

# Partial-Order Reduction for Supervisory Controller Synthesis

**Citation for published version (APA):**

van der Sanden, L. J., Geilen, M. C. W., Reniers, M. A., & Basten, A. A. (2022). Partial-Order Reduction for Supervisory Controller Synthesis. *IEEE Transactions on Automatic Control*, 67(2), 870-885.  
<https://doi.org/10.1109/TAC.2021.3129161>

**DOI:**

[10.1109/TAC.2021.3129161](https://doi.org/10.1109/TAC.2021.3129161)

**Document status and date:**

Published: 01/02/2022

**Document Version:**

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

**Please check the document version of this publication:**

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

**General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

[www.tue.nl/taverne](http://www.tue.nl/taverne)


**Take down policy**

If you believe that this document breaches copyright please contact us at:

[openaccess@tue.nl](mailto:openaccess@tue.nl)

providing details and we will investigate your claim.

# Partial-Order Reduction for Supervisory Controller Synthesis

Bram van der Sanden , Marc Geilen , *Member, IEEE*, Michel Reniers , *Senior Member, IEEE*, and Twan Basten , *Senior Member, IEEE*

**Abstract**—A key challenge in the synthesis and subsequent 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 (POR) is a well-established technique that alleviates this issue in the field of model checking. It does so by exploiting redundancy in the model with respect to the properties of interest. For controller synthesis, the functional properties of interest are nonblockingness, controllability, and least-restrictiveness, but also performance properties, such as throughput and latency are of interest. We propose an on-the-fly POR on the input model that preserves both functional and performance properties in the synthesized supervisory controller. This improves the scalability of the synthesis (and any subsequent performance analysis). Synthesis experiments show the effectiveness of the POR on a set of realistic manufacturing system models.

**Index Terms**—Control system analysis, control systems, system analysis and design, systems engineering and theory, supervisory control, system performance.

## I. INTRODUCTION

**S**UPERVISORY 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 specification states of the combined plant and requirement automata, and  $\Sigma$  is 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 the scalability of the synthesis and subsequent 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 of the composite automaton, a subset of redundant-enabled transitions is removed, in turn, reducing the number of reachable states. Synthesis can then be performed on this smaller automaton leading to a smaller supervisor, and a smaller state space for performance analysis.

Our POR aims to preserve functional and performance aspects. As functional properties, we consider controllability, non-blockingness, and least-restrictiveness. The supervisor needs to be *controllable* with respect to the plant, meaning that it should not disable uncontrollable events. Further, the composition of the supervisor with the plant, the *controlled system*, should be *nonblocking*, meaning that from every reachable state in the controlled system, a *marked state* can be reached. This typically indicates the completion of an operation. The supervisor is (non)blocking if the controlled system is (non)blocking [3]. Finally, the supervisor should be *least-restrictive*, meaning that it restricts the system as little as possible while still being controllable and nonblocking. As performance aspects, we consider throughput and latency. *Throughput* describes the system production rate, for example the number of products produced by the system per hour. *Latency* describes the temporal distance between certain events, for example the time between the start and end of processing a product.

Fig. 1 shows the proposed POR approach. The system behavior is modeled by plant  $\mathcal{P}$ , which is a composition of automata  $P^1 \parallel \dots \parallel P^k$ . Synthesis can be applied on  $\mathcal{P}$  directly to obtain a supervisor  $\mathcal{A}_{\text{sup}}$  (an automaton). Alternatively, using

Manuscript received November 11, 2019; revised August 19, 2020; accepted December 13, 2020. Date of publication November 18, 2021; date of current version January 28, 2022. This work was supported by NWO-AES through the Robust Cyber-Physical Systems (RCPS) program under Grant 12694 and in part by ESI (TNO) through the Maestro project with ASML as the Carrying Industrial Partner. Recommended by Associate Editor Dr. Jan Komenda. (*Corresponding author: Bram van der Sanden.*)

Bram van der Sanden was with the Eindhoven University of Technology, Eindhoven, The Netherlands. He is now with ESI (TNO), HTC25, 5656AE Eindhoven, The Netherlands (e-mail: bram.vandersanden@tno.nl).

Marc Geilen and Michel Reniers are with the Eindhoven University of Technology, 5600MB Eindhoven, The Netherlands (e-mail: m.c.w.geilen@tue.nl; M.A.Reniers@tue.nl).

Twan Basten is with the Eindhoven University of Technology, Eindhoven, The Netherlands, and also with ESI (TNO), Eindhoven, The Netherlands (e-mail: a.a.basten@tue.nl).

Color versions of one or more figures in this article are available at <https://doi.org/10.1109/TAC.2021.3129161>.

Digital Object Identifier 10.1109/TAC.2021.3129161

<sup>1</sup>For the purpose of supervisory controller synthesis, we assume that all requirements are translated into plant automata using a *plantify* transformation [7], such that we can treat all automata similarly.

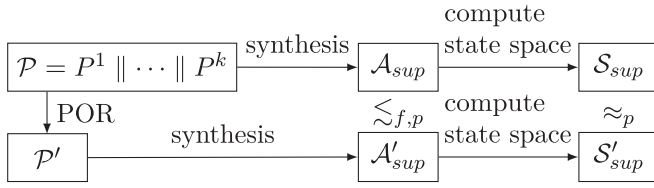


Fig. 1. POR approach.

POR, 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 Definition 25). 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 (see Definition 10) that considers redundancy. The relation also guarantees that the performance aspects are preserved in the corresponding timed state spaces  $\mathcal{S}_{sup}$  and  $\mathcal{S}'_{sup}$ , denoted  $\mathcal{S}_{sup} \approx_p \mathcal{S}'_{sup}$ . We, moreover, provide a concrete POR algorithm (Algorithm 1) that gives a valid on-the-fly reduction to compute reduced plant  $\mathcal{P}'$  during composition.

The proposed POR extends the POR introduced in [8] for the performance analysis of timed systems. We consider controller synthesis and the preservation of both functional and performance properties during synthesis. An integrated overview of both approaches can be found in [9].

We follow [10] 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 position, consisting of several actions on one or more motor resources in a specific order. Actions of different activities may not interfere with each other except through resource claims and releases. During the execution of an activity, the relevant resources are claimed and no interference on these resources is possible. Resources are assigned to activities in the order in which activities are executed. Activities that use different sets of resources may execute concurrently. Activities can then be treated as atomic events, abstracting from the execution orders of activity-internal concurrent actions. As shown in [10], this improves the scalability of controller synthesis.

Various well-known ways to capture the timing behavior of supervisory controllers are real-valued clocks as used in timed automata [11], discrete-valued clocks as used in tick-based models [12], and (max,+) algebra [13]–[15]. We use (max,+) algebra (see for instance [16]), 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 [15]. Supervisor synthesis on (max,+) 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 the scalability of the supervisor synthesis. It preserves both functional and performance properties, which in turn improves scalability of any subsequent performance analysis. Our POR technique can also be used on conventional finite-state automata with events, by assuming activities do not claim or release resources and are assigned the empty  $0 \times 0$  (max,+) timing matrix (implying that they are timeless).

The rest of this article is organized as follows. Section II introduces the modeling framework. Section III defines the functional and performance aspects considered in the POR. Section IV introduces performance equivalence of timed state spaces. Section V defines the POR conditions and shows that these conditions preserve the desired aspects. Section VI introduces an on-the-fly reduction that uses local conditions to compute a reduced automaton directly from a composition of automata. The experimental evaluation in Section VII shows the effectiveness of our POR technique. Related work is described in Section VIII. Finally, Section IX concludes this article.

## 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 is described by a (max,+) matrix, as explained in the following. The (max,+) automata of the running example are shown in Fig. 2. We use a representation of (max,+) automata that extends the definition of [15] with rewards, and we restrict ourselves to a setting with deterministic (max,+) automata. This corresponds to the construction discussed in Section VI in [15], 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 [15])):* A (max,+) automaton  $\mathcal{A}$  is a tuple  $\langle S, \hat{s}, S^m, Act, \text{reward}, M, T \rangle$ , 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*,  $Act$  is a nonempty set of activities,  $\text{reward} : Act \rightarrow \mathbb{R}^{\geq 0}$  quantifies the progress per activity,  $M$  maps each activity to its associated (max,+) matrix, and  $T \subseteq S \times Act \times S$  is the transition relation. We assume that  $\mathcal{A}$  is deterministic; for any  $s, s', s'' \in S$  and  $A \in Act$ ,  $\langle s, A, s' \rangle \in T$  and  $\langle s, A, s'' \rangle \in T$  imply  $s' = s''$ .

Both plants and supervisors are described as (max,+) automata. Marked states define a notion of completion of the plant for supervisor synthesis, as illustrated in the following.

Let set  $Act^*$  contain all finite *strings* over  $Act$ , including the empty string  $\varepsilon$ . We write  $s_1 \rightarrow^A s_2$  if  $\langle s_1, A, s_2 \rangle \in T$ . The transition relation is extended to  $Act^*$  in the relation  $\rightarrow^*$  by

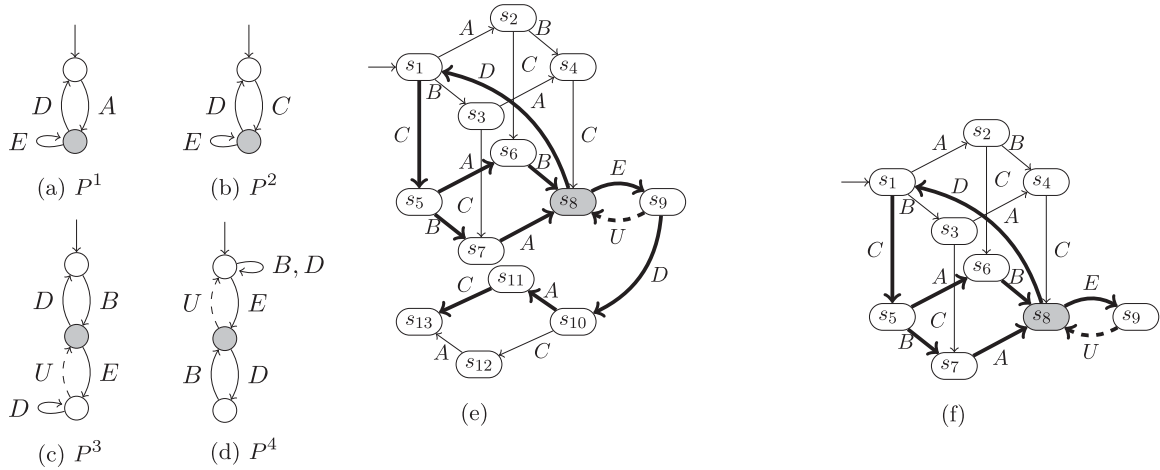


Fig. 2. Running example: plants  $P^1$ ,  $P^2$ ,  $P^3$ , and  $P^4$ , and the composition  $\mathcal{P}$ . Transitions of uncontrollable activities are denoted with dashed arrows. Marked states are indicated in gray.

letting  $s \xrightarrow{\varepsilon} s$  for all  $s \in S$ , and for all  $\alpha \in Act^*$ ,  $A \in Act$ ,  $s, s' \in S$ , and  $s \xrightarrow{\alpha A} s'$  if  $s \xrightarrow{\alpha A} s''$  and  $s'' \xrightarrow{A} s'$  for some  $s'' \in S$ . We write  $s \xrightarrow{*} s'$  if  $s \xrightarrow{\alpha} s'$  for some  $\alpha \in Act^*$ . Set  $enabled(s) = \{A \in Act \mid \exists s' : s \xrightarrow{A} s'\}$  contains all activities *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  $A(s)$ . 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. A possible behavior of an automaton is described in a *path*. A(n) (in)finite path  $p$  of  $\mathcal{A}$  is a(n) (in)finite, alternating sequence of states and activities:  $p = s_0 A_1 s_1 A_2 s_2 A_3 \dots$  such that  $s_0 = \hat{s}$  and  $s_{i+1} = A_{i+1}(s_i)$  for all  $i \geq 0$ . A finite path always ends in a state. Given path  $p$  and  $i \geq 0$ ,  $p[\dots i]$  denotes prefix  $s_0 A_1 \dots s_i$ , and  $p[i, j] = s_i A_{i+1} \dots s_j$  with  $0 \leq i \leq j$  is the path fragment from state  $s_i$  up to  $s_j$ . Further,  $p[i]$  denotes state  $s_i$  in  $p$ .

**Definition 2 (Synchronous Composition of (Max,+ Automata):** Given automata  $\mathcal{A}_1 = \langle S_1, \hat{s}_1, S_1^m, Act_1, reward_1, M_1, T_1 \rangle$  and  $\mathcal{A}_2 = \langle S_2, \hat{s}_2, S_2^m, Act_2, reward_2, M_2, T_2 \rangle$ , with for each activity  $A \in Act_1 \cap Act_2$ ,  $reward_1(A) = reward_2(A)$  and  $M_1(A) = M_2(A)$ , the synchronous composition  $\mathcal{A}_1 \parallel \mathcal{A}_2 = \langle S_1 \times S_2, \langle \hat{s}_1, \hat{s}_2 \rangle, S_1^m \times S_2^m, Act_1 \cup Act_2, reward_1 \cup reward_2, M_1 \cup M_2, T_{12} \rangle$ , with

$$\langle s_1, s_2 \rangle \xrightarrow{A}_{12} \langle s_1', s_2' \rangle \text{ if } A \in Act_1 \cap Act_2, s_1 \xrightarrow{A}_1 s_1', s_2 \xrightarrow{A}_2 s_2'$$

$$\langle s_1, s_2 \rangle \xrightarrow{A}_{12} \langle s_1', s_2 \rangle \text{ if } A \in Act_1 \setminus Act_2, s_1 \xrightarrow{A}_1 s_1'$$

$$\langle s_1, s_2 \rangle \xrightarrow{A}_{12} \langle s_1, s_2' \rangle \text{ if } A \in Act_2 \setminus Act_1, s_2 \xrightarrow{A}_2 s_2'$$

Fig. 2(e) 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  $D$ . The synchronous composition of deterministic automata is again deterministic.

$$\begin{array}{ccc} \begin{bmatrix} 4 & 5 & -\infty \\ -\infty & 3 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix} & \begin{bmatrix} 1 & 3 & -\infty \\ 1 & 3 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix} & \begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 4 \end{bmatrix} \\ M_A & M_B & M_C \\ \\ \begin{bmatrix} 2 & -\infty & 3 \\ -\infty & 0 & -\infty \\ 2 & -\infty & 3 \end{bmatrix} & \begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 5 \end{bmatrix} & \begin{bmatrix} 2 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix} \\ M_D & M_E & M_U \end{array}$$

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

Activities abstract from low-level activity-internal action executions. For our intended supervisor synthesis and performance analysis, we only need 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 release time of a resource in terms of all the availability times of resources. To illustrate, consider matrix  $M_A$  of activity  $A$ . The first row describes the release time of resource  $r_1$  in terms of when resources  $r_1, r_2$ , and  $r_3$  are available at the start of executing  $A$ . The execution of  $A$  implies a time delay of 4 time units between the claiming of  $r_1$  and its subsequent release. Similarly, a delay of 5 occurs between the claiming of  $r_2$  and the release of  $r_1$ . There is no dependency on the availability of  $r_3$ , indicated by  $-\infty$ . We define a (global) resource set  $Res = \{r_i \mid 1 \leq i \leq s\}$ , where  $s$  is the size of the matrices. In our running example, we have  $Res = \{r_1, r_2, r_3\}$ . Function  $R$  maps each activity to the set of resources used. For activity  $A$  with (max,+) matrix  $M_A$ ,  $r \notin R(A)$  iff  $[M_A]_{r,r} = 0$ ,  $[M_A]_{i \neq r, r} = -\infty$ , and  $[M_A]_{r, j \neq r} = -\infty$ . For example, given  $M_B$  in Fig. 3,  $R(B) = \{r_1, r_2\}$ . As  $r_3 \notin R(B)$ ,  $r_3$  does not influence the result of multiplying a vector with  $M_B$ .

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

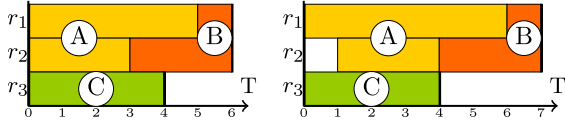


Fig. 4. Gantt chart of activity sequence  $ABC$  when all resources are initially available (left) and when resource  $r_2$  is available after 1 time unit (right).

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  $m \times p$  matrix  $\mathbf{A}$  and  $p \times n$  matrix  $\mathbf{B}$ ,  $\mathbf{A} \otimes \mathbf{B}$  denotes  $(\max, +)$  matrix multiplication, resulting in matrix  $\mathbf{A} \otimes \mathbf{B}$  with elements  $[\mathbf{A} \otimes \mathbf{B}]_{ij} = \max_{k=1}^p ([\mathbf{A}]_{ik} + [\mathbf{B}]_{kj})$ . Adding a constant  $c$  to matrix  $\mathbf{A}$  yields a new matrix  $\mathbf{A} + c$  with  $[\mathbf{A} + c]_{ij} = [\mathbf{A}]_{ij} + c$ . For any vector  $\mathbf{x}$  of size  $n$ ,  $\|\mathbf{x}\| = \max_{i=1}^n [\mathbf{x}]_i$  denotes the vector norm of  $\mathbf{x}$ . For vector  $\mathbf{x}$ , with  $\|\mathbf{x}\| > -\infty$ ,  $norm(\mathbf{x})$  denotes  $\mathbf{x} - \|\mathbf{x}\|$ , the normalized vector, with  $\|norm(\mathbf{x})\| = 0$ .  $\mathbf{0}$  denotes a vector with only zero entries.

Resource availability times are captured in a  $(\max, +)$  vector, typically denoted  $\gamma$ . 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. Timing evolution is, therefore, expressed using  $(\max, +)$  vector-matrix multiplication. When all resources are initially available at time 0, captured in vector  $\mathbf{0}$ , the new availability times of the resources after executing  $A$  are computed as follows:

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

Resources  $r_1$  and  $r_2$  are available again after 5 and 3 time units, respectively. Resource  $r_3$  is not present in  $R(A)$ , but a row is present for  $r_3$  to carry over its time stamp. Its availability time stays 0, since it is not used by  $A$ . The timing semantics of an activity sequence is defined in terms of repeated matrix multiplication. For example, the resource availability after executing activity sequence  $ABC$  given vector  $\mathbf{0}$  (see Fig. 4) is computed as  $\mathbf{M}_C \otimes \mathbf{M}_B \otimes \mathbf{M}_A \otimes \mathbf{0} = [6, 6, 4]^T$ . This representation of activity timing generalizes the well-known heaps-of-pieces model [17], [18], where pieces have a rigid structure (i.e., have a fixed shape). In our approach, however, the  $(\max, +)$  matrices encode flexible pieces, as illustrated in Fig. 4 for  $A$ .

We define the input model for our POR as a composition of  $(\max, +)$  automata. The composition of all the individual automata is again an automaton. We assume that the matrices of the constituent automata all have the same size to ensure that they can be multiplied. Additional resources can 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.

**Definition 3 ((Max,+) Timed System):** A  $(\max, +)$  timed system  $\mathcal{M}$  is described by  $\mathcal{M} = \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$  with  $(\max, +)$  automata  $\mathcal{A}_i = \langle S_i, \hat{s}_i, S_i^m, Act_i, reward_i, M_i, T_i \rangle$  with  $1 \leq i \leq n$ . We assume that all matrices have size  $|\text{Res}| \times |\text{Res}|$  and

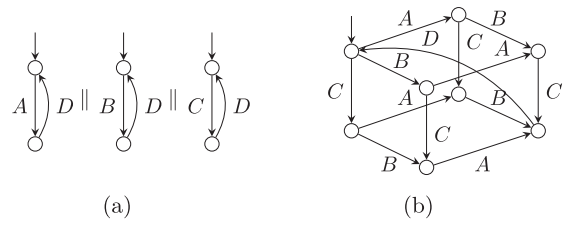


Fig. 5. Running example with  $(\max, +)$  automata  $\bar{P}^1$ ,  $\bar{P}^2$ , and  $\bar{P}^3$ , and composition  $\bar{P}$  (a)  $\bar{P}^1, \bar{P}^2, \bar{P}^3$ . (b)  $\bar{P}^1 \parallel \bar{P}^2 \parallel \bar{P}^3$ .

that for all  $1 \leq i, j \leq n$  and each activity  $A \in Act_i \cap Act_j$ ,  $reward_i(A) = reward_j(A)$ , and  $M_i(A) = M_j(A)$ .

A  $(\max, +)$  automaton can be interpreted as a normalized  $(\max, +)$  state space that captures each path of the automaton as a *run*, 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  $c = \langle s, \gamma \rangle$  in the state space consists of a state  $s$  of the  $(\max, +)$  automaton and a normalized resource availability vector  $\gamma$ . Fig. 6 shows the normalized  $(\max, +)$  state space of a simplified running example, discussed in more detail in the following.

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

- 1) set  $C = S \times \mathbb{R}^{-\infty|\text{Res}|}$  of configurations;
- 2) initial configuration  $\hat{c} = \langle \hat{s}, \mathbf{0} \rangle$ ;
- 3) labeled transition relation  $\Delta \subseteq C \times Act \times C$  that consists of the transitions in the set  $\{ \langle \langle s, \gamma \rangle, A, \langle s', norm(\gamma') \rangle \rangle \mid s \xrightarrow{A} s' \wedge \gamma' = M(A) \otimes \gamma \}$ ;
- 4) function  $w_1$  that assigns a weight  $w_1(c, A, c') = reward(A)$  to each transition  $\langle c, A, c' \rangle \in \Delta$ ; and
- 5) function  $w_2$  that assigns a weight  $w_2(c, A, c') = \|M(A) \otimes \gamma\|$  to each transition  $\langle c, A, c' \rangle \in \Delta$  with  $c = \langle s, \gamma \rangle$  indicating the time passed during execution.

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

**Proposition 5:** Let  $\mathcal{S}$  be a normalized  $(\max, +)$  state space with run  $\rho$  such that  $c_i = \langle s_i, \gamma_i \rangle$  for each  $i$ . Then, for all  $n \geq 0$ ,  $\bar{\gamma}_n = \sum_{k=0}^{n-1} w_2(c_k, A_{k+1}, c_{k+1}) + \gamma_n$ .

*Proof:* Proved with induction over  $n$  using Definition 4 and the property that  $c + M \otimes \gamma = M \otimes (\gamma + c)$  given a constant  $c$ , matrix  $M$ , and vector  $\gamma$ .

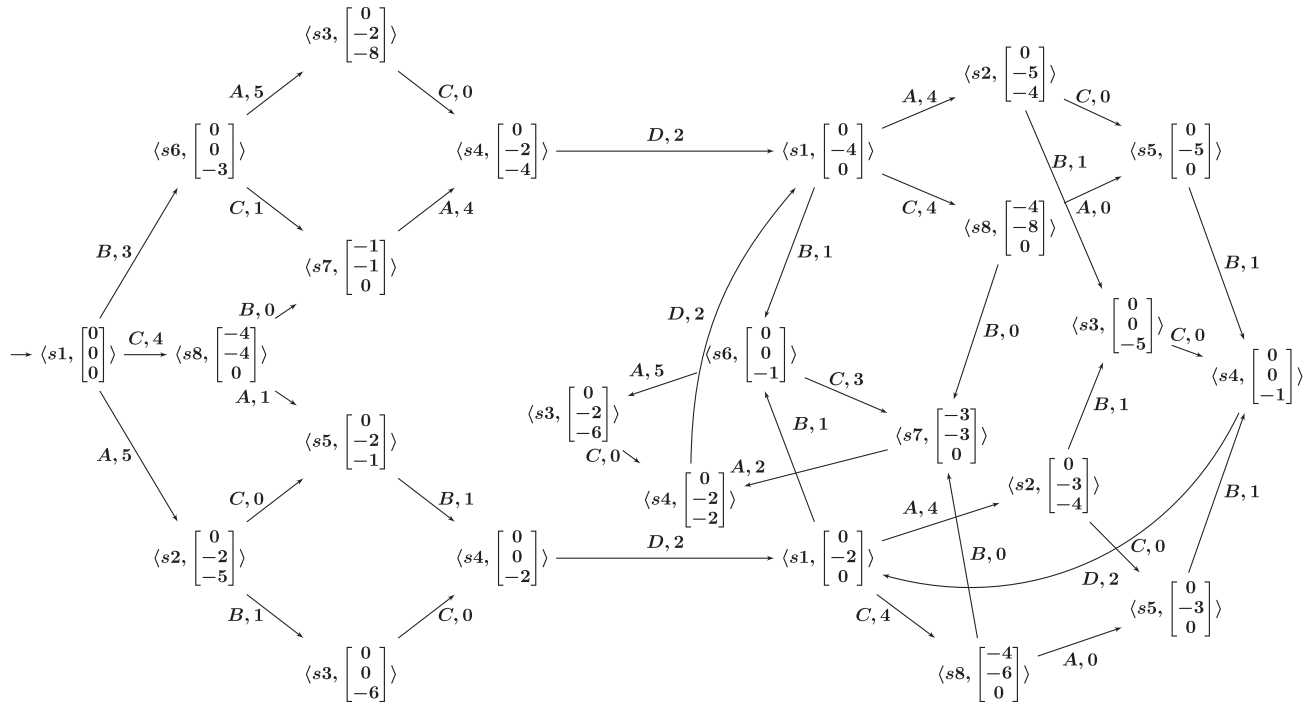


Fig. 6. Normalized (max,+) state space of  $\bar{\mathcal{P}}$  [Fig. 5(b)]. Transitions are annotated with the activity and  $w_2$  value.

In the remainder, we restrict ourselves to the reachable part of the normalized (max,+) state space, and we assume that this is finite. The reachable part might be infinite [19]. It is guaranteed to be finite, however, if for each simple cycle (no repetition of transitions is allowed) in the (max,+) automaton with corresponding activity sequence  $A_1 \dots A_k$ , the matrix  $M_{A_k} \otimes \dots \otimes M_{A_1}$  is irreducible. This can be practically verified in polynomial time by finding the simple cycles [20] and checking that the corresponding matrices have no entries  $-\infty$ . Practically, this means 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 independently.

### III. PROPERTIES TO BE PRESERVED

In this section, we introduce the properties that we want to preserve: controllability, nonblockingness, and least-restrictiveness as functional properties, and latency and throughput as performance properties.

#### A. Controllability, Nonblockingness, and Least-Restrictiveness

In supervisory control, the activity alphabet  $Act$  is partitioned into set  $Act_c$  of controllable activities and set  $Act_u$  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 a reduction of a given (max,+) automaton, where a subset of the states and transitions is preserved.

**Definition 6 (Subautomaton):** Let  $\mathcal{A}_i = \langle S_i, \hat{s}, S_i^m, Act, reward, M, T_i \rangle$  for  $i \in \{1, 2\}$  be two (max,+) automata with the same alphabet  $Act$  (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  $S_1^m = S_1 \cap S_2^m$ . The latter ensures that marking in  $\mathcal{A}_1$  is consistent with marking in  $\mathcal{A}_2$ .

**Definition 7 (Union of (Max,+) Automata):** Let  $\mathcal{A}_i = \langle S_i, \hat{s}, S_i^m, Act, reward, M, T_i \rangle$ ,  $i \in I$ , 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  $\bigcup_{i \in I} \mathcal{A}_i = \langle \bigcup_{i \in I} S_i, \hat{s}, \bigcup_{i \in I} S_i^m, Act, reward, M, \bigcup_{i \in I} T_i \rangle$ .

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  $\mathcal{A} = \langle S, \hat{s}, S^m, Act, reward, M, T \rangle$  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  $\mathcal{P}$  in Fig. 2(e) is blocking, since states  $s_{10}, s_{11}, s_{12}$ , and  $s_{13}$  are blocking.  $\mathcal{A}_{sup}$  in Fig. 2(f) is nonblocking, as it is always possible to return to  $s_8$ . This shows how marked states can be used to ensure a notion of completion.

We assume that  $\mathcal{A}_{sup} \preceq \mathcal{P}$ , since the computed supervisor is often a restriction of the plant by disabling transitions that lead to undesired behavior, violating the nonblocking requirement or the controllability requirement, defined in the following. 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 in any state reachable in the controlled system.

**Definition 9 (Controllability):** Let  $\mathcal{P} = \langle S_{\mathcal{P}}, \hat{s}_{\mathcal{P}}, S_{\mathcal{P}}^m, Act_{\mathcal{P}}, reward_{\mathcal{P}}, M_{\mathcal{P}}, T_{\mathcal{P}} \rangle$  and  $\mathcal{A} = \langle S_{\mathcal{A}}, \hat{s}_{\mathcal{A}}, S_{\mathcal{A}}^m, Act_{\mathcal{A}}, reward_{\mathcal{A}}, M_{\mathcal{A}}, T_{\mathcal{A}} \rangle$  be  $(\max, +)$  automata, with  $Act_u \subseteq Act_{\mathcal{P}} \cup Act_{\mathcal{A}}$  the set of uncontrollable activities.  $\mathcal{A}$  is *controllable with respect to*  $\mathcal{P}$  iff, for every string  $\alpha \in (Act_{\mathcal{P}} \cup Act_{\mathcal{A}})^*$ , every state  $s \in S_{\mathcal{A}}$ , and every  $U \in Act_u$  such that  $\hat{s}_{\mathcal{A}} \xrightarrow{\alpha} s$  and  $\hat{s}_{\mathcal{P}} \xrightarrow{\alpha U} s'$  for some  $s' \in S_{\mathcal{P}}$ , it holds that  $s \xrightarrow{U} s''$  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  $s_1 \xrightarrow{CABE} s_9$  in supervisor  $\mathcal{A}'_{\text{sup}}$  shown in Fig. 2(f). 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  $\mathcal{A}'_{\text{sup}}$ , activity  $U$  is also present in state  $s_9$ . Since this also holds for the other strings and uncontrollable activities,  $\mathcal{A}'_{\text{sup}}$  is controllable with respect to  $\mathcal{P}$ .

The union of controllable and nonblocking subautomata of a given automaton is again controllable and nonblocking [7]. 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  $\text{supCN}(\mathcal{A}) = \bigcup_{\mathcal{A}' \text{ s.t. } CN(\mathcal{A}', \mathcal{A})} \mathcal{A}'$ .  $\mathcal{A}'$  is *least restrictive* with respect to  $\mathcal{A}$  iff  $\mathcal{A}' = \text{supCN}(\mathcal{A})$ .

Both  $\mathcal{A}_{\text{sup}}$  and  $\mathcal{A}'_{\text{sup}}$  in Fig. 2(f) are nonblocking and controllable with respect to  $\mathcal{P}$ . Given  $\mathcal{P}$ , calculating  $\mathcal{A}_{\text{sup}} = \text{supCN}(\mathcal{P})$  is called *synthesis*. Synthesis algorithms implement fixed-point computations [1], [2], where in each iteration *bad* states are removed. Bad states are those states that are blocking or that lead to a blocking state through uncontrollable activities. 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 corresponding normalized  $(\max, +)$  state space. Throughput is defined as the ratio between the cumulative reward and the cumulative execution time of a run. For throughput analysis, we only consider infinite runs. 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 reachable state has at least one outgoing transition. A nontotal supervisor has a throughput lower bound of zero. The presented latency analysis applies to nontotal supervisors as well. As supervisor synthesis guarantees that from each state a marked state is reachable, the supervisor is total if each marked state has at least one outgoing transition. A total supervisor can be seen as an

$\omega$ -automaton [21] accepting infinite words over  $Act$ . There are no specific acceptance conditions, so any infinite word starting in the initial state is accepted. Marked states are not used to define acceptance conditions. For the performance analysis of a 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 cumulative reward (sum of  $w_1$  weights) and the cumulative execution time (sum of  $w_2$  weights).

**Definition 11 (Throughput):** The ratio of a run (fragment)  $\rho = c_0 A_1 c_1 A_2 c_2 A_3 \dots$  is the ratio of the sums of weights  $w_1$  and  $w_2$ , defined as follows:

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

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

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

Since the reachable part of  $\mathcal{S}$  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, we can determine the minimum ratio value of the graph from a *minimum cycle ratio* (MCR) analysis [22].

**Proposition 12 (Adapted from Proposition 1 in [15]):**  $\tau_{\min}(\mathcal{S}) = \inf_{\text{cycle} \in \text{cycles}(\mathcal{S})} \text{Ratio}(\text{cycle}) = \text{MCR}(\mathcal{S})$ , where  $\text{cycles}(\mathcal{S})$  denotes all cycles in  $\mathcal{S}$  and  $\text{MCR}(\mathcal{S})$  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. We use this small 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  $(CBAD)^\omega$ :

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

In this cycle, the first transition represents the execution of  $C$  on resource  $r_3$ , computed through matrix multiplication with  $M_C$ ; it adds 4 time units to the total execution time (weight  $w_2$ ), 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 similarly. 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  $A_{\text{src}}$  and sink activity  $A_{\text{snk}}$ . In the state space, consider some run  $\rho = c_0 A_1 c_1 A_2 \dots$  with  $c_i = \langle s_i, \gamma_i \rangle$  containing run fragment  $\rho[i, j+1] = c_i A_{i+1} \dots c_j A_{j+1} c_{j+1}$ , with  $A_{i+1} = A_{\text{src}}$  and  $A_{j+1} = A_{\text{snk}}$ . We define the start-to-start latency  $\lambda$  between the resource availability times of resource  $r$  in  $\gamma_i$  and  $\gamma_j$  as  $\lambda(\rho, i, j, r) = (\bar{\gamma}_j)_r - (\bar{\gamma}_i)_r$ .

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  $\rho =$

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

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  $\bar{\gamma}_2 = [5, 3, 4]^T$ . Then, we compute the latency

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

We assume that the occurrences of  $A_{\text{src}}$  and  $A_{\text{snk}}$  activities are related. In any run, for any  $k \geq 1$ , the  $k$ th occurrence of  $A_{\text{src}}$  is paired with the  $k$ th occurrence of  $A_{\text{snk}}$ . We refer to such a pair of related activities as a source-sink pair. Let  $\text{getOccurrence}(\rho, A, k)$  denote the index of the  $k$ th occurrence of activity  $A$  in run  $\rho$ . The *start-to-start latency* for resource  $r$  in  $\rho$  with source-sink pair  $A_{i+1} = A_{\text{src}}$  and  $A_{j+1} = A_{\text{snk}}$  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

$$\lambda_{\max}(\rho, A_{\text{src}}, A_{\text{snk}}, r) = \sup_{k \geq 1} \lambda_k(\rho)$$

where  $\lambda_k(\rho) = \lambda(\rho, i, j, r)$ ,  $i = \text{getOccurrence}(\rho, A_{\text{src}}, k)$ , and  $j = \text{getOccurrence}(\rho, A_{\text{snk}}, k)$ .

**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  $A_{\text{src}}, A_{\text{snk}}$  is found by taking the maximum latency over all runs in the state space

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

#### IV. STATE-SPACE PERFORMANCE EQUIVALENCE

The performance of the supervisory controller is analyzed on the corresponding normalized  $(\max,+)$  state space. In our POR, we ensure that performance properties are preserved in the normalized  $(\max,+)$  state space of the reduced supervisor. We capture this preservation in the notion of state-space performance equivalence. Throughput and latency are both expressed

in terms of runs in the state space. We introduce the notion of equivalent runs that have the same ratio value (Definition 11). We show that equivalent runs have the same throughput and latency values. In Section V, we show that our POR preserves equivalent runs in the corresponding normalized  $(\max,+)$  state spaces, thereby preserving performance properties.

**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 equivalent runs with the same ratio value caused by the interleaving of *ratio-independent* activities that have no mutual influence. Such runs have the same throughput and latency values, as shown in the following. The first property of ratio-independent 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  $B$  are 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  $\mathcal{S} = \langle C, \hat{c}, Act, \Delta, M, w_1, w_2 \rangle$  be a normalized  $(\max,+)$  state space,  $c \in C$  a configuration, and  $A, B \in \text{enabled}(c)$  activities enabled in  $c$ . Activities  $A$  and  $B$  are *ratio-independent* in  $c$  iff they satisfy the following conditions<sup>2</sup>:

- 1)  $B \in \text{enabled}(A(c))$ ,  $A \in \text{enabled}(B(c))$ , 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  $i \in \{1, 2\}$ ; and
- 3)  $R(A) \cap R(B) = \emptyset$ .

Two activities are *ratio-dependent* in configuration  $c$  iff they are not ratio-independent in  $c$ .

To illustrate, consider configuration  $c_0 = \langle s_1, \mathbf{0} \rangle$  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 Definition 15) and ratio-independent with activity  $C$ .

To formalize the equivalence on runs, 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  $\rho[\dots m] = c_0 A_1 \dots A_m c_m$  of some run  $\rho$ , let  $Act(\rho[\dots m])$  denote the activity sequence  $A_1 \dots A_m$ .

**Definition 16 (String Equivalence):** Let  $\mathcal{S} = \langle C, \hat{c}, Act, \Delta, M, w_1, w_2 \rangle$  be a normalized  $(\max,+)$  state space. Strings  $\alpha, \beta \in Act^*$  are equivalent [24] in a configuration  $c$ , denoted  $\alpha \equiv_c \beta$ , iff there exists a list of strings  $v_0, v_1, \dots, v_n$ , where  $v_0 = \alpha$ ,  $v_n = \beta$ , and for each  $0 \leq i < n$ ,  $v_i = \bar{v}AB\hat{v}$  and  $v_{i+1} = \bar{v}BA\hat{v}$  for some  $\bar{v}, \hat{v} \in Act^*$  and  $A, B \in Act$  such that  $A$  and  $B$  are ratio-independent in  $\bar{v}(c)$ .

<sup>2</sup>For state spaces generated from  $(\max,+)$  automata, as in this article, condition 2 follows from 1 and 3.



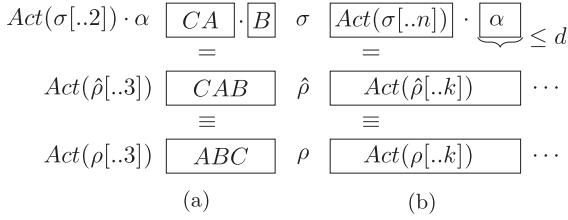


Fig. 7. Illustration of Definition 18. (a) Example  $\rho \succeq_{\frac{1}{2}} \sigma$ . (b) Illustration of  $\rho \succeq_n^d \sigma$

**Definition 17 (Prefix Equivalence):** Prefixes  $\rho[..m]$  and  $\sigma[..m]$  of runs  $\rho$  and  $\sigma$  starting in configuration  $c$  are equivalent in configuration  $c$ , denoted  $\rho[..m] \equiv_c \sigma[..m]$ , iff  $Act(\rho[..m]) \equiv_c Act(\sigma[..m])$ .

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 Definition 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  $\rho$  and  $\sigma$  be two runs in  $\mathcal{S} = \langle C, \hat{c}, Act, \Delta, M, w_1, w_2 \rangle$ . 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  $k \geq n$ , a run  $\hat{\rho} \in \mathcal{R}(\mathcal{S})$  with run prefix  $\hat{\rho}[..k]$  with  $Act(\hat{\rho}[..k]) \equiv_c Act(\rho[..k])$  such that  $Act(\hat{\rho}[..k]) = Act(\sigma[..n]) \cdot \alpha$  for some activity sequence  $\alpha$ , and  $k - n \leq d$ . Runs  $\rho$  and  $\sigma$  are equivalent, denoted  $\rho \equiv \sigma$ , iff  $\rho \succeq \sigma$  and  $\sigma \succeq \rho$ .

Two finite runs  $\rho$  and  $\sigma$  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  $\rho$  with activity sequence  $(ABCD)^\omega$  in the state space in Fig. 6. In this state space, we can also find run  $\sigma = (CABD)^\omega$ , which 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. 7(a). Then, we need to match two ( $n$ ) activities of  $\rho$  to the first two activities of  $\sigma$ . The prefix consisting of the first two activities of  $\rho$ ,  $AB$ , is not a prefix of  $\sigma$ . In  $\sigma$ , independent activity  $C$  is performed first. Run  $\hat{\rho}$ , equivalent (in the sense of Definition 17) to  $\rho$  and identical to  $\sigma$  for  $n = 2$  activities and  $d = 1$ , which can be constructed by moving  $C$  to the front.  $\rho$  and  $\sigma$  must be such that this can be done for any  $n \geq 0$ . The general case is shown in Fig. 7(b). It is crucial for the preservation of throughput that the length  $k$  that one needs to consider in  $\rho$  to find the first  $n$  activities of  $\sigma$  exceeds  $n$  by maximally a finite amount  $d$ , independent of  $n$ .

The property that equivalent runs have the same throughput and latency values was stated in [8] without proof. It is included here with proof for completeness purposes.

**Proposition 19 (Equivalent Runs Have The Same Throughput And Latency Values):** Let  $\rho, \sigma \in \mathcal{R}(\mathcal{S})$ , be runs in  $\mathcal{S}$ . Let  $A_{src}$  and  $A_{snk}$  be any source–sink pair, and let  $r \in Res$ . If  $\rho \equiv \sigma$ , then  $\text{Ratio}(\rho) = \text{Ratio}(\sigma)$  and  $\lambda_{\max}(\rho, A_{src}, A_{snk}, r) = \lambda_{\max}(\sigma, A_{src}, A_{snk}, r)$ .

*Proof:* By Definition 18, for each prefix in one run we can match a (possibly) shorter prefix in the other run. The difference between the prefixes is a suffix  $\alpha$  bounded in length by  $d$ .

For throughput preservation, observe that the maximum weight difference between the prefixes for both weights  $w_1$  and  $w_2$  is bounded by the maximum cumulative sum of the weights  $w_1$  and  $w_2$  of  $\alpha$ . By Proposition 12, it suffices to consider the maximum cumulative weights  $w_1$  and  $w_2$  over all simple cycles in the state space to determine the maximum cumulative sum for  $\alpha$  for both weights. In the limit, a bounded weight difference can be ignored. Therefore,  $\rho$  and  $\sigma$  have the same ratio value, and consequently also the same throughput value.

The maximum latency in a run between  $A_{src}$  and  $A_{snk}$  on resource  $r$  is determined by some source–sink occurrence pair. The prefixes of  $\rho$  and  $\sigma$  are matched by swapping the ratio-independent activities. Given condition 3 of Definition 15, the source and sink activity can only be part of swaps where the resource availability time of  $r$  is not affected. These swaps thus do not affect the latency value for the source–sink occurrence pair, and hence,  $\rho$  and  $\sigma$  have the same maximum latency.  $\square$

## V. (MAX,+) AUTOMATON REDUCTION

In the previous section, we defined when two state spaces are performance-equivalent, preserving throughput and latency aspects. We proceed to define POR conditions at the level of a (max,+) automaton, which ensure that functional properties are preserved during synthesis and that the corresponding state spaces of the supervisor and reduced supervisor are performance-equivalent.

To reuse existing POR techniques, we want to remove the need to identify the marked states with special set  $S^m$ . We add a self-loop with a special *controllable* activity  $\omega$  to marked states. Nonblockingness of  $\mathcal{A}$  can then be replaced by the notion of  $\omega$ -reachability of the resulting automaton  $\mathcal{A}_\omega$ . The fact that  $\mathcal{A}$  is nonblocking iff  $\mathcal{A}_\omega$  is  $\omega$ -reachable follows directly from the definitions.

**Definition 20 ( $\omega$ -Reachability):** Let  $\mathcal{A} = \langle S, \hat{s}, S^m, Act, reward, M, T \rangle$  be a (max,+) automaton. The  $\omega$ -extension of  $\mathcal{A}$  is the automaton  $\mathcal{A}_\omega = \langle S_\omega, \hat{s}, S^m, Act_\omega, reward_\omega, M_\omega, T_\omega \rangle$  with  $\omega \notin Act$ ,  $Act_\omega = Act \cup \{\omega\}$ ,  $reward_\omega = reward \cup \{(\omega, 0)\}$ ,  $M_\omega = M \cup \{(\omega, I(\omega))\}$ , where  $I(\omega)$  is an identity matrix of size  $|\text{Res}| \times |\text{Res}|$  with for each  $1 \leq i, j \leq |\text{Res}|$ ,  $[I(\omega)]_{ij} = -\infty$  if  $i \neq j$  and  $[I(\omega)]_{ij} = 0$  if  $i = j$ , and  $T_\omega = T \cup \{s^m \xrightarrow{\omega} s^m \mid s^m \in S^m\}$ .  $\mathcal{A}_\omega$  is  $\omega$ -reachable iff from each reachable state  $s \in S$  (i.e.,  $\hat{s} \rightarrow^* s$ ) a state  $s_\omega$  is reachable in which  $\omega$  is enabled.

We want to exploit redundancy in the plant to obtain a reduced plant while preserving the properties of interest. Part of the redundancy is caused by the interleaving of activities that have no mutual influence on these properties. Such activities are referred to as *uncontrollable-independent*. Two activities  $A$  and  $B$  are uncontrollable-independent in state  $s$  if (i) they are independent in the classical sense, no uncontrollable activities are enabled in any of the states  $s, A(s), B(s)$ , and  $AB(s)$ , and they do not share resources. Condition (ii) guarantees that uncontrollable activities do not influence the execution of both activities. It

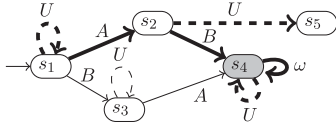


Fig. 8. Example to show the need for condition 2 of Definition 21.

ensures that both the  $AB$  and  $BA$  paths are either preserved or removed during synthesis. The need for this condition is illustrated with Fig. 8. In the example, we have paths  $s_1 \xrightarrow{BA^*} s_4$  and  $s_1 \xrightarrow{AB^*} s_4$  in the full automaton, and we may only have path  $s_1 \xrightarrow{AB^*} s_4$  in the reduced automaton if the reduction considers both paths equivalent based on the independence of  $A$  and  $B$ . Synthesis on the reduced automaton yields an empty supervisor because state  $s_2$  leads to blocking state  $s_5$  via uncontrollable activity  $U$ . Synthesis on the full automaton yields a nonempty supervisor. Although in states  $s_1$ ,  $s_2$ ,  $s_3$ , and  $s_4$ , the same uncontrollable activity  $U$  is enabled, only in state  $s_2$  it takes the system to a blocking state. Therefore, both paths cannot be considered equivalent and activities  $A$  and  $B$  cannot be considered uncontrollable-independent. Condition (iii) guarantees that the activities have no mutual influence on their timing behavior.

**Definition 21 (Uncontrollable-Independence):** Given (max,+) automaton  $\mathcal{A} = \langle S, \hat{s}, S^m, Act, reward, M, T \rangle$  and state  $s \in S$ , activities  $A, B \in enabled(s)$  are *uncontrollable-independent* in  $s$  iff they satisfy the following conditions:

- 1)  $B \in enabled(A(s))$ ,  $A \in enabled(B(s))$ , and  $AB(s) = BA(s)$ ;
- 2)  $enabled(s) \cap Act_u = enabled(A(s)) \cap Act_u = enabled(B(s)) \cap Act_u = enabled(AB(s)) \cap Act_u = \emptyset$ ; and
- 3)  $R(A) \cap R(B) = \emptyset$ .

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. 2(e). 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.

Uncontrollable-independence lifts the notion of ratio-independence 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. As  $R(A) \cap R(B) = \emptyset$ , 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.

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

**Definition 22:** Let  $\mathcal{A} = \langle S, \hat{s}, S^m, Act, reward, M, T \rangle$  be a (max,+) automaton. Strings  $\alpha, \beta \in Act^*$  are equivalent in a state  $s$ , denoted  $\alpha \equiv_{s,u} \beta$ , iff there exists a list of strings  $v_0, v_1, \dots, v_n$ , where  $v_0 = \alpha$ ,  $v_n = \beta$  and, for each  $0 \leq i < n$ ,  $v_i = \bar{v}AB\hat{v}$

and  $v_{i+1} = \bar{v}BA\hat{v}$  for some  $\bar{v}, \hat{v} \in Act^*$  and activities  $A, B \in Act$  such that  $A$  and  $B$  are uncontrollable-independent in  $\bar{v}(s)$ . Paths  $p = s \rightarrow^\alpha s_1$  and  $p' = s \rightarrow^\beta s_2$  are equivalent iff  $\alpha \equiv_{s,u} \beta$ .

A reduction of a (max,+) automaton is defined as follows.

**Definition 23 ((Max,+) Automaton Reduction Function):** A reduction function *reduce* for a (max,+) automaton  $\mathcal{A} = \langle S, \hat{s}, S^m, Act, reward, M, T \rangle$  is a mapping from  $S$  to  $2^{Act}$  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  $\mathcal{A}' = \langle S', \hat{s}, S^{m'}, Act, reward, M, T' \rangle$  that satisfies the following conditions:

- 1)  $S' \subseteq S$ ,  $S^{m'} = S^m \cap S'$ ,  $T' \subseteq T$ ;
- 2) for every  $s \in S'$  and  $A \in reduce(s)$ ,  $\langle s, A, A(s) \rangle \in T'$ .

A reduction on plant  $\mathcal{P}$  gives a reduced plant  $\mathcal{P}'$ , as shown in Fig. 2(e). The supervisor  $\mathcal{A}'_{sup}$  synthesized for  $\mathcal{P}'$  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 path in  $\mathcal{P}'$  is preserved for each path in  $\mathcal{P}$  to a marked state.

**Definition 24 (Reduced Least-Restrictiveness):** Let  $\mathcal{A}_1 = \langle S_1, \hat{s}, S_1^m, Act_1, reward_1, M_1, T_1 \rangle$  and  $\mathcal{A}_2 = \langle S_2, \hat{s}, S_2^m, Act_2, reward_2, M_2, T_2 \rangle$  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  $\hat{s} \rightarrow^{\beta^*}_{\mathcal{A}_2} s_m$  with  $s_m \in S_2^m$ , there exists a path  $\hat{s} \rightarrow^{\beta^*}_{\mathcal{A}_1} s_m$  in  $\mathcal{A}_1$  with  $s_m \in S_1^m$  such that  $\alpha \equiv_{\hat{s},u} \beta$ .

In our running example,  $\mathcal{A}'_{sup}$  is reduced least-restrictive with respect to  $\mathcal{A}_{sup}$ , since it preserves a representative path to the marked state  $s_8$  for all pruned paths.

We apply POR to obtain  $\mathcal{P}'$  from  $\mathcal{P}$ . 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 25:** 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  $\mathcal{A}'_{sup} \lesssim_{f,p} \mathcal{A}_{sup}$  iff

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

Conditions 1–3 guarantee preservation of functional aspects. Condition 4 ensures the preservation of performance. Condition 3 does not automatically imply condition 4, as throughput is defined over infinite runs (going through simple cycles), where cycles in the state space may not have corresponding marked states in the automaton. To preserve the properties of interest, we impose the following ample conditions on a reduction of a (max,+) automaton.

**Definition 26 (Ample Conditions (Max,+) Automaton):** A reduction function *ample* is an ample reduction if it satisfies all the following conditions in each state  $s$ .

- (A1) Nonemptiness condition: if  $enabled(s) \neq \emptyset$ , then  $ample(s) \neq \emptyset$ .

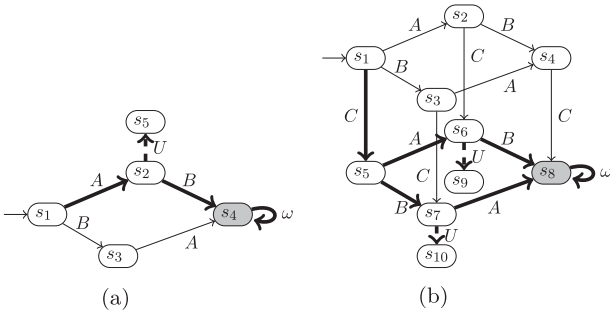


Fig. 9. Necessity of conditions (A5.1) and (A5.2). a) Plant  $\mathcal{P}_1$  with reduced plant  $\mathcal{P}'_1$  that satisfies all conditions except condition (A5.1). (b) Plant  $\mathcal{P}_2$  with reduced plant  $\mathcal{P}'_2$  that satisfies all conditions except condition (A5.2).

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

We refer to set  $\text{ample}(s)$  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  $\text{ample}(s)$  remains enabled as long as no activity in  $\text{ample}(s)$  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 marked states can still be identified. Note that  $\omega$  acts like an uncontrollable activity, since it must remain in the ample set. We have chosen it to be controllable, however, as we do not want  $\omega$  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 for any path to a marked state, where  $\omega$  is enabled, is preserved in the reduced plant. Conditions (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.

To illustrate the need for conditions (A5.1) and (A5.2), consider the plants shown in Fig. 9. Fig. 9(a) 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  $\text{ample}(s_1) \neq \text{enabled}(s_1)$ . 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  $s_1$ . This is needed since state  $s_2$  becomes a bad state during synthesis, and the alternative path to  $s_4$  is then preserved. To illustrate the need for condition (A5.2) consider plant  $\mathcal{P}_2$  shown in Fig. 9(b). Here, the reduction yielding  $\mathcal{P}'_2$  satisfies conditions (A1) till (A5.1), but not condition (A5.2); activity  $U \in \text{enabled}(BC(s_1))$ , but only  $C$  is present in  $\text{ample}(s_1)$ , and not  $B$ . The result after synthesis on  $\mathcal{P}_2$  contains the paths  $s_1 \xrightarrow{A} s_2 \xrightarrow{B} s_4 \xrightarrow{C} s_8$  and  $s_1 \xrightarrow{B} s_3 \xrightarrow{A} s_4 \xrightarrow{C} s_8$ , whereas synthesis on the reduced plant  $\mathcal{P}'_2$  yields an empty supervisor since states  $s_9$  and  $s_{10}$  are not marked.

We now prove the main result that synthesis after reduction preserves both functional and performance aspects.

*Theorem 27:* Let  $\mathcal{P}$  be a plant, and  $\mathcal{P}'$  the reduced plant obtained from an ample reduction. Then,  $\text{supCN}(\mathcal{P}') \lesssim_{f,p} \text{supCN}(\mathcal{P})$ .

*Proof:* Let  $\mathcal{A}'_{\text{sup}} = \text{supCN}(\mathcal{P}')$  and  $\mathcal{A}_{\text{sup}} = \text{supCN}(\mathcal{P})$ . To show that  $\mathcal{A}'_{\text{sup}} \lesssim_{f,p} \mathcal{A}_{\text{sup}}$ , we need to prove the four conditions as stated in Definition 25.

- 1) *Nonblockingness:* Synthesis guarantees that  $\text{supCN}(\mathcal{P}')$  is nonblocking, for any  $\mathcal{P}'$ .
- 2) *Controllability:* Condition (A3) guarantees that uncontrollable actions are preserved in an ample reduction.
- 3) *Reduced least-restrictiveness:* We need to show that  $\text{supCN}(\mathcal{P}') \preceq \text{supCN}(\mathcal{P})$  and that for each path  $p$  to a marked state in  $\text{supCN}(\mathcal{P})$ ,  $\text{supCN}(\mathcal{P}')$  has a path  $p'$  to a marked state in  $\text{supCN}(\mathcal{P}')$  that is equivalent to  $p$ . First, we show that  $\text{supCN}(\mathcal{P}') \preceq \text{supCN}(\mathcal{P})$ . By induction on the iterations of the synthesis algorithm given in [2], we can show that states of  $\mathcal{P}'$  are bad in iteration  $k$  of synthesis on  $\mathcal{P}'$  if and only if those states are bad in iteration  $k$  of synthesis on  $\mathcal{P}$ . Consequently, since  $\mathcal{P}' \preceq \mathcal{P}$ , also  $\text{supCN}(\mathcal{P}') \preceq \text{supCN}(\mathcal{P})$ .

Second, we show that a path  $p'$  exists in  $\text{supCN}(\mathcal{P}')$  equivalent to  $p$ . As  $\text{supCN}(\mathcal{P}) \preceq \mathcal{P}$ , by our assumption below Definition 8,  $p$  is also a path in  $\mathcal{P}$ . By structural induction, we can find an equivalent path  $p'$  in  $\mathcal{P}'$ . In each step, we reorder uncontrollable-independent activities in  $p$  to extend the prefix of  $p'$ , in the end obtaining path  $p'$ . To prove that  $p'$  is also a path in  $\text{supCN}(\mathcal{P}')$ , we assume toward a contradiction that path  $p'$  is not preserved in  $\text{supCN}(\mathcal{P}')$ . Assuming that  $p'$  is not in  $\text{supCN}(\mathcal{P}')$ , there must be a state on  $p'$  that turns bad during synthesis. Path  $p'$  is also a path in  $\mathcal{P}$ , as  $\mathcal{P}' \preceq \mathcal{P}$ . The earlier observation that states of  $\mathcal{P}'$  are bad in iteration  $k$  of synthesis on  $\mathcal{P}'$  if and only if those states are bad in iteration  $k$  of synthesis on  $\mathcal{P}$  implies that  $p'$  is also not in  $\text{supCN}(\mathcal{P})$ . Path  $p'$  is obtained from  $p$  by swapping uncontrollable-independent activities. By induction on the number of swaps to obtain a path from an equivalent one, it can be shown that synthesis preserves equivalent paths. The key observation is that a swap of two uncontrollable-independent activities preserves bad states (via condition 2 of Definition 21). Given that  $p'$  is not in  $\text{supCN}(\mathcal{P})$ , and  $p$  and  $p'$  being equivalent, this implies that also  $p$  is not in  $\text{supCN}(\mathcal{P})$ ,

which contradicts our initial assumption. Therefore,  $p'$  must be present in  $\text{supCN}(\mathcal{P}')$ , showing reduced least-restrictiveness of  $\text{supCN}(\mathcal{P}')$ .

- 4) *Performance*: Let  $\mathcal{S}$  and  $\mathcal{S}'$  be the state spaces of  $\text{supCN}(\mathcal{P})$  and  $\text{supCN}(\mathcal{P}')$ , respectively. We first observe that any run of  $\mathcal{S}'$  is also a run of  $\mathcal{S}$ . This follows from the fact that  $\text{supCN}(\mathcal{P}') \preceq \text{supCN}(\mathcal{P})$ . Thus,  $\mathcal{S}'$  does not introduce any new ratio or latency values not also present in  $\mathcal{S}$ . Second, we show that there is a run in  $\mathcal{S}'$  with the worst-case throughput (ratio value) of  $\mathcal{S}'$ . Finally, we show that for any run in  $\mathcal{S}$  with maximum latency, there is a run with the same latency value in  $\mathcal{S}'$ .

### A. Throughput

By Proposition 12, the worst-case throughput in  $\mathcal{S}$  is exhibited by a run  $\rho$  with  $\text{Act}(\rho) = \alpha_1 \cdot (\alpha_2)^\omega$  (for some  $\alpha_1, \alpha_2 \in \text{Act}^*$ ) such that  $\alpha_2$  corresponds to a cycle in the state space, i.e.,  $\alpha_1(\hat{c}) = \alpha_1 \cdot \alpha_2(\hat{c})$  with  $\hat{c}$  the initial configuration of  $\mathcal{S}$  and  $\mathcal{S}'$ . This run corresponds to a path  $p$  in  $\text{supCN}(\mathcal{P})$  with  $\text{Act}(p) = \alpha_1 \cdot \alpha_2 \cdot \alpha_3$ , where  $\alpha_2$  corresponds to a cycle in  $\text{supCN}(\mathcal{P})$  and where  $\alpha_3$  represents a path from that cycle to a marked state. We can extend this path to a path  $\bar{p}$  with  $\text{Act}(\bar{p}) = \alpha_1 \cdot \alpha_2^N \cdot \alpha_3$  for arbitrary  $N > 0$ , where we traverse cycle  $\alpha_2$   $N$  times. By reduced least-restrictiveness of  $\text{supCN}(\mathcal{P}')$ , for each such path there exists an equivalent path  $\bar{p}'$  in  $\text{supCN}(\mathcal{P}')$ . If  $N \geq |\mathcal{S}'|$ ,  $\bar{p}'$  is such that there exist  $\beta_1, \beta_2, \beta_3$  with  $\beta_1 \cdot \beta_2 \cdot \beta_3 \equiv_{\hat{s}, u} \alpha_1 \cdot \alpha_2^N \cdot \alpha_3$ , where  $\hat{s}$  is the initial state in  $\mathcal{P}$  and  $\mathcal{P}'$ ,  $|\beta_1| \geq |\alpha_1|$ ,  $|\beta_3| \geq |\alpha_3|$ ,  $|\beta_2| = k \cdot |\alpha_2|$  for some  $k > 0$ . Then, for all  $m > 0$ ,  $\alpha_1 \cdot \alpha_2^{N+(m-1) \cdot k} \cdot \alpha_3 \equiv_{\hat{s}, u} \beta_1 \cdot \beta_2^m \cdot \beta_3$ . As uncontrollable-independence in the automaton implies ratio-independence in the state space,  $\alpha_1 \cdot \alpha_2^{N+(m-1) \cdot k} \cdot \alpha_3 \equiv_{\hat{c}} \beta_1 \cdot \beta_2^m \cdot \beta_3$ . It follows that  $\mathcal{S}'$  has a run  $\rho'$  with  $\text{Act}(\rho') = \beta_1 \cdot (\beta_2)^\omega$  that repeats a cycle that is equivalent to  $\alpha_2$  in run  $\rho$ . Hence,  $\rho$  and  $\rho'$  have the same ratio values, i.e.,  $\text{Ratio}(\rho') = \text{Ratio}(\rho)$ .

### B. Latency

The maximum latency between source activity  $A_{\text{src}}$  and sink activity  $A_{\text{snk}}$  for resource  $r$  is determined by some source–sink occurrence pair in some run  $\rho$  in  $\mathcal{S}$ . Let prefix  $\rho[\dots m]$ , for some  $m$ , with  $\text{Act}(\rho[\dots m]) = \alpha_1 \cdot A_{\text{src}} \cdot \alpha_2 \cdot A_{\text{snk}}$  contain this occurrence pair.  $\rho[\dots m]$  corresponds to a path  $p$  in  $\text{supCN}(\mathcal{P})$  with  $\text{Act}(p) = \text{Act}(\rho[\dots m])$ . We can extend  $p$  to path  $\bar{p}$  with  $\text{Act}(\bar{p}) = \text{Act}(p) \cdot \alpha_3$  such that  $\bar{p}$  leads to a marked state. By reduced least-restrictiveness of  $\text{supCN}(\mathcal{P}')$ , for each such path, there exists an equivalent path  $\bar{p}'$  with  $\text{Act}(\bar{p}') \equiv_{\hat{s}, u} \text{Act}(\bar{p})$ , where  $\hat{s}$  is the initial state in  $\mathcal{P}'$  and  $\mathcal{P}$ . Path  $\bar{p}'$  can be obtained from  $\bar{p}$  by swapping uncontrollable-independent activities such that  $\text{Act}(\bar{p}') = \beta_1 \cdot A_{\text{src}} \cdot \beta_2 \cdot A_{\text{snk}} \cdot \beta_3$ . As uncontrollable-independence in the automaton implies ratio-independence in the state space, we obtain a run  $\rho'$  in  $\mathcal{S}'$  such that  $\text{Act}(\rho'[\dots m']) = \text{Act}(\bar{p}') \equiv_{\hat{c}} \text{Act}(\rho[\dots m]) \cdot \alpha_3$ . Note that path  $\bar{p}'$  can always be extended to an infinite run, because it ends in a marked state having an  $\omega$ -self loop. Activities  $A_{\text{src}}$  and  $A_{\text{snk}}$  can only be a part of a swap where the resource availability time of  $r$  is not affected (as explained in the proof of Proposition 19). Hence, the latency of this occurrence is identical

to the one in  $\rho$ . Given the assumption that the latency value of the considered source–sink occurrence pair in  $\rho$  is the maximum value in  $\mathcal{S}$  and since  $\mathcal{S}'$  does not introduce new latency values, the latency value of  $\rho'$  must be the same worst-case value as the latency value of  $\rho$ .  $\square$

## VI. ON-THE-FLY REDUCTION

Section V gives sufficient conditions for a reduction function on a single (max,+) automaton to preserve functional and performance properties. 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 to compute a reduced composition on-the-fly. Given (max,+) timed system  $\mathcal{M} = \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$  and  $\mathcal{A} = \{\mathcal{A}_i \mid 1 \leq i \leq n\}$ , the ample function selects a set  $\text{ample}(s)$  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  $\text{ample}(s) = \text{enabled}_{\mathcal{C}}(s) = \text{enabled}(s) \cap \text{Act}(\mathcal{C})$ , where  $\text{Act}(\mathcal{C}) = \bigcup_{\mathcal{A}_i \in \mathcal{C}} \text{Act}_i$  and  $\text{Act}_i$  is the alphabet of (max,+) automaton  $\mathcal{A}_i$ . This cluster-inspired ample approach, based on [23], is a generalization of the traditional on-the-fly method of Peled [5] that selects the enabled activities  $\text{enabled}_i(s) = \text{enabled}(s) \cap \text{Act}_i$  of one (max,+) automaton  $\mathcal{A}_i$  as ample set, if possible, while exploring a state  $s = \langle s_1, s_2, \dots, s_n \rangle$ . If this is not possible, all enabled activities in  $s$  are selected as ample set.

A cluster that ensures that the ample conditions are met is called a *safe cluster*. To consider the local state  $\pi_{\mathcal{C}}(s)$  in a cluster  $\mathcal{C} \subseteq \mathcal{A}$ , we define a projection  $\pi$

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

Given local state  $\pi_{\mathcal{C}}(s)$ ,  $\text{enabled}(\pi_{\mathcal{C}}(s))$  denotes the set of enabled activities in the composition of the (max,+) automata in  $\mathcal{C}$ . Note that  $\text{enabled}_{\mathcal{C}}(s) \subseteq \text{enabled}(\pi_{\mathcal{C}}(s))$ , since the latter might contain activities that are enabled in the local state of the cluster-composition  $\mathcal{A}_{c_1} \parallel \dots \parallel \mathcal{A}_{c_k}$  for cluster  $\mathcal{C} = \{\mathcal{A}_{c_1}, \dots, \mathcal{A}_{c_k}\}$ , 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 28 (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  $\text{enabled}(s) \neq \emptyset$ , then  $\text{enabled}_{\mathcal{C}}(s) \neq \emptyset$ ;
- (C2.1) for any  $A \in \text{enabled}_{\mathcal{C}}(s)$  and  $B \notin \text{Act}(\mathcal{C})$ ,  $A$  and  $B$  are uncontrollable-independent;
- (C2.2) for any  $A \in \text{enabled}(\pi_{\mathcal{C}}(s)) \cap \text{Act}_i$ ,  $\mathcal{A}_i \in \mathcal{C}$ ;
- (C3)  $\text{enabled}_{\mathcal{C}}(s) \supseteq \text{enabled}(s) \cap \text{Act}_u$ ;
- (C4)  $\text{enabled}_{\mathcal{C}}(s) \supseteq \text{enabled}(s) \cap \{\omega\}$ ;

(C5.1) if  $A \in \text{enabled}_{\mathcal{C}}(s)$  and  $\text{enabled}(A(s)) \cap \text{Act}_u \neq \emptyset$ , then  $\text{enabled}_{\mathcal{C}}(s) = \text{enabled}(s)$ ;

(C5.2) for  $A, B \in \text{enabled}(s)$  if  $\text{enabled}(AB(s)) \cap \text{Act}_u \neq \emptyset$ , then  $A \in \text{enabled}_{\mathcal{C}}(s) \Leftrightarrow B \in \text{enabled}_{\mathcal{C}}(s)$ .

Condition (C2.2) requires that each activity in  $\text{enabled}(\pi_{\mathcal{C}}(s))$  does not occur outside of the cluster. Together with (C2.1), this ensures that no activity  $A \notin \text{enabled}_{\mathcal{C}}(s)$ , dependent on some activity in  $\text{enabled}_{\mathcal{C}}(s)$ , becomes enabled by executing only activities outside the cluster. Condition (C3) ensures that uncontrollable activities are always preserved. Condition (C4) ensures that  $\omega$ , 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 29 (Cluster-Inspired Ample Reduction):* A cluster-inspired ample reduction for a (max,+) timed system  $\mathcal{M} = \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$  is a mapping from  $S = S_1 \times \dots \times S_n$  to  $2^{\text{Act}}$  such that  $\text{ample}(s)$  for all states  $s \in S$  satisfies the following condition:

(M1)  $\text{ample}(s) = \text{enabled}_{\mathcal{C}}(s)$ , where  $\mathcal{C} \subseteq \mathcal{A}$  is safe in  $s$ .

*Theorem 30:* Let  $\mathcal{P} = \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$  be a plant modeled as (max,+) timed system, and  $\mathcal{P}'$  be the (max,+) automaton obtained by a cluster-inspired ample reduction. Then,  $\text{supCN}(\mathcal{P}') \lesssim_{f,p} \text{supCN}(\mathcal{P})$ .

*Proof:* Let ample be a cluster-inspired ample reduction. We show that ample satisfies the conditions of Definition 26. The result then immediately follows from Theorem 27.

Consider any state  $s$  in the composition of  $\mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$ . Let  $\text{ample}(s) = \text{enabled}_{\mathcal{C}}(s)$ , where  $\mathcal{C}$  is any safe cluster in  $s$ , satisfying condition (M1). Conditions (A1), (A3), (A4), (A5.1), and (A5.2) follow directly from (C1), (C3), (C4), (C5.1), and (C5.2). We prove (A2) by contraposition, for the case that  $\text{enabled}(s) \neq \emptyset$ . If  $\text{enabled}(s) = \emptyset$ , (A2) is trivially satisfied. Assume that (A2) does not hold. This means that there exists a finite path fragment  $p = s \xrightarrow{A_1} s_1 \xrightarrow{A_2} s_2 \xrightarrow{A_3} \dots \xrightarrow{A_{n-1}} s_{n-1} \xrightarrow{A_n}$ , where  $A_1 \dots A_{n-1}$  are uncontrollable-independent with  $\text{ample}(s) = \text{enabled}_{\mathcal{C}}(s)$ , and  $A_n$  is uncontrollable-dependent with some activity in  $\text{enabled}_{\mathcal{C}}(s)$ . Since  $A_n$  is uncontrollable-dependent with some activity in  $\text{enabled}_{\mathcal{C}}(s)$ , by (C2.1),  $A_n \in \text{Act}(\mathcal{C})$ . Moreover, we have  $A_n \notin \text{enabled}_{\mathcal{C}}(s)$ . Since activities  $A_1, \dots, A_{n-1}$  are uncontrollable-independent with  $\text{ample}(s)$ ,  $A_1, \dots, A_{n-1} \notin \text{Act}(\mathcal{C})$ , they do not affect the state of  $\mathcal{C}$ , and therefore,  $\pi_{\mathcal{C}}(s)$  does not change in the first  $n - 1$  steps. As  $A_n \in \text{enabled}(s_{n-1})$ ,  $A_n \in \text{enabled}(\pi_{\mathcal{C}}(s))$ . Since  $A_n \notin \text{enabled}_{\mathcal{C}}(s)$ ,  $A_n$  becomes enabled in  $\pi_{\mathcal{C}}(s)$  by executing one of the activities in set  $A_1, \dots, A_{n-1}$ . Since  $A_1, \dots, A_{n-1}$  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 (C2.2).  $\square$

In each composition state  $s$ , we compute a safe cluster starting from a candidate activity. To guarantee conditions (C5.1) and (C5.2), we check whether an activity  $A$  might enable an uncontrollable activity in  $\text{enabled}(A(s))$  or  $\text{enabled}(AB(s))$  for some uncontrollable-independent activity  $B$ . To avoid

---

**Algorithm 1: Algorithm To Compute A Safe Cluster.**


---

```

1: Proc ComputeClusters, candidate
2:  $A \leftarrow \text{candidate}$ ;  $\mathcal{C} \leftarrow \emptyset$ ; processed  $\leftarrow \emptyset$ 
3: if  $\omega \in \text{enabled}(s)$  then
4:   return  $\mathcal{A}$ 
5: for  $U \in \text{enabled}(s) \cap \text{Act}_u$  do
6:    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathcal{A}_i \mid U \in \text{Act}_i\}$ 
7:   while  $A \neq \perp$  do
8:     processed  $\leftarrow$  processed  $\cup \{A\}$ 
9:     if  $A \in \text{enabled}(s)$  then
10:      if  $A \in \mathcal{U}$  then
11:        return  $\mathcal{A}$ 
12:       $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathcal{A}_i \mid A \in \text{Act}_i\}$ 
13:      for  $B \in \{D \in \text{Act} \mid R(D) \cap R(A) \neq \emptyset\}$  do
14:        if  $B \notin \text{Act}(\mathcal{C})$  then
15:           $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathcal{A}_i \mid B \in \text{Act}_i\}.\text{first}()$ 
16:      if  $A \notin \text{enabled}(s) \wedge A \in \text{enabled}(\pi_{\mathcal{C}}(s))$  then
17:        for  $\mathcal{A}_i \in \mathcal{A}$  do
18:          if  $A \in \text{Act}_i \wedge \mathcal{A}_i \notin \mathcal{C} \wedge A \notin \text{enabled}(s_i)$  then
19:             $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathcal{A}_i\}$ ; break
20:      if processed  $\neq \text{enabled}(\pi_{\mathcal{C}}(s))$  then
21:         $A \leftarrow [\text{enabled}(\pi_{\mathcal{C}}(s)) \setminus \text{processed}].\text{first}()$ 
22:      else
23:         $A \leftarrow \perp$ 
24:      return  $\mathcal{C}$ 

```

---

computing these enabled sets during the on-the-fly reduction, we introduce a new set  $\mathcal{U}$  that can be generated *a priori* from the network of (max,+) automata.  $\mathcal{U}$  contains all activities that enable (or do not disable) an uncontrollable activity in any of the automata

$$\mathcal{U} = \bigcup_{\mathcal{A}_i \in \mathcal{A}} \{A \in \text{Act}_i \mid \text{enabled}(A(s)) \cap \text{Act}_u \neq \emptyset, s \in S_i\}.$$

The reduction is 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, so  $\mathcal{C}$  is conservative.

A safe cluster in a state  $s$  can be computed with Algorithm 1. In the algorithm,  $[\mathcal{A}_i \mid B \in \text{Act}_i]$  denotes a list comprehension that creates a list of all elements  $\mathcal{A}_i$  for which  $B \in \text{Act}_i$ . Function  $\text{first}()$  picks the first element. With  $A \leftarrow \perp$ , we denote that no activity is assigned to  $A$ . The algorithm first ensures that conditions (C3) and (C4) are met (lines 2-8). If  $\text{enabled}(s)$  contains  $\omega$ , then set  $\mathcal{A}$  is returned, since all (max,+) automata have  $\omega$  in their alphabet and will be added to the cluster. If not, then for each enabled uncontrollable activity, all (max,+) automata having this activity in the alphabet are added to the cluster (lines 6-8). The algorithm then checks for each activity enabled in the current cluster  $\mathcal{C}$  whether condition (C2.1) or (C2.2) is violated, ensuring that also (C5.1) and (C5.2) are satisfied. The algorithm starts with candidate activity  $A$ . If  $A$  is enabled in the composition (line 11), we check if  $A \in \mathcal{U}$ , as  $A$  might then enable an uncontrollable activity and condition (C5.1) or (C5.2) might get violated. When  $A \in \mathcal{U}$  and  $A$  is

**Algorithm 2:** Algorithm To Select A Candidate Activity.

---

```

1: Proc Select Candidates  $s, \mathcal{A}$ 
2:   if  $\omega \in \text{enabled}(s)$  then
3:     return  $\omega$ 
4:   else if  $\text{Act}_u \cap \text{enabled}(s) \neq \emptyset$  then
5:     return  $[\text{Act}_u \cap \text{enabled}(s)].\text{first}()$ 
6:   else
7:     return  $\arg \min_{A \in \text{enabled}(s)} \text{GetWeight}A, s, \mathcal{A}$ 
1: Proc GetWeight  $A, s, \mathcal{A}$ 
2:    $w \leftarrow 0$ 
3:   for  $\mathcal{A}_i \in \mathcal{A}$  do
4:     if  $A \in \text{Act}_i$  then
5:       for  $B \in \text{enabled}(s_i)$  do
6:         if  $B \in \text{enabled}(s)$  then
7:            $w \leftarrow w + |\mathcal{A}| \cdot |\text{Act}|$ 
8:         else
9:            $w \leftarrow w + 1$ 
10:  return  $w$ 

```

---

enabled within the current cluster, the set of all automata is returned (line 13); otherwise, we add all (max,+) automata containing  $A$  (line 15) and add a (max,+) automaton for each dependent activity outside the current cluster (lines 16–18). This ensures that condition (C2.1) is satisfied for activity  $A$  and the cluster obtained after executing lines 11–21. If  $A$  is enabled in the composition of (max,+) automata in 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 22–28. After handling  $A$ , we check whether there are other unprocessed activities that are locally enabled in the new cluster (line 29–33). The algorithm continues until all locally enabled activities are processed.

*Theorem 31:* Let  $s$  be a state in the composition  $\mathcal{M} = \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$  and  $A \in \text{enabled}(s)$  be the candidate activity. Then,  $\text{COMPUTECLUSTER}(s, A)$  returns a safe cluster in  $s$ .

*Proof:* From the previous reasoning, it follows that  $\text{COMPUTECLUSTER}(s, A)$  returns a safe cluster  $\mathcal{C}$  in  $s$  that satisfies conditions (C1) through (C5.2). ■

In each composition state, there are typically multiple valid safe clusters. Heuristics can be used to select a cluster that likely yields a large reduction. One approach is to select a safe cluster yielding the smallest ample set starting from each candidate activity in the enabled set. This heuristic often performs well [25], since it allows to prune most enabled transitions. A disadvantage is that a safe cluster is constructed starting from each candidate. For larger enabled sets, this leads to a significant overhead.

Algorithm 2 is an alternative heuristic that selects only one candidate activity. First, it checks if  $\text{enabled}(s)$  contains  $\omega$  or an uncontrollable activity. As they will always be in the ample set, they are a good candidate choice; otherwise, a weight is computed for each candidate based on two aspects, and a candidate with the minimum weight is selected. The first aspect considers whether an activity is selected that does not occur in (max,+) automata that have locally enabled activities that are

also in  $\text{enabled}(s)$ , since this implies that the ample set will increase. We add weight  $|\mathcal{A}| \cdot |\text{Act}|$  as this is the maximum total sum of locally enabled activities. The second aspect considers minimizing the number of other enabled activities within the cluster, since possibly (max,+) automata need to be added where these are not enabled. For each other locally enabled activity, we increase the weight by one.

## 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 system is a simplification of the ASML lithography scanner using similar types of resources. The first variant (TW1) is described in [10]. 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. TW3 extends 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 and TW3-10s, with 10 processing stations. 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 were performed with a 2.40 GHz Intel i5-6300 U CPU processor and with 4 GB Java heap space to run the algorithms.

We evaluate the achievable reductions while preserving only functional aspects, as well as preserving both functional and performance aspects. As Supremica models do not describe the resource usage, we assume that activities do not claim or release resources and are assigned the  $0 \times 0$  (max,+) timing matrix. This implies that they are timeless, and that we effectively preserve only functional aspects. For the Twilight and ASML models, we have the timing matrices and also consider performance aspects.

### A. Reduction Preserving Only Functional Aspects

Before applying POR, we compute set  $\mathcal{U}$  to check whether a reduction is possible. For some models in our test set,  $\mathcal{U}$  contains all activities and no reductions are possible: DosingTank, MachineBufferMachine, TankProcess, AutomaticCarParkGate, TransferLine, and TransferLine3. For the Twilight models and the ASML model, we disregard the performance aspects by assigning the  $0 \times 0$  (max,+) timing matrix to activities. Table I shows the POR results 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 interleaving of activities that can be exploited to obtain a smaller composition. A similar reasoning applies to TW2, TW3, and

TABLE I

REDUCTIONS ACHIEVED PRESERVING FUNCTIONAL ASPECTS.  $|S|$  IS THE NUMBER OF STATES AND  $|T|$  IS THE NUMBER OF TRANSITIONS IN THE AUTOMATA COMPOSITION ( $\mathcal{P}$  IN FIG. 1). THE RUNNING TIMES TO COMPUTE THE COMPOSITION  $T_P$  AND THE SUPERVISOR  $T_S$  ARE IN MILLISECONDS. THE HIGHEST REDUCTIONS ARE HIGHLIGHTED IN BOLD

	full composition				<i>AllCandidates</i> heuristic				<i>SmartCandidate</i> 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	<b>96.9%</b>	<b>98.9%</b>	X	X	<b>96.2%</b>	<b>98.6%</b>	X	X
ASML	2412	7662	174.9	394.6	86.1%	94.0%	82%	62%	65.5%	83.7%	64%	50%

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

	automata composition				<i>AllCandidates</i> heuristic				<i>SmartCandidate</i> 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	<b>80.4%</b>	<b>91.4%</b>	122%	108%	<b>39.6%</b>	<b>67.2%</b>	118%	127%

	normalized (max,+) state space		<i>AllCandidates</i> heuristic				<i>SmartCandidate</i> 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

TW4. The reduction for TW1 is very small, as there is a lot of activity synchronization by the product-location and life-cycle automata. Recall condition (C2.2), requiring 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), which limits reduction possibilities. The reductions for TW2, TW3, and TW4 are 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 all activities in this model are controllable and requirements are local.

As expected, *AllCandidates* yields similar or better reductions than *SmartCandidate* in terms of states and transitions remaining 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*, where Algorithm 1 is run only once in each state. 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 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 required 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 larger compositions, 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 TW3-10 s 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 specific structure, with resource booking/unbooking and acyclic product life cycles where each resource can occur only once. 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 required conditions are different from the ones we use. 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 the verification of similar timing relations. Theelen *et al.* [32] applied ideas from POR on scenario-aware data flow models, using an independence relation among actions to resolve nondeterministic 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 are also 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. A synthesis algorithm is provided that computes such a controller. Huang and Kumar [35] described an approach to generate a reduced controller under the traditional notion of nonblockingness. Another approach is to find a smaller equivalent least-restrictive supervisor [36]. 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 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, such as performance-related properties. Our POR achieves this by adding sufficient conditions to the reduction function and adapting the notion of event-dependence used.

Su *et al.* [17] described 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 [13] and [14]. 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. CONCLUSION

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. The reduction helps in synthesis and performance analysis of supervisory controllers, as 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 nontimed 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 where automata describe local parts of the system. We obtained reductions up to 80.4% and 91.4% in the number of states and transitions. The possible reductions are highly dependent on 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.

## 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, Jan. 1989.
- [2] L. Ouedraogo, R. Kumar, R. Malik, and K. Akesson, "Nonblocking and safe control of discrete-event systems modeled as extended finite automata," *IEEE Trans. Automat. Sci. Eng.*, vol. 8, no. 3, pp. 560–569, Jul. 2011.
- [3] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. New York, NY, USA: Springer, 2010.
- [4] W. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems: A brief history," *Annu. Rev. Control*, vol. 25, pp. 250–256, 2018.
- [5] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods Syst. Des.*, vol. 8, no. 1, pp. 39–64, 1996.
- [6] D. Peled, "Ten years of partial order reduction," in *Proc. Int. Conf. Comput. Aided Verification*, 1998, pp. 17–28.
- [7] H. Flordal, R. Malik, M. Fabian, and K. Akesson, "Compositional synthesis of maximally permissive supervisors using supervision equivalence," *Discrete Event Dyn. Syst.*, vol. 17, no. 4, pp. 475–504, 2007.
- [8] B. van der Sanden, M. Geilen, M. Reniers, and T. Basten, "Partial-order reduction for performance analysis of max-plus timed systems," in *Proc. IEEE 18th Int. Conf. Appl. Concurrency Syst. Des.*, 2018, pp. 40–49.
- [9] B. van der Sanden *et al.*, "Partial-order reduction for synthesis and performance analysis of supervisory controllers," *Eindhoven Univ. Technol.*, Eindhoven, The Netherlands, Tech. Rep. ESR-2019-02, 2019.



- [10] B. van der Sanden *et al.*, “Compositional specification of functionality and timing of manufacturing systems,” in *Proc. IEEE Forum Specification Des. Lang.*, 2016, pp. 1–8.
- [11] R. Alur and D. L. Dill, “A theory of timed automata,” *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.
- [12] S. Miremadi, Z. Fei, K. Akesson, and B. Lennartson, “Symbolic representation and computation of timed discrete-event systems,” *IEEE Autom. Sci. Eng.*, vol. 11, no. 1, pp. 6–19, Jan. 2014.
- [13] J. Komenda, S. Lahaye, and J.-L. Boimond, “Supervisory control of (max,+) automata: A behavioral approach,” *Discrete Event Dyn. Syst.*, vol. 19, pp. 525–549, 2009.
- [14] S. Lahaye, J. Komenda, and J.-L. Boimond, “Supervisory control of (max,+) automata: Extensions towards applications,” *Int. J. Control*, vol. 88, no. 12, pp. 2523–2537, 2015.
- [15] S. Gaubert, “Performance evaluation of (max,+) automata,” *IEEE Trans. Autom. Control*, vol. 40, no. 12, pp. 2014–2025, Dec. 1995.
- [16] F. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat, *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Hoboken, NJ, USA: Wiley, 1992.
- [17] R. Su, J. van Schuppen, and J. Rooda, “The synthesis of time optimal supervisors by using heaps-of-pieces,” *IEEE Trans. Autom. 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. Autom. Control*, vol. 44, no. 4, pp. 683–697, Apr. 1999.
- [19] M. Geilen and S. Stuijk, “Worst-case performance analysis of synchronous dataflow scenarios,” in *Proc. IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synth.*, 2010, pp. 125–134.
- [20] D. B. Johnson, “Finding all the elementary circuits of a directed graph,” *SIAM J. Comput.*, vol. 4, no. 1, pp. 77–84, 1975.
- [21] W. Thomas, “Automata on infinite objects,” *Handbook Theor. Comput. Sci., Vol. B*, pp. 133–191, 1990.
- [22] A. Dasdan, “Experimental analysis of the fastest optimum cycle ratio and mean algorithms,” *ACM Trans. Des. Automat. Electron. Syst.*, vol. 9, no. 4, pp. 385–418, 2004.
- [23] T. Basten, D. Bořnački, and M. Geilen, “Cluster-based partial-order reduction,” *Automated Softw.*, vol. 11, no. 4, pp. 365–402, 2004.
- [24] A. Mazurkiewicz, “Trace theory,” in *Petri Nets: Appl. Relationships Other Models Concurrency*. Springer, 1987, pp. 278–324.
- [25] J. Geldenhuys, H. Hansen, and A. Valmari, “Exploring the scope for partial order reduction,” in *Proc. Int. Symp. Automated Technol. Verification Anal.*, 2009, pp. 39–53.
- [26] K. Akesson, M. Fabian, H. Flordal, and R. Malik, “Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems,” in *Proc. 8th Int. Workshop Discrete Event Syst.*, 2006, pp. 384–385.
- [27] B. van der Sanden, “Performance analysis and optimization of supervisory controllers,” Ph.D. dissertation, Dept. Elect. Eng., Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2018.
- [28] A. Hellgren, M. Fabian, and B. Lennartson, “Deadlock detection and controller synthesis for production systems using partial order techniques,” *Proc. IEEE Int. Conf. Control Appl.*, vol. 2, pp. 1472–1477, 1999.
- [29] A. M. Shaw, “Partial order reduction with compositional verification,” M.S. thesis, Dept. Comput. Sci., Univ. Waikato, Hamilton, New Zealand, 2014.
- [30] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, “Partial order reductions for timed systems,” in *Proc. Int. Conf. Concurrency Theory*, 1998, pp. 485–500.
- [31] T. Yoneda and B.-H. Schlingloff, “Efficient verification of parallel real-time systems,” *Formal Methods Syst. Des.*, vol. 11, no. 2, pp. 187–215, 1997.
- [32] B. Theelen, M. Geilen, and J. Voeten, “Performance model checking scenario-aware dataflow,” in *Proc. Int. Conf. Formal Model. Anal. Timed Syst.*, 2011, pp. 43–59.
- [33] P. Dietrich *et al.*, “Implementation considerations in supervisory control,” in *Synthesis and Control of Discrete Event Systems*. New York, NY, USA: Springer, 2002, pp. 185–201.
- [34] A. Morgenstern and K. Schneider, “Synthesizing deterministic controllers in supervisory control,” in *Proc. Inform. Control, Automat. Robot. II.*, 2007, pp. 95–102.
- [35] J. Huang and R. Kumar, “Directed control of discrete event systems for safety and nonblocking,” *IEEE Automat. Sci. Eng.*, vol. 5, no. 4, pp. 620–629, Oct. 2008.
- [36] R. Su and W. Wonham, “Supervisor reduction for discrete-event systems,” *Discrete Event Dyn. Syst.*, vol. 14, no. 1, pp. 31–53, Jan. 2004.



**Bram van der Sanden** received the M.Sc. degree in computer science and the Ph.D. degree in electrical engineering from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 2014 and 2018.

He is currently a Research Fellow with ESI (TNO), Eindhoven, The Netherlands. 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** (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1996 and 2002.

He is currently an Associate Professor with the Department of Electrical Engineering at Eindhoven University of Technology, Eindhoven, The Netherlands. 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 multiobjective optimization and trade-off analysis.



**Michel Reniers** (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computing science from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1994 and 1999.

He is currently an Associate Professor in model-based engineering of supervisory control with the Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands. He has authored more than 100 journal and conference papers.

His research interests include model-based systems engineering and model-based validation and testing to novel approaches for supervisory control synthesis, and applications of this work are mostly in the areas of cyber-physical systems. (photograph by Angeline Swinkels)



**Twan Basten** (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computing science from the Eindhoven University of Technology, (TU/e), Eindhoven, The Netherlands, in 1993 and 1998.

He is currently a Professor with the Department of Electrical Engineering, TU/e. He is also a Senior Research Fellow with ESI (TNO), Eindhoven, The Netherlands. His current research interests include the design of embedded and cyber-physical systems, dependable computing,

and computational models.