\boldsymbol{M}_E}{\begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 5 \end{bmatrix}} \quad \underset{\boldsymbol{M}_U}{\begin{bmatrix} 2 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix}}$$

Fig. 3: (max,+) matrices of activities A, B, C, D, E and U.

A possible behavior of an automaton is described in a *run*. A(n in)finite run ρ of $\mathcal{A}$ is a(n in)finite, alternating sequence of states and activities: $\rho = s_0 A_1 s_1 A_2 s_2 A_3 \ldots$ such that $s_0 = \hat{s}$ and $s_{i+1} = A_{i+1}(s_i)$ for all $i \geq 0$. Given run ρ and $i \geq 0$, let $\rho[..i]$ denote the prefix $s_0 A_1 \ldots s_i$, and let $\rho[i,j] = s_i A_{i+1} \ldots s_j$ with $0 \leq i \leq j$ denote a run fragment from state s_i until s_j. Furthermore, let $\rho[i]$ denote state s_i in ρ.

Synchronous composition is used to compose (max,+) automata.

Definition 2 (Synchronous composition)**.** *Given automata* $\mathcal{A}_1 = \langle S_1, \hat{s}_1, S_1^m, \mathcal{A}ct_1, reward_1, M_1, T_1 \rangle$ *and* $\mathcal{A}_2 = \langle S_2, \hat{s}_2, S_2^m, \mathcal{A}ct_2, reward_2, M_2, T_2 \rangle$*, we define the synchronous composition* $\mathcal{A}_1 \parallel \mathcal{A}_2 = \langle S_1 \times S_2, \langle \hat{s_1}, \hat{s_2} \rangle, S_1^m \times S_2^m, \mathcal{A}ct_1 \cup \mathcal{A}ct_2, reward_1 \cup reward_2, M_1 \cup M_2, T_{12} \rangle$*, where*

$\langle s_1, s_2 \rangle \xrightarrow{A}_{12} \langle s_1', s_2' \rangle$ *if* $A \in \mathcal{A}ct_1 \cap \mathcal{A}ct_2, s_1 \xrightarrow{A}_1 s_1', s_2 \xrightarrow{A}_2 s_2'$;

$\langle s_1, s_2 \rangle \xrightarrow{A}_{12} \langle s_1', s_2 \rangle$ *if* $A \in \mathcal{A}ct_1 \setminus \mathcal{A}ct_2, s_1 \xrightarrow{A}_1 s_1'$;

$\langle s_1, s_2 \rangle \xrightarrow{A}_{12} \langle s_1, s_2' \rangle$ *if* $A \in \mathcal{A}ct_2 \setminus \mathcal{A}ct_1, s_2 \xrightarrow{A}_2 s_2'$.

It is assumed that for each activity $A \in \mathcal{A}ct_1 \cap \mathcal{A}ct_2$*,* $reward_1(A) = reward_2(A)$ *and* $M_1(A) = M_2(A)$.

Fig. 2e shows the synchronous composition of $P^1 \parallel P^2 \parallel P^3 \parallel P^4$ (composition is associative). An activity is disabled in a state, if it is disabled in the current state of one of the automata that has the activity in its alphabet. For example, D is initially disabled because plant P^3 initially disables activity D. Note that the synchronous composition of deterministic automata is again deterministic.

As explained in the introduction, activities abstract from low-level activity-internal action executions. For our intended supervisor synthesis and performance analysis, we only need to capture the timing of the claims and releases of resources. So we abstract from the low-level actions. The (max,+) matrices of the activities of the running example are shown in Fig. 3. Each matrix row represents the symbolic release time of a resource in terms of all the availability times of resources. Assume a (global) resource set $Res = \{r_1, r_2, r_3\}$. Let R map each activity to the set of resources the activity uses. As an example, consider matrix $\boldsymbol{M}_A$ of activity A with $R(A) = \{r_1, r_2\}$. The first row describes the release time of resource $r_1 \in Res$, expressed in terms of when resources r_1, r_2 and r_3 are available at the start of executing A. In the execution of activity A, there is a time delay of 4 time units between the claiming of resource r_1 and its subsequent release. Similarly, a delay of 5 occurs between the claiming of resource r_2 and the release of r_1. There is no dependency on the availability of r_3, indicated by $-\infty$.

The two essential characteristics in the execution of an activity are *synchronization*, when an action needs to wait until resources are available, and *delay* to model the execution time of actions on the resource. These characteristics correspond to the (max,+) operators *maximum* (max) and *addition* (+), defined over the set $\mathbb{R}^{-\infty} = \mathbb{R} \cup \{-\infty\}$. Operators max and + are defined as usually, with the additional convention that $-\infty$ is the unit element of max: $\max(-\infty, x) = \max(x, -\infty) = x$, and

the zero-element of $+$: $-\infty + x = x + -\infty = -\infty$. Since (max,+) algebra is a linear algebra, it can be extended to matrices and vectors in the usual way. Given matrix $\boldsymbol{A}$ and matrix $\boldsymbol{B}$, we use $\boldsymbol{A} \otimes \boldsymbol{B}$ to denote the $(\max, +)$ matrix multiplication. Given $m \times p$ matrix $\boldsymbol{A}$ and $p \times n$ matrix $\boldsymbol{B}$, the elements of the resulting matrix $\boldsymbol{A} \otimes \boldsymbol{B}$ are determined by: $[\boldsymbol{A} \otimes \boldsymbol{B}]_{ij} = \max_{k=1}^{p}([\boldsymbol{A}]_{ik} + [\boldsymbol{B}]_{kj})$. Adding a constant c to matrix $\boldsymbol{A}$ yields a new matrix $\boldsymbol{A} + c$ with elements $[\boldsymbol{A} + c]_{ij} = [\boldsymbol{A}]_{ij} + c$. For any vector $\boldsymbol{x}$ of size n, $\|\boldsymbol{x}\| = \max_{i=1}^{n}[\boldsymbol{x}]_i$ denotes the vector norm of $\boldsymbol{x}$. For vector $\boldsymbol{x}$, with $\|\boldsymbol{x}\| > -\infty$, $norm(\boldsymbol{x})$ denotes $\boldsymbol{x} - \|\boldsymbol{x}\|$, the normalized vector, such that $\left\|norm(\boldsymbol{x})\right\| = 0$. We use $\mathbf{0}$ to denote a vector with only zero entries.

Resource availability times are captured in a (max,+) vector, typically denoted using γ. Given such a *resource availability vector*, we obtain the new resource availability vector after executing an activity by multiplying it with the corresponding (max,+) matrix. The timing evolution of the system is therefore expressed using (max,+) vector-matrix multiplication. When all resources are initially available, captured in vector $\mathbf{0}$, the new availability times of the resources after executing A are computed as follows:

$$\boldsymbol{M}_A \otimes \mathbf{0} = \begin{bmatrix} \max(4+0, 5+0, -\infty+0) \\ \max(-\infty+0, 3+0, -\infty+0) \\ \max(-\infty+0, -\infty+0, 0+0) \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 0 \end{bmatrix}.$$

Resources r_1 and r_2 are available again after 5 and 3 time units, respectively. Resource r_3 is not present in set $R(A)$, but a row is present for this resource to carry over the time stamps of all system resources. The availability time of r_3 stays 0, since it is not used by activity A. Note that this representation of activity timing generalizes the well-known heaps-of-pieces model [17], [18], in which the relative release times of resources are fixed, independent of the resource availability times.

The timing semantics of an activity sequence is defined in terms of repeated matrix multiplication. For example, the resource availability vector after executing activity sequence ABC given vector $\mathbf{0}$ (also shown in Fig. 4) is computed as follows:

$$\boldsymbol{M}_C \otimes \boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \mathbf{0} = [6, 6, 4]^{\intercal}.$$

We can now define the input model for our synthesis as a composition of (max,+) automata, referred to as a (max,+) timed system.

Definition 3 ((max,+) timed system)**.** *A* (max,+) timed system $\mathcal{M}$ *is described by* $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ *with (max,+) automata* $\mathcal{A}_i = \langle S_i, \hat{s}_i, S_i^m, \mathcal{A}ct_i, reward_i, M_i, T_i \rangle$ *with* $1 \leq i \leq n$*. We assume that all matrices have the size* $|Res|^2$*, and that for all* $1 \leq i, j \leq n$ *and each activity* $A \in \mathcal{A}ct_i \cap \mathcal{A}ct_j$*,* $reward_i(A) = reward_j(A)$ *and* $M_i(A) = M_j(A)$*.*

The composition of all the individual automata is again an automaton. We assume that the matrices have the same size to ensure that they can be multiplied. Additional resources be added to a matrix, by adding a new row and column for the resource and having $-\infty$ in all the new positions, except on the diagonal where the value is 0.

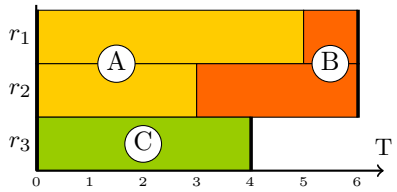


Fig. 4: Gantt chart of activity sequence ABC when all resources are initially available. The execution of activities A, B, and C is indicated in yellow, orange, and green.

A (max,+) automaton can be interpreted as a normalized $(\max, +)$ state space that captures all the accepted runs, and contains all the necessary information for evaluation of performance properties. The state space records normalized resource availability vectors, with transitions between them. Each configuration $c = \langle s, \boldsymbol{\gamma} \rangle$ in the state space consists of a state s of the (max,+) automaton and a normalized resource availability vector $\boldsymbol{\gamma}$. Fig. 6 shows the normalized (max,+) state spaced of a simplified running example, discussed in more detail below.

Definition 4 (Normalized (max,+) state space (adapted from [19]))**.** *Given (max,+) automaton* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T \rangle$ *with matrices of size* $|Res|^2$*, we define the normalized* $(\max, +)$ *state space* $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$ *as follows:*

- *set* $C = S \times \mathbb{R}^{-\infty^{|Res|}}$ *of configurations;*
- *initial configuration* $\hat{c} = \langle \hat{s}, \mathbf{0} \rangle$*;*
- *labeled transition relation* $\Delta \subseteq C \times \mathcal{A}ct \times C$ *that consists of the transitions in the set* $\{\langle \langle s, \boldsymbol{\gamma} \rangle, A, \langle s', norm(\boldsymbol{\gamma}') \rangle \rangle \mid s \xrightarrow{A} s' \wedge \boldsymbol{\gamma}' = M(A) \otimes \boldsymbol{\gamma}\}$*;*
- *function* w_1 *that assigns a weight* $w_1(c, A, c') = reward(A)$ *to each transition* $\langle c, A, c' \rangle \in \Delta$*;*
- *function* w_2 *that assigns a weight* $w_2(c, a, c') = \left\|M(A) \otimes \boldsymbol{\gamma}\right\|$ *to each transition* $\langle c, A, c' \rangle \in \Delta$ *with* $c = \langle s, \boldsymbol{\gamma} \rangle$*. This weight indicates the execution time.*

The set of enabled activities and runs in a normalized (max,+) state space $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$ is defined in a similar way as in a (max,+) automaton. A(n in)finite run ρ of $\mathcal{S}$ is a(n in)finite, alternating sequence of configurations and activities: $\rho = c_0 A_1 c_1 A_2 c_2 A_3 \ldots$ such that $c_0 = \hat{c}$ and $c_{i+1} = A_{i+1}(c_i)$ for all $i \geq 0$. We define run prefix $\rho[..i] = c_0 A_1 \ldots c_i$, run fragment $\rho[i, j] = c_i A_{i+1} \ldots c_j$ from configuration c_i until c_j, and $\rho[i] = c_i$. We define vector $\bar{\boldsymbol{\gamma}}_n = (\bigotimes_{k=1}^{n} M(A_k)) \otimes \mathbf{0}$, the resulting resource availability vector after executing activities $A_1 \cdots A_n$ (without normalization). This vector can also be derived from the normalized (max,+) state space as the summation of encountered w_2 values.

Proposition 5. *Let* $\mathcal{S}$ *be a normalized (max,+) state space, and* $\rho = c_0 A_1 c_1 A_2 c_2 A_3 \ldots$ *be a run in* $\mathcal{S}$ *with* $c_i = \langle s_i, \boldsymbol{\gamma}_i \rangle$ *for each* i*. Then, for all* $n \geq 0$ *it holds that* $\bar{\boldsymbol{\gamma}}_n = \sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + \boldsymbol{\gamma}_n$*.*

In some cases, the state space might be infinite [19]. It is guaranteed to be finite, if for every activity sequence

$A_1A_2\ldots$ allowed by the (max,+) automaton and any $k \geq 0$, there is some $m > k$ such that the matrix related to the subsequence $A_k \ldots A_m$, $\boldsymbol{M}_{A_m} \otimes \cdots \otimes \boldsymbol{M}_{A_k}$ contains no entries $-\infty$ [19]. This means that a complete resource-to-resource synchronization is achieved. A sufficient condition for this is that the claim of each resource is linked to the claim of each other resource via the resource dependencies over each cycle in the automaton. In practical systems, resources typically do not operate fully independent and have dependencies. If this is not the case, then an optimal controller can be generated for each part of the system that operates fully independently.

III. Properties to be preserved

In this section we introduce the properties that we want to preserve. On the one hand, these are functional properties, namely controllability, nonblockingness, and least-restrictiveness. On the other, hand these are performance properties, namely latency and throughput.

A. Controllability, nonblockingness, least-restrictiveness

In supervisory control, the activity alphabet $\mathcal{A}ct$ is partitioned into set $\mathcal{A}ct_c$ of *controllable* activities and set $\mathcal{A}ct_u$ of *uncontrollable* activities. In the running example, A, B, C, D, and E are controllable and U is uncontrollable.

To define least-restrictiveness, we need the notions of subautomata and union of automata. A *subautomaton* is obtained as reduction of a given (max,+) automaton, where a subset of the states and transitions is preserved.

Definition 6 (Subautomaton)**.** *Let* $\mathcal{A}_1 = \langle S_1, \hat{s}, S_1^m, \mathcal{A}ct, reward, M, T_1\rangle$ *and* $\mathcal{A}_2 = \langle S_2, \hat{s}, S_2^m, \mathcal{A}ct, reward, M, T_2\rangle$ *be two (max,+) automata with the same alphabet* $\mathcal{A}ct$ *(and derived reward function and timing matrices) and initial state* $\hat{s}$*. Automaton* $\mathcal{A}_1$ *is a* subautomaton *of* $\mathcal{A}_2$*, denoted* $\mathcal{A}_1 \preceq \mathcal{A}_2$*, iff* $S_1 \subseteq S_2, T_1 \subseteq T_2$*, and* $S_1^m = S_1 \cap S_2^m$*. We require that* $S_1^m = S_1 \cap S_2^m$ *to ensure that marking in* $\mathcal{A}_1$ *is consistent with marking in* $\mathcal{A}_2$*.*

Definition 7 (Union of (max,+) automata)**.** *Let* $\mathcal{A}_i = \langle S_i, \hat{s}, S_i^m, \mathcal{A}ct, reward, M, T_i\rangle$*,* $i \in I$ *be a set of automata all having the same activity alphabet, reward function, timing matrices, and initial state. The union of these (max,+) automata is defined as* $\bigcup_{i\in I} \mathcal{A}_i = \langle \bigcup_{i\in I} S_i, \hat{s}, \bigcup_{i\in I} S_i^m, \mathcal{A}ct, reward, M, \bigcup_{i\in I} T_i\rangle$*.*

The behavior of an *uncontrolled system* is represented by a plant $\mathcal{P}$. A supervisor $\mathcal{A}_{sup}$ is added to ensure that the *controlled system*, formed by $\mathcal{P} \parallel \mathcal{A}_{sup}$, is *nonblocking* [1].

Definition 8 (Nonblockingness)**.** *A (max,+) automaton* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ *is* nonblocking *iff, for every state* $s \in S$ *such that* $\hat{s} \rightarrow^* s$*,* s *is a nonblocking state. A state* $s \in S$ *is a* nonblocking *state iff* $s \rightarrow^* s^m$ *for some marked state* $s^m \in S^m$*; otherwise it is a* blocking *state.*

Plant $\mathcal{P}$ shown in Fig. 2e is blocking, since states s_{10}, s_{11}, s_{12}, and s_{13} are blocking. The automaton in Fig. 2f is nonblocking. This illustrates how marked states can be used to ensure a notion of progress.

We assume that $\mathcal{A}_{sup} \preceq \mathcal{P}$, since the computed supervisor is often a refinement of the plant by disabling transitions that lead to undesired behavior, violating the nonblocking requirements or controllability requirements, defined below. Since $\mathcal{A}_{sup} \preceq \mathcal{P}$, $\mathcal{P} \parallel \mathcal{A}_{sup} = \mathcal{A}_{sup}$, which means that $\mathcal{P} \parallel \mathcal{A}_{sup}$ is nonblocking iff $\mathcal{A}_{sup}$ is nonblocking.

A supervisor is required to be *controllable* with respect to the plant it needs to control, such that it does not disable any uncontrollable activity that the plant defines after a sequence allowed in the composition. Controllability of a (max,+) automaton $\mathcal{A}$ with respect to an automaton $\mathcal{P}$ is defined in the following way.

Definition 9 (Controllability)**.** *Let* $\mathcal{P} = \langle S_\mathcal{P}, \hat{s}_\mathcal{P}, S_\mathcal{P}^m, \mathcal{A}ct_\mathcal{P}, reward_\mathcal{P}, M_\mathcal{P}, T_\mathcal{P}\rangle$ *and* $\mathcal{A} = \langle S_\mathcal{A}, \hat{s}_\mathcal{A}, S_\mathcal{A}^m, \mathcal{A}ct_\mathcal{A}, reward_\mathcal{A}, M_\mathcal{A}, T_\mathcal{A}\rangle$ *be (max,+) automata, and* $\mathcal{A}ct_u \subseteq \mathcal{A}ct_\mathcal{P} \cup \mathcal{A}ct_\mathcal{A}$ *the set of uncontrollable activities.* $\mathcal{A}$ *is* controllable with respect to $\mathcal{P}$ *iff, for every string* $\alpha \in (\mathcal{A}ct_\mathcal{P} \cup \mathcal{A}ct_\mathcal{A})^*$*, every state* $s \in S_\mathcal{A}$*, and every* $U \in \mathcal{A}ct_u$ *such that* $\hat{s}_\mathcal{A} \xrightarrow{\alpha}_\mathcal{A}^* s$ *and* $\hat{s}_\mathcal{P} \xrightarrow{\alpha U}_\mathcal{P}^* s'$ *for some* $s' \in S_\mathcal{P}$*, it holds that* $s \xrightarrow{U}_\mathcal{A} s''$ *for some* $s'' \in S_\mathcal{A}$*. Any state* s *of* $\mathcal{A}$ *that satisfies this property is called a* controllable *state; otherwise it is* uncontrollable*.*

As an example, consider string $CABE$ and path $s_1 \xrightarrow{CABE}_{\mathcal{A}'_{sup}}^* s_9$ in supervisor $\mathcal{A}'_{sup}$ shown in Fig. 2f. In plant $\mathcal{P}$, the execution of string $CABE$, leads to state s_9. In state s_9 in $\mathcal{P}$, uncontrollable activity U is enabled. In $\mathcal{A}'_{sup}$, activity U is also present in state s_9 in $\mathcal{A}'_{sup}$. Since this also holds for the other strings and uncontrollable activities, $\mathcal{A}'_{sup}$ is controllable with respect to $\mathcal{P}$.

The union of controllable and nonblocking subautomata of a given automaton is again controllable and nonblocking [8]. A subautomaton is least-restrictive iff it is the union of all the subautomata of $\mathcal{P}$ that satisfy both properties.

Definition 10 (Least-restrictiveness)**.** *Let* $\mathcal{A}, \mathcal{A}'$ *be (max,+) automata. Define predicate* $CN(\mathcal{A}', \mathcal{A})$ *to be true iff* $\mathcal{A}' \preceq \mathcal{A}$ *and* $\mathcal{A}'$ *is nonblocking and controllable with respect to* $\mathcal{A}$*. The supremal controllable and nonblocking subautomaton of* $\mathcal{A}$ *is* $sup\mathcal{CN}(\mathcal{A}) = \bigcup_{\mathcal{A}', CN(\mathcal{A}', \mathcal{A})} \mathcal{A}'$*.* $\mathcal{A}'$ *is* least restrictive *with respect to* $\mathcal{A}$ *iff* $\mathcal{A}' = sup\mathcal{CN}(\mathcal{A})$*.*

Both $\mathcal{A}_{sup}$ and $\mathcal{A}'_{sup}$, shown in Fig. 2f, are nonblocking and controllable with respect to $\mathcal{P}$. Given $\mathcal{P}$, calculating $\mathcal{A}_{sup} = sup\mathcal{CN}(\mathcal{P})$ is called *synthesis*. The synthesis algorithms to compute $\mathcal{A}_{sup}$ implement fixed-point computations [1], [2], where in each iteration blocking states are removed. As a consequence, other states can become *bad* states, when they are blocking themselves or uncontrollable. These bad states are identified and removed iteratively, until no more such states remain in the resulting supervisor.

B. Throughput and latency

The performance of a supervisor is quantified using throughput and latency metrics, defined on the normalized (max,+) state space of the supervisor. Throughput is

defined as the ratio between the total reward and the total execution time of a run over time. For throughput analysis, we only want to consider infinite runs. For a finite run, throughput is considered to be zero, because its reward tends to zero over time when time proceeds indefinitely. Latency is the temporal distance that separates the resource availability times of a resource at the start of two activity instances in a run. Latency analysis can be performed on both finite and infinite runs. For readability, we assume for performance analysis that a supervisor is *total*, meaning that each state has at least one outgoing transition. A non-total supervisor has a throughput lower bound of zero and the presented latency analysis applies to non-total supervisors as well. Since supervisor synthesis guarantees that from each state a marked state is reachable, totality can be verified by checking that each marked state in a supervisor has at least one outgoing transition.

A total supervisor can be seen as an ω-automaton [20] that accepts infinite words over $\mathcal{A}ct$. There are no specific acceptance conditions on these words, so any infinite word starting in the initial state is accepted. Marked states are not used to define acceptance conditions. For performance analysis of a total supervisor all infinite runs in the timed states space $\mathcal{S}$ are considered, contained in set $\mathcal{R}(\mathcal{S})$.

We define *throughput* of a run as the ratio between the total reward (sum of w_1 weights) and the total execution time (sum of w_2 weights).

Definition 11 (Ratio value of a run (fragment))**.** *The ratio of a run (fragment) $\rho = c_0A_1c_1A_2c_2A_3\ldots$ is the ratio of the sums of weights w_1 and w_2, defined as follows*

$$\mathsf{Ratio}(\rho) = \limsup_{l\to\infty} \frac{\sum_{i=0}^{l} w_1(c_i, A_{i+1}, c_{i+1})}{\sum_{i=0}^{l} w_2(c_i, A_{i+1}, c_{i+1})}.$$

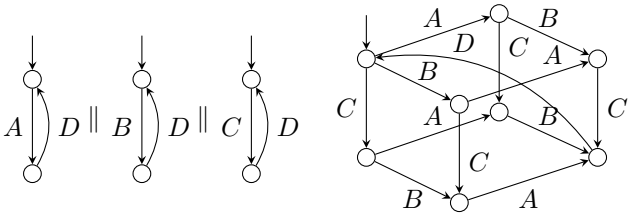
The throughput guarantee corresponds to the *minimum ratio value* achieved by any of those runs:

$$\tau_{min}(\mathcal{S}) = \inf_{\rho\in\mathcal{R}(\mathcal{S})} \mathsf{Ratio}(\rho).$$

Since $\mathcal{S}$ is finite, infinite runs pass recurrent configurations infinitely often. Thus infinite runs are composed of simple cycles of the state space. The minimum ratio value of the state space is hence determined by the simple cycle with the lowest ratio value, since this behavior can be continuously repeated in a run. This cycle thus provides a lower bound on the throughput (hence a throughput guarantee). Since $\mathcal{S}$ has a finite number of simple cycles (no repetition of transitions is allowed), we can determine the minimum ratio value of the graph from a *minimum cycle ratio* (MCR) analysis [21].

Proposition 12 (Adapted from Proposition 1 in [22])**.** $\tau_{min}(\mathcal{S}) = \inf_{cycle\in cycles(\mathcal{S})} \mathsf{Ratio}(cycle) = MCR(\mathcal{S})$*, where $cycles(\mathcal{S})$ denotes all cycles in $\mathcal{S}$ and $MCR(\mathcal{S})$ is the MCR of $\mathcal{S}$.*

To illustrate the MCR, consider the normalized (max,+) state space shown in Fig. 6, obtained from the composition of plants $\bar{P}^1$, $\bar{P}^2$, and $\bar{P}^3$ shown in Fig. 5. These are inspired by plants P^1, P^2, and P^3 shown in Fig. 2. We use this



(a) $\bar{P}^1$, $\bar{P}^2$, and $\bar{P}^3$. (b) $\bar{\mathcal{P}} = \bar{P}^1 \parallel \bar{P}^2 \parallel \bar{P}^3$.

Fig. 5: Running example with (max,+) automata $\bar{P}^1$, $\bar{P}^2$, and $\bar{P}^3$, and composition $\bar{\mathcal{P}}$.

smaller model such that we can show the full (max,+) state space. In this state space, activities A, B and D have reward 0, (weight w_1), and activity C has reward 1. Then, the MCR is $1/8$, with a total reward of 1 and a total execution time of 8, which can for instance be found in the cycle corresponding to the execution of $(CBAD)^\omega$:

$$\langle s_1, \begin{bmatrix} 0 \\ -4 \\ 0 \end{bmatrix}\rangle \xrightarrow{C,4} \langle s_8, \begin{bmatrix} -4 \\ -8 \\ 0 \end{bmatrix}\rangle \xrightarrow{B,0} \langle s_7, \begin{bmatrix} -3 \\ -3 \\ 0 \end{bmatrix}\rangle \xrightarrow{A,2} \langle s_4, \begin{bmatrix} 0 \\ -2 \\ -2 \end{bmatrix}\rangle.$$
$$\xleftarrow{\qquad D,2 \qquad}$$

In this cycle, the first transition represents the execution of C on resource r_3, which is computed through matrix multiplication with $\boldsymbol{M}_C$, adds 4 times units to the total execution time, and results in vector $[0, -4, 4]^T$. Since the vector is normalized, 4 time units are deducted from each value. The other transitions can be computed in a similar way. Other periodic executions where B precedes A, i.e. $(BACD)^\omega$ and $(BCAD)^\omega$, have the same MCR value.

Latency is the time delay between a stimulus and its effect. A well-known related notion in the field of production systems is makespan, where the stimulus is the start of the schedule and the effect is the completion of the product. In the context of (max,+) timed systems, we define *latency* in terms of the temporal distance that separates the resource availability times of a resource at a designated source activity A_{src} and sink activity A_{snk}. In the state space, consider some run $\rho = c_0A_1c_1A_2\ldots$ with $c_i = \langle s_i, \boldsymbol{\gamma}_i\rangle$ containing run fragment $\rho[i, j+1] = c_iA_{i+1}\ldots c_jA_{j+1}c_{j+1}$, with $A_{i+1} = A_{src}$ and $A_{j+1} = A_{snk}$. We define the start-to-start latency λ between the resource availability times of resource r in $\boldsymbol{\gamma}_i$ and $\boldsymbol{\gamma}_j$ as

$$\lambda(\rho, i, j, r) = [\bar{\boldsymbol{\gamma}}_j]_r - [\bar{\boldsymbol{\gamma}}_i]_r.$$

To illustrate, consider the execution of activity sequence ABC. Suppose we want to compute the start-to-start latency between the availability times of resource r_1 in $\bar{\boldsymbol{\gamma}}_0 = \mathbf{0}$ (start of activity A) and $\bar{\boldsymbol{\gamma}}_2 = \boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \mathbf{0} = [6, 6, 0]^\intercal$ (start of activity C). The latency is now computed as

$$\lambda(\rho, 0, 2, r_1) = [\bar{\boldsymbol{\gamma}}_2]_{r_1} - [\bar{\boldsymbol{\gamma}}_0]_{r_1} = [\begin{bmatrix} \mathbf{6} \\ 6 \\ 0 \end{bmatrix}]_{r_1} - [\begin{bmatrix} \mathbf{0} \\ 0 \\ 0 \end{bmatrix}]_{r_1} = 6.$$

We assume that the occurrences of A_{src} and A_{snk} activities are related. In any run, for any $k \geq 1$, the k-th

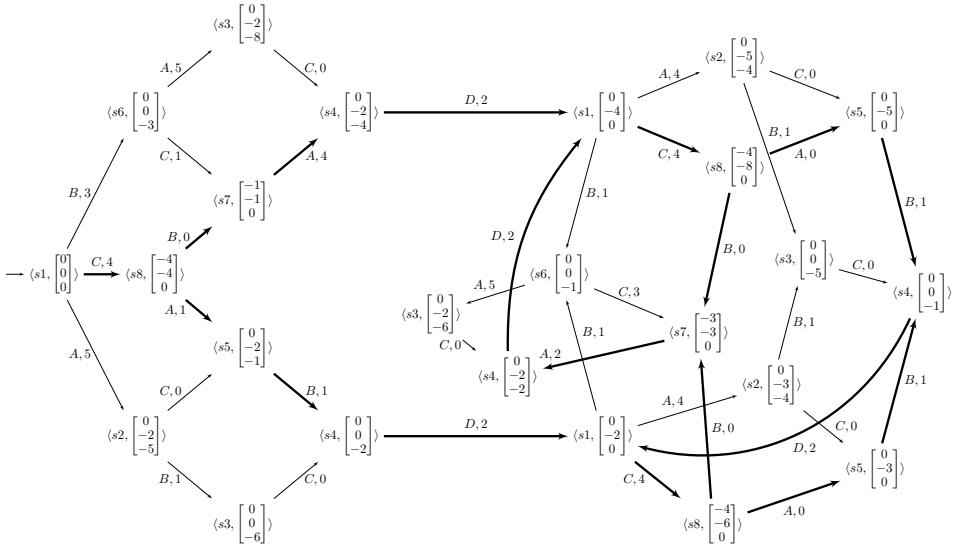


Fig. 6: Normalized (max,+) state space of $\bar{\mathcal{P}}$ in Fig. 5b. The reduction is shown with thick transitions. Transitions are annotated with the activity and w_2 value (added execution time).

occurrence of A_{src} is paired with the k-th occurrence of A_{snk}. We refer to such a pair of related activities as a source-sink pair. Let $\mathsf{getOccurrence}(\rho, A, k)$ denote the index of the k-th occurrence of activity A in run ρ. The *start-to-start latency* for resource r in ρ with source-sink pair $A_{i+1} = A_{src}$ and $A_{j+1} = A_{snk}$ in run fragment $\rho[i, j+1]$, is equal to $\lambda(\rho, i, j, r)$. The maximum start-to-start latency in a run is obtained by looking at all source-sink pairs:

$$\begin{aligned}\lambda_{max}(\rho, A_{src}, A_{snk}, r) &= \sup_{k\geq 1} \lambda_k(\rho) \textbf{ where}\\ \lambda_k(\rho) &= \lambda(\rho, i, j, r),\\ i &= \mathsf{getOccurence}(\rho, A_{src}, k), \text{ and}\\ j &= \mathsf{getOccurence}(\rho, A_{snk}, k).\end{aligned}$$

Definition 13 (Latency)**.** *Given normalized (max,+) state space $\mathcal{S}$, the maximum start-to-start latency of $\mathcal{S}$ with resource r and source-sink pair A_{src}, A_{snk} is found by taking the maximum latency over all runs in the state space:*

$$\lambda_{max}(\mathcal{S}) = \sup_{\rho\in\mathcal{R}(\mathcal{S})} \lambda_{max}(\rho, A_{src}, A_{snk}, r).$$

As an example, consider the execution of activity sequence ACB starting from the initial configuration in the normalized (max,+) state space shown in Fig. 6. This corresponds to run fragment $\rho =$

$$\langle s_1, \begin{bmatrix}0\\0\\0\end{bmatrix}\rangle \xrightarrow{A,5} \langle s_2, \begin{bmatrix}0\\-2\\-5\end{bmatrix}\rangle \xrightarrow{C,0} \langle s_5, \begin{bmatrix}0\\-2\\-1\end{bmatrix}\rangle \xrightarrow{B,1} \langle s_4, \begin{bmatrix}0\\0\\-2\end{bmatrix}\rangle.$$

Say that we want to compute the start-to-start latency between the resource availability times of r_2 in $\bar{\gamma}_0$ (start of activity A) and in $\bar{\gamma}_2$ (start of activity B). First, we compute the vectors without normalization from the state space using Proposition 5 to find $\bar{\gamma}_0 = \gamma_0$ and $\bar{\gamma}_2 = [5, 3, 4]^\intercal$. Then, we compute the latency:

$$\lambda(\rho, 0, 2, r_2) = [\bar{\gamma}_2]_{r_2} - [\bar{\gamma}_0]_{r_1} = [\begin{bmatrix}5\\\mathbf{3}\\4\end{bmatrix}]_{r_2} - [\begin{bmatrix}0\\\mathbf{0}\\0\end{bmatrix}]_{r_2} = 3.$$

IV. Normalized (max,+) State Space Reduction

Performance analysis of the supervisory controller is done on the corresponding normalized (max,+) state space. In this section, we consider necessary conditions on a reduction function to preserve throughput and latency. In the next section, we lift these conditions to the level of a (max,+) automaton and add conditions to preserve functional aspects in the reduced supervisor. The reduced (max,+) state space should be performance-equivalent to the full (max,+) state space, defined as follows.

Definition 14 (State-space performance equivalence)**.** *Normalized (max,+) state spaces $\mathcal{S}$ and $\mathcal{S}'$ are performance-equivalent, denoted $\mathcal{S} \approx_p \mathcal{S}'$ iff $\tau_{min}(\mathcal{S}) = \tau_{min}(\mathcal{S}')$ and $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$.*

A state space may have multiple runs with the same ratio value caused by the interleaving of *ratio-independent* activities that have no mutual influence. The first property of such activities, say A and B, is the classical notion of *independence*: in every configuration where A and B are both enabled, the execution of one activity cannot disable the other activity, and the resulting configuration after executing both activities in any order is the same. The second property requires that the summed weights w_1 and w_2 of the corresponding transitions of A and B is the same. The third property requires that A and B do not share resources. As an example, consider the initial configuration in Fig. 6, where activities A and C are ratio-independent.

Definition 15 (Ratio independent)**.** *Let $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2\rangle$ be a normalized (max,+)*

state space, $c \in C$ be a configuration, and $A, B \in enabled(c)$ be activities enabled in c. Activities A and B are ratio independent *in c iff they satisfy all the following conditions:*

1) *if $A, B \in enabled(c)$, then $B \in enabled(A(c))$, $A \in enabled(B(c))$, and $AB(c) = BA(c)$;*
2) *$w_i(c, A, A(c)) + w_i(A(c), B, AB(c)) = w_i(c, B, B(c)) + w_i(B(c), A, BA(c))$ for $i \in \{1, 2\}$;*
3) *$R(A) \cap R(B) = \emptyset$.*

Two activities are ratio dependent *iff they are not ratio independent.*

We reduce the size of the state space by removing redundant interleaving of ratio-independent activities that lead to multiple equivalent runs with the same ratio value. To formalize this equivalence, we first define the equivalence of run prefixes. Two run prefixes are equivalent iff their corresponding activity sequences can be obtained from each other by repeatedly commuting adjacent ratio-independent activities. Given prefix $\rho[..m] = c_0 A_1 \dots A_m c_m$ of some run ρ, let $Act(\rho[..m])$ denote the activity sequence $A_1 \dots A_m$.

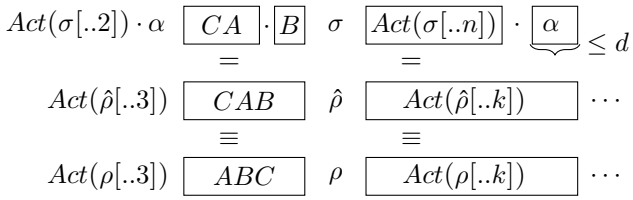
Definition 16 (String equivalence)**.** *Let $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$ be a normalized (max,+) state space. Strings $\alpha, \beta \in \mathcal{A}ct^*$ are equivalent [23] in a configuration c, denoted $\alpha \equiv_c \beta$, iff there exists a list of strings $\upsilon_0, \upsilon_1, \dots, \upsilon_n$, where $\upsilon_0 = \alpha$, $\upsilon_n = \beta$, and for each $0 \leq i < n$, $\upsilon_i = \bar{\upsilon} AB \hat{\upsilon}$ and $\upsilon_{i+1} = \bar{\upsilon} BA \hat{\upsilon}$ for some $\bar{\upsilon}, \hat{\upsilon} \in \mathcal{A}ct^*$ and activities $A, B \in \mathcal{A}ct$ such that A and B are ratio-independent in $\bar{\upsilon}(c)$.*

Definition 17 (Prefix equivalence)**.** *Prefixes $\rho[..m]$ and $\sigma[..m]$ of runs ρ and σ starting in configuration c are equivalent in configuration c, denoted $\rho[..m] \equiv_c \sigma[..m]$, iff $Act(\rho[..m]) \equiv_c Act(\sigma[..m])$ as defined in Def. 16.*

Throughput is defined as a limit on prefix ratios of infinite runs. To define equivalence of runs in terms of throughput, we need to consider equivalent run prefixes (in the sense of Def. 17) with a bounded difference in the number of activities following those prefixes. This bounded difference ensures that the resulting weight difference can be ignored in the limit.

Definition 18 (Run equivalence)**.** *Let ρ and σ be two runs in $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$. We define $\rho \succeq \sigma$ iff there exists a $d \in \mathbb{N}$ such that for all $n \geq 0$ it holds that $\rho \succeq_n^d \sigma$. We define $\rho \succeq_n^d \sigma$ iff there exists some $k \geq n$, run prefixes $\rho[..k]$ and $\hat{\rho}[..k]$ with $Act(\hat{\rho}[..k]) \equiv_{\hat{c}} Act(\rho[..k])$ such that $Act(\hat{\rho}[..k]) = Act(\sigma[..n]) \cdot \tau$ for some activity sequence τ, and $k - n \leq d$. Runs ρ and σ are equivalent, denoted $\rho \equiv \sigma$, iff $\rho \succeq \sigma$ and $\sigma \succeq \rho$.*

Two finite runs ρ and σ are equivalent if $\rho \succeq_n^d \sigma$ for $0 \leq n \leq |\rho|$ and $\sigma \succeq_n^d \rho$ for $0 \leq n \leq |\sigma|$, where $d = max(|\sigma|, |\rho|)$. As an example, consider run ρ with activity sequence $(ABCD)^\omega$ in the full state space. We want to construct an equivalent run σ in the reduced state space, say $(CABD)^\omega$. This run is equivalent because C is ratio-independent of A and B. The run should satisfy $\rho \succeq_n^d \sigma$ for some $d \in \mathbb{N}$ and for all $n \geq 0$. Consider the case for $n = 2$, shown in Fig. 7a. Then, we need to match two (n) activities



(a) Example $\rho \succeq_2^1 \sigma$. (b) Illustration of $\rho \succeq_n^d \sigma$.

Fig. 7: Illustration of Def. 18.

of ρ to the first two activities of σ in the reduced space. The prefix consisting of the first two activities of ρ, AB, is not a valid run fragment after reduction. In the reduced space the independent activity C must be performed first. Run $\hat{\rho}$, equivalent (in the sense of Def. 17) to ρ and identical to σ for $n = 2$ activities and $d = 1$, can be constructed by moving C to the front. ρ and σ must be such that this can be done for any $n \geq 0$. The general case is shown in Fig. 7b. It is crucial for the preservation of throughput that the length k that one needs to consider in ρ to find the first n activities of σ exceeds n by maximally a finite amount d, independent of n.

The following two theorems show that equivalent runs indeed have the same throughput and latency values.

Theorem 19 (Equivalent runs have the same throughput)**.** *Let $\rho, \sigma \in \mathcal{R}(\mathcal{S})$ be two runs in $\mathcal{S}$. If $\rho \equiv \sigma$, then $\mathit{Ratio}(\rho) = \mathit{Ratio}(\sigma)$.*

Theorem 20 (Equivalent runs have the same latency)**.** *Let $\rho, \sigma \in \mathcal{R}(\mathcal{S})$ be two runs in $\mathcal{S}$. Let A_{src} and A_{snk} be any source-sink pair, and let r be the resource for which we want to calculate the start-to-start latency. If $\rho \equiv \sigma$, then $\lambda_{max}(\rho, A_{src}, A_{snk}, r) = \lambda_{max}(\sigma, A_{src}, A_{snk}, r)$.*

To reduce the (max,+) state space, we introduce a reduction function and state the conditions that should be imposed on this reduction to ensure that for each run in the full (max,+) state space there exists an equivalent run in the reduced (max,+) state space. We refer to these conditions as *ample conditions* and call a reduction that satisfies the ample conditions an *ample reduction*, as in [5].

Definition 21 ((max,+) state-space reduction)**.** *A reduction function reduce for a (max,+) state space $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$ is a mapping from C to $2^{\mathcal{A}ct}$ such that $reduce(c) \subseteq enabled(c)$ for each configuration $c \in C$. We define the reduction of $\mathcal{S}$ induced by reduce as the smallest (max,+) state space $\mathcal{S}' = \langle C', \hat{c}', \mathcal{A}ct', \Delta', M', w_1', w_2' \rangle$ that satisfies the following conditions:*

- *$C' \subseteq C$, $\hat{c}' = \hat{c}$, $\mathcal{A}ct' = \mathcal{A}ct$, $\Delta' \subseteq \Delta$, $M' = M$;*
- *for every $c \in C'$ and $A \in reduce(c)$, $(c, A, A(c)) \in \Delta'$, $w_1'(c, A, A(c)) = w_1(c, A, A(c))$, and $w_2'(c, A, A(c)) = w_2(c, A, A(c))$.*

Definition 22 ((max,+) state-space ample reduction)**.** *Let $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$ be a (max,+) state space. A reduction function ample is an ample reduction if it satisfies the following conditions in each configuration c:*

(R1) Non-emptyness condition: *if* $enabled(c) \neq \emptyset$, *then* $ample(c) \neq \emptyset$.

(R2) Ratio-dependency condition: *For any configuration* $c_0 \in C'$ *and run* $c_0A_1c_1A_2 \dots A_mc_m$ *with* $m \geq 1$ *in* $\mathcal{S}$, *if activity* A_m *and some activity in* $ample(c_0)$ *are ratio dependent in* c_0, *then there is an index* i *with* $1 \leq i \leq m$ *with* $A_i \in ample(c_0)$.

We refer to $ample(c)$ *as an* ample set.

Condition (R1) ensures that the reduction does not introduce new deadlocks. Condition (R2) implies that starting from some configuration c_i, any activity in $ample(c_i)$ remains enabled as long as no activity in $ample(c_i)$ has been executed. To illustrate, consider configuration $c_0 = \langle s1, \mathbf{0} \rangle$ in the state space shown in Fig. 6. Activities A and B are ratio dependent in c_0 (they do not satisfy condition 1 in Def. 15) and ratio independent with activity C. Ample sets $\{A, B\}$ and $\{C\}$ both satisfy conditions (R1) and (R2).

An ample reduction ensures that for each run in a given normalized (max,+) state space $\mathcal{S}$ we can find an equivalent run in the reduced (max,+) state space $\mathcal{S}'$. By Theorems 19 and 20, it follows that such a reduction thereby also preserves throughput and latency aspects in the reduced (max,+) state space, and by Def. 14, $\mathcal{S} \approx_p \mathcal{S}'$.

Theorem 23 (Equivalent runs)**.** *Let* $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2 \rangle$ *be a finite normalized (max,+) state space, and* $\mathcal{S}'$ *be a reduced (max,+) state space induced by an ample reduction function. Then for each run* $\rho \in \mathcal{R}(\mathcal{S})$, *there exists a run* $\sigma \in \mathcal{R}(\mathcal{S}')$ *with* $\rho \equiv \sigma$.

V. (max,+) Automaton Reduction

In the previous section, we described the notion of ratio-independence and the ample conditions to preserve the performance aspects of the supervisor in the corresponding normalized (max,+) state space. In this section, we lift these conditions to the level of a (max,+) automaton, so that we do not have to first compute the full (max,+) state space. We now also consider the functional aspects.

To reuse existing POR techniques, we want to remove the need to identify the marked states with special set S^m. Instead, we add a self-loop with a special controllable activity ω to indicate marked states.

Definition 24. *Let* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T \rangle$ *be a (max,+) automaton. The* ω*-extension of* $\mathcal{A}$ *is the automaton* $\mathcal{A}_\omega = \langle S_\omega, \hat{s}_\omega, S^m_\omega, \mathcal{A}ct_\omega, reward_\omega, M_\omega, T_\omega \rangle$ *with* $\omega \notin \mathcal{A}ct$, $\mathcal{A}ct_\omega = \mathcal{A}ct \cup \{\omega\}$, $reward_\omega = reward \cup \{\langle \omega, 0 \rangle\}$, $M_\omega = M \cup \{\langle \omega, M(\omega) \rangle\}$, *where* $M(\omega)$ *is an identity matrix of size* $|Res|^2$ *with for each* $1 \leq i, j \leq |Res|$, $[M(\omega)]_{ij} = -\infty$ *if* $i \neq j$ *and* $[M(\omega)]_{ij} = 0$ *if* $i = j$, *and* $T_\omega = T \cup \{s^m \xrightarrow{\omega} s^m \mid s^m \in S^m\}$.

We can then replace nonblockingness of $\mathcal{A}$ by the notion of ω-reachability of $\mathcal{A}_\omega$, that uses the ω-self-loops instead of the marked state labeling.

Definition 25. *Let* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T \rangle$ *be a (max,+) automaton.* $\mathcal{A}$ *is* ω-reachable *iff from each reachable state* $s \in S$ *(i.e.* $\hat{s} \rightarrow^* s$*) a state* s_ω *is reachable in which* ω *is enabled.*

Proposition 26. *Let* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T \rangle$ *be a (max,+) automaton and let* $\mathcal{A}_\omega = \langle S_\omega, \hat{s}_\omega, S^m_\omega, \mathcal{A}ct_\omega, reward_\omega, M_\omega, T_\omega \rangle$ *be the* ω*-extension of* $\mathcal{A}$ *with* $\omega \notin \mathcal{A}ct$. *Then* $\mathcal{A}$ *is nonblocking if and only if* $\mathcal{A}_\omega$ *is* ω*-reachable.*

In the plant $\mathcal{P}$, there can be multiple paths that are nonblocking and controllable, and multiple runs with the same throughput and latency values. Part of this redundancy is caused by the interleaving of activities that have no mutual influence on these properties. Such activities are referred to as *uncontrollable-independent*. For two activities A and B to be uncontrollable-independent, we first require that no uncontrollable activity is enabled in $s, A(s), B(s)$, and $AB(s)$. This property ensures that there is no influence of uncontrollable activities on both activities. Second, we also require that activities A and B do not share resources. This guarantees that the activities have no mutual influence on their timing behavior.

Definition 27 (Uncontrollable-independence)**.** *Given (max,+) automaton* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T \rangle$ *and state* $s \in S$, *activities* $A, B \in enabled(s)$ *are* uncontrollable-independent *in* s *iff they satisfy the following conditions:*

- $B \in enabled(A(s))$ *and* $A \in enabled(B(s))$;
- $AB(s) = BA(s)$;
- $R(A) \cap R(B) = \emptyset$;
- $enabled(s) \cap \mathcal{A}ct_u = enabled(A(s)) \cap \mathcal{A}ct_u = enabled(B(s)) \cap \mathcal{A}ct_u = enabled(AB(s)) \cap \mathcal{A}ct_u = \emptyset$

Two activities are uncontrollable-dependent *in* s, *iff they are not uncontrollable-independent in* s.

As an example, consider state s_1 in $\mathcal{P}$ shown in Fig. 2e. Here, activities A and B are uncontrollable-dependent since they both use resources R_1 and R_2 (as shown in Fig. 3). They are uncontrollable-independent with activity C, since they do not share a resource, are independent, and do not enable an uncontrollable activity.

The uncontrollable-independence lifts the ratio-independence notion of the previous section to the level of a (max,+) automaton. If two activities A and B are uncontrollable independent in some state in the (max,+) automaton, then they are also ratio independent in the corresponding configurations in the underlying state space. Because $R(A) \cap R(B) = \emptyset$, their corresponding (max,+) matrices commute. As a result, the resulting normalized vector after multiplication is the same, and the sum of the weights w_1 and w_2 is the same, independent of the execution order.

Theorem 28. *Given are a (max,+) automaton* $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T \rangle$ *and a state* $s \in S$ *with activities* $A, B \in enabled(s)$. *Consider any configuration* $c = \langle s, \boldsymbol{\gamma} \rangle$ *in the underlying normalized (max,+) state space. If* A *and* B *are uncontrollable-(in)dependent in* s, *then they are ratio-(in)dependent in* c.

Two runs in a (max,+) automaton are equivalent iff they can be obtained from each other by repeatedly commuting adjacent uncontrollable-independent activities.

Definition 29. *Let $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ be a (max,+) automaton. Strings $\alpha, \beta \in \mathcal{A}ct^*$ are equivalent [23] in a state s, denoted $\alpha \equiv_{s,u} \beta$, iff there exists a list of strings $\upsilon_0, \upsilon_1, \ldots \upsilon_n$, where $\upsilon_0 = \alpha$, $\upsilon_n = \beta$, and for each $0 \leq i < n$, $\upsilon_i = \bar{\upsilon}AB\hat{\upsilon}$ and $\upsilon_{i+1} = \bar{\upsilon}BA\hat{\upsilon}$ for some $\bar{\upsilon}, \hat{\upsilon} \in \mathcal{A}ct^*$ and activities $A, B \in \mathcal{A}ct$ such that A and B are uncontrollable-independent in $\bar{\upsilon}(s)$.*

A reduction function on a (max,+) automaton is defined in the following way.

Definition 30 ((max,+) automaton reduction function)**.** *A reduction function reduce for a (max,+) automaton $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ is a mapping from S to $2^{\mathcal{A}ct}$ such that $reduce(s) \subseteq enabled(s)$ for each state $s \in S$. We define the reduction of $\mathcal{A}$ induced by reduce as the smallest (max,+) automaton $\mathcal{A}' = \langle S', \hat{s}', S^{m'}, \mathcal{A}ct', reward', M', T'\rangle$ that satisfies the following conditions:*

- *$S' \subseteq S,\ \hat{s}' = \hat{s},\ S^{m'} = S^m \cap S'\ \mathcal{A}ct' = \mathcal{A}ct,\ T' \subseteq T$;*
- *for every $s \in S'$ and $A \in reduce(s)$, $\langle s, A, A(s)\rangle \in T'$, $reward'(A) = reward(A)$, and $M'(A) = M(A)$.*

By applying a reduction on plant $\mathcal{P}$, we get a reduced plant $\mathcal{P}'$, as is shown in Fig. 2e. The supervisor $\mathcal{A}'_{sup}$ that is synthesized for this reduced model will typically not be least-restrictive with respect to $\mathcal{P}$. Therefore, we introduce a new notion of *reduced least-restrictiveness*, that defines that an equivalent run in $\mathcal{P}'$ is preserved for each run in $\mathcal{P}$ to a marked state.

Definition 31 (Reduced least-restrictiveness)**.** *Let $\mathcal{A}_1 = \langle S_1, \hat{s}, S_1^m, \mathcal{A}ct_1, reward_1, M_1, T_1\rangle$ and $\mathcal{A}_2 = \langle S_2, \hat{s}, S_2^m, \mathcal{A}ct_2, reward_2, M_2, T_2\rangle$ be two (max,+) automata such that $\mathcal{A}_1 \preceq \mathcal{A}_2$. $\mathcal{A}_1$ is reduced least-restrictive with respect to $\mathcal{A}_2$ iff for every path $\hat{s} \xrightarrow{\alpha}{}^*_{\mathcal{A}_2} s_m$ with $s_m \in S_2^m$ there exists a path $\hat{s} \xrightarrow{\beta}{}^*_{\mathcal{A}_1} s_m$ in $\mathcal{A}_1$ with $s_m \in S_1^m$ such that $\alpha \equiv_{\hat{s},u} \beta$.*

In our running example, plant $\mathcal{P}'$ is reduced least-restrictive with respect to $\mathcal{P}$, since it preserves a representative run to the marked state s_8.

In the reduction of $\mathcal{P}$ to $\mathcal{P}'$, we want to preserve controllability, nonblockingness, and reduced least-restrictiveness with respect to the full plant $\mathcal{P}$, as well as throughput and latency. After synthesis, we have supervisors $\mathcal{A}_{sup} = sup\mathcal{CN}(\mathcal{P})$ and $\mathcal{A}'_{sup} = sup\mathcal{CN}(\mathcal{P}')$. Reduced supervisor $\mathcal{A}'_{sup}$ should preserve the functional aspects and performance aspects of supervisor $\mathcal{A}_{sup}$, defined by $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$.

Definition 32. *Let $\mathcal{P}$ be a plant and $\mathcal{A}'_{sup}$ and $\mathcal{A}_{sup}$ be supervisors for $\mathcal{P}$. Let $\mathcal{S}'$ and $\mathcal{S}$ be the corresponding normalized (max,+) state spaces of $\mathcal{A}'_{sup}$ and $\mathcal{A}_{sup}$. We define $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$ if and only if*

1) *if $\mathcal{A}_{sup}$ is nonblocking then $\mathcal{A}'_{sup}$ is nonblocking,*
2) *if $\mathcal{A}_{sup}$ is controllable w.r.t. $\mathcal{P}$ then $\mathcal{A}'_{sup}$ is controllable with respect to $\mathcal{P}$,*
3) *$\mathcal{A}'_{sup}$ is reduced least-restrictive w.r.t. $\mathcal{A}_{sup}$, and*
4) *$\mathcal{S}' \approx_p \mathcal{S}$.*

In this definition, conditions 1,2, and 3 guarantee that the functional aspects are preserved. Condition 4 ensures that the timed state spaces have the same throughput and latency values. Note that condition 3 does not automatically imply condition 4, since throughput is defined over infinite runs (going through simple cycles), and there can be cycles in the state space where none of the corresponding states are marked. To preserve the properties of interest, we impose the following ample conditions on a reduction function of a (max,+) automaton.

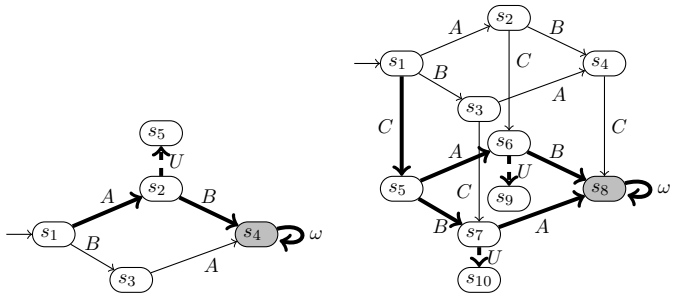
Definition 33 (Ample conditions (max,+) automaton)**.** *A reduction function ample is an ample reduction if it satisfies all the following conditions in each state s:*

- ***(A1) Non-emptyness condition:*** *if $enabled(s) \neq \emptyset$, then $ample(s) \neq \emptyset$.*
- ***(A2) Dependency condition:*** *For any run $s_0 A_1 s_1 A_2 \ldots A_m s_m$ with $s = s_0$ and $m \geq 1$ in $\mathcal{A}$, if activity A_m and some activity in $ample(s_0)$ are uncontrollable dependent in s_0, then there is an index i with $1 \leq i \leq m$ with $A_i \in ample(s_0)$;*
- ***(A3) Controllability condition:*** *$ample(s) \supseteq enabled(s) \cap \mathcal{A}ct_u$;*
- ***(A4) Nonblockingness condition:*** *$ample(s) \supseteq enabled(s) \cap \{\omega\}$;*
- ***(A5.1) Synthesis condition 1:*** *If $A \in ample(s)$ and $enabled(A(s)) \cap \mathcal{A}ct_u \neq \emptyset$ then $ample(s) = enabled(s)$;*
- ***(A5.2) Synthesis condition 2:*** *For $A, B \in enabled(s)$ that are uncontrollable independent in s if $enabled(AB(s)) \cap \mathcal{A}ct_u \neq \emptyset$, then $A \in ample(s) \Leftrightarrow B \in ample(s)$.*

We refer to set $ample(s)$ for any state s as an ample set.

Condition (A1) ensures that deadlock-freedom is preserved. Condition (A2) ensures that starting from some state s, any activity in $ample(s)$ remains enabled as long as no activity in $ample(s)$ has been executed. Condition (A3) ensures that all uncontrollable activities in the enabled set remain in the ample set to avoid that they become disabled by the reduction. Condition (A4) ensures that in the reduced automaton the marked states can still be identified. Note that ω behaves like an uncontrollable activity, since it must remain in the ample set. We have chosen it to be a controllable activity however, since we do not want ω to have an impact on conditions (A5.1) and (A5.2), and subsequently on the reductions that can be achieved. Condition (A2) ensures that an equivalent path to a marked state, where ω is enabled, is preserved in the reduced plant. Condition (A5.1) and (A5.2) ensure that if a path to a marked state is present in the supervisor, then an equivalent path is also present in the reduced supervisor.

We use these conditions rather than the weaker dependency relation, because the conditions can be checked efficiently using local information as we will see in the next section. To check the weaker dependency relation, we would need to explore states $s, A(s), B(s)$, and $AB(s)$ and compute the enabled set in each of these states to check whether there are uncontrollable activities enabled.



(a) Plant $\mathcal{P}_1$ with reduced plant $\mathcal{P}_1'$ that satisfies all conditions except condition (A5.1).

(b) Plant $\mathcal{P}_2$ with reduced plant $\mathcal{P}_2'$ that satisfies all conditions except condition (A5.2).

Fig. 8: Necessity of conditions (A5.1) and (A5.2).

To illustrate the need for conditions (A5.1) and (A5.2), consider the plants shown in Fig. 8. Fig. 8a shows plant $\mathcal{P}_1$, where the reduction satisfies conditions (A1) till (A4) and (A5.2), but not (A5.1). Condition (A5.1) is not satisfied, since uncontrollable activity U is enabled after A, but $ample(s_1) \neq enabled(s_1)$. Synthesis on $\mathcal{P}_1'$ yields an empty supervisor, whereas synthesis on $\mathcal{P}_1$ yields a supervisor with path $s \xrightarrow{BA} s_4$ to marked state s_4. If condition (A5.1) is satisfied, both A and B are in the ample set of s_1, since state s_2 might become a blocking state, and the alternative path to s_4 is preserved. To illustrate the need for condition (A5.2) consider plant $\mathcal{P}_2$ shown in Fig. 8b. Here, the reduction yielding $\mathcal{P}_2'$ satisfies conditions (A1) till (A5.1), but not condition (A5.2); activity $U \in enabled(BC(s_1))$, but only C is present in the reduction, and not B. The result after synthesis on $\mathcal{P}_2$ contains the paths $s_1 \xrightarrow{A} s_2 \xrightarrow{B} s_4 \xrightarrow{C} s_8 \xrightarrow{\omega} s_8$ and $s_1 \xrightarrow{B} s_3 \xrightarrow{A} s_4 \xrightarrow{C} s_8 \xrightarrow{\omega} s_8$, whereas synthesis on the reduced plant $\mathcal{P}_2'$ yields an empty supervisor since states s_9 and s_{10} are not marked.

Given reduction function $ample_{\mathcal{A}}$ on a (max,+) automaton $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$, we define reduction function $ample_{\mathcal{S}}$ on the corresponding (max,+) state space $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2\rangle$ in the following way: for any $c \in C$ with $c = \langle s, \gamma\rangle$ and $s \in S$, define $ample_{\mathcal{S}}(c) = ample_{\mathcal{A}}(s)$. The following theorem shows that an ample reduction at (max,+) automaton level is also an ample reduction at (max,+) state space level, thereby preserving latency and throughput. This follows from the fact that conditions (A1) and (A2) on the (max,+) automaton imply (R1) and (R2) in the reduction of the underlying state space.

Theorem 34. *Let $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ be a (max,+) automaton with corresponding state space $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2\rangle$. Let $ample_{\mathcal{A}}$ be an ample reduction on $\mathcal{A}$ that yields $\mathcal{A}'$ with corresponding state space $\mathcal{S}'$. Let $ample_{\mathcal{S}}$ be the ample reduction corresponding to $ample_{\mathcal{A}}$ on $\mathcal{S}$ that yields state space $\hat{\mathcal{S}}$, then $\hat{\mathcal{S}} = \mathcal{S}'$.*

The ample reduction of plant $\mathcal{P}$ yields a reduced plant $\mathcal{P}'$, such that synthesis on both plants yields supervisors $\mathcal{A}_{sup}$ and $\mathcal{A}'_{sup}$ respectively, such that $\mathcal{A}'_{sup}$ preserves functionality and performance, i.e. $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$.

Theorem 35 (Ample reduction preserves functionality and performance)**.** *Let $\mathcal{P}$ be a plant, and $\mathcal{P}'$ the reduced plant obtained by an ample reduction that satisfies Def. 33. Let $\mathcal{A}_{sup}$ be the supervisor with $\mathcal{A}_{sup} = sup\mathcal{CN}(\mathcal{P})$, and $\mathcal{A}'_{sup}$ be the reduced supervisor with $\mathcal{A}'_{sup} = sup\mathcal{CN}(\mathcal{P}')$. Then $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$.*

VI. On-the-fly reduction

Section V gives sufficient conditions on a reduction function on a single (max,+) automaton to preserve functional properties as well as performance properties in the corresponding normalized (max,+) state space. For an efficient reduction, we avoid first computing the full composition of the (max,+) automata. Rather, we use sufficient local conditions on the network of (max,+) automata specifying a system and its requirements to compute a reduced composition on-the-fly.

Given (max,+) timed system $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ and $\mathcal{A} = \{\mathcal{A}_i \mid 1 \leq i \leq n\}$, the ample function selects a set $ample(s)$ in each state s of the composition, such that conditions (A1) till (A5) are met. The ample set is induced by a *cluster* $\mathcal{C} \subseteq \mathcal{A}$, and computed as $ample(s) = enabled_{\mathcal{C}}(s) = enabled(s) \cap \mathcal{A}ct(\mathcal{C})$, where $\mathcal{A}ct(\mathcal{C}) = \bigcup_{\mathcal{A}_i \in \mathcal{C}} \mathcal{A}ct_i$ denotes the set of activities that occur in $\mathcal{C}$ and $\mathcal{A}ct_i$ is the alphabet of (max,+) automaton $\mathcal{A}_i$. A cluster that ensures that the ample conditions are met is called a *safe cluster*. This cluster-inspired ample approach, based on [24], is a generalization of the traditional on-the-fly method of Peled [5] that selects the enabled activities $enabled_i(s) = enabled(s) \cap \mathcal{A}ct_i$ of one (max,+) automaton $\mathcal{A}_i$ as ample set if possible, while exploring a state $s = \langle s_1, s_2, \ldots, s_n\rangle$. Otherwise, all enabled activities in s are selected as ample set. In an experimental evaluation, we found that this approach did not yield any reduction on most of the models described in Section VII. Therefore, we consider the generalization of the approach to clusters. We need some additional notation to define a safe cluster. First, we define a projection to consider the local state $\pi_{\mathcal{C}}(s)$ in a cluster $\mathcal{C} \subseteq \mathcal{A}$:

$$\pi_{\mathcal{A}_i}(s) = s_i$$
$$\pi_{\mathcal{C}}(s) = \langle \pi_{\mathcal{A}_{c_1}}(s), ..., \pi_{\mathcal{A}_{c_k}}(s)\rangle \text{ where } \mathcal{C} = \{\mathcal{A}_{c_1}, ..., \mathcal{A}_{c_k}\} \text{ and } c_j < c_{j+1} \text{ for all } 1 \leq j < k.$$

Given local state $\pi_{\mathcal{C}}(s)$, $enabled(\pi_{\mathcal{C}}(s))$ denotes the set of enabled activities in the composition of precisely the (max,+) automata in $\mathcal{C}$. Note that $enabled_{\mathcal{C}}(s) \subseteq enabled(\pi_{\mathcal{C}}(s))$, since the latter might contain activities that are enabled in the local state of the cluster-composition, but disabled in the global composition due to a (max,+) automaton outside the cluster disabling the activity. We only consider independence of activities across (max,+) automata, and not within the same (max,+) automaton. The former can be checked locally, whereas the latter requires an exploration on the internal transition structure. We treat activities inside the same (max,+) automaton as dependent.

Definition 36 (Cluster safety)**.** *Let $\mathcal{C} \subseteq \mathcal{A}$ be any cluster, and s be a state in the composition of $\mathcal{A}$. Cluster $\mathcal{C}$ is safe in s if the following conditions are satisfied.*

- ***(C1)*** *if $enabled(s) \neq \emptyset$, then $enabled_{\mathcal{C}}(s) \neq \emptyset$;*
- ***(C2.1)*** *for any $A \in enabled_{\mathcal{C}}(s)$ and $B \in \mathcal{A}ct(\mathcal{A}) \setminus \mathcal{A}ct(\mathcal{C})$, A and B are uncontrollable-independent activities;*
- ***(C2.2)*** *for any $A \in enabled(\pi_{\mathcal{C}}(s))$, if $A \in \mathcal{A}ct_i$ then $\mathcal{A}_i \in \mathcal{C}$;*
- ***(C3)*** *$enabled_{\mathcal{C}}(s) \supseteq enabled(s) \cap \mathcal{A}ct_u$;*
- ***(C4)*** *$enabled_{\mathcal{C}}(s) \supseteq enabled(s) \cap \{\omega\}$;*
- ***(C5.1)*** *if $A \in enabled_{\mathcal{C}}(s)$ and $enabled(A(s)) \cap \mathcal{A}ct_u \neq \emptyset$ then $enabled_{\mathcal{C}}(s) = enabled(s)$;*
- ***(C5.2)*** *for $A, B \in enabled(s)$ if $enabled(AB(s)) \cap \mathcal{A}ct_u \neq \emptyset$, then $A \in enabled_{\mathcal{C}}(s) \Leftrightarrow B \in enabled_{\mathcal{C}}(s)$.*

Condition (C2.1) requires that each enabled activity in $enabled_{\mathcal{C}}(s)$ is independent in s with any activity outside $\mathcal{A}ct(\mathcal{C})$. Condition (C2.2) requires that each activity in $enabled(\pi_{\mathcal{C}}(s))$ does not occur outside of the cluster. Together, these two conditions ensure that no activity $A \in \mathcal{A}ct(\mathcal{A}) \setminus enabled_{\mathcal{C}}(s)$, dependent on some activity in $enabled_{\mathcal{C}}(s)$, becomes enabled by executing only activities outside the cluster. Condition (C3) ensures that uncontrollable activities are always preserved. Condition (C4) ensures that ω, if enabled, is preserved. Conditions (C5.1) and (C5.2) ensure that if a path to a marked state remains after synthesis on the full composition, then also an equivalent path remains after synthesis on the reduced composition. Note that $\mathcal{C} = \mathcal{A}$ is a safe cluster in any state. We define a cluster-inspired ample reduction based on cluster safety on a (max,+) timed system.

Definition 37 (Cluster-inspired ample reduction)**.** *A cluster-inspired ample reduction ample for a (max,+) timed system $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ is a mapping from $S = S_1 \times \ldots \times S_n$ to $2^{\mathcal{A}ct}$ such that $ample(s) = enabled_{\mathcal{C}}(s)$ for some cluster $\mathcal{C} \subseteq \mathcal{A}$, and satisfies the following condition:*

- ***(M) Cluster-safety condition:*** *for any state s, $ample(s) = enabled_{\mathcal{C}}(s)$ where $\mathcal{C}$ is safe in s.*

Theorem 38. *Let ample be a cluster-inspired ample reduction on $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$. Then ample satisfies conditions (A1), (A2), (A3), (A4), (A5.1), and (A5.2).*

In each state in the composition, we compute a safe cluster starting from a candidate activity. To check whether an activity might enable an uncontrollable activity, we introduce a new set $\mathcal{U}$, defined as follows:

$$\mathcal{U} = \bigcup_{\mathcal{A}_i \in \mathcal{A}} \{A \in \mathcal{A}ct_i \mid U \in enabled(A(s)) \cap \mathcal{A}ct_u, s \in S^i\}.$$

This set can be generated a priori from the network of (max,+) automata. The reduction will be most effective if this set $\mathcal{U}$ is small, and not effective if it contains all activities. After executing some activity $A \in \mathcal{U}$, an uncontrollable activity U might still be disabled by some other (max,+) automaton, even though A enables it locally.

Algorithm 1 shows the algorithm to compute a safe cluster in a state s. The algorithm starts in lines 2-6 with ensuring that conditions (C3) and (C4) are met. If $enabled(s)$ contains ω, then the algorithm can immediately return set $\mathcal{A}$, since all (max,+) automata have ω in their alphabet and will be added to the cluster. If not, then for each enabled uncontrollable activity, all (max,+) automata that have this activity in the alphabet are added to the cluster (lines 5-6). The remainder of the algorithm checks for each activity enabled in the current cluster $\mathcal{C}$ whether condition (C2.1) or (C2.2) is violated. The algorithm starts with initial *candidate* activity A. If A is enabled in the composition (line 9), we add all (max,+) automata that contain A (line 12) and add a (max,+) automaton for each dependent activity outside the current cluster (lines 13-15). This ensures that condition (C2.1) is satisfied for activity A and the cluster obtained after executing lines 9-15. If $A \in \mathcal{U}$, then A might enable an uncontrollable activity, and condition (C5.1) or (C5.2) might get violated. When $A \in \mathcal{U}$ and A is enabled within the current cluster, the full enabled set is returned as ample set by the algorithm (line 11). If A is enabled in the composition of (max,+) automata in

Algorithm 1 Algorithm to compute a safe cluster.

```
1:  proc ComputeCluster(s, candidate)
2:    A ← candidate; C ← ∅; processed ← ∅
3:    if ω ∈ enabled(s) then
4:      return A
5:    for U ∈ enabled(s) ∩ Act_u do
6:      C ← C ∪ {A_i | U ∈ Act_i}
7:    while A ≠ ⊥ do
8:      processed ← processed ∪ {A}
9:      if A ∈ enabled(s) then
10:       if A ∈ U then
11:         return A
12:       C ← C ∪ {A_i | A ∈ Act_i}
13:       for B ∈ {D ∈ Act | R(D) ∩ R(A) ≠ ∅} do
14:         if B ∉ Act(C) then
15:           C ← C ∪ {[A_i | B ∈ Act_i].first()}
16:     if A ∉ enabled(s) ∧ A ∈ enabled(π_C(s)) then
17:       for A_i ∈ A \ C do
18:         if A ∈ Act_i ∧ A_i ∧ A ∉ enabled(s_i) then
19:           C ← C ∪ {A_i} ; break
20:     if processed ≠ enabled(π_C(s)) then
21:       A ← [enabled(π_C(s)) \ processed].first()
22:     else
23:       A ← ⊥
24:   return C
```

Algorithm 2 Algorithm to select a candidate activity.

```
1:  proc SelectCandidate(s, A)
2:    if ω ∈ enabled(s) then
3:      return ω
4:    else if Act_u ∩ enabled(s) ≠ ∅ then
5:      return [Act_u ∩ enabled(s)].first()
6:    else
7:      return min_{A∈enabled(s)} GetWeight(A, s, A)

1:  proc GetWeight(A, s, A)
2:    w ← 0
3:    for A_i ∈ A do
4:      if A ∈ Act_i then
5:        for B ∈ enabled(s_i) do
6:          if B ∈ enabled(s) then
7:            w ← w + |A| · |Act|
8:          else
9:            w ← w + 1
10:   return w
```

the cluster, but not in the full composition, then we add a (max,+) automaton that causes A to be disabled in the full composition. This ensures that condition (C2.2) is satisfied for A for the cluster obtained after executing lines 16-19. Notation $[\mathcal{A}_i \mid B \in Act_i]$ denotes a list comprehension that creates a list of all elements $\mathcal{A}_i$ for which $B \in Act_i$. Function $first()$ picks the first element from a list. After handling the activity, we check whether there are other activities that are locally enabled in the new cluster and not yet processed (line 20-23). The algorithm continues until all locally enabled activities are processed.

Theorem 39. *Let s be a state in the composition $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ and $A \in enabled(s)$ be the candidate activity. Then,* ComputeCluster(s, A) *returns a safe cluster in s.*

In each state in the composition, there are often multiple valid safe clusters. Heuristics can be used to select a cluster that likely leads to a large reduction. One approach is to select a safe cluster that yields the smallest ample set in each state. A safe cluster is computed starting from each candidate activity in the enabled set, and then the smallest cluster is selected. This heuristic often performs well [25], since it allows to prune most enabled transitions. A disadvantage is that a safe cluster needs to be constructed starting from each candidate. When the enabled sets become larger, this leads to a significant runtime overhead due to the computations of the enabled sets at each iteration of the algorithm. Algorithm 2 is an alternative approach, that selects only one activity as a candidate using a heuristic. First, it checks whether $enabled(s)$ contains ω or an uncontrollable activity. Since they will always be in the ample set, they are a good candidate choice. Otherwise, an activity is selected that doesn't occur in (max,+) automata that have locally enabled activities that are also in $enabled(s)$, since this implies that the ample set will increase. We add weight $|\mathcal{A}| \cdot |\mathcal{A}ct|$ since this is the maximum total sum of locally enabled activities. In a similar way we want to minimize the number of other enabled activities within the cluster, since possibly (max,+) automata need to be added where these are not enabled. To find the best candidate, a weight is computed for each activity in $enabled(s)$ based on these two properties, and the activity with the minimum weight is selected.

VII. Experimental evaluation

To test the effectiveness of the on-the-fly reduction, we use a set of models without data variables available from Supremica [26], the ASML lithography scanner model described in [27], and four variants of the Twilight system [27]. Twilight is an imaginary manufacturing system with two processing stations (conditioning, drilling) that processes balls according to a given recipe. This manufacturing system is a simplification of the ASML lithography scanner model using similar types of resources. The first variant (TW1) is the model described in [9]. Here, the life cycle and location of each product is explicitly modeled. In TW2, we remove these product-location and life-cycle automata, and instead use automata that ensure that products are always moved forward in the production process. In TW3, we extend TW2 with a polish station, where each product undergoes a polish and drill step after the condition step but in arbitrary order. To analyze the scalability of synthesis with POR, we also used a variant of TW3, TW3-10s, where the number of processing stations is increased to 10. In TW4, we fix the order so that a product is always first conditioned, then drilled, and then polished.

We use two heuristics to compute ample sets; *AllCandidates* (Algorithm 1) that tries all candidate activities to find the smallest ample set, and *SmartCandidate* (Algorithm 2) that selects one candidate activity. All experiments are performed with a 2.40GHz Intel i5-6300U CPU processor and with 4GB Java heap space to run the algorithms.

In the experimental evaluation, we consider the reductions that can be achieved while preserving only functional aspects, as well as preserving both functional and performance aspects. In the Supremica models Supremica, there is no accompanying specification describing the resources used by each activity. Therefore, we assume that activities do not share resources, effectively preserving only functional aspects. For the Twilight models and the ASML model, we do have the activity matrices, and therefore also consider the reductions preserving performance aspects. Performance aspects can be ignored by setting $R(A) = \emptyset$ for each activity $A \in \mathcal{A}ct$. In the remainder of this section, we give the results for the two classes of experiments.

a) Reduction preserving only functional aspects: Before applying POR, we compute set $\mathcal{U}$ to check whether a reduction is possible. For these models in our test set, $\mathcal{U}$ contains all activities and not reductions are possible: DosingTank, MachineBufferMachine, TankProcess, AutomaticCarParkGate, TransferLine, and TransferLine3. For the Twilight models and the ASML model, we disregard the performance aspects using the approach described above. Table I shows the results of the on-the-fly reduction where reductions are possible. The highest reductions are achieved in VolvoCell and RobotAssemblyCell, where all activities are controllable and plants describe local parts of the system. This leaves a lot of redundant interleavings of activities that can be exploited to obtain a smaller composition. A similar reasoning is applicable to TW2, TW3, and TW4. The reduction for TW1 is very small, because there is a lot of activity synchronization by the product-location and life-cycle automata. Recall condition (C2.2), that requires that each enabled activity in the local state of a safe cluster must be independent with activities outside the cluster. During state-space exploration of TW1, the algorithm often needs to add product-location or life-cycle automata to the cluster to satisfy this condition (C2.2); this limits reduction possibilities. The reduction for TW2, TW3, and TW4 is much larger, since we do not explicitly model the product-location and life-cycle automata. For TW3-10, we had to compute the full state space on a server with much more heap space. Therefore, we have no running times on the same hardware, indicated by an X, and cannot compare the running times. In the ASML model a large reduction of 86.1% is achieved, since

	full composition				*AllCandidates* heuristic				*SmartCandidate* heuristic			
	$\|S\|$	$\|T\|$	T_P (ms)	T_S (ms)	% of $\|S\|$	% of $\|T\|$	% of T_P	% of T_S	% of $\|S\|$	% of $\|T\|$	% of T_P	% of T_S
CircularTable	442	1357	52.5	31.6	72.9%	76.5%	155%	376%	72.9%	76.5%	84%	249%
IntertwinedProductCycles	65280	277441	6229.9	4746.6	0.0%	0.8%	216%	387%	0.0%	0.8%	131%	283%
RobotAssemblyCell	4741	21107	2071.2	660.6	93.4%	98.0%	10%	35%	90.6%	97.0%	3%	14%
VolvoCell	8871	29230	3919.0	518.4	83.9%	93.6%	169%	1287%	52.6%	73.8%	125%	974%
WeldingRobots	198	544	16.3	35.6	0.0%	7.0%	198%	194%	0.0%	5.7%	179%	180%
VelocityBalancing	372	775	22.0	40.8	9.1%	20.5%	155%	171%	9.1%	20.5%	131%	165%
TW1	1280	2171	322.0	61.0	0.5%	0.6%	1661%	9193%	0.5%	0.6%	819%	4440%
TW2	63	132	1.6	13.1	34.9%	50.0%	394%	148%	33.3%	47.0%	206%	122%
TW3	343	761	14.4	29.3	48.4%	67.0%	220%	162%	42.6%	59.0%	143%	130%
TW4	318	776	15.9	36.7	30.8%	47.8%	602%	327%	29.6%	42.9%	237%	174%
TW3-10s	1887381	10907298	X	X	**96.9%**	**98.9%**	X	X	**96.2%**	**98.6%**	X	X
ASML	2412	7662	174.9	394.6	86.1%	94.0%	82%	62%	65.5%	83.7%	64%	50%

TABLE I: Reductions achieved preserving functional aspects. $|S|$ is the number of states and $|T|$ is the number of transitions. The running times to compute the composition T_P and the supervisor T_S are in milliseconds. The highest reductions are highlighted in bold.

all activities in this model are controllable and requirements are formulated in a local fashion.

As expected, *AllCandidates* yields similar or better reductions than *SmartCandidate* in terms of states and transitions that remain in the composition by selecting the smallest safe cluster in each state. However, there is a significant runtime overhead in computing the reduced composition (T_P) with *AllCandidates* due to the computations involved in Algorithm 1. This overhead is much lower for *SmartCandidate*, because we run Algorithm 1 only once in each state in the composition. The additional runtime induced by the computations in *SmartCandidate* is in most cases a factor of 2. We also considered the total time T_S needed to apply supervisory controller synthesis. For the heuristics, T_S includes time T_P that is needed to compute the reduced composition. For *AllCandidates*, the median additional synthesis runtime overhead is a factor of 3.3, and for *SmartCandidate* it is a factor 2.3. Again, in almost all cases, the POR algorithm gives some runtime overhead. Note that in practice the bottleneck in synthesis is not the runtime, but the memory that is needed to store the composition. This means that for scalability, the most important metrics are the reductions that can be achieved in terms of the number of states and transitions.

b) Reduction preserving both functional and performance aspects: Table II shows the results of the reduction that preserves both functional and performance aspects. This reduction yields a larger composition, thus achieving less reduction, than the reduction preserving only functional aspects. This is as expected, since resource sharing between activities is also considered. The normalized (max,+) state spaces of the full ASML and full TW3-10s models could not be computed due to insufficient memory.

VIII. Related work

The application of POR techniques in the domain of supervisory control theory has been first investigated by Hellgren et al. [28]. There, POR is used to reduce the state space when checking deadlocks. A setting with only controllable actions is considered and the models adhere to a very specific structure, with resource booking/unbooking and product life cycles that contain no cycles and do not use the same resource multiple times. Shaw [29] introduces an on-the-fly model checking approach for both controllability and nonblockingness. Because the aim is model checking rather than synthesis, the conditions that are needed are different from the ones used in this paper. For example, for checking controllability, a reduction might remove uncontrollable events from a plant model that are independent with controllable events. This is not valid if one wants to apply synthesis in a subsequent step.

There has been some initial work in applying POR techniques to timed systems. Bengtsson et al. [30] apply POR on timed automata for reachability analysis. These automata execute asynchronously, in their own local time scale, and synchronize their time scales on communication transitions. Yoneda et al. [31] investigated POR for timed Petri nets, for verification of similar timing relations. Theelen et al. [32] apply ideas from POR on Scenario-Aware Data Flow models, using an independence relation among actions to resolve non-deterministic choices that have no impact on the performance metrics.

Our POR technique can be used to obtain a smaller supervisory controller for the given plant. There have been other approaches to construct a reduced nonblocking supervisor. Dietrich et al. [33] impose three sufficient conditions on a restricted supervisor to preserve nonblockingness. Morgenstern and Schneider [34] propose a stronger notion of nonblockingness called *forceable nonblockingness*. Given a plant, a controller is forceable nonblocking if every (in)finite run of the controlled system visits a marked state. This means that a marked state *will* be reached, no matter how the plant behaves, whereas in the original setting, the plant only *has the possibility* to reach a marked state. They provide a synthesis algorithm that directly computes such a controller. Huang and Kumar [35] describe an approach to generate a controller under the traditional notion of nonblockingness. Another approach is to find a smaller equivalent supervisor [36] that preserves maximal permissiveness. A supervisor typically has information about the enabling and disabling of events, and information to keep track of the plant evolution. The latter may contain

	automata composition				*AllCandidates* heuristic				*SmartCandidate* heuristic			
	$\|S\|$	$\|T\|$	T_P (ms)	T_S (ms)	% of $\|S\|$	% of $\|T\|$	% of T_P	% of T_S	% of $\|S\|$	% of $\|T\|$	% of T_P	% of T_S
TW1	1280	2171	234.5	54.8	0.5%	0.6%	2334%	10205%	0.5%	0.6%	1177%	5249%
TW2	63	132	1.3	11.4	33.3%	46.2%	1084%	527%	31.7%	44.7%	502%	295%
TW3	343	761	11.0	31.2	27.1%	39.8%	604%	340%	21.9%	32.2%	334%	274%
TW4	318	776	23.9	27.8	29.6%	44.3%	368%	487%	27.4%	39.6%	200%	339%
ASML	2412	7662	195.6	317.5	**80.4%**	**91.4%**	122%	108%	**39.6%**	**67.2%**	118%	127%

	normalized (max,+) state space		*AllCandidates* heuristic				*SmartCandidate* heuristic			
	$\|C\|$	$\|\Delta\|$	$\|C\|$	% of $\|C\|$	$\|\Delta\|$	% of $\|\Delta\|$	$\|C\|$	% of $\|C\|$	$\|\Delta\|$	% of $\|\Delta\|$
TW1	2967	5277	2959	5261	0.3%	0.3%	2959	5261	0.3%	0.3%
TW2	282	564	200	332	29.1%	41.1%	206	344	27.0%	39.0%
TW3	3282	7136	2169	3848	33.9%	46.1%	2266	4175	31.0%	41.5%
TW4	11637	26147	8018	14907	31.1%	43.0%	8283	16068	28.8%	38.5%
ASML	X	X	69659	X	97702	X	1022464	X	1690309	X

TABLE II: Reductions achieved preserving both functional and performance aspects. The running times to compute the composition T_P and the supervisor T_S are in milliseconds. The highest reductions are highlighted in bold.

redundancy. The technique exploits this redundancy to obtain a smaller supervisor. Compared to our approach, all approaches except the last one do not ensure a property of least-restrictiveness. Also, it is not straightforward to combine these reduction techniques with preserving other aspects of interest, such as performance-related properties. In our POR this is achieved by adding the sufficient conditions to the reduction function and adapting the notion of event-dependence used.

Su et al. [17] describe a related approach to compute a maximally-permissive supervisor that optimizes makespan. It does not consider latency and throughput preservation. Parallelism is encoded using a mutual exclusion function. In our approach, we encode the specific resource usage, and thereby the mutual exclusion on resources, in the activities. The evaluation in the approach of [17] relies on the construction of a tree automaton, which grows exponentially in size in the worst case. In our (max,+) state space, redundancy in subsequences with the same timing information is encoded more efficiently.

Supervisory control of (max,+) automata is also considered in [12], [13]. Here, the conventional (max,+) automaton definition is used, where the timing aspects are coupled to the system states. As a result, synthesis is performed on a model including timing information. In our approach, we abstract from the timing information during synthesis, which benefits scalability of the approach.

IX. Conclusions

We presented a POR technique for a network of (max,+) automata specifying a plant and its requirements to obtain a smaller supervisor, while preserving controllability, nonblockingness, reduced least-restrictiveness, throughput, and latency. This reduction helps in synthesis and performance analysis of supervisory controllers, since less memory is needed to perform synthesis and to store the resulting reduced supervisor and timed state space. The technique is inspired by an existing cluster-based ample set reduction for non-timed systems. The reduced supervisor is computed by exploiting the structure of the automata and information about the (in)dependence among activities. The experimental evaluation shows that our POR technique is successful for models where a small set of states has uncontrollable activities enabled, and automata describing local parts of the system. We obtained reductions up to 80.4% and 91.4% in the number of states and transitions while preserving both functional and performance aspects. The possible reductions are highly dependent on the model structure, the amount of synchronization on activities among automata, and the extent to which activities use the same resources. In our models, the POR technique successfully exploits redundant interleaving related to processing stations that can perform operations in parallel and robot movements that can be executed simultaneously.

Acknowledgment

This research is supported by the Dutch NWO-TTW, carried out as part of the Robust Cyber-Physical Systems (RCPS) program, project number 12694, and carried out as part of the Concerto project, under the responsibility of ESI (TNO) with ASML as the carrying industrial partner.

References

[1] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[2] L. Ouedraogo, R. Kumar, R. Malik *et al.*, "Nonblocking and safe control of discrete-event systems modeled as extended finite automata," *IEEE Trans. Autom. Sci. Eng.*, vol. 8, no. 3, pp. 560–569, July 2011.

[3] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. Springer, 2010.

[4] W. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems: A brief history," *Annual Reviews in Control*, 2018.

[5] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8, no. 1, pp. 39–64, 1996.

[6] ——, "Ten years of partial order reduction," in *Computer Aided Verification*, A. J. Hu and M. Y. Vardi, Eds. Springer, 1998, pp. 17–28.

[7] B. van der Sanden, M. Geilen, M. Reniers *et al.*, "Partial-order reduction for performance analysis of max-plus timed systems," in *Int. Conf. on Application of Concurrency to System Design (ACSD)*. IEEE Computer Society, 2018.

[8] H. Flordal, R. Malik, M. Fabian *et al.*, "Compositional synthesis of maximally permissive supervisors using supervision equivalence," *Discrete Event Dynamic Systems*, vol. 17, no. 4, pp. 475–504, 2007.

[9] B. van der Sanden, J. Bastos, J. Voeten *et al.*, "Compositional specification of functionality and timing of manufacturing systems," in *Forum on Spec. and Design Languages*. IEEE, 2016, pp. 1–8.

[10] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[11] S. Miremadi, Z. Fei, K. Åkesson *et al.*, "Symbolic representation and computation of timed discrete-event systems," *IEEE Trans. on Autom. Science and Eng.*, vol. 11, no. 1, pp. 6–19, 2014.

[12] J. Komenda, S. Lahaye, and J.-L. Boimond, "Supervisory control of (max,+) automata: A behavioral approach," *Discrete Event Dynamic Systems*, vol. 19, pp. 525–549, 2009.

[13] S. Lahaye, J. Komenda, and J.-L. Boimond, "Supervisory control of (max,+) automata: extensions towards applications," *Int. Journal of Control*, vol. 88, no. 12, pp. 2523–2537, 2015.

[14] F. Baccelli, G. Cohen, G. J. Olsder *et al.*, *Synchronization and linearity: an algebra for discrete event systems*. John Wiley and Sons, 1992.

[15] B. van der Sanden, M. Geilen, M. Reniers *et al.*, "Partial-order reduction for synthesis and performance analysis of supervisory controllers," Eindhoven University of Technology, Tech. Rep. ESR-2019-02, 2019.

[16] S. Gaubert, "Performance evaluation of (max,+) automata," *IEEE Trans. on Automatic Control*, vol. 40, no. 12, Dec 1995.

[17] R. Su, J. van Schuppen, and J. Rooda, "The synthesis of time optimal supervisors by using heaps-of-pieces," *IEEE Trans. on Automatic Control*, vol. 57, no. 1, pp. 105–118, Jan 2012.

[18] S. Gaubert and J. Mairesse, "Modeling and analysis of timed petri nets using heaps of pieces," *IEEE Trans. on Automatic Control*, vol. 44, no. 4, pp. 683–697, 1999.

[19] M. Geilen and S. Stuijk, "Worst-case performance analysis of Synchronous Dataflow scenarios," *Int. Conf. on Hardware/Software Codesign and System Synthesis*, pp. 125–134, 2010.

[20] W. Thomas, "Automata on infinite objects," *Handbook of theoretical computer science, Volume B*, pp. 133–191, 1990.

[21] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM-TODAES*, vol. 9, no. 4, pp. 385–418, 2004.

[22] S. Gaubert, "Performance evaluation of (max,+) automata," *IEEE Trans. on Automatic Control*, vol. 40, no. 12, pp. 2014–2025, Dec 1995.

[23] A. Mazurkiewicz, "Trace theory," in *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer Berlin Heidelberg, 1987, pp. 278–324.

[24] T. Basten, D. Bošnački, and M. Geilen, "Cluster-based partial-order reduction," *Automated Software Engineering*, vol. 11, no. 4, pp. 365–402, 2004.

[25] J. Geldenhuys, H. Hansen, and A. Valmari, "Exploring the scope for partial order reduction," in *Proc. of the 2009 Int. Symposium ATVA*, Z. Liu and A. P. Ravn, Eds. Springer, 2009, pp. 39–53.

[26] K. Åkesson, M. Fabian, H. Flordal *et al.*, "Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems," in *Workshop on Discrete Event Systems (WODES)*, July 2006, pp. 384–385.

[27] B. van der Sanden, "Performance analysis and optimization of supervisory controllers," Ph.D. dissertation, Eindhoven University of Technology, 2018.

[28] A. Hellgren, M. Fabian, and B. Lennartson, "Deadlock detection and controller synthesis for production systems using partial order techniques," *Proc. of the 1999 IEEE Int. Conf. on Control Applications*, vol. 2, pp. 1472–1477 vol. 2, 1999.

[29] A. M. Shaw, "Partial order reduction with compositional verification," Master's thesis, University of Waikato, Hamilton, New Zealand, 2014.

[30] J. Bengtsson, B. Jonsson, J. Lilius *et al.*, "Partial order reductions for timed systems," in *Proc. of Int. Conf. on Concurrency Theory (CONCUR)*. Berlin, Heidelberg: Springer, 1998, pp. 485–500.

[31] T. Yoneda and B.-H. Schlingloff, "Efficient verification of parallel real–time systems," *Formal Methods in System Design*, vol. 11, no. 2, pp. 187–215, 1997.

[32] B. Theelen, M. Geilen, and J. Voeten, "Performance model checking scenario-aware dataflow," in *Int. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS)*. Berlin, Heidelberg: Springer, 2011, pp. 43–59.

[33] P. Dietrich, R. Malik, W. M. Wonham *et al.*, *Implementation Considerations in Supervisory Control*. Springer, 2002, pp. 185–201.

[34] A. Morgenstern and K. Schneider, "Synthesizing deterministic controllers in supervisory control," in *Informatics in Control, Automation and Robotics II*. Springer, 2007, pp. 95–102.

[35] J. Huang and R. Kumar, "Directed control of discrete event systems for safety and nonblocking," *IEEE Trans. on Automation Science and Engineering*, vol. 5, no. 4, pp. 620–629, Oct 2008.

[36] R. Su and W. Wonham, "Supervisor reduction for discrete-event systems," *Discrete Event Dynamic Systems*, vol. 14, no. 1, pp. 31–53, Jan 2004.

[37] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

Bram van der Sanden holds an M.Sc. in Computer Science and a Ph.D. in Electrical Engineering from Eindhoven University of Technology (TU/e). He is currently a Research Fellow at ESI (TNO). His research interests include model-based systems engineering, formal models-of-computation, optimization, performance analysis, supervisory controller synthesis, game theory and formal verification methods.

Marc Geilen is currently an Assistant Professor in the Department of Electrical Engineering at TU/e. His research interests include modeling, simulation and programming of multimedia systems, formal models-of-computation, model-based design processes, multiprocessor systems-on-chip, networked embedded systems and cyber-physical systems, and multi-objective optimization and trade-off analysis. He is a member of IEEE.

Michel Reniers (S'17) is currently an Associate Professor in model-based engineering of supervisory control at the Department of Mechanical Engineering at TU/e. He has authored over 100 journal and conference papers. His research portfolio ranges from model-based systems engineering and model-based validation and testing to novel approaches for supervisory control synthesis. Applications of this work are mostly in the areas of cyber-physical systems.

Twan Basten (M'98-SM'06) received the M.Sc. and Ph.D. degrees in computing science from TU/e, Eindhoven, the Netherlands. He is currently a Professor with the Department of Electrical Engineering, TU/e . He is also a Senior Research Fellow with ESI (TNO), Eindhoven. His current research interests include the design of embedded and cyber-physical systems, dependable computing, and computational models.

Appendix

Proofs for Section II (Modeling)

Proposition 5. *Let $\mathcal{S}$ be a normalized (max,+) state space, and $\rho = c_0 A_1 c_1 A_2 c_2 A_3 \dots$ be a run in $\mathcal{S}$ with $c_i = \langle s_i, \boldsymbol{\gamma}_i \rangle$ for each i. Then, for all $n \geq 0$ it holds that $\bar{\boldsymbol{\gamma}}_n = \sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + \boldsymbol{\gamma}_n$.*

Proof. Proof by induction over n. First consider the base case $n = 0$. Then, $\bar{\boldsymbol{\gamma}}_0 = \boldsymbol{\gamma}_0 = \mathbf{0}$. Now, consider the induction step. As induction hypothesis assume $\bar{\boldsymbol{\gamma}}_n = \sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + \boldsymbol{\gamma}_n$. Then:

$$
\begin{aligned}
&\sum_{k=0}^{n} w_2(c_k, A_{k+1}, c_{k+1}) + \boldsymbol{\gamma}_{n+1} \\
=&\sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + w_2(c_n, A_{n+1}, c_{n+1}) + \boldsymbol{\gamma}_{n+1} \\
=\ &\{\text{Def. 4}\} \\
&\sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + \left\| M(A_{n+1}) \otimes \boldsymbol{\gamma}_n \right\| + M(A_{n+1}) \otimes \boldsymbol{\gamma}_n - \left\| M(A_{n+1}) \otimes \boldsymbol{\gamma}_n \right\| \\
=&\sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + M(A_{n+1}) \otimes \boldsymbol{\gamma}_n \\
=\ &\{c + \boldsymbol{M} \otimes \boldsymbol{\gamma} = \boldsymbol{M} \otimes (\boldsymbol{\gamma} + c)\} \\
&M(A_{n+1}) \otimes \left(\sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + \boldsymbol{\gamma}_n \right) \\
=\ &\{\text{induction hypothesis}\} \\
&M(A_{n+1}) \otimes \left((\bigotimes_{k=1}^{n} M(A_k)) \otimes \mathbf{0} \right) \\
=\ &(\bigotimes_{k=1}^{n+1} M(A_k)) \otimes \mathbf{0} \\
=\ &\bar{\boldsymbol{\gamma}}_{n+1}
\end{aligned}
$$

□

Proofs for Section IV (Normalized (max,+) State Space Reduction)

Theorem 19 (Equivalent runs have the same throughput)**.** *Let $\rho, \sigma \in \mathcal{R}(\mathcal{S})$ be two runs in $\mathcal{S}$. If $\rho \equiv \sigma$, then $\mathit{Ratio}(\rho) = \mathit{Ratio}(\sigma)$.*

Proof. Since $\rho \equiv \sigma$, by Def. 18, we have $\rho \succeq_n \sigma$ for $n \geq 0$. This means that there exists a $k \geq n$, and run $\hat{\rho}$ such that $\hat{\rho}^k = \sigma^n \cdot \tau$ for some τ, and $k - n \leq c$.

The maximum difference between $w_i(\rho^n)$ and $w_i(\sigma^n)$ for $i \in \{1, 2\}$ and any $n \geq 0$ is bounded by the maximum total w_i sum of suffix τ, whose length is bounded by c. Let k_i denote the maximum total w_i sum of τ, i.e. the maximum total w_i sum over all simple cycles in the graph:

$$w_i(\rho^n) \leq w_i(\sigma^n) \leq w_i(\rho^n) + k_i \text{ for some } k_i \geq 0.$$

Since $\limsup_{n\to\infty} w_i(\rho^n) = \infty$ for $i \in \{1, 2\}$, the constant k_i can be ignored, and we obtain the following result:

$$\mathsf{Ratio}(\rho) = \limsup_{n\to\infty} \frac{w_1(\rho^n)}{w_2(\rho^n)} = \limsup_{n\to\infty} \frac{w_1(\sigma^n)}{w_2(\sigma^n)} = \mathsf{Ratio}(\sigma).$$

□

Theorem 20 (Equivalent runs have the same latency)**.** *Let $\rho, \sigma \in \mathcal{R}(\mathcal{S})$ be two runs in $\mathcal{S}$. Let A_{src} and A_{snk} be any source-sink pair, and let r be the resource for which we want to calculate the start-to-start latency. If $\rho \equiv \sigma$, then $\lambda_{max}(\rho, A_{src}, A_{snk}, r) = \lambda_{max}(\sigma, A_{src}, A_{snk}, r)$.*

Proof. Consider any source-sink pair instance k in run ρ with latency $\lambda_k(\rho) = \lambda(\rho, i, j, r) = [\bar{\boldsymbol{\gamma}}_j]_r - [\bar{\boldsymbol{\gamma}}_i]_r$ for some $0 \leq i < j$. Furthermore, let $\lambda_k(\sigma) = \lambda(\sigma, m, n, r)$ for some $0 \leq m < n$.

The order of A_{src} and A_{snk} activities in σ is the same as in run ρ, since $R(A_{src}) \cap R(A_{snk}) \neq \emptyset$ and activities A_{src} and A_{snk} are ratio-dependent in any configuration. The corresponding run fragments $\rho[i,j]$ and $\sigma[m,n]$ start with the k-th occurrence of A_{src} and end with the k-th occurrence of A_{snk}.

Let $l = max(j,n)$. Since $\rho \equiv \sigma$, also $\rho \succeq_l \sigma$ and there exists some $\hat{\rho}$ and $k \geq l$ such that $\rho^k \equiv \hat{\rho}^k = \sigma^l \cdot \tau$ for some τ. Prefix $\hat{\rho}^k$ is obtained from ρ by repeatedly commuting ratio-independent activities. We need to show that each such swap has no influence on the latency of source-sink pair instance k. Let A, B be any pair of such activities, ratio-independent in some configuration c_s.

First, consider the case where both A and B are different from A_{src} and A_{snk}. Let $\bar{\gamma}_j$ be the resource vector of interest. Assume that $s \leq j$, since for $s > j$, obviously the swap has no impact on the value. Since A, B are ratio-independent, their matrices commute, and therefore the value of $\bar{\gamma}_j$ remains the same:

$$\begin{aligned}\bar{\gamma}_j &= \bigotimes_{k=1}^{j} M(A_k) \otimes \mathbf{0} \\ &= \bigotimes_{k=s+2}^{j} M(A_k) \otimes M(A_{s+1}) \otimes M(A_s) \otimes \bigotimes_{k=1}^{s-1} M(A_k) \otimes \mathbf{0} \\ &= \bigotimes_{k=s+2}^{j} M(A_k) \otimes M(A_s) \otimes M(A_{s+1}) \otimes \bigotimes_{k=1}^{s-1} M(A_k) \otimes \mathbf{0}.\end{aligned}$$

Now assume a swap of activities A_{src} and B, ratio-independent in configuration c_s. Assume that $A_{src} = A^{\rho}_{s+1}$ in run ρ, and $B = A^{\sigma}_{s+1}$ in run σ. We need to show that $[\bar{\gamma}_s]_r = [\boldsymbol{\gamma}^{\bar{\sigma}}_{s+1}]_r$, where $\boldsymbol{\gamma}^{\bar{\sigma}}_{s+1}$ is the resource vector after executing B from configuration c_s. Since A_{src} and B are ratio-independent, resource r is not used by B. This means that the resource availability time before and after executing B stays the same for resource r. This implies that $[\bar{\gamma}_s]_r = [\boldsymbol{\gamma}^{\bar{\sigma}}_{s+1}]_r$. A similar reasoning can be used for the case where A_{snk} and B are swapped, and when B is executed first in run ρ instead of in run σ. Since the resource availability vectors corresponding to the start of the k-th occurrences of A_{src} and A_{snk} are the same, we conclude that $\lambda_k(\rho) = \lambda_k(\sigma)$ for any $k \geq 0$. It follows that $\lambda_{max}(\rho, A_{src}, A_{snk}, r) = \lambda_{max}(\sigma, A_{src}, A_{snk}, r)$. □

Lemma 40 (adapted from [37, Lemma 8.15])**.** *Let $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2\rangle$ be a finite normalized (max,+) state space and let $\rho[i,j] = c_i B_i c_{i+1} \ldots B_j c_j$ be a run fragment in $\mathcal{S}$. If $ample(c_i)$ satisfies condition (R2) and $\{B_i, \ldots, B_j\} \cap ample(c_i) = \emptyset$, then all activities $A \in ample(c_i)$ are ratio-independent of $\{B_i, \ldots, B_j\}$. In addition, we have $A \in enabled(c_k)$ for $i < k \leq j$.*

Proof. Proof follows the same structure as the proof of [37, Lemma 8.15]. □

Theorem 23 (Equivalent runs)**.** *Let $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2\rangle$ be a finite normalized (max,+) state space, and $\mathcal{S}'$ be a reduced (max,+) state space induced by an ample reduction function. Then for each run $\rho \in \mathcal{R}(\mathcal{S})$, there exists a run $\sigma \in \mathcal{R}(\mathcal{S}')$ with $\rho \equiv \sigma$.*

Proof. Let $\rho \in \mathcal{R}(\mathcal{S})$ be some run in $\mathcal{S}$. Now, we need to show that there exists an equivalent run $\sigma \in \mathcal{R}(\mathcal{S}')$ such that $\rho \succeq \sigma$ and $\sigma \succeq \rho$.

First, we show that there exists a run $\sigma \in \mathcal{R}(\mathcal{S}')$ such that $\rho \succeq \sigma$ with $c = |C|$. To this end, we first define σ^n recursively as follows:

$$\begin{aligned}\sigma^0 &= \varepsilon \\ \sigma^{n+1} &= \sigma^n \cdot A, \text{where } A \in ample(\rho[n]) \text{ is the first such activity in } \rho[n..].\end{aligned}$$

We now prove that such an activity A can be found in $\rho[n..] = c_n B_1 c_{n+1} B_2 \ldots$, with $c_n = \rho[n]$. Let $Res' \subseteq Res$ denote the set of resources used by activities in $ample(c_n)$, i.e. $Res' = \bigcup_{A \in ample(c_n)} R(A)$. Since the normalized (max,+) state space is finite, eventually we always reach a cycle, and on this cycle there is at least one activity that uses one of the resources in Res' that is also used by some activity $A \in ample(c_n)$. Let $A = B_m$ be the first such activity.

Let σ^n for each $n \geq 0$ be constructed using this procedure, and let $\sigma = \bigsqcup_{n \geq 0} \sigma^n$. We now prove that $\rho \succeq_n \sigma$ for all $n \geq 0$ by induction on n. As a base case, consider $n = 0$. Then, obviously $\sigma^0 = \rho^0 = \varepsilon$ satisfies $\rho \succeq_0 \sigma$. As induction hypothesis assume that $\rho \succeq_n \sigma$. This means that there exists a $k \geq n$, and $\hat{\rho}^k \equiv \rho^k$ such that $\hat{\rho}^k = \sigma^n \cdot \tau$ for some τ, and $k - n \leq |C|$. In the construction, we find some $l \geq n+1$, such that $\sigma^{n+1} = \sigma^n \cdot A$, $\tau' = B_1 \ldots B_{m-1}$ and $\tilde{\rho}^l = \sigma^{n+1} \cdot \tau'$. By Lemma 40, $A = B_m \in ample(c_n)$, and B_m is ratio-independent with $B_1 \ldots B_{m-1}$, which means that $AB_1 \ldots B_{m-1} \equiv B_1 \ldots B_{m-1} A$. The value of m (and subsequently also the length of τ') is bounded by the maximum simple cycle length in the state space, i.e. $m \leq |C|$. Since $\sigma^n \equiv \hat{\rho}^n$ and $AB_1 \ldots B_{m-1} \equiv B_1 \ldots B_{m-1} A$, we have that $\tilde{\rho}^l \equiv \rho^l$. By the principle of induction, $\rho \succeq_n \sigma$ for all $n \geq 0$.

Since $\rho \succeq_n \sigma$ for all $n \geq 0$, run σ satisfies $\rho \succeq \sigma$. The fact that $\sigma \succeq \rho$, follows from the observation that each run $\sigma \in \mathcal{R}(\mathcal{S}')$ is also a run in $\mathcal{R}(\mathcal{S})$. Therefore, we have shown that for each run $\rho \in \mathcal{R}(S)$, there exists a run $\sigma \in \mathcal{R}(S')$ such that $\rho \equiv \sigma$. □

From Theorems 20 and 23, it immediately follows that an ample reduction preserves throughput and latency aspects.

Corollary 41. *Let $\mathcal{S}$ be a finite normalized (max,+) state space, and $\mathcal{S}'$ the reduced (max,+) state space induced by an ample reduction according to Def. 22. Then $\tau_{min}(\mathcal{S}) = \tau_{min}(\mathcal{S}')$ and $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$, and therefore $\mathcal{S} \approx_p \mathcal{S}'$ by Def. 14.*

Proofs for Section V ((max,+) Automaton Reduction)

Proposition 26. *Let $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ be a (max,+) automaton and let $\mathcal{A}_\omega = \langle S_\omega, \hat{s}_\omega, S^m_\omega, \mathcal{A}ct_\omega, reward_\omega, M_\omega, T_\omega\rangle$ be the ω-extension of $\mathcal{A}$ with $\omega \notin \mathcal{A}ct$. Then $\mathcal{A}$ is nonblocking if and only if $\mathcal{A}_\omega$ is ω-reachable.*

Proof. Straightforward, using the definitions of ω-reachability and nonblockingness. □

Lemma 42. *Let A and B be activities with corresponding matrices $\boldsymbol{M}_A$ and $\boldsymbol{M}_B$. If $R(A) \cap R(B) = \emptyset$, then $\boldsymbol{M}_A \otimes \boldsymbol{M}_B = \boldsymbol{M}_B \otimes \boldsymbol{M}_A$.*

If the (max,+) matrices of two activities commute, then the resulting normalized vector in the (max,+) state space is the same.

Lemma 43. *Consider (max,+) matrices $\boldsymbol{M}_A, \boldsymbol{M}_B$ and vector $\boldsymbol{\gamma}$. If $\boldsymbol{M}_A \otimes \boldsymbol{M}_B = \boldsymbol{M}_B \otimes \boldsymbol{M}_A$, then the resulting normalized vector after multiplication in the normalized (max,+) state space is the same.*

Proof.

$$
\begin{aligned}
& norm(\boldsymbol{M}_B \otimes norm(\boldsymbol{M}_A \otimes \boldsymbol{\gamma})) \\
&= \boldsymbol{M}_B \otimes (\boldsymbol{M}_A \otimes \boldsymbol{\gamma} \; - \|\boldsymbol{M}_A \otimes \boldsymbol{\gamma}\|) - \big\|\boldsymbol{M}_B \otimes (\boldsymbol{M}_A \otimes \boldsymbol{\gamma} \; - \|\boldsymbol{M}_A \otimes \boldsymbol{\gamma}\|)\big\| \\
&= \boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \; - \|\boldsymbol{M}_A \otimes \boldsymbol{\gamma}\| - \|\boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \boldsymbol{\gamma}\| + \|\boldsymbol{M}_A \otimes \boldsymbol{\gamma}\| \\
&= \boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \; - \|\boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \boldsymbol{\gamma}\| \\
&= \{\text{using } \boldsymbol{M}_A \otimes \boldsymbol{M}_B = \boldsymbol{M}_B \otimes \boldsymbol{M}_A\} \\
& \quad \boldsymbol{M}_A \otimes \boldsymbol{M}_B \otimes \boldsymbol{\gamma} \; - \|\boldsymbol{M}_A \otimes \boldsymbol{M}_B \otimes \boldsymbol{\gamma}\| \\
&= \{ \text{ same steps in reverse direction } \} \\
& \quad norm(\boldsymbol{M}_A \otimes norm(\boldsymbol{M}_B \otimes \boldsymbol{\gamma})).
\end{aligned}
$$

□

Given resource-independent activities, the sum of the weights for both w_1 and w_2 after execution are the same, independent of the execution order.

Lemma 44. *Let c be a configuration and A, B be resource-independent activities. Then, the sum of weights w_1 and w_2 of the run fragments after execution of both A and B starting from c is the same, independent of the relative order of A and B.*

Proof. For weight w_1: trivial. For weight w_2:

$$
\begin{aligned}
& w_2(c, A, A(c)) + w_2(A(c), B, AB(c)) \\
&= \; \| \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \| + \| \boldsymbol{M}_B \otimes norm(\boldsymbol{M}_A \otimes \boldsymbol{\gamma}) \| \\
&= \; \| \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \| + \| \boldsymbol{M}_B \otimes (\boldsymbol{M}_A \otimes \boldsymbol{\gamma} - \| \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \|) \| \\
&= \{\text{using } \boldsymbol{M}_A \otimes (\boldsymbol{\gamma} - c) = \boldsymbol{M}_A \otimes \boldsymbol{\gamma} - c\} \\
& \quad \| \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \| + \| \boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \boldsymbol{\gamma} - \| \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \| \| \\
&= \{\text{using } \; \| \boldsymbol{\gamma} - c \| = \| \boldsymbol{\gamma} \| - c\} \\
& \quad \| \boldsymbol{M}_B \otimes \boldsymbol{M}_A \otimes \boldsymbol{\gamma} \| \\
&= \; \| \boldsymbol{M}_A \otimes \boldsymbol{M}_B \otimes \boldsymbol{\gamma} \| \\
&= \{ \text{ same steps in reverse direction } \} \\
& \quad w_2(c, B, B(c)) + w_2(B(c), A, BA(c)).
\end{aligned}
$$

□

Theorem 28. *Given are a (max,+) automaton $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ and a state $s \in S$ with activities $A, B \in enabled(s)$. Consider any configuration $c = \langle s, \boldsymbol{\gamma}\rangle$ in the underlying normalized (max,+) state space. If A and B are uncontrollable-(in)dependent in s, then they are ratio-(in)dependent in c.*

Proof. Assume that A and B are uncontrollable independent in s. To prove that A and B are ratio independent in c, we show that the three conditions stated in Def. 15 hold. The first part of condition 1 follows directly by independence of A and B. The second part requires a unique configuration $c' = \langle s', \boldsymbol{\gamma}'\rangle$ after executing A and B in arbitrary order. The fact that the same state s' is reached follows from the independence of A and B, and that the same resource availability vector $\boldsymbol{\gamma}'$ is reached follows from Lemma 43. Condition 2 requires that the sum of the weights for both w_1 and w_2 is the same, independent of the execution order. This follows from Lemma 44. Condition 3 requires that A and B have no resources in common, which follows directly from the definition of uncontrollable independence.

The proof for the case that A and B are uncontrollable dependent is analogous. □

The following lemma shows that conditions (A2), (A5.1), and (A5.2) guarantee that each activity in the ample set is uncontrollable independent with all activities outside the ample set.

Lemma 45. *Let $ample(s) \subseteq enabled(s)$ be an ample set that satisfies conditions (A2), (A5.1), and (A5.2). Then for activities $A, B \in enabled(s)$ with $A \notin ample(s)$ and $B \in ample(s)$, it holds that A and B are uncontrollable independent in s.*

Proof. We show that $enabled(s) = enabled(A(s)) \cap \mathcal{A}ct_u = enabled(B(s)) \cap \mathcal{A}ct_u = enabled(AB(s)) \cap \mathcal{A}ct_u = \emptyset$. The fact that A and B are independent can be shown from condition (A2).

- $enabled(s) = \emptyset$: Assume that there exist an uncontrollable activity $U \in enabled(s) \cap \mathcal{A}ct_u$. Since $A \in enabled(s) \setminus ample(s)$, by condition (A2), U and A must be independent in s, and therefore also $U \in enabled(A(s))$. Then, the same reasoning applies as in the case that $enabled(A(s)) \cap \mathcal{A}ct_u \neq \emptyset$.
- $enabled(B(s)) \cap \mathcal{A}ct_u = \emptyset$: Suppose towards a contradiction that $enabled(B(s)) \cap \mathcal{A}ct_u \neq \emptyset$. Then by condition (A5.1), $ample(s) = enabled(s)$, contradicting that $A \notin ample(s)$.
- $enabled(A(s)) \cap \mathcal{A}ct_u = \emptyset$: Assume that there exist an uncontrollable activity $U \in enabled(A(s)) \cap \mathcal{A}ct_u$. Then we distinguish the cases where either $U \notin enabled(B(s))$ or $U \in enabled(B(s))$.
  In case $U \notin enabled(B(s))$, activity B is dependent with U in s, and by condition (A2) it follows that $A \in ample(s)$, contradicting the assumption that $A \notin ample(s)$.
  In case $U \in enabled(B(s))$, by condition (A5.1) it follows that $ample(s) = enabled(s)$ contradicting that $A \notin ample(s)$.
- $enabled(AB(s)) \cap \mathcal{A}ct_u = \emptyset$: Suppose towards a contradiction that $enabled(AB(s)) \cap \mathcal{A}ct_u \neq \emptyset$. Then according to condition (A5.2), activities A and B must be either both in or not in the ample set. This contradicts the assumption that $A \notin ample(s)$ and $B \in ample(s)$.

□

Theorem 34. *Let $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ be a (max,+) automaton with corresponding state space $\mathcal{S} = \langle C, \hat{c}, \mathcal{A}ct, \Delta, M, w_1, w_2\rangle$. Let $ample_{\mathcal{A}}$ be an ample reduction on $\mathcal{A}$ that yields $\mathcal{A}'$ with corresponding state space $\mathcal{S}'$. Let $ample_{\mathcal{S}}$ be the ample reduction corresponding to $ample_{\mathcal{A}}$ on $\mathcal{S}$ that yields state space $\hat{\mathcal{S}}$, then $\hat{\mathcal{S}} = \mathcal{S}'$.*

Proof. To illustrate the approach, consider the following figure.

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{\quad ample_{\mathcal{A}} \quad} & \mathcal{A}' \\ \downarrow & & \downarrow \\ \mathcal{S} & \xrightarrow[\quad ample_{\mathcal{S}} \quad]{} & \hat{\mathcal{S}} = \mathcal{S}' \end{array}$$

For each $s \in S$, it holds that $enabled_{\mathcal{S}'}(s) = ample_{\mathcal{A}}(s)$. For each configuration $c \in C$ with $c = \langle s, \gamma\rangle$ we have $enabled_{\mathcal{S}'}(c) = enabled_{\mathcal{A}'}(s)$ by Def. 4. Also, $enabled_{\mathcal{S}}(c) = enabled_{\mathcal{A}}(s)$ by Def. 4, and $enabled_{\hat{\mathcal{S}}}(c) = ample_{\mathcal{S}}(c) = ample_{\mathcal{A}}(s)$ by definition of $ample_{\mathcal{S}}$. This implies that $enabled_{\hat{\mathcal{S}}}(c) = enabled_{\mathcal{S}'}(c)$ for each configuration $c \in C$. Since the set of enabled activities is the same, the set of configurations and transitions in both $\hat{\mathcal{S}}$ and $\mathcal{S}'$ is also the same. Also $\mathcal{A}ct' = \hat{\mathcal{A}ct} = \mathcal{A}ct$, $M' = \hat{M} = M$, and $w'_i = \hat{w_i} = w_i$ for $i \in \{1, 2\}$. This proves that $\hat{\mathcal{S}} = \mathcal{S}'$. □

Lemma 46. *If ample reduction $ample_{\mathcal{A}}$ on $\mathcal{A}$ satisfies conditions (A1) and (A2), then the corresponding ample reduction $ample_{\mathcal{S}}$ on the corresponding state space $\mathcal{S}$ satisfies (R1) and (R2).*

Proof. Condition (R1) follows directly from (A1). By Lemma 28, if activities are resource (in)dependent in s, then they are also ratio-(in)dependent in any c with $c = \langle s, \gamma\rangle$. Combined with the definition of $ample_{\mathcal{S}}$, (R2) now also follows from (A2). □

An ample reduction on a (max,+) automaton preserves the minimum throughput and maximum latency values.

Theorem 47. *Let $\mathcal{A}$ be any (max,+) automaton with corresponding state space $\mathcal{S}$. Let ample be an ample reduction on $\mathcal{A}$ satisfying conditions (A1) and (A2) that yields $\mathcal{A}'$ with corresponding state space $\mathcal{S}'$. Then $\tau_{min}(\mathcal{S}) = \tau_{min}(\mathcal{S}')$ and $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$.*

Proof. Let $ample_{\mathcal{S}}$ be the corresponding ample reduction on $\mathcal{S}$ that yields state space $\hat{\mathcal{S}}$. By Lemma 34, $\hat{\mathcal{S}} = \mathcal{S}'$. Since $ample_{\mathcal{A}}$ satisfies conditions (A1) and (A2), by Lemma 46, $ample_{\mathcal{S}}$ satisfies conditions (R1) and (R2). By Corollary 41, this implies that $\tau_{min}(\mathcal{S}) = \tau_{min}(\mathcal{S}')$ and $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$. Therefore, if $ample_{\mathcal{A}}$ satisfies (A1) and (A2), then the minimum throughput and maximum latency values are preserved in the reduced state space $\mathcal{S}'$. □

The following theorem shows that if a reduction satisfies conditions (A1) to (A4), then the reduction is reduced least-restrictive; for each run to a marked state in the full automaton we can find an equivalent run to the same marked state in the reduced automaton. Conditions (A5.1) and (A5.2) are not needed yet at this point. They are needed after this theorem to ensure that if a path to a marked state is present in the supervisor, then an equivalent path is also present in the reduced supervisor obtained after synthesis.

Theorem 48. *Let $\mathcal{A} = \langle S, \hat{s}, S^m_{\mathcal{A}}, \mathcal{A}ct, T_{\mathcal{A}}\rangle$ be an automaton extended with ω-self-loops. Let $\mathcal{A}_r = \langle S_r, \hat{s}, S^m_r, \mathcal{A}ct, T_r\rangle$ be the reduced automaton induced by ample reduction ample. Let $s \in S_r$ and let $\rho = s \xrightarrow{\alpha}{}^*_{\mathcal{A}} s_m$ be a run in $\mathcal{A}$ with $\alpha \in \mathcal{A}ct^*$ and $s_m \in S^m$. Then in $\mathcal{A}_r$, there exists a run $\rho_r = s \xrightarrow{\beta}{}^*_r s_m$ with $\beta \in \mathcal{A}ct^*$ and $\rho \equiv \rho_r$.*

Proof. Let $\rho = s \xrightarrow{\alpha}{}^* s_m$ be some run in $\mathcal{A}$. Now, we need to show that there exists a run $\rho_r = s \xrightarrow{\beta}{}^*_r s_m$ in $\mathcal{A}_r$ such that $\rho \equiv \rho_r$. First, we show that $\rho \succeq_n \rho_r$ for all $n \leq |\rho|$. We define ρ^n_r recursively as follows:

$$\rho^0_r = \varepsilon$$
$$\rho^{n+1}_r = \rho^n_r \cdot A, \text{where } A \in ample(\rho[n]) \text{ is the first such activity in } \rho[n..].$$

We prove that such an activity A can be found in $\rho[n..] = s_n \xrightarrow{B_1 \cdots B_2} s_m$. We distinguish the cases $\omega \in enabled(s)$ and $\omega \notin enabled(s)$. First consider the case where $\omega \in enabled(s)$. Then by condition (A4) also $\omega \in ample(s)$, which means that we can choose $A = \omega$. Now consider the case where $\omega \notin enabled(s)$. We know that $s_m \xrightarrow{\omega}$ and since $\omega \notin enabled(s)$, ω is dependent with all activities in $enabled(s)$. Since $enabled(s) \neq \emptyset$, by condition (A1) also $ample(s) \neq \emptyset$. By condition (A2), it follows that there exists some $B_m \in ample(s_n)$. Let $A = B_m$ be the first such activity.

Let ρ^n_r for each $0 \leq n \leq |\rho|$ be constructed using this procedure. We now prove that $c \leq |\rho|$ and $\rho \succeq_n \rho_r$ for all $0 \leq n \leq |\rho|$ by induction on n. As a base case, consider $n = 0$. Then, obviously $\rho^0_r = \rho^0 = \varepsilon$ satisfies $\rho \succeq_0 \rho_r$. As induction hypothesis assume that $\rho \succeq_n \rho_r$. This means that there exists a $k \geq n$, and $\hat{\rho}^k \equiv \rho^k$ such that $\hat{\rho}^k = \rho^n_r \cdot \tau$ for some τ, and $k - n \leq |\rho|$. In the construction, we find some $l \geq n+1$, such that $\rho^{n+1}_r = \rho^n_r \cdot A$, $\tau' = B_1 \dots B_{m-1}$ and $\tilde{\rho}^l = \rho^{n+1}_r \cdot \tau'$. By Lemma 40, $A = B_m \in ample(s_n)$, and B_m is resource independent with $B_1 \dots B_{m-1}$ in s_n, which means that $AB_1 \dots B_{m-1} \equiv_{s_n} B_1 \dots B_{m-1}A$. The value of m (and subsequently also the length of τ') is bounded by the length of ρ, i.e. $m \leq |\rho|$. Since $\rho^n_r \equiv \hat{\rho}^n$ and $AB_1 \dots B_{m-1} \equiv B_1 \dots B_{m-1}A$, we have that $\tilde{\rho}^l \equiv \rho^l$. By the principle of induction, $\rho \succeq_n \rho_r$ for all $n \geq 0$.

Since $\rho \succeq_n \rho_r$ for all $n \geq 0$, run ρ_r satisfies $\rho \succeq \rho_r$. The fact that $\rho_r \succeq \rho$, follows from the observation that each run $\rho_r \in \mathcal{A}'$ is also a run in $\mathcal{A}$. Therefore, we have shown that for each run ρ in $\mathcal{A}$, there exists a run ρ' in $\mathcal{A}'$ such that $\rho \equiv \rho_r$. □

Theorem 49. *Let $\mathcal{A}$ be an automaton that has been extended with ω-self-loops, and let $\mathcal{A}'$ be the reduced automaton induced by reduction function ample. Let $\mathcal{A}$ be nonblocking. If ample satisfies conditions (A1), (A2), (A3), and (A4) then $\mathcal{A}'$ is nonblocking and controllable with respect to $\mathcal{A}$.*

Proof of Theorem 49. First we show that $\mathcal{A}'$ is nonblocking. By Prop. 26, this is equivalent to showing that $\mathcal{A}'$ is ω-reachable. Let $s \in S'$ be any reachable state in $\mathcal{A}'$, such that $\hat{s} \xrightarrow{\alpha_1}{}^*_{\mathcal{A}'} s$ for some $\alpha_1 \in \mathcal{A}ct^*$. Then, also $\hat{s} \xrightarrow{\alpha_1}{}^*_{\mathcal{A}} s$ since $\mathcal{A}' \preceq \mathcal{A}$. Since $\mathcal{A}$ is nonblocking, it is also ω-reachable. Therefore, there exists a run $\rho = s \xrightarrow{\alpha_2}{}^*_{\mathcal{A}} s_m$ for some $\alpha_2 \in \mathcal{A}ct^*$ such that $\omega \in enabled(s_m)$. By Theorem 48, there exists a run $\rho_r = s \xrightarrow{\beta}{}^*_{\mathcal{A}'} s_m$ in $\mathcal{A}'$ for some $\beta \in \mathcal{A}ct^*$ with $\rho \equiv \rho_r$.

We also need to show that $\mathcal{A}'$ is controllable with respect to $\mathcal{A}$. Let $\alpha \in \mathcal{A}ct^*$ and $U \in \mathcal{A}ct_u$, and let $s, s' \in S_{\mathcal{A}}$ such that $\hat{s} \xrightarrow{\alpha}{}^*_{\mathcal{A}'} s$ and $\hat{s} \xrightarrow{\alpha}{}^*_{\mathcal{A}} s \xrightarrow{U}_{\mathcal{A}} s'$. Since $s \in S'_{\mathcal{A}}$, and condition (S3) holds in s, we have that $U \in ample(s)$. By Def. 33, then also $\hat{s} \xrightarrow{\alpha}{}^*_{\mathcal{A}'} s \xrightarrow{U}_{\mathcal{A}'} s'$, proving that $\mathcal{A}'$ is controllable with respect to $\mathcal{A}$. □

Lemma 50 (adapted from [37, Lemma 8.15])**.** *Let $\rho[i,j] = c_i B_i c_{i+1} \dots B_j c_j$ be a run fragment in $\mathcal{S}$. If $ample(c_i)$ satisfies condition (A2), (A5.1) and (A5.2) and $\{B_i, \dots, B_j\} \cap ample(c_i) = \emptyset$, then all activities $A \in ample(c_i)$ are uncontrollable-independent of $\{B_i, \dots, B_j\}$. In addition, we have $A \in enabled(c_k)$ for $i < k \leq j$.*

Proof. Proof follows the same structure as the proof of [37, Lemma 8.15]. □

Algorithm 3 Synthesis algorithm SUPCN, adapted from [2]

1: **proc** SUPCN($\mathcal{P} = \langle S, \hat{s}, S^m, \mathcal{A}ct, T\rangle$)
2: $k \leftarrow 0, X^k \leftarrow S$
3: **repeat**
4: $i \leftarrow 0, N_0^k \leftarrow S^m \cap X^k$
5: **repeat** ▷ compute the nonblocking states
6: $N_{i+1}^k \leftarrow N_i^k \cup \{s \in X^k \mid s \rightarrow s', s' \in N_i^k\}$
7: $i \leftarrow i + 1$
8: **until** $N_i^k = N_{i-1}^k$
9: $N^k \leftarrow N_i^k$
10: $j \leftarrow 0, B_0^k \leftarrow X^k \setminus N^k$
11: **repeat** ▷ compute the bad states
12: $B_{j+1}^k \leftarrow B_j^k \cup \{s \in X^k \mid s \xrightarrow{U} s', s' \in B_j^k, U \in \mathcal{A}ct_u\}$
13: $j \leftarrow j + 1$
14: **until** $B_j^k = B_{j+1}^k$
15: $B^k \leftarrow B_j^k$
16: $X^{k+1} \leftarrow X^k \setminus B^k$
17: **until** $X^k = X^{k-1}$
18: **if** $\hat{s} \in X^k$ **then**
19: **return** $\langle X^k, \hat{s}, S^m \cap X^k, \mathcal{A}ct, T \cap (X^k \times \mathcal{A}ct \times X^k)\rangle$
20: **else**
21: **return** $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle$

Theorem 51 (Ample reduction preserves a path to each marked state)**.** *Let $\mathcal{A} = \langle S, \hat{s}, S^m, \mathcal{A}ct, reward, M, T\rangle$ be an automaton that has been extended with ω-self-loops. Let $\mathcal{A}_r = \langle S_r, \hat{s}_r, S_r^m, \mathcal{A}ct_r, reward_r, M_r, T_r\rangle$ be the reduced automaton induced by ample reduction ample. Let $s \in S_r$ and let $\rho = s \xrightarrow{\alpha}_{\mathcal{A}}^* s_m$ be a path in $\mathcal{A}$ with $\alpha \in \mathcal{A}ct^*$ and $s_m \in S^m$. Then in $\mathcal{A}_r$, there exists a path $\rho_r = s \xrightarrow{\beta}_r^* s_m$ with $\beta \in \mathcal{A}ct^*$ and $\alpha \equiv_{s,u} \beta$.*

Proof. The proof follows the same structure as the proof of Theorem 48, using Lemma 50 rather than Lemma 40. □

To prove reduced least-restrictiveness of the reduced supervisor, we need a definition of the synthesis procedure, shown in Algorithm 3.

Theorem 52 (Existence of an equivalent path in the reduced supervisor)**.** *Let $\mathcal{P}$ be a plant, and let $\mathcal{P}'$ be the reduced plant after applying an ample reduction on $\mathcal{P}$. Let $\mathcal{A}_{sup} = sup\mathcal{CN}(\mathcal{P})$ be the supervisor after applying synthesis on $\mathcal{P}$, and $\mathcal{A}'_{sup} = sup\mathcal{CN}(\mathcal{P}')$ be the reduced supervisor after applying synthesis on $\mathcal{P}'$. Let $\rho = \hat{s} \xrightarrow{\alpha}_{\mathcal{A}_{sup}}^* s_m$ be a path in $\mathcal{A}_{sup}$ with $\alpha \in \mathcal{A}ct^*$ and $s_m \in S^m$. Then in $\mathcal{A}'_{sup}$ there exists a path $\rho' = \hat{s} \xrightarrow{\beta}_{\mathcal{A}'_{sup}}^* s_m$ with $\beta \in \mathcal{A}ct^*$ and $\rho \equiv_{s,u} \rho'$.*

Proof. Let $\rho = \hat{s} \xrightarrow{\alpha}_{\mathcal{A}_{sup}}^* s_m$ be any path in $\mathcal{A}_{sup}$ to a marked state. Since $\mathcal{A}_{sup} \preceq \mathcal{P}$, path ρ is also a path in $\mathcal{P}$. By Theorem 51, there exists at least one equivalent path $\rho' = \hat{s} \xrightarrow{\beta}^* s_m$ in $\mathcal{P}'$ with $\rho' \equiv_{\hat{s},u} \rho$. We need to show that this path is also present in $\mathcal{A}'_{sup}$, which means that it is not removed during synthesis on $\mathcal{P}'$.

A state s is not removed during synthesis using Algorithm 3 if $s \in N^k$ and $s \notin B^k$ for each iteration $k \geq 0$ in the algorithm. Since ρ is a path in $\mathcal{A}_{sup}$, for each state s on ρ it holds that $s \in N^l$ and $s \notin B^l$ for each $l \geq 0$ in the synthesis on $\mathcal{P}$. To show path ρ' is present in $\mathcal{A}'_{sup}$, we need to show that all states s' on ρ' are initially nonblocking in $\mathcal{P}'$, i.e. $s' \in N^0$ at the start of synthesis on $\mathcal{P}'$, and during synthesis no state is added to set B^k for some $k \geq 0$.

Path ρ' can be obtained from ρ in steps $\rho = \rho^0 \rightarrow \rho^1 \rightarrow \ldots \rightarrow \rho^n = \rho'$ by repeatedly commuting uncontrollable-independent activities. Consider the step from ρ^i to ρ^{i+1}, where we permute two activities A and B that are uncontrollable-independent in state s_i. We first show that for all states s' in ρ^{i+1} , s' is initially nonblocking. Let

$$\rho^i = \hat{s} \rightarrow^* s_i \xrightarrow{AB}^* \bar{s} \rightarrow^* s_m, \text{ and}$$
$$\rho^{i+1} = \hat{s} \rightarrow^* s_i \xrightarrow{BA}^* \bar{s} \rightarrow^* s_m.$$

Since $\bar{s} \rightarrow^* s_m$ is also in ρ^{i+1}, we know that all states on path segment $\rho^{i+1}[\bar{s}, s_m]$ are initially nonblocking. Since $\bar{s}$ is nonblocking, all states before $\bar{s}$ on path ρ^{i+1} are also initially nonblocking. By induction on the number n of permutations needed to transform ρ into ρ', $\rho^n = \rho'$, and all states in ρ' are initially nonblocking.

Now, we show that no state s' on path ρ' is added to set B_{j+1}^k for some $k, j \geq 0$ on line 12 of SUPCN($\mathcal{P}'$). Suppose towards a contradiction that there is a state $s_b \in B_j^k$ such that $s' \xrightarrow{U} s_b$ and $U \in \mathcal{A}ct_u$. We know that ρ' is also a path in plant $\mathcal{P}$ and in $\mathcal{A}_{sup}$ by Theorem 51. Since $\mathcal{A}_{sup}$ is controllable with respect to $\mathcal{P}$, $U \in enabled(s)$ in $\mathcal{A}_{sup}$

and $s_b \in S_{sup}$. If now $s_b \in B^l$ for some $l \geq 0$ in the synthesis of $\mathcal{P}$, then s_b would not be present in S_{sup}, leading to a contradiction. Therefore, state s' is not added to B^k_j for any $k, j \geq 0$ in the synthesis on $\mathcal{P}'$.

By combining these two parts, we conclude that no state s' of path ρ' is removed during synthesis on $\mathcal{P}'$, and as a result, path ρ' is also present in $\mathcal{A}'_{sup}$.

□

Theorem 35 (Ample reduction preserves functionality and performance)**.** *Let $\mathcal{P}$ be a plant, and $\mathcal{P}'$ the reduced plant obtained by an ample reduction that satisfies Def. 33. Let $\mathcal{A}_{sup}$ be the supervisor with $\mathcal{A}_{sup} = sup\mathcal{CN}(\mathcal{P})$, and $\mathcal{A}'_{sup}$ be the reduced supervisor with $\mathcal{A}'_{sup} = sup\mathcal{CN}(\mathcal{P}')$. Then $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$.*

Proof. To show that $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$, we need to prove the four conditions as stated in Def. 32.

1. Nonblockingness: We need to show that if $\mathcal{A}_{sup}$ is nonblocking then $\mathcal{A}'_{sup}$ is nonblocking. The synthesis algorithm always guarantees that the resulting supervisor $\mathcal{A}'_{sup} = sup\mathcal{CN}(\mathcal{P}')$ is nonblocking, independent of whether $\mathcal{A}_{sup}$ is nonblocking.

2. Controllability: We need to show that if $\mathcal{A}_{sup}$ is controllable w.r.t. $\mathcal{P}$ then $\mathcal{A}'_{sup}$ is controllable with respect to $\mathcal{P}$. Consider any string $\sigma \in \mathcal{A}ct^*$ and state $s \in S_{\mathcal{A}'_{sup}}$ such that $\hat{s} \xrightarrow{\sigma}{}^*_{\mathcal{A}'_{sup}} s$. Let $enabled_{\mathcal{A}}(s)$ denote the enabled set in state s in automaton $\mathcal{A}$. Since by Theorem 49, $\mathcal{P}'$ is controllable with respect to $\mathcal{P}$, $enabled'_{\mathcal{P}}(s) \cap \mathcal{A}ct_u = enabled_{\mathcal{P}}(s) \cap \mathcal{A}ct_u$. Synthesis guarantees that $\mathcal{A}'_{sup}$ is controllable with respect to $\mathcal{P}'$, which means that $enabled_{\mathcal{A}_{sup}}(s) \cap \mathcal{A}ct_u = enabled'_{\mathcal{P}}(s) \cap \mathcal{A}ct_u$. Therefore, $enabled_{\mathcal{A}_{sup}}(s) \cap \mathcal{A}ct_u = enabled_{\mathcal{P}}(s) \cap \mathcal{A}ct_u$.

3. Reduced least-restrictiveness: We need to show that $\mathcal{A}'_{sup}$ is reduced least-restrictive w.r.t. $\mathcal{A}_{sup}$. Supervisor $\mathcal{A}'_{sup}$ is reduced least-restrictive with respect to supervisor $\mathcal{A}_{sup}$, if for each path $\hat{s} \xrightarrow{\sigma}{}^*_{\mathcal{A}_{sup}} s_m$ in $\mathcal{A}_{sup}$ with $\sigma \in \mathcal{A}ct^*$ and $s_m \in S^m$, there exists a path $\hat{s} \xrightarrow{\tau}{}^*_{\mathcal{A}'_{sup}} s_m$ in $\mathcal{A}'_{sup}$ with $\sigma' \in \mathcal{A}ct^*$ and $\sigma \equiv \tau$. This follows by Theorem 52.

4. Performance: Let $\mathcal{S}'$ and $\mathcal{S}$ be the corresponding normalized (max,+) state spaces of $\mathcal{A}'_{sup}$ and $\mathcal{A}_{sup}$. We need to show that $\mathcal{S}' \approx_p \mathcal{S}$. By Def. 14, this implies that we need to show that $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$ and $\tau_{min}(\mathcal{S}) = \tau_{min}(\mathcal{S}')$.

- $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$**:** Let A_{src} and A_{snk} be the source and sink activity and let r be the resource on which we want to compute the start-to-start latency. Then by Def. 13,

$$\lambda_{max}(\mathcal{S}) = \sup_{\rho \in \mathcal{R}(\mathcal{S})} \lambda_{max}(\rho, A_{src}, A_{snk}, r) = \sup_{\rho \in \mathcal{R}(\mathcal{S})} \sup_{k \geq 1} \lambda_k(\rho) \textbf{ where}$$
$$\lambda_k(\rho) = \lambda(\rho, i, j, r),$$
$$i = \mathsf{getOccurence}(\rho, A_{src}, k), \text{ and}$$
$$j = \mathsf{getOccurence}(\rho, A_{snk}, k).$$

  Consider the start-to-start latency for source-sink pair A_{src}, A_{snk} on resource r is determined by occurrences $i = \mathsf{getOccurence}(\rho, A_{src}, k)$ and $j = \mathsf{getOccurence}(\rho, A_{snk}, k)$ in a run $\rho \in \mathcal{R}(S)$. Then, the maximum latency is found in run fragment $\rho[..j+1] = \hat{c}A_1 \dots c_i A_{src} \dots c_j A_{snk} c_{j+1}$, with for all $c_h = \langle s_h, \boldsymbol{\gamma}_h \rangle$ with $h \leq j+1$. This run fragment corresponds to a path $\hat{s} \xrightarrow{\alpha_1}{}^* s_i \xrightarrow{A_{src}\alpha_2}{}^* s_j \xrightarrow{A_{snk}} s_{j+1}$ in $\mathcal{A}_{sup}$. Since $\mathcal{A}_{sup}$ is nonblocking, by Def. 8, there exists a path $\hat{s} \xrightarrow{\alpha_1}{}^* s_i \xrightarrow{A_{src}\alpha_2}{}^* s_j \xrightarrow{A_{snk}} s_{j+1} \xrightarrow{\alpha_3}{}^* s_m$ for some marked state $s^m \in S^m$ and $\alpha_1, \alpha_2, \alpha_3 \in \mathcal{A}ct^*$. Since $\mathcal{A}'_{sup}$ is reduced least-restrictive with respect to $\mathcal{A}_{sup}$, there exists a path $\hat{s} \xrightarrow{\alpha_4}{}^* s_k \xrightarrow{A_{src}\alpha_5}{}^* s_l \xrightarrow{A_{snk}\alpha_6}{}^* s_m$ with $\alpha_4, \alpha_5, \alpha_6 \in \mathcal{A}ct^*$ in $\mathcal{A}'_{sup}$ where $\alpha_1 A_{src} \alpha_2 A_{snk} \alpha_3 \equiv_{\hat{s},u} \alpha_4 A_{src} \alpha_5 A_{snk} \alpha_6$. In $\mathcal{S}'$, we can therefore find run fragment $\rho'[..l+1] = \hat{c}A_1 \dots c_k A_{src} \dots c_l A_{snk} c_{l+1}$, for which $\lambda(\rho', k, l, r) = \lambda(\rho, i, j, r)$. Since this can be done for any source-sink occurrence pair, the suprema are the same and $\lambda_{max}(\mathcal{S}) = \lambda_{max}(\mathcal{S}')$.
- $\tau_{min}(\mathcal{S}) = \tau_{min}(\mathcal{S}')$**:** As stated in Section III, $\tau_{min}(\mathcal{S}) = \inf_{cycle \in cycles(\mathcal{S})} \mathsf{Ratio}(cycle)$. Let $cycle_{min}$ be the cycle in $\mathcal{S}$ such that $\mathsf{Ratio}(cycle_{min}) = \tau_{min}(\mathcal{S})$. Then, there is a run $\rho \in \mathcal{R}(\mathcal{S})$ with $\hat{c} \xrightarrow{\alpha_1}{}^* (c_i \xrightarrow{\alpha_2}{}^*)^\omega$, where $c_i = \langle s_i, \boldsymbol{\gamma}_i \rangle$ is a configuration on cycle $cycle_{min} = c_i \xrightarrow{\alpha_2}{}^* c_i$, $\alpha_1, \alpha_2 \in \mathcal{A}ct^*$, and α_2 is the sequence of activities on cycle $cycle_{min}$. Since $\mathcal{A}_{sup}$ is nonblocking, by Def. 8, there exists a path $\hat{s} \xrightarrow{\alpha_1}{}^* s_i \xrightarrow{\alpha_2}{}^* s_i \xrightarrow{\alpha_3}{}^* s_m$ in $\mathcal{A}_{sup}$ for some marked state $s^m \in S^m$ and $\alpha_1, \alpha_2, \alpha_3 \in \mathcal{A}ct^*$. We can extend this path to $\hat{s} \xrightarrow{\alpha_1}{}^* (s_i \xrightarrow{\alpha_2}{}^*)^N s_i \xrightarrow{\alpha_3}{}^* s_m$, for arbitrary $N \geq 0$, where we traverse N times through the cycle. Since $\mathcal{A}'_{sup}$ is reduced least-restrictive with respect to $\mathcal{A}_{sup}$, there exists a path $\hat{s} \xrightarrow{\beta}{}^* s_m$ with $\beta \in \mathcal{A}ct^*$ in $\mathcal{A}'_{sup}$ where $\alpha_1 \alpha_2^N \alpha_3 \equiv_{\hat{s},u} \beta$. For each such path, there is a corresponding run $\rho' \in \mathcal{R}(\mathcal{S}')$ with prefix $\rho'[..m] = \hat{c}\beta c_m$. We show that in $\mathcal{S}'$ there is a run that contains a cycle with the same ratio $\mathsf{Ratio}(cycle_{min})$ as in $\mathcal{S}$. If $N \geq |S'|$, there exist β_1, β_2, β_3 with $\beta_1\beta_2\beta_3 \equiv_{\hat{s},u} \alpha_1\alpha_2^N\alpha_3$, $|\beta_1| \geq |\alpha_1|$, $|\beta_3| \geq |\alpha_3|$ and the length $|\beta_2| = k \cdot |\alpha_2|$, such that $\rho' \in \mathcal{R}(\mathcal{S}')$ with $\rho'[..m] = \hat{c} \xrightarrow{\beta_1}{}^* c_j \xrightarrow{\beta_2}{}^* c_j \xrightarrow{\beta_3}{}^* c_m$. Let $cycle'_{min}$ denote the cycle $c_j\beta_2 c_j$ in $\mathcal{S}'$. Then for all $m > 0$, $\alpha_1\alpha_2^{N+(m-1)\cdot k}\alpha_3 \equiv_{\hat{s},u} \beta_1\beta_2^m\beta_3$. Since the former repeats the cycle $cycle_{min}$ and the latter repeats the cycle $cycle'_{min}$ both cycles have the same cycle ratio. This shows that $\tau_{min}(\mathcal{S}) \geq \tau_{min}(\mathcal{S}')$, since $\mathsf{Ratio}(cycle_{min}) = \mathsf{Ratio}(cycle^N_{min}) = \mathsf{Ratio}(cycle'_{min})$. Clearly, also $\tau_{min}(\mathcal{S}') \geq \tau_{min}(\mathcal{S})$ as $\mathcal{S}'$ is included in $\mathcal{S}$. Therefore, $\tau_{min}(\mathcal{S}') = \tau_{min}(\mathcal{S})$.

□

Proofs for Section VI (On-the-fly reduction)

Theorem 38. *Let ample be a cluster-inspired ample reduction on $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$. Then ample satisfies conditions (A1), (A2), (A3), (A4), (A5.1), and (A5.2).*

Proof. Consider any state s in the composition of $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$. First consider the case where $enabled(s) = \emptyset$. Then by Def. 37 $ample(s) = \emptyset$, and conditions (A1) and (A2) are satisfied. Now assume that $enabled(s) \neq \emptyset$, and let $ample(s) = enabled_{\mathcal{C}}(s)$, where $\mathcal{C}$ is any safe cluster in s. Since cluster $\mathcal{C}$ is safe, by Def. 36 it satisfies conditions (C1), (C2.1) and (C2.2). We need to show that $enabled_{\mathcal{C}}(s)$ satisfies conditions (A1) and (A2).

Condition (A1) follows directly from the definition. We prove condition (A2) by contraposition. Assume that condition (A2) does not hold. This means that there exists a finite run fragment $\rho = s \xrightarrow{A_1} s_1 \xrightarrow{A_2} s_2 \xrightarrow{A_3} \ldots s_{n-1} \xrightarrow{A_n}$, where $A_1 \ldots A_{n-1}$ are resource-independent with $ample(s) = enabled_{\mathcal{C}}(s)$, and A_n is resource-dependent with some activity in $enabled_{\mathcal{C}}(s)$.

Since A_n is resource-dependent with some activity in $enabled_{\mathcal{C}}(s)$, by condition (C2.1), $A_n \in \mathcal{A}ct(\mathcal{C})$. Moreover, we have $A_n \notin enabled_{\mathcal{C}}(s)$. Since activities $A_1 \ldots A_{n-1}$ are resource-independent with $ample_C(s)$, $A_1 \ldots A_{n-1} \in \mathcal{A}ct(\mathcal{A}) \setminus \mathcal{A}ct(\mathcal{C})$ and they do not affect the state of $\mathcal{C}$, which means that $\pi_{\mathcal{C}}(s)$ does not change in the first $n-1$ steps. As $A_n \in enabled(s_{n-1})$, $A_n \in enabled(\pi_{\mathcal{C}}(s))$. We also have that $A_n \notin enabled_{\mathcal{C}}(s)$, since otherwise $A_n \in enabled_{\mathcal{C}}(s)$, contradicting our assumption. This means that A_n becomes enabled in $\pi_{\mathcal{C}}(s)$ by executing one of the activities in set $A_1, \ldots, A_{n-1}$. Since $A_1, \ldots, A_{n-1}$ are activities outside of $\mathcal{C}$, there must be some A_i with $1 \leq i \leq n-1$ that enabled A_n, which can only happen if A_n occurs outside of cluster $\mathcal{C}$ by definition of synchronous composition. This contradicts condition (C2.2).

Conditions (A3), (A4), (A5.1), and (A5.2) follow directly from conditions (C3), (C4), (C5.1), and (C5.2). □

Theorem 39. *Let s be a state in the composition $\mathcal{M} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ and $A \in enabled(s)$ be the candidate activity. Then,* ComputeCluster(s, A) *returns a safe cluster in s.*

Proof. Let $\mathcal{C}_1$ denote the cluster after executing lines 5-6. Set $enabled(\pi_{\mathcal{C}_1}(s))$ contains all uncontrollable enabled activities from $enabled(s) \cap \mathcal{A}ct_u$ and the special activity ω if it is enabled in state s. This means that conditions (C3) and (C4) are satisfied for $\mathcal{C}_1$ and any superset of $\mathcal{C}_1$.

Now, consider activity A and let $\mathcal{C}_k$ denote the value of $\mathcal{C}$ at line 8, and $\mathcal{C}_{k+1}$ the new cluster after executing lines 8-23. We show that conditions (C2.1) and (C2.2) hold for A in $\mathcal{C}_{k+1}$ and any superset of $\mathcal{C}_{k+1}$. We consider two cases:

- case $A \in enabled(s)$: $\mathcal{C}_{k+1}$ contains all (max,+) automata $\mathcal{A}_i$ with $A \in \mathcal{A}ct_i$ (added at line 12), which means that condition (C2.2) is satisfied for A. Since there are no (max,+) automata outside $\mathcal{A}_i$ with A, condition (C2.2) holds also for any superset of $\mathcal{C}_{k+1}$. After executing lines 13-15, there is no resource-dependent activity outside cluster $\mathcal{C}_{k+1}$, which means that condition (C2.1) is satisfied. It suffices to add only one automaton with such an activity to the cluster to satisfy condition (A2.1), since then this resource-dependent activity becomes part of the cluster. Condition (C2.1) is also satisfied in any superset of $\mathcal{C}_{k+1}$.
- case $A \notin enabled(s) \wedge A \in enabled(\pi_{\mathcal{C}}(s))$: after executing lines 17-19, condition (C2.1) trivially holds since $A \notin enabled(s)$. Since $A \notin enabled(s)$ and $A \in enabled(\pi_{\mathcal{C}}(s))$, there exists at least one (max,+) automaton $\mathcal{A}_i \notin \mathcal{C}_k$ where $A \notin enabled(\pi_i(s))$. Let $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \{\mathcal{A}_i\}$, obtained at line 19. Then $A \notin enabled(\pi_{\mathcal{C}_{k+1}}(s))$, which means that condition (C2.2) holds. Since $\mathcal{A}_i \in \mathcal{C}_{k+1}$, A will remain disabled in any superset of $\mathcal{C}_{k+1}$.

After executing lines 8-19 for activity A, conditions (C2.1) and (C2.2) hold for A and keep holding for any extension to the cluster.

In line 10 we check whether $A \in \mathcal{U}$. If this is the case, then possibly an uncontrollable event becomes enabled in state $A(s)$, and the set of all (max,+) automata is returned. This guarantees that condition (C5.1) is never violated. Furthermore, it guarantees that condition (C5.2) is never violated, since it guarantees that no uncontrollable activity becomes enables after executing A and subsequently some other resource-independent activity from s.

Upon termination of the algorithm, we obtain some cluster $\mathcal{C}_l$ where conditions (C1) till (C5.2) hold for each activity in $enabled(\pi_{\mathcal{C}_l}(s))$. From this it follows that $\mathcal{C}_l$ is safe in s. □