**Preprints of the 19th World Congress**
**The International Federation of Automatic Control**
**Cape Town, South Africa. August 24-29, 2014**

# State-Partition-Based Control of Discrete Event Systems for Enforcement of Regular Language Specifications ⋆

**Antoine Butez** * **Stéphane Lafortune** ** **Yin Wang** ***

* *Ecole Nomale Supérieure de Cachan, France; (antoine.butez@lurpa.ens-cachan.fr)*
** *Department of EECS, University of Michigan, Ann Arbor, USA; (stephane@umich.edu)*
*** *Facebook, Menlo Park, CA, USA; (yinwang@fb.com)*

**Abstract:** We consider the solution of supervisory control problems for discrete event systems modeled by automata or bounded Petri nets where the specification is expressed as a regular sublanguage of the system language and where the supervisor is restricted to be state-partition-based with respect to original system state space, i.e., the state space of the automaton or the set of reachable markings of the Petri net. State-partition-based supervisors are completely characterized by a partition of the original system state space into legal and illegal states: transitions between legal states are always enabled while transitions from legal to illegal states are always disabled. We present a general algorithm that calculates all state-partition-based supervisors that result in safe and non-blocking controlled languages. The algorithm uses a vertex-cover-type algorithm on the representation of the supremal controllable sublanguage in order to obtain the desired partitions. This work is motivated by the application of discrete event control techniques to the avoidance of classes of concurrency bugs in multithreaded programs. State-partition-based supervisors are especially advantageous in that application as they allow more concurrency at run-time. More generally, this class of supervisors is required when the representation of the supervisor must be based on the system's original state space; this occurs for memoryless supervisors in automaton-based control or in supervision based on place invariants in Petri-net-based control, for instance.

## 1. INTRODUCTION

The control theory of discrete event systems modeled by automata provides algorithmic techniques for handling a large class of control specifications related to safety and non-blockingness. Specifically, the theory initiated by Ramadge & Wonham [Ramadge and Wonham, 1987, 1989], often referred to as Supervisory Control Theory (SCT hereafter), provides algorithmic techniques based on the notion of the supremal controllable sublanguage [Wonham and Ramadge, 1987] for handling safety and non-blockingness, when these are expressed in terms of a regular sublanguage of the language marked by the uncontrolled system and when the control capabilities are limited by the presence of uncontrollable events. The corresponding solution is guaranteed to be maximally permissive, with respect to language inclusion. Let the original model of the uncontrolled system be denoted by $G$. In the process of calculating the supremal controllable sublanguage and of synthesizing the supervisor that will implement the associated control law, it may be necessary to refine the state space of $G$, if the safety specification requires memorizing how a given state is reached. Specifically, the automaton $H \times G$ needs to be constructed, where $H$ is the automaton that represents the regular language safety specification. The control law associated with the supremal controllable sublanguage solution will then be encoded as a sub-automaton of $H \times G$.

Our interest in this paper is on synthesizing *state-partition-based* feedback control laws that guarantee safety and non-blockingness for a given discrete event system modeled as a finite-state automaton $G$ when the specification is a regular sublanguage $L_{am}$ of the language marked by $G$. We assume that the states of the original model $G$ have physical meaning and that the implementation of the control law should be in terms of legal vs. illegal states of the original model, not on the basis of a refined state space as would occur when constructing $H \times G$ in standard SCT. In other words, given a computed set of *legal* states of $G$, the supervisor will consider as legal any transition between legal states, and it will consider as illegal any transition from a legal state to an illegal one. One may wonder why it is not permitted to define the supervisor as a function over the state set of $H \times G$; such a supervisor would then be implemented as a look-up table whose "state" is updated upon each event occurrence in $G$. To explain our motivation for requiring state-partition-based supervisors, we need to digress to review one control technique for Petri nets and discuss our prior work on controlling the

⋆ The research of the first author was partially supported by Ecole Normale Supérieure de Cachan. The research of the second author was partially supported by US NSF grant CCF-1138860 (Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering).

execution of concurrent software. This is done in the next two paragraphs.

**Control of Petri nets by control places:** A well-known control technique for systems modeled by Petri nets that is maximally permissive for safety control specifications based on linear inequalities on the marking of the Petri net is that of Supervision Based on Place Invariants (or SBPI) [Moody and Antsaklis, 1998]. In that technique, the control specification is given as a set of linear constraints of the form $(l, b)$ that characterize as *illegal* all Petri net markings whose dot product with the vector $l$ is greater than the scalar $b$, for at least one pair $(l, b)$. Thus, the specification results in a partition of the reachable set of markings of the Petri net into legal and illegal ones. This is a convenient way of capturing legal and illegal behavior and it results in a simple state-partition-based implementation of the control law: a transition from a legal marking to a legal one should be enabled, while a transition from a legal marking to an illegal one must be disabled. The SBPI control technique enforces the linear constraints by building a place invariant with a control place added to the net for each linear inequality. Control places result in a "local" implementation of the control law because only the transitions connected to a control place are affected at run-time. This avoids the global bottleneck of an implementation of the control law where the control action has to be updated upon the occurrence of each event (as in standard SCT), which effectively allows only serial execution of the system. Controlling Petri nets by the SBPI technique and their associated control places results in a distributed implementation of the control law that allows for concurrency at run-time. SBPI can also handle uncontrollable events by performing constraint transformation [Moody and Antsaklis, 1998, Iordache and Antsaklis, 2006]. There are several other control techniques for Petri nets that address different types of safety specifications and exploit the structural properties of Petri nets; see, e.g., Holloway et al. [1997], Sreenivas [1997], Seatzu et al. [2013].

**Gadara project:** In our work on deadlock avoidance in multithreaded software treated in [Wang et al., 2008, 2009, Liao et al., 2013c] and referred to as the "Gadara Project [1] ", we use control places as the implementation mechanism of the control law on the Petri net model of the concurrent program. In this case, we showed that the goal of deadlock-avoidance in the program is exactly captured by a set of linear inequalities on the state space of its Petri net model (subject to model accuracy) [Liao et al., 2013b]; structural properties of the net, in the form of siphons, can be exploited to iteratively construct this set of linear inequalities [Liao et al., 2013a]. However, there are other types of concurrency bugs where the control goals can only be captured by regular language specifications. For instance, a class of concurrency bugs called *order violations* requires specifications such as "$b$ occurs only after $a$ has occurred, and no more $a$'s can occur once $b$ has occurred," where $a$ and $b$ are program statements corresponding to transitions in the Petri net model [Lu et al., 2008]. Another line of work tries to enforce only successfully tested thread interleavings in production runs [Yu and Narayanasamy, 2009], where certain interleavings can only be modeled by regular expressions. To solve these problems, we wish to build the reachability graph of the Petri net $N$ (which is assumed to be bounded), work with the automaton representation $G$ of this graph, and synthesize a state-partition-based supervisor for $G$ with respect to the original regular language specification on the language of $N$. The state-partition-based supervisor should be safe and non-blocking, and it should not disable uncontrollable events. If the set of legal states of $G$ corresponding to this state-partition-based supervisor is *linearly separable* from the set of illegal states, SBPI can be used directly to synthesize control places for the resulting linear inequalities. In this regard, the methodology and algorithms in [Nazeem et al., 2011] can be used to obtain a *minimum* set of linear inequalities that effect the desired separation, which will result in a minimum number of control places. More general cases can be handled by dividing the set of legal states into linearly-separable subregions and using a disjunction of linear classifiers, as done in [Cordone et al., 2012]. In either case, the synthesized supervisor will be implementable by control places. In practice, the control places often connect to very few transitions in the Petri net that models the multithreaded program. Most of the transitions (program statements) are not affected by the control law, which preserves concurrency well and incurs very little runtime overhead. The transitions affected by the control places determine which lines of code need to be instrumented to implement the feedback control law captured by the state-partition-based supervisor.

The above discussion provides the motivation for the problem considered in this paper. In summary, we wish to synthesize state-partition-based supervisors for an automaton $G$ subject to a regular language specification $L_{am} \subseteq \mathcal{L}_m(G)$. The state-partition-based supervisor will be completely characterized by a subset of legal states of the state set of $G$, $X_{legal} \subseteq X_G$. This will in general come with a loss of maximal permissiveness as compared with the optimal solution of SCT. But we are concerned with applications where this solution, which is represented as a sub-automaton of $H \times G$, cannot be practically implemented, e.g., for the reasons described above in controlling software execution. More generally, state-partition-based supervisors have the benefit of being "memoryless" in automaton-based control. To the best of our knowledge, none of the existing literature on supervisory control directly addresses the problem we have formulated.

This paper is organized as follows. Section 2 provides a brief summary of notations used throughout the paper and also a review of the basic supervisory control problem, nonblocking version, of SCT. Section 3 formally defines the state-partition-based supervisory control problem considered in this paper. A general algorithm for solving that problem is presented in Section 4. The correctness of that algorithm is demonstrated in Section 5. Finally Section 6 concludes the paper.

## 2. PRELIMINARIES

Due to space limitations, we assume the reader is familiar with the basic results and notations of SCT. Hereafter, we employ the notation of Chapter 3 in [Cassandras and Lafortune, 2008]. The uncontrolled system is

[1] http://gadara.eecs.umich.edu/

modeled by deterministic finite-state automaton $G = (X_G, E, f_G, x_0^G, X_m^G)$ with associated languages $\mathcal{L}_m(G)$ and $\mathcal{L}(G)$, where $\overline{\mathcal{L}_m(G)} \subseteq \mathcal{L}(G)$. It is assumed that the state set $X_G$ of $G$ has *special meaning* inherited from the modeling of the system. For implementation purposes, we shall require that the feedback control law be expressed in terms of a partition of $X_G$ into $X_{legal}$ and $X_{illegal}$. We assume that all events in $E$ are observable and that $E = E_c \cup E_{uc}$, where $E_c$ and $E_{uc}$ are the sets of controllable and uncontrollable events, respectively. For notational convenience, we define the set of transitions of automaton $G$ as :

$$Tr(G) = \{(x, e, y) \in X_G \times E \times X_G : f_H(x, e) = y\}$$

Consider regular language specification $L_{am} \subseteq \mathcal{L}_m(G)$, and such that $\overline{L_{am}} \cap \mathcal{L}_m(G) = L_{am}$, i.e., $L_{am}$ is $\mathcal{L}_m(G)$-closed. Let $L_{am}$ be marked by trim automaton $H$, i.e., $\mathcal{L}(H) = \overline{L_{am}}$ and $\mathcal{L}_m(H) = L_{am}$. Automaton $H$ is defined as $H = (X_H, E, f_H, x_0^H, X_m^H)$. From a language viewpoint, the associated supervisory control problem is, in the terminology of [Cassandras and Lafortune, 2008], the Basic Supervisory Control Problem: Non-Blocking Version, or BSCP-NB:

*Problem 1.* **BSCP-NB**
Given DES $G$ with event set $E$, uncontrollable event set $E_{uc} \subseteq E$, and admissible marked language $L_{am} \subseteq \mathcal{L}_m(G)$, with $L_{am}$ assumed to be $\mathcal{L}_m(G)$-closed, find a *non-blocking* supervisor $S$ such that:

(1) $\mathcal{L}_m(S/G) \subseteq L_{am}$
(2) $\mathcal{L}_m(S/G)$ is "the largest it can be," that is, for any other non-blocking supervisor $S_{other}$:

$$\mathcal{L}_m(S_{other}/G) \subseteq L_{am} \Rightarrow \mathcal{L}_m(S_{other}/G) \subseteq \mathcal{L}_m(S/G) \ .$$

It is well known that the solution of BSCP-NB is the supervisor $S$ such that

$$\mathcal{L}(S/G) = \overline{L_{am}^{\uparrow C}} \ \text{ and } \ \mathcal{L}_m(S/G) = L_{am}^{\uparrow C}$$

as long as $L_{am}^{\uparrow C} \neq \emptyset$. The $\uparrow C$ operation consists of taking the supremal controllable sublanguage (with respect to $\mathcal{L}(G)$ and $E_{uc}$). The desired supervisor $S$ is encoded by the trim automaton $H_{sol}$ such that

$$\mathcal{L}_m(H_{sol}) = L_{am}^{\uparrow C} = \mathcal{L}_m(H)^{\uparrow C}$$

From the standard algorithm for the $\uparrow C$ operation, $H_{sol}$ is a sub-automaton of $H \times G$, denoted by $H_{sol} \sqsubseteq (H \times G)$.

## 3. PROBLEM STATEMENT

In this paper, we take a more restrictive approach that the language-based BSCP-NB reviewed in the preceding section. We assume that we are given $G$ and $L_{am}$ (represented by trim automaton $H$), and we wish to synthesize a *state-partition-based* supervisor, denoted by $S_{spb}$, which will be encoded by a sub-automaton of $G$, denoted by $G_{legal}$. The notion of state-partition-based supervisor is defined as follows. Let $X_G$, the state space of $G$, be partitioned into $X_G = X_{legal} \cup X_{illegal}$, the "legal" and "illegal" state sets, respectively. Let $G_{legal} = Trim\left(G|_{X_{legal}}\right)$, i.e., $G_{legal}$ is the restriction of $G$ to its subset of legal states, $X_{legal}$. Any transition in $Tr(G)$ that originates or ends at a state in $X_{illegal}$ is removed, and then the trim operation is performed. Hence, $G_{legal}$ encodes a supervisor for $G$, $S_{spb} : \mathcal{L}(G) \times E \to 2^E$, defined as follows:

$$e \in S_{spb}(s) \text{ iff } \exists y \in X \text{ s.t. } (f_G(x_0, s), e, y) \in Tr(G_{legal})$$

This definition implies that $S_{spb}$ is defined over the state space of $G$, and the control law of $S_{npb}$ is completely characterized by $X_{legal}$. Any transition between legal states of $G$ is legal, and any transition to/from an illegal state of $G$ (i.e., a state in $X_{illegal}$) is illegal; note that legal transitions may be removed by the trim operation. Of course, an arbitrary $X_{legal}$ may not yield an *admissible* corresponding (state-partition-based) supervisor, as it may violate the *controllability* condition of supervisory control theory, which states that uncontrollable transitions must always be enabled by a supervisor.

In view of the above, we formally formulate the problem addressed in this paper as follows.

*Problem 2.* **State-Partition-Based Control of DES**
Given DES $G$ with event set $E$, uncontrollable event set $E_{uc} \subseteq E$, and admissible marked language $L_{am} \subseteq \mathcal{L}_m(G)$, with $L_{am}$ assumed to be $\mathcal{L}_m(G)$-closed, find a set $X_{legal} \subseteq X$, with its corresponding $G_{legal} = Trim\left(G|_{X_{legal}}\right)$, such that:

(1) $\mathcal{L}(G_{legal})$ is controllable with respect to $\mathcal{L}(G)$ and $E_{uc}$
(2) $\mathcal{L}_m(G_{legal}) \subseteq L_{am}^{\uparrow C}$.

The above definition of $G_{legal}$ ensures that $\mathcal{L}(G_{legal}) = \overline{\mathcal{L}_m(G_{legal})}$, i.e., $G_{legal}$ is non-blocking. Thus, the aim in state-partition-based control is to find $X_{legal}$ such that its corresponding $G_{legal}$ marks a *controllable* language that is a *safe* solution given the specification $L_{am}$. Consequently, $G_{legal}$ will indeed encode a safe and non-blocking state-partition-based supervisor.

The solution to Problem 2 may not be unique. In fact, the requirement of global maximal permissiveness that is an integral part of the formulation of BSCP-NB does not hold anymore in the present context, since there may be incomparable solutions $G_{legal}$ that mark maximal controllable subsets of $L_{am}^{\uparrow C}$. The methodology provided in Section 4 will find *one* solution respecting the set of constraints in Problem 2; it may also be used to find *all* possible solutions $X_{legal}$, if so desired, by repeated application of the algorithms provided therein.

## 4. STATE-PARTITION-BASED CONTROL

### *4.1 Main Algorithm*

We provide in this section an algorithm, called Main Algorithm and formally stated as Algorithm 1 below, that enforces regular language specifications on an automaton using a partition of the state-space of the automaton, according to the requirements of Problem 2.

The first part of Algorithm 1 is to find the optimal solution of Problem 1. As explained in Section 2, the optimal solution is the supremal controllable sublanguage of $L_{am} = \mathcal{L}_m(H)$ with respect to $\mathcal{L}(G)$ and $E_{uc}$. Assume that it is not the empty solution. Let $H_{sol} = (X_{sol}, E, f_{sol}, x_0^{sol}, X_m^{sol})$ be the trim automaton that marks this solution, i.e., $\mathcal{L}_m(H_{sol}) = (\mathcal{L}_m(H))^{\uparrow C}$ and $\mathcal{L}(H_{sol}) = \overline{\mathcal{L}_m(H_{sol})}$.

Automaton $H_{sol}$ is a sub-automaton of the product automaton $H \times G$. Thus, any state in $H_{sol}$ can be mapped back to a state of $G$: $\forall x \in X_{sol}, \exists (x_H, x_G) \in X_H \times X_G : x = (x_H, x_G)$. This means that some states of $G$ are now split in $H_{sol}$; for instance, for some $x_G \in X_G$ there may exist $(x_1, x_2) \in X_H^2$ such that $(x_1, x_G)$ and $(x_2, x_G)$ are both in $X_{sol}$. *The fundamental problem of state based-control is to avoid those split states.* Furthermore, the language generated by $H_{sol}$ is a sublanguage of the one generated by $G$, i.e., $\mathcal{L}(H_{sol}) \subseteq \mathcal{L}(G)$. Hence, some transitions which were initially in $G$ may not have corresponding ones in $H_{sol}$. Those transitions were in fact deleted during the product between $G$ and $H$ or during the $\uparrow C$ operation, as they were either unsafe, violated controllability, or caused blocking. We refer to them as the *illegal* transitions. As we want to create a solution that is sub-automaton of $G$, we have to know what states of $G$ are connected to those deleted transitions. To do so, we introduce the set of transitions denoted by $TRed$, which is the set of transitions that no longer exist in $H_{sol}$:

$$TRed = \{((x_H, x_G), e, (y_H, y_G)) \in X_{sol} \times E \times X_{sol} : ((x_G, e, y_G) \in Tr(G)) \wedge (((x_H, x_G), e, (y_H, y_G)) \notin Tr(H_{sol}))\}$$

From $H_{sol}$ and $TRed$, we construct a new automaton, called $H_{total}$, where we add all the transitions in $TRed$ to $H_{sol}$. In other words, $H_{total} = (X_{sol}, E, f_{total}, x_0^{sol}, X_m^{sol})$, where $f_{total}$ is built in terms of its corresponding $Tr(H_{total})$:

$$Tr(H_{total}) = Tr(H_{sol}) \cup TRed$$

The next step of our algorithm to solve Problem 2 is to find a set of states of $H_{total}$ that *covers* the set $TRed$, in the following sense: for each transition in $TRed$, either its origin state or its destination state should be contained in the covering set. In this manner, if this cover set is deleted from $H_{total}$, then by a trim operation, all the transitions in $TRed$ (which are illegal) will be deleted as well. To solve this problem, we apply a vertex-cover-type algorithm, Algorithm 2, which is described below in Section 4.2. Its output is the set $X_{cover} \subseteq X_{sol}$. By construction of $X_{cover}$, if the state set of $H_{total}$ is restricted only to the set $X_{sol} \setminus X_{cover}$, then the trim automaton $Trim\left(H_{total}|_{X_{sol} \setminus X_{cover}}\right)$ does not contain any illegal transitions of $TRed$. Hence, the language generated by this automaton is by construction safe, but it might not be controllable anymore.

To address the above lack of controllability, the last step of Algorithm 1 is to do another supremal controllable operation, namely, the supremal controllable sublanguage of the language marked by automaton $Trim\left(H_{total}|_{X_{sol} \setminus X_{cover}}\right)$, with respect to $\mathcal{L}(G)$ and $E_{uc}$. That is why we introduce the automaton $H_{final}$ such that $\mathcal{L}_m(H_{final}) = \left(\mathcal{L}_m\left(Trim\left(H_{total}|_{X_{sol} \setminus X_{cover}}\right)\right)\right)^{\uparrow C}$ (with respect to $\mathcal{L}(G)$ and $E_{uc}$). By construction, $\mathcal{L}_m(H_{final})$ is safe and controllable, and automaton $H_{final}$ is non-blocking. When $\left(\mathcal{L}_m\left(Trim\left(H_{total}|_{X_{sol} \setminus X_{cover}}\right)\right)\right)^{\uparrow C}$ is performed, there is no need to build the product automaton

$$\left(Trim\left(H_{total}|_{X_{sol} \setminus X_{cover}}\right)\right) \times G$$

because the necessary state refinement $G$ has already been done in the construction of $H_{sol}$. Hence, in that manner, $H_{final}$ is a guaranteed to be a sub-automaton of $H_{sol}$; this implies that $X_{final} \subseteq X_H \times X_G$.

$H_{final}$ respects all the properties needed and all of its states can be mapped back to a state of $G$. Therefore, we finally obtain the desired set of legal states of $G$, $X_{legal}$, which is defined as follows :

$$X_{legal} = \{x \in X_G : \exists x_H \in X_H, (x_H, x) \in X_{final}\}$$

This selection of $X_{legal}$ is justified by two results that we present and prove in Section 5. For now, we briefly describe the essence of these results. Property 4 insures that each state of $G$ that was initially split in $H_{sol}$ is no longer split in $H_{final}$. In turn, this implies Proposition 6, which proves that the languages generated by $H_{final}$ and $G_{legal}$ are identical. We conclude that $G_{legal}$ also respects the safety, controllability and non-blocking properties, as proved in Theorem 8. The above results are formally stated and proved in the next section, Section 5. Before that, we complete the description of the Vertex Cover Algorithm, Algorithm 2, and present an example.

**Algorithm 1** Main Algorithm

**Input:** $G$, $H$ *s.t.* $\mathcal{L}_m(H) = L_{am}$, $E = E_c \cup E_{uc}$
**Output:** $X_{legal}$
1. *Build* $H_{sol}$*, the solution of Problem 1*
$\mathcal{L}_m(H_{sol}) = (\mathcal{L}_m(H))^{\uparrow C}$
2. *Calculate* $TRed$ *and build* $H_{total}$
$H_{total} = (X_{sol}, E, f_{total}, x_0^{sol}, X_M^{sol})$ and $Tr(H_{total}) = Tr(H_{sol}) \cup TRed$
3. *Run Algorithm 2 on* $H_{total}$ *to generate a solution* $X_{cover}$
4. *Build* $H_{final}$ *by*
$\mathcal{L}_m(H_{final}) = \left(\mathcal{L}_m\left(Trim\left(H_{total}|_{X_{sol} \setminus X_{cover}}\right)\right)\right)^{\uparrow C}$
5. *Build* $X_{legal}$
$X_{legal} = \{x \in X_G : \exists x_H \in X_H, (x_H, x) \in X_{final}\}$

### *4.2 Vertex Cover Algorithm*

In order to select the states that, once deleted from $H_{total}$, will delete all the transitions in $TRed$, we propose Algorithm 2. This is a vertex-cover-type algorithm that allows to find all the possible cover solutions, if it is run in an exhaustive search mode.

The algorithm requires as input the states of $H_{total}$ and also its set of transitions that is divided into two disjoint sets $Tr(H_{sol})$ and $TRed$, as well as the event set $E$. The output of the algorithm is a set of states of $H_{total}$ that is sufficient to cover all the transitions in $TRed$; we denote this output set by $X_{cover}$. We now explain in detail the steps of Algorithm 2.

The first step of the algorithm consists in initializing the local variables. $D$ is the set of states to be deleted to ensure that a transition in $TRed$ will also be deleted. $V$ denotes the set of states that have already been visited and do not need to be expanded anymore. We also introduce $X_1$ which is the set of states that are direct children of the initial state. If a transition from the initial state to one of its children is in $TRed$, then we set the *child* to be deleted, and also set it to "visited", because it is not necessary anymore to expand the search from this successor in later steps as it has already been deleted. It is not appropriate to select the initial state of $H_{total}$ to be deleted because in that case the solution to the state-partition-based control

**Algorithm 2** Vertex Cover Algorithm

**Input:** $X_{sol},\ Tr(H_{total}) = Tr(H_{sol}) \cup TRed,\ E$
**Output:** $X_{cover}$
1. *Initialization*
$X_{cover} = \emptyset$, $D = \emptyset$, $V = \{x_0\}$, $i = 1$
$X_1 = \{x_1 \in X_{sol} : \exists e \in E, (x_0, e, x_1) \in Tr(H_{total})\}$
**for all** $x_1 \in X_1$ **do**
  **if** $(x_0, e, x_1) \in TRed$ **then**
    $D \leftarrow D \cup \{x_1\}$
    $V \leftarrow V \cup \{x_1\}$
    $X_1 \leftarrow X_1 \setminus \{x_1\}$
  **end if**
**end for**
2. *States reachable by strings of length* $i + 1$
**for all** $x_i \in X_i$ **do**
  **if** $x_i \in V$ **then**
    Break
  **else**
    $X_{i+1} = \{x_{i+1} \in X_{sol} : \exists e \in E, (x_i, e, x_{i+1}) \in Tr(H_{total})\}$
    **for all** $x_{i+1} \in X_{i+1}$ **do**
      **if** $(x_i, e, x_{i+1}) \in TRed$ **then**
        Select $x_d = x_i$ or $x_d = x_{i+1}$
        $D \leftarrow D \cup \{x_d\}$
        $V \leftarrow V \cup \{x_d\}$
        **if** $x_d = x_i$ **then**
          $X_{i+1} \leftarrow X_{i+1} \setminus \{x \in X_{sol} : \exists e \in E, (x_i, e, x) \in Tr(H_{total}) \wedge \nexists (y_i, a) \in X_i \times E, (y_a, e, x) \in Tr(H_{sol})\}$
        **else**
          $X_{i+1} \leftarrow X_{i+1} \setminus \{x_d\}$
        **end if**
      **end if**
    **end for**
    $V \leftarrow V \cup \{x_i\}$
  **end if**
**end for**
$i \leftarrow i + 1$
3. *Check if there is any state in* $X_i$ *to explore*
**if** $X_i \neq \emptyset$ **then**
  Go to 2.
**else**
  Go to 4.
**end if**
4. *Return the cover set*
$X_{cover} = D$
Return $X_{cover}$

problem would be empty; hence, that choice need not be considered for these types of transitions in $TRed$.

The iteration step is Step 2. In this step we consider that we have a set $X_i$ of $x_i$'s to be expanded. First, if $x_i \in V$, it means it has already been deleted or expanded in a previous step $k < i$. So it does not need to be examined, and the algorithm heads back for another $x_i \in X_i$. Else, if $x_i \notin V$, then we add to the set $X_{i+1}$ all the direct children of $x_i$, which will be visited in the next iteration. And for all those children $x_{i+1}$, we test if they are connected to $x_i$ with a transition in $TRed$. If yes, we make an *arbitrary choice* between $x_i$ and $x_{i+1}$ for the state to be added into the deleted state set $D$. We call this state $x_d$. Then $x_d$ is set to be visited, i.e., $x_d \in V$ because it is deleted. If $x_d = x_{i+1}$, then we remove it from the set of states to be examined in the next iteration. However if $x_d = x_i$, then we remove all its children from $X_{i+1}$, in order not to examine them (as it is pointless) in iteration $i+1$. Notice however that a child is *not* removed from $X_{i+1}$ if there exists a transition from *another* $x \in X_i$ to the child. We then set the state $x_i$ as visited, i.e., $x_i \in V$. When this procedure is done for all $x_i \in X_i$, we increment the iteration counter and we test if we have other states to examine. If the new $X_i$ is empty, then we are done exploring and the algorithm returns the cover set $X_{cover} = D$; else we go Step 2 again.

*Remark 3.* In an exhaustive search mode for all possible solutions $X_{cover}$, we would consider both choices for $x_d$ at Step 2 and run the remaining of the algorithm for both instances; this would be done each time there is a choice, thereby generating a tree of solutions.

Note that it is possible that the states of $H_{total}$ might not all be examined by the algorithm. This is because the cover algorithm takes care of the connectivity in the transition structure of $H_{total}$. If a state is not visited, it means all the states from which the unvisited state is accessible have been deleted. To save computation time, the algorithm will not analyze this state, and it will be removed by the trim operation when building $H_{final}$.

### *4.3 Example*

To illustrate Algorithm 1 and Algorithm 2, consider Example 3.3 from Chapter 3 in [Cassandras and Lafortune, 2008], which is inspired by concurrency control in database management systems [Lafortune, 1988].

Consider two database transactions, $T_1 = a_1 b_1$ and $T_2 = a_2 b_2$. Event $x_i$ models an operation by transaction $i$ on database item $x$. The uncontrolled execution of $T_1$ and $T_2$ in modeled by automaton $G$ in Fig. 1. From the theory of database concurrency control, the only admissible strings are those where $a_1$ precedes $a_2$ if and only if $b_1$ precedes $b_2$. This specification is modeled by automaton $H$ in Fig. 1. The objective is to build sub-automaton $G_{legal}$ of $G$ such that the generated language is safe with respect to the specification, is controllable, and is non-blocking. In this example, for the sake of simplicity, we assume that all the events are controllable, i.e., $E_{uc} = \emptyset$.

The first step of the main algorithm is to build $H_{sol}$ such that $\mathcal{L}_m(H_{sol}) = (\mathcal{L}_m(H))^{\uparrow C}$. In this particular case, as every event is controllable and as $H \times G$ is non-blocking, $H_{sol}$ and $H$ are isomorphic. Due to space limitations, we set $H_{sol} = H$ (i.e., the states of $H_{sol}$ are renamed). The second step of the main algorithm is construct the set of illegal transitions $TRed$. Since state 5 is the only state of $G$ that is split in $H_{sol}$, we have to include in TRed all the transitions in $G$ that leave or enter state 5. This leads to

$$TRed = \{(2, a_2, 10), (4, a_1, 5), (5, b_2, 8), (10, b_1, 6)\}$$

Indeed, if we consider for example the transition $(2, a_2, 5)$ in $G$, then all the copies of 2 in $H_{sol}$ have to be connected to all the copies of 5 in $H_{sol}$. State 2 is already connected to 5 but not to 10 of $H_{sol}$, so $(2, a_2, 10)$ is added as an illegal transition in $TRed$. Automaton $H_{total}$ is built as automaton $H_{sol}$ with the transitions in $TRed$ added to its transition function; it is shown in Fig. 2.

Fig. 1. Automata $G$ and $H$

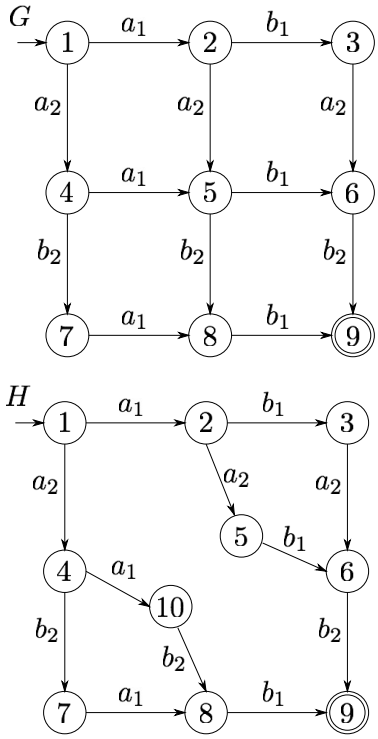
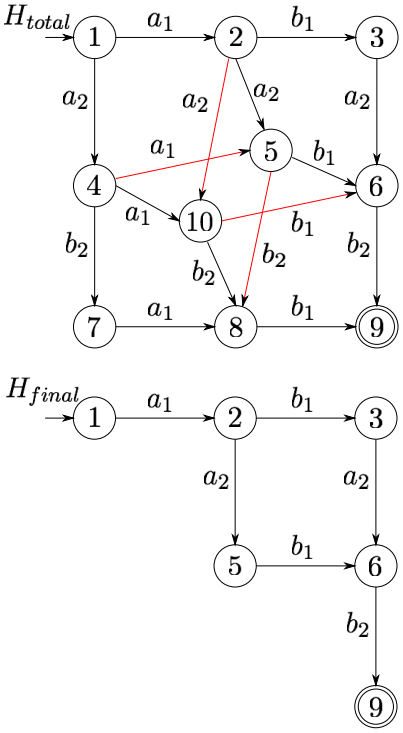



Fig. 2. Automata $H_{total}$ and $H_{final}$




The cover algorithm provides a cover set $X_{cover}$ that will ensure that all the transitions in $TRed$ will be deleted by the trim operation on $H_{total}$ restricted to the set of states not included in the cover. In this example, we consider that the cover algorithm returns $X_{cover} = \{4, 8, 10\}$. From this cover set we build the final automaton $H_{final}$, shown in Fig. 2 and such that $\mathcal{L}_m(H_{final}) = \left(\mathcal{L}_m\left(Trim\left(H_{total}|_{X_{sol}\setminus X_{Cover}}\right)\right)\right)^{\uparrow C}$. Since in this example all events are controllable, this results in $H_{final}$ being the trim of automaton $H_{total}$ restricted to the set of states $X_{sol} \setminus X_{cover}$. The last step is to identify the states of $G$ that are legal. The set of legal states $X_{legal}$ is the set of states of $G$ such that one copy remains in $H_{final}$. In this case, we obtain $X_{legal} = \{1, 2, 3, 5, 6, 9\}$. Finally, $G_{legal}$, shown in Fig. 2, has the same structure as $H_{final}$, so $G_{legal} = H_{final}$. The resulting state-partition-based supervisor is more restrictive than the supremal solution $H_{sol} = H$, but its implementation is state-based as it does not need to memorize how state 5 was reached, which is the essential requirement for state-partition-based supervisors.

We briefly discuss other solutions. The other two interesting (i.e., minimal in terms of set inclusion) solutions for $X_{cover}$ are $X^2_{cover}\{2, 5, 6\}$ (the symmetric solution to above), and $X^3_{cover}\{5, 10\}$. In the case of $X^2_{cover}$, we would obtain $X^2_{legal} = \{1, 4, 7, 10, 8, 9\}$, while for $X^3_{cover}$, we would obtain $X^3_{legal} = \{1, 4, 7, 8, 2, 3, 6, 9\}$. The latter solution represents serial executions of $T_1$ and $T_2$. If some events are uncontrollable, then this may rule out some of the above solutions. For instance, if $a_2$ is uncontrollable, then the first choice of $X_{cover}$, $\{4, 8, 10\}$, now leads to an empty solution (at Step 4 of Algorithm 1). In this case, the symmetric solution $X^2_{cover}\{2, 5, 6\}$ is preferable, as it does not require disabling $a_2$.

## 5. PROPERTIES OF ALGORITHMS

In this section, we prove the correctness of Algorithm 1. We first prove a result about $H_{final}$, Proposition 4, then a result about $G_{legal}$, Proposition 6, and conclude with the main result, Theorem 8.

*Proposition 4.* At most one copy of each state of $G$ remains in $H_{final}$:

$$\forall (x^1_H, x_G) \in X_{final}, \nexists x^2_H \in X_H, x^1_H \neq x^2_H : (x^2_H, x_G) \in X_{final}$$

*Proof 5.* The proof is by contradiction. Let us suppose that two copies of a state of $G$ remain in $H_{final}$:

$$\exists (x^1_H, x^2_H, x_G) \in X_H \times X_H \times X_G, x^1_H \neq x^2_H, \text{ such that: } \begin{cases} (x^1_H, x_G) \in X_{final} \\ (x^2_H, x_G) \in X_{final} \end{cases}$$

We use the notation $x_1 = (x^1_H, x_G)$ and $x_2 = (x^2_H, x_G)$. Since $x_1 \in X_{final}$, we know that this state is accessible from at least one state (because a Trim operation was performed). Hence, there exists a pair $(x_p, e) \in X_{final} \times E$ such that there exists a transition between $x_p$ and $x_1$ in $H_{final}$, i.e., $(x_p, e, x_1) \in Tr(H_{final})$. But by construction of $H_{total}$, there exists a red transition between $x_p$ and $x_2$, i.e., $\exists t \in TRed$ such that $t = (x_p, e, x_2)$. From the cover algorithm and from the Trim operation on $H_{total}$, we know that all the red transitions are deleted by deleting either one of the two states connected to each transition. So for transition $t$, either $x_p$ or $x_2$ should have been deleted in $H_{final}$. If $x_2$ was deleted, then the hypothesis is wrong. On the other hand, if $x_p$ was deleted, then $x_1$ was not accessible anymore and was also deleted. (The same argument can be repeated for each $x_p$ such that

$(x_p, e, x_1) \in Tr(H_{final})$, since each one of them will be connected to $x_2$ by a red transition in $H_{total}$.) We conclude that the hypothesis that $x_1$ and $x_2$ both exist in $H_{final}$ is wrong. Q.E.D.

*Proposition 6.* The languages generated and marked by $G_{legal}$ and $H_{final}$ are the same:

$$\mathcal{L}(G_{legal}) = \mathcal{L}(H_{final})$$
$$\mathcal{L}_m(G_{legal}) = \mathcal{L}_m(H_{final})$$

*Proof 7.* Let us prove first that the generated languages of $G_{legal}$ and $H_{final}$ are equal, i.e., $s \in \mathcal{L}(G_{legal}) \Leftrightarrow s \in \mathcal{L}(H_{final})$. The proof is by induction on the length of strings.
Base Case : $\varepsilon \in \mathcal{L}(G_{legal})$ and $\varepsilon \in \mathcal{L}(H_{final})$.
Induction hypothesis: Assume that $t \in \mathcal{L}(G_{legal})$ iff $t \in \mathcal{L}(H_{final})$ for any $t$ of length $n$.
Induction step: ($\Leftarrow$) : Consider $\sigma \in E$ such that $t\sigma \in \mathcal{L}(H_{final})$. There exists $\left((x_H^1, x_G^1), (x_H^2, x_G^2)\right) \in (X_{final})^2$ such that $\left((x_H^1, x_G^1), \sigma, (x_H^2, x_G^2)\right) \in Tr(H_{final})$. Since $\mathcal{L}(H_{final}) \subseteq \mathcal{L}(G)$, we have that $t\sigma \in \mathcal{L}(G)$. By definition of $X_{legal}$, $x_G^1$ and $x_G^2$ are elements of $X_{legal}$. So $t\sigma$ is also an element of the language of $G_{legal}$, i.e., $t\sigma \in \mathcal{L}(G_{legal})$.
($\Rightarrow$) : Consider $\sigma \in E$ such that $t\sigma \in \mathcal{L}(G_{legal})$. There exists $\left(x_G^1, x_G^2\right) \in X_{legal}^2$ such that $\left(x_G^1, \sigma, x_G^2\right) \in Tr(G_{legal})$. Since $x_G^1$ and $x_G^2$ are elements of $X_{legal}$, then there exists $\left(x_H^1, x_H^2\right) \in X_H^2$ such that $\left((x_H^1, x_G^1), \sigma, (x_H^2, x_G^2)\right) \in Tr(H_{final})$. So $t\sigma$ is also an element of the language of $H_{final}$, i.e., $t\sigma \in \mathcal{L}(H_{final})$.
This completes the proof that $\mathcal{L}(G_{legal}) = \mathcal{L}(H_{final})$.

The second part of the proposition is so a consequence of the first part. By construction, if $x_G$ is a marked state of $G_{legal}$, then there exists $x_H \in X_H$ such that $(x_H, x_G)$ is also marked in $H_{final}$. And as the languages generated by the two automata are the same, then so are their marked languages, i.e., $\mathcal{L}_m(G_{legal}) = \mathcal{L}_m(H_{final})$. Q.E.D.

*Theorem 8.* Algorithm 1 always produces a safe and non-blocking solution.

*Proof 9.* First, we argue that $H_{final}$ represents a safe and non-blocking solution: (i) By construction, $\mathcal{L}H_{final} \subseteq \overline{L_{am}^{\uparrow C}}$ and it is controllable with respect to $\mathcal{L}(G)$ and $E_{uc}$; and (ii) Moreover, $\mathcal{L}_m(H_{final}) \subseteq L_{am}^{\uparrow C}$ and $\overline{\mathcal{L}_m(H_{final})} = \mathcal{L}(H_{final})$. The result is then immediate from Proposition 6. Q.E.D.

*Remark 10.* Algorithm 2 allows to choose between which state to consider as "illegal" for each transition $t \in TRed$. Since this choice is arbitrary, the solution provided by Algorithm 1 will not be unique as it will depend on the instantiation of the choices in Algorithm 2. Let us denote by $S$ the set of all the possible solutions to Problem 2:

$$S = \Big\{ X_{legal} \subseteq X_G : \mathcal{L}_m(G_{legal}) \subseteq L_{am}^{\uparrow C},$$
$$\mathcal{L}(G_{legal}) \text{ is controllable, and } \overline{\mathcal{L}_m(G_{legal})} = \mathcal{L}(G_{legal}) \Big\}.$$

It is not hard to show that any $X_{legal} \in S$ can be obtained from Algorithm 1 by proper selection of the set $X_{cover}$ in Step 3, which in turn is achieved by making the proper choices in Algorithm 2. This means that we can, is so desired, generate all the solutions in $S$ by exhaustively considering all the admissible choices in Algorithm 2. And if we have generated all the solutions in $S$, we will thereby have generated all the *maximal* state-partition-based solutions, in the sense of set inclusion. We can only find maximal solutions because in general the union of two elements of $S$ may not by in $S$ itself, i.e., $(X_1, X_2) \in S^2 \not\Rightarrow X_1 \cup X_2 \in S$.

## 6. CONCLUSION

Motivated by the fact that state-partition-based supervisors are especially advantageous and/or required in several application domains, we presented a new state-partition-based method of controlling automata for enforcing a regular language specification in a non-blocking manner. Starting from a system model in the form of an automaton, the algorithm that we presented constructs a partition of the state set of the automaton that characterizes each reachable state as legal or illegal. This partition must satisfy the following properties: (i) transitions between legal states are always legal while transitions from legal to illegal states are always illegal and controllable; (ii) the restriction of the original system to the set of legal states is non-blocking. Hence, the partition induces a state-partition-based supervisor that is safe and non-blocking with respect to the given regular language specification. When applied exhaustively over all possible selections within it, our algorithm generates all partitions that satisfy the above two properties. Hence, all maximal solutions are also generated. The choice of which maximal solution(s) is/are better is application-dependent, and it will be dictated by the chosen performance optimization goal. For instance, in the concurrent software application area discussed in Section 1, compilers could use profiling to find the maximal solution that results in "maximum" concurrency in practice.

Our methodology can be applied to controlling labeled bounded Petri nets subject to regular language specifications, by constructing the reachability graph of the Petri net and working with its automaton representation. In the case where the obtained partition can be effected by linear inequalities, the resulting supervisor can be implemented by control places, which is advantageous in terms of run-time overhead.

## REFERENCES

C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.

R. Cordone, A. Nazeem, L. Piroddi, and S. Reveliotis. Maximally permissive deadlock avoidance for sequential resource allocation systems using disjunctions of linear classifiers. In *IEEE 51st Annual Conference on Decision and Control (CDC), 2012*, pages 7244–7251, 2012.

L. Holloway, B. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7(2): 151–190, 1997.

M.V. Iordache and P.J. Antsaklis. Supervision based on place invariants: A survey. *Discrete Event Dynamic Systems: Theory and Applications*, 16(4):451–492, 2006. ISSN 0924-6703.

S. Lafortune. Modeling and analysis of transaction execution in database systems. *IEEE Transactions on Automatic Control*, 33(5):439–447, May 1988.

H. Liao, S. Lafortune, S. Reveliotis, Y. Wang, and S. Mahlke. Optimal liveness-enforcing control of a class of Petri nets arising in multithreaded software. *IEEE Transactions on Automatic Control*, 58(5):1123–1138, May 2013a.

H. Liao, Y. Wang, H.K. Cho, J. Stanley, T. Kelly, S. Lafortune, S. Mahlke, and S. Reveliotis. Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets. *Discrete Event Dynamic Systems: Theory & Applications*, 23(2):157–195, June 2013b.

H. Liao, Y. Wang, J. Stanley, S. Lafortune, S. Reveliotis, T. Kelly, and S. Mahlke. Eliminating concurrency bugs in multithreaded software: A new approach based on discrete-event control. *IEEE Transactions on Control Systems Technology*, 21(6):2067–2082, November 2013c.

S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems*, 2008.

J. O. Moody and P. J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers, Boston, MA, 1998.

A. Nazeem, S. Reveliotis, Y. Wang, and S. Lafortune. Designing compact and maximally permissive deadlock avoidance policies for complex resource allocation systems through classification theory: The linear case. *IEEE Transactions on Automatic Control*, 56(8):1818 – 1833, August 2011.

P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control & Optimization*, 25(1), 1987.

P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. IEEE*, 77(1):81–98, January 1989.

C. Seatzu, M. Silva, and J. H. Van Schuppen. *Control of Discrete-Event Systems. Automata and Petri net Perspectives*, volume 433. Springer London, 2013.

R.S. Sreenivas. On the existence of supervisory policies that enforce liveness in discrete-event dynamic systems modeled by controlled Petri nets. *IEEE Transactions on Automatic Control*, 42(7):928–945, 1997.

Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI '08 – Proceedings of the 8th Usenix Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL '09 – Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–263, 2009.

W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM J. Control & Optimization*, 25(3):637–659, May 1987.

J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proc. 36th International Symposium on Computer Architecture*, June 2009.