fmsd manuscript No.
(will be inserted by the editor)

Achieving Distributed Control through Model Checking

Susanne Graf · Doron Peled · Sophie Quinton

Received: date / Accepted: date

Abstract We apply model checking of knowledge properties to the design of distributed controllers that enforce global constraints on concurrent systems. The problem of synthesizing a distributed controller is undecidable in the general case, and local knowledge of processes may not suffice to control them so as to achieve the global constraint without introducing deadlocks. We calculate when processes can decide autonomously, whether to take or block an action so that the global constraint is not violated. When individual processes cannot take a decision alone, one may coordinate several processes in order to achieve stronger, joint knowledge and take joint decisions. A fixed coordination grouping sets of processes to a single process may severely degrade concurrency; therefore, we propose the use of temporary coordinations. Since realizing such coordinations on a distributed platform induces communication overhead, we strive to minimize their number, again using model checking techniques. We show how this framework is applied to the case of synthesizing a distributed controller for enforcing a priority order, as inspired by the BIP component framework. Finally, we show that the general undecidability holds even for the particular problem of enforcing a priority order.

Keywords distributed control · knowledge · model checking · prioritized systems

1 Introduction

Consider a concurrent system, where some global safety constraint, say of prioritizing transitions, needs to be imposed. In a centralized implementation, a global coordinator can control this system and allow only maximal priority actions to progress from each state. We are

Susanne Graf
VERIMAG, Centre Équation, 2, avenue de Vignate, 38610 Gières, FRANCE
E-mail: susanne.graf@imag.fr

Doron Peled
Department of Computer Science, Bar Ilan University, Ramat Gan 52900, ISRAEL
E-mail: doron.peled@gmail.com

Sophie Quinton
Institute of Computer and Network Engineering, TU Braunschweig, 38106 Braunschweig, GERMANY
E-mail: quinton@ida.ing.tu-bs.de
This work was undertaken while Sophie Quinton was at VERIMAG.

however interested in distributed implementations and distributed controllers [14,18]: local controllers, one per process, may forbid the execution of some transitions if their occurrence leads to the violation of the imposed constraint. Due to the distributed nature of the system, each local controller has only a limited view of the system. In the general case, the problem of synthesizing a distributed controller that imposes some global constraint on a system is undecidable [17,15].

One can achieve decidability at the expense of reducing concurrency. In the worst case, no concurrency remains and a completely global controller is built. Even under this flexible design assumption, the general synthesis problem remains highly intractable. One practical method for designing controllers is based on checking *knowledge properties* which the local controllers may use to take a decision whether to allow or block a transition. In [13,1], *knowledge* is used as a tool for constructing a distributed controller. The *knowledge* of a process in any particular local state s is a set of properties which hold in all the reachable (global) states containing s. There are several definitions for knowledge, and what is *known* in a local state s depends on this definition and on how much of the local history may be encoded into the local state.

A *conjunctive* controller allows a transition to be fired only if *all* local controllers allow it. In other words, it is sufficient for a transition to be blocked that one local controller decides it. This is the approach followed by [13], where knowledge-based controllability (termed *Kripke observability*) is studied as a basis for constructing a distributed controller. A *disjunctive* controller as in [1] allows a transition to be fired if *at least one* of the local controllers supports it. This is the approach that we follow here.

In [1], distributed control is achieved by first calculating for each local state using model-checking *knowledge* about the permission to fire transitions without violating the imposed constraint. Based on that knowledge, which reflects in a given local state all the possible current situations of the other processes, the local *controller* of a process decides at runtime whether a transition of that process can be safely fired. Local knowledge is not always sufficient to build a distributed controller that guarantees also deadlock-freedom, that is, a controller which allows at least one transition in any global state in which a transition preserving the constraint exists. This occurs when none of the individual local states has sufficient knowledge, and *joint knowledge* of several processes (sometimes called *distributed knowledge*) may be used in that case. In [1] it is proposed to group processes statically and to use a unique controller for each (fixed) process set, which consequently has stronger local knowledge. Unfortunately, this approach causes the loss of concurrency among processes with a common controller.

The approach presented in this article extends the knowledge-based approach of [1]. Instead of permanent synchronizations via fixed process groups, we suggest here a method for constructing distributed controllers that synchronize processes *temporarily*. We use model-checking techniques to precalculate a minimal set of synchronizations allowing to achieve joint knowledge during these short coordination phases. After each synchronization, the participating processes can again progress independently until a further synchronization is called for. Temporary multiprocess synchronizations are achieved by a coordination algorithm based on asynchronous message passing, such as the α-core algorithm [12]. Such synchronizations are expensive as they incur communication overhead. Therefore, an important goal is keeping the number of such synchronizations and the number of involved processes minimal.

Organization. This article is organized as follows: Section 2 introduces the required background notation and theory, including the notions of local state and of knowledge that we consider. Section 3 generalizes the construction of [1] for priority constraints to arbitrary safety constraints. Section 4 presents our method for constructing distributed controllers by synchronizing processes *temporarily*, and how we reduce the communication overhead induced by these additional synchronizations. Section 5 illustrates our method on an example and provides some experimental results obtained using a prototype implementation of our techniques. Section 6 shows some connections between the classical controller synthesis problem and knowledge-based control. In particular, we show that even for the special case of priority constraints, the distributed control problem is undecidable. This shows the interest of our approach as a practical solution to this problem.

2 Preliminaries

Before presenting a generalization of the support policy introduced in [1], we need to define some concepts related to Petri nets, distributed control and knowledge.

Petri nets. We represent distributed systems as (safe) Petri nets, but the method and algorithms developed here can equally apply to other models, e.g., communicating automata or transition systems, which form the basis of the formalism used in [8].

Definition 1 A *Petri net* N is a tuple (P, T, E, s_0) where

- P is a finite set of *places*. The set of *states* (markings) is defined as $S = 2^P$.
- T is a finite set of *transitions*.
- $E \subseteq (P \times T) \cup (T \times P)$ is a bipartite relation between the places and the transitions.
- $s_0 \subseteq 2^P$ is the *initial state* (initial marking).

For a transition $t \in T$, we define the set of *input places* ${}^\bullet t$ as $\{p \in P | (p,\, t) \in E\}$, and the set of *output places* $t^\bullet$ as $\{p \in P | (t,\, p) \in E\}$.

Definition 2 A transition t is *enabled* in a state s if ${}^\bullet t \subseteq s$ and $(t^\bullet \backslash {}^\bullet t) \cap s = \emptyset$. We denote the fact that t is enabled from s by $s[t\rangle$.

This means that we force Petri nets to be safe by not enabling transitions that would lead to a situation in which some place holds more than one token. A state s is in *deadlock* if there is no enabled transition from it.

Definition 3 A transition t can be *fired* (*executed*) from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled in s. Then, $s' = (s \backslash {}^\bullet t) \cup t^\bullet$.

We use the Petri net of Figure 1 as a running example. As usual, transitions are represented as segments, places as circles, and the relation E as a set of arrows from transitions to places and from places to transitions. The Petri net of Figure 1 has places named $p_1, p_2, \ldots, p_8$ and transitions $a, b, \ldots, e$. We represent a state s by putting *tokens* inside the places of s. In the example of Figure 1, the depicted initial state s_0 is $\{p_1, p_2\}$. The transitions enabled in s_0 are a and b. Firing a from s_0 means removing the token from p_1 and adding one to p_3.

Definition 4 An *execution* is a maximal (i.e., it cannot be extended) alternating sequence of states and transitions $s_0 t_1 s_1 t_2 s_2 \ldots$ with s_0 the initial state of the Petri net, such that for each state s_i in the sequence with $i > 0$, it holds that $s_{i-1}[t_i\rangle s_i$.

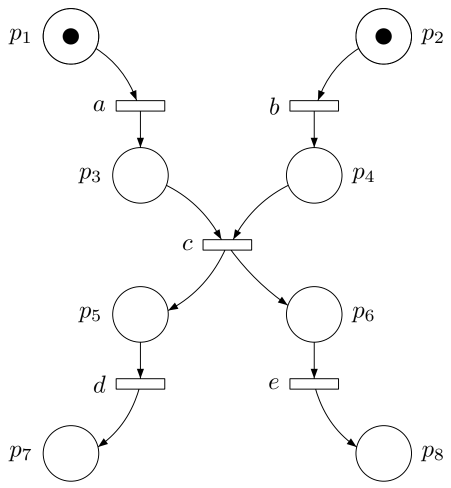


Fig. 1 A Petri net with initial state $\{p_1, p_2\}$

We denote the set of executions of a Petri net N by $exec(N)$. The set of prefixes of the executions in a set X is denoted by $pref(X)$. A state is *reachable* in N if it appears in at least one execution of N. We denote the set of reachable states of N by $reach(N)$. The reachable states of our running example are: $\{p_1, p_2\}$, $\{p_1, p_4\}$, $\{p_2, p_3\}$, $\{p_3, p_4\}$, $\{p_5, p_6\}$, $\{p_5, p_8\}$, $\{p_6, p_7\}$ and $\{p_7, p_8\}$.

Constraints. The constraints that we want to enforce in a distributed way are of the form $\Psi \subseteq S \times T$ with respect to a given Petri net N. If $(s,t) \in \Psi$, then transition t may be safely fired in state s according to Ψ. We denote (N,Ψ) the pair made of a Petri net N and the constraint Ψ that we want to enforce.

Definition 5 A transition t of N is *enabled with respect to* Ψ in a state s if $s[t\rangle$ and, furthermore, $(s,t) \in \Psi$. An *execution* of (N,Ψ) is a maximal prefix $s_0 t_1 s_1 t_2 s_2 t_3 \ldots$ of an execution of N such that for each state s_i in the sequence, $(s_i, t_{i+1}) \in \Psi$.

We denote the executions of (N,Ψ) by $exec(N,\Psi)$, and the set of states reachable by these executions by $reach(N,\Psi)$. Both sets are nonempty as at least the initial state is reachable. Clearly, $reach(N,\Psi) \subseteq reach(N)$. Furthermore, note that N may have states in which some transition is enabled but no transition is enabled *with respect to* Ψ. That is, restricting N according to Ψ may introduce deadlocks. Hence $exec(N,\Psi) \subseteq pref(exec(N))$. The problem we want to solve is the following:

> Given a Petri net with a constraint (N,Ψ), we want to obtain a Petri net N' such that $exec(N') \subseteq exec(N,\Psi)$. In particular, this means that the states in $reach(N')$ which are deadlocks of N' must also be deadlocks of (N,Ψ).

It is sometimes also desired to avoid that (N,Ψ) introduces deadlocks which are not in N. This may be achieved by computing a constraint Ψ' based on Ψ such that: (1) $\Psi' \subseteq \Psi$ and (2) Ψ' blocks only transitions leading to states in which the system cannot progress without violating Ψ. Thus, (N,Ψ') does not introduce deadlocks compared to N. Building Ψ' may be achieved using game theory [16]. Here, we do not focus on this issue and we consider that deadlocks introduced by Ψ are not a problem.

We are interested in particular in enforcing priority orders, which do not introduce deadlocks. Indeed, a priority order $\ll$ is a partial order relation among the transitions T of N and thus defines a constraint Ψ in a straightforward manner. Given a priority order $\ll$, Ψ is defined by: $(s,t) \in \Psi$ if and only if t is enabled in s and has *maximal priority* among the transitions enabled in s — that is, there is no other transition r with $s[r\rangle$ such that $t \ll r$. We write $\mathit{exec}(N,\ll)$ and $\mathit{reach}(N,\ll)$ instead of $\mathit{exec}(N,\Psi)$ and $\mathit{reach}(N,\Psi)$, respectively. As priority orders do not introduce new deadlocks, we have $\mathit{exec}(N,\ll) \subseteq \mathit{exec}(N)$.

Let us now consider the Petri net of Figure 1 and the priorities $a \ll b$ and $d \ll e$. When the priorities are *not* taken into account, there are four different executions of N, namely $abcde$, $bacde$, $abced$ and $baced$ (states are abstracted away). However, when taking the priorities into account, there is only one execution left: $baced$. Thus, priorities are used in this context for scheduling purposes.

Distributed setting. A Petri net can be seen as a distributed system, consisting of a set of concurrently executing and temporarily synchronizing processes. There are several options for defining the notion of process in Petri nets: we choose to consider transition sets as processes.

Definition 6 A *process* π of a Petri net N is a subset of the transitions of N, i.e., $\pi \subseteq T$.

We assume a given set of processes Π_N that covers all the transitions of net N. That is, $\bigcup_{\pi \in \Pi_N} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. Note that we do not require our processes to be sequential, i.e., to hold no more than a single token at any time. In this section, all the notions and notations related to processes extend naturally to sets of processes. Thus, we usually provide definitions directly for sets of processes. Then, when a formula refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π instead.

The neighborhood of a process π describes the places of the system that π can observe. Note that our definition of neighborhood is only one among others, and all our results apply also to other notions of neighborhood. In particular, it may be more realistic not to consider output places as part of the neighborhood, or a process may not even see all input places of its transitions.

Definition 7 The *neighborhood* $\mathit{ngb}(\pi)$ of π is the set of places $\bigcup_{t \in \pi}({}^\bullet t \cup t^\bullet)$.
For a set of processes Π, $\mathit{ngb}(\Pi) = \bigcup_{\pi \in \Pi} \mathit{ngb}(\pi)$.

Definition 8 The *local state* of a set of processes Π in a state $s \in S$ is $s|_\Pi = s \cap \mathit{nbg}(\Pi)$.

That is, the local state of a process π in a global state s consists of the restriction of s to the neighborhood of π. It describes what π can see of s based on its limited view. In particular, according to this definition, any process π can see whether one of its transitions is enabled.

Definition 9 Define an equivalence on states $\equiv_\Pi \subseteq S \times S$ such that $s \equiv_\Pi s'$ when $s|_\Pi = s'|_\Pi$.

Thus, if $t \in \pi$ and $s \equiv_\pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$. Figure 2 represents one possible distribution of our running example. We represent processes by drawing dashed lines between them. Here, the left process π_l consists of transitions a, c and d while the right process π_r

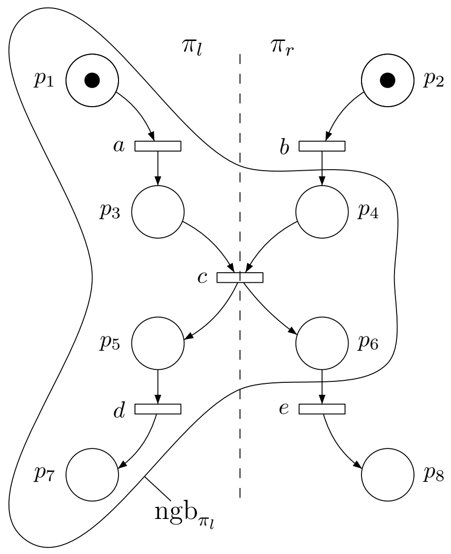


Fig. 2 A distributed Petri net with two processes π_l and π_r

consists of transitions b, c and e. The neighborhood of π_l contains all the places of the Petri net except p_2 and p_8. The local state $s_0|_{\pi_l}$ corresponding to the initial state $s_0 = \{p_1, p_2\}$ is $\{p_1\}$. Note that the local state $s|_{\pi_l}$ corresponding to $s = \{p_1, p_8\}$ is also $\{p_1\}$, hence $s_0 \equiv_{\pi_l} s$.

Representing properties. We identify properties with the sets of states in which they hold. Formally, a *property* is a Boolean formula in which places in P are used as atomic predicates. Then, given a state $s \in S$ and a place $p_i \in P$, we have $s \models p_i$ if and only if $p_i \in s$. For a state s, we denote by φ_s the conjunction of the places of s and the negated places that are not in s. Thus, φ_s is satisfied by state s and by no other state. For the Petri net of Figure 2, the initial state s is characterized by $\varphi_s = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge \neg p_7 \wedge \neg p_8$. A set of states $Q \subseteq S$ can be characterized by a property $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or any equivalent Boolean formula.

Definition 10 A property φ is an *invariant* of a Petri net N if $s \models \varphi$ for each $s \in \mathit{reach}(N)$, i.e., if φ holds in every reachable state.

Below, we provide notations for some properties useful for defining the distributed controllers we are aiming at. For any given Petri net, these properties represent sets of states that can be denoted by a characteristic formula as just explained:

– $\varphi_{reach(N)}$: all the reachable states of N.
  Similarly, $\varphi_{reach(N,\Psi)}$ denotes the reachable states of (N, Ψ).
– $\varphi_{en(t)}$: the states in which transition t is enabled.
– $\varphi_{\Psi(t)}$: the states s in which transition t is enabled and $(s,t) \in \Psi$.
  Formally: $\varphi_{\Psi(t)} = \varphi_{en(t)} \wedge \bigvee_{(s,t)\in\Psi} \varphi_s$.

– φ_{df}^{Ψ}: the reachable states in which at least one transition is enabled w.r.t. Ψ, i.e., the reachable states which are deadlock-free w.r.t. Ψ.
Formally: $\varphi_{df}^{\Psi} = \varphi_{reach(N,\Psi)} \wedge \bigvee_{t \in T} \varphi_{\Psi(t)}$.
– $\varphi_{s|_\pi}$: the states in which the local state of process π is $s|_\pi$.

We can perform model checking in order to calculate these formulae, and store them in a compact way, e.g., using BDDs. For Ψ representing priority constraints, we denote $\varphi_{\Psi(t)}$ by $\varphi_{max(t)}$: it corresponds to the states in which transition t has a maximal priority among all the enabled transitions of the system. That is, $\varphi_{max(t)} = \varphi_{en(t)} \wedge \bigwedge_{t \ll r} \neg\varphi_{en(r)}$.

Knowledge. Our approach for a local or semi-local decision on firing transitions is based on the *knowledge* of processes [6] or of sets of them. Basically, the knowledge of a process π in a given state s is defined by the set of reachable global states s' that are equivalent to s with respect to π, i.e., such that $s \equiv_\pi s'$. For example, in the initial state represented in Figure 2, the left process π_l *knows* that the current global state is $\{p_1, p_2\}$, because it is the only *reachable* state that projects onto local state $\{p_1\}$. Indeed, neither $\{p_1, p_8\}$, nor $\{p_1\}$ nor $\{p_1, p_2, p_8\}$ are reachable. In fact, in this example, both processes always *know* the exact global state of the system based on their local state.

Definition 11 Considering a set of processes Π and a property φ, we define the property $K_\Pi\varphi$ as the set of global states s such that for each *reachable* s' with $s \equiv_\Pi s'$, $s' \models \varphi$. Whenever $s \models K_\Pi\varphi$ for some state s, we say that *Π knows φ in s.*

Equivalently, we sometimes write $s|_\Pi \models K_\Pi\varphi$ rather than $s \models K_\Pi\varphi$ to emphasize that this knowledge property is calculated based on the local state of Π. We easily obtain that if $s \models K_\pi\varphi$ and $s \equiv_\pi s'$ then $s' \models K_\pi\varphi$. Furthermore, process π knows φ in state s exactly when $(\varphi_{reach(N)} \wedge \varphi_{s|_\pi}) \rightarrow \varphi$ is a tautology. Given a Petri net and a property φ, one can perform model checking in order to decide whether $s \models K_\pi\varphi$ for some state s.

3 The Support Policy

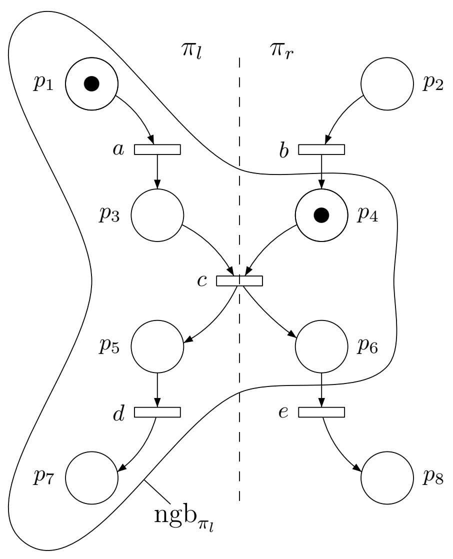
In this section, we generalize the *support policy* introduced in [1] for priorities to any constraint Ψ of the form defined in Section 2. This method uses model checking to analyze the system and identify when a process can decide, based only on its local state, whether some enabled transition is also enabled with respect to Ψ or whether it should be blocked. The support policy is based on a *support table* Δ which indicates, for each local state of each process π the transitions which are known to be enabled with respect to Ψ. These transitions are then *supported* as they may be fired safely according to Ψ, independently of the actual state of the other processes. The basic principle of the support policy is the following:

> In a state s, a transition t is *supported* by a process π containing t if and only if π knows in s (based on its limited view of the system) that $(s,t) \in \Psi$. That is, t is supported by π if and only if $s \models K_\pi\varphi_{\Psi(t)}$. Furthermore, a transition can be fired in a state if and only if, in addition to its original enabledness condition, at least one of the processes containing it, supports it.

The disjunctive nature of the controller resulting from this policy appears in the fact that a transition needs only one local controller to support it in order to be fired.

Building the support table. Given a Petri net N and a constraint Ψ, the corresponding support table Δ is built as follows: we check for each process π, for each local state $s|_\pi$ corresponding to a reachable state $s \in reach(N)$ and each (enabled) transition $t \in \pi$, whether $s|_\pi \models K_\pi \varphi_{\Psi(t)}$. If it holds, we add to the support table an entry $s|_\pi$ and we associate with it the transition t. Note that to check that $s|_\pi \models K_\pi \varphi_{\Psi(t)}$, one has to check that $s' \models \varphi_{\Psi(t)}$ for any reachable state s' such that $s'|_\pi = s|_\pi$. The construction of the support table is simple and its size is limited to the sum of the local state spaces in contrast to the global state space, which might be as large as their product.

The support table Δ for our running example with priorities is shown in Figure 3. For readability, we also show the local states in which no transition is supported. Δ is split into two parts, one per process. The arrows point to the entries in the table (i.e., the local states) corresponding to the global state represented on the left. In this state, process π_r does not support any of its transitions as none of them is enabled. On the other hand, process π_l supports a because it knows that the current global state is $\{p_1, p_4\}$, thus also knowing that b — which has higher priority than a — is not enabled. Note that here one local state corresponds to exactly one global state, but this is in general not the case (see e.g. Figure 4).



Δ^{π_l}

p_1	$\emptyset$
$\longrightarrow$ p_1, p_4	$\{a\}$
p_3	$\emptyset$
p_3, p_4	$\{c\}$
p_5, p_6	$\emptyset$
p_5	$\{d\}$
p_6, p_7	$\emptyset$
p_7	$\emptyset$

Δ^{π_r}

p_2	$\{b\}$
$\longrightarrow$ p_4	$\emptyset$
p_2, p_3	$\{b\}$
p_3, p_4	$\{c\}$
p_5, p_6	$\{e\}$
p_5, p_8	$\emptyset$
p_6	$\{e\}$
p_8	$\emptyset$

Fig. 3 A Petri net with priorities $a \ll b$ and $d \ll e$ along with its support table Δ

Distributed control based on a support table. We use the support table Δ to control (restrict) the executions of N so as to enforce Ψ. Each process π in Π_N is equipped with the entries of this table for its reachable local states $s|_\pi$. Before firing a transition in local state $s|_\pi$, process π consults the entry for this local state in Δ, and supports only the transitions that appear in that entry. This can be represented as an extended Petri net N^Δ, as we explain now. For simplicity of the transformation, we consider extended Petri nets [7], where processes may have local variables, and transitions have an enabling condition and a data transformation.

Definition 12 An *extended Petri net* has, in addition to the Petri net components, for each process $\pi \in \Pi_N$ a finite set of variables V_π and (1) for each variable $v \in V_\pi$, an initial value v_0 (2) for each transition $t \in T$, an enabling condition en_t and a transformation predicate f_t on the variables $V_t = \bigcup_{\pi \in proc(t)} V_\pi$, where $proc(t)$ is the set of processes to which t belongs. In order to fire t, en_t must hold in addition to the usual Petri net enabling condition on the input and output places of t. When t is executed, in addition to the usual changes to the tokens, the variables in V_t are updated according to f_t.

A Petri net N' *extends* N if N' is an extended Petri net obtained from N according to Definition 12. The comparison between the original Petri net N and N' extending it is based only on places and transitions. That is, we ignore (project out) the additional variables.

Lemma 1 *For a Petri net N' extending N, $exec(N') \subseteq pref(exec(N))$.*

Proof The extended Petri net N' can only strengthen the enabling condition of the transitions of N, thus it can only restrict the set of executions and of reachable states. These restrictions may result in new deadlocks, not present in N.

We have the following monotonicity property.

Theorem 1 *Consider a Petri net N and an extension N' of N. Let φ be a property and s a state of N. If $s \models K_\pi \varphi$ in N, then $s \models K_\pi \varphi$ also in N'.*

Proof The extended Petri net N' restricts the set of executions, and possibly the set of reachable states, of N. Each local state $s|_\pi$ is part of fewer global states, and thus the knowledge about φ in $s|_\pi$ can only increase.

Monotonicity is important to ensure Ψ in N^Δ. Indeed, the knowledge allowing to enforce Ψ by the imposed transformation is calculated based on N, but is used to control the execution of the transitions of N^Δ. Monotonicity thus ensures the correctness of N^Δ with respect to Ψ.

The extended Petri net N^Δ is obtained from N and Ψ by defining the additional condition en_t for an enabled transition t to be fired as: $\bigvee_{\pi \in proc(t)} K_\pi \varphi_{\Psi(t)}$. That is, t can be fired if is supported by at least one process containing it. The knowledge properties calculated in Δ are encoded in the variables and updated as transitions are fired. Note that N^Δ is indeed a controlled version of N, as it can only restrict the executions of N. It is distributed, because one set of variables per process is used to define the additional enabledness conditions. Only variables of processes involved in a transition t can be used to determine whether t can be fired or not, and only these variables are updated when t is fired. Finally, it is disjunctive, because a transition can be fired if at least one process supports it.

Note that defining controllers as extended Petri nets allows the use of some finite memory that is updated with the execution of observable transitions. This can be useful, e.g., when constructing a controller based on knowledge with perfect recall [10]. However, a controller based on simple knowledge, as in Definition 11, does not need this capability.

Deadlock-freedom. The extended Petri net N^Δ obtained from N and Ψ obviously enforces Ψ, since only supported transitions are fired, and only transitions which are known to be enabled with respect to Ψ are supported. However, N^Δ does not ensure that no deadlock is added with respect to (N, Ψ). If N^Δ does not introduce any deadlock with respect to (N, Ψ), we say that it *implements* (N, Ψ). We now focus on the issue of determining whether an extended Petri net N^Δ implements (N, Ψ) or not.

Definition 13 We define the following properties k_i^π for a process π:

- $k_1^\pi = \bigvee_{t\in\pi} K_\pi \varphi_{\Psi(t)}$: process π has at least one transition t which it knows to be enabled with respect to Ψ and can thus support.
- $k_2^\pi = \neg k_1^\pi \wedge K_\pi \bigvee_{\rho\neq\pi} k_1^\rho$: process π cannot support any transition, but in all the global states s' with $s'|_\pi = s|_\pi$ some other process ρ is in a local state where k_1^ρ holds. This allows π to remain inactive without risk of introducing a deadlock.
- $k_3^\pi = \neg k_1^\pi \wedge \neg k_2^\pi$: π does not know whether or not there is a supported transition.

k_1^π can be extended to sets of processes: $k_1^\Pi = \bigvee_{t\in T_\Pi} K_\Pi \varphi_{\Psi(t)}$, where $T_\Pi = \bigcup_{\pi\in\Pi} \pi$. Note that $k_1^\pi \vee k_2^\pi \vee k_3^\pi \equiv true$.

The construction in [1] checks whether $\bigvee_{\pi\in\Pi_N} k_1^\pi$ holds in all reachable states of the original system which are neither deadlock nor termination states. If so, it is sufficient that each process supports a transition t when it knows that t does not violate the constraint Ψ (only priority policies are handled in [1]) in order to enforce Ψ without introducing any additional deadlock. The next section discusses what to do when this check fails.

4 A Synchronization-Based Approach

It is not possible in general to decide, based only on the local state of a process, whether some enabled transition is allowed by Ψ. That is, there are cases where the support policy introduced in the previous section fails in the sense that N^Δ as defined by the support policy does not implement (N,Ψ), because N^Δ has more deadlocks than (N,Ψ). Before discussing existing solutions to this issue and presenting a new one, let us look at an example where this situation arises.

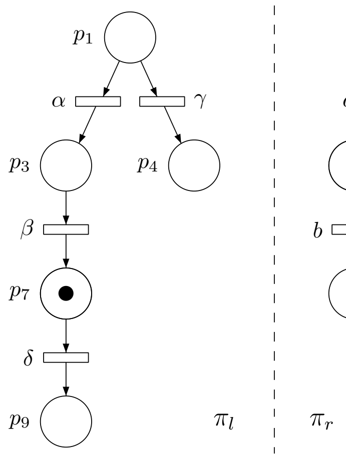
An example where the support policy fails. Consider a concurrent system as in Figure 4, with two processes π_l (left) and π_r (right) with disjoint sets of transitions, each of them having initially a nondeterministic choice. The priorities in this system are $\delta \ll b \ll \beta$. Note that each process can observe only its own transitions.

In the initial state, all four enabled transitions α, γ, a, c are maximal as they are not ordered by priorities. If α is fired and subsequently a (or vice versa), we reach a global state in which process π_r does not have any enabled transition with maximal priority since $b \ll \beta$. Process π_l does, and it can execute β. Thereafter, since $\delta \ll b$, process π_l cannot execute δ and must wait for process π_r to execute b. Now, with its limited observability, in its local state $\{p_7\}$, π_l cannot distinguish the states $\{p_7, p_5\}$, in which b is enabled and δ not allowed, and $\{p_7, p_8\}$, in which no other transition is enabled and δ can be fired. Thus in $\{p_7\}$, π_l does not have sufficient knowledge for executing δ.

This shows that local knowledge of the processes is not always sufficient to construct a controller. Here, in the initial state, both processes can progress freely, but then reach a situation where they cannot decide locally on how to progress.

Existing solutions. To handle situations where the support policy fails, two suggestions have been made:

1. Use knowledge of perfect recall [10,1]. This means that the knowledge is not based only on the local state, but also on the limited history that each process can observe. Although the history is not finitely bounded, it is enough to calculate the set of states where the rest of the system can reside at each point. A subset construction can be used



Δ^{π_l}

	p_1	$\{\alpha, \gamma\}$
	p_3	$\{\beta\}$
	p_4	$\emptyset$
$\longrightarrow$	p_7	$\emptyset$
	p_9	$\emptyset$

Δ^{π_r}

	p_2	$\{a, c\}$
$\longrightarrow$	p_5	$\emptyset$
	p_6	$\emptyset$
	p_8	$\emptyset$

Fig. 4 A Petri net with priorities $\delta \ll b \ll \beta$

to supply for each process an automaton that is updated according to the local history. This construction is very expensive: the size of this automaton can be exponential in the number of global states. Furthermore, although in this way we extend our knowledge (by separating local states according with different histories), this still does not guarantee that a distributed controller can be found. In particular, knowledge of perfect recall is useless in the situation of Figure 4.

2. Combine the knowledge of some processes by synchronizing them to form a single process [1]. Such a combined process may *know* more than its constituting processes separately. This approach may however lead to an important loss of concurrency: in the worst case, all processes must be combined and no concurrency remains.

Adding synchronizations to provide sufficient knowledge. Instead of the fixed synchronization between processes suggested in [1], we propose to use *temporary* synchronizations: processes coordinate to achieve *joint* knowledge (i.e., knowledge of a set of processes), whenever their local knowledge is not sufficient to ensure deadlock-freedom. This does not reduce the concurrency as much as the previous method, but induces some communication overhead as the temporary synchronizations are achieved at the coordination level through additional exchange of messages.

We now propose to calculate the support table Δ iteratively, by first adding entries corresponding to local states of single processes as described in Section 3, then (joint) local states of pairs of processes, then triples etc. At each stage of the construction, Δ is identified with its set of entries, which are (joint) local states $s|_{\Pi}$ satisfying k_1^{Π} — that is, local states in which the joint knowledge of the processes in Π is sufficient to ensure progress. Note that the joint local state of a set of processes Π can be seen as a tuple consisting of the local states of the processes in Π.

Definition 14 A set of (joint) local states Δ is an *invariant* if for each reachable global state s that is not a deadlock in (N, Ψ), Δ contains at least one (joint) local state in s.

The first iteration includes in Δ, for every $\pi \in \Pi_N$, the singleton local states satisfying k_1^{π}, i.e. states in which progress of π is guaranteed. With each entry corresponding to such a local state $s|_\pi$, we associate the actual transitions t that make k_1^π hold. If after the first iteration $\varDelta$ is invariant, then the method presented in the previous section is sufficient to build $N^\varDelta$ implementing $(N, \varPsi)$. Otherwise, we consider also joint local states.

Definition 15 A *synchronization state* is a reachable global state s that is a not a deadlock in $(N, \varPsi)$ and such that $s \not\models k_1^\pi$ for any process $\pi \in \varPi_N$.

The existence of a synchronization state means that $\varDelta$ is not invariant after the first iteration described above. We propose adding some joint local states for achieving invariance. That is, such states will require additional synchronization, hence their name.

Consider a process $\pi \in \varPi_N$. We first calculate for each local state of π not satisfying k_1^π whether it satisfies k_2^π. Let U_π be the set of local states of process π satisfying k_3^π, that is, satisfying neither k_1^π nor k_2^π. Now, in a second iteration, we add to $\varDelta$ pairs $(s_\pi, s_\rho) \in U_\pi \times U_\rho$ for $\pi \neq \rho$ such that there exists a synchronization state s with $s|_\pi = s_\pi$, $s|_\rho = s_\rho$ and furthermore $s \models k_1^{\{\pi,\rho\}}$. Again, we associate with that entry of the table $\varDelta$ the transitions t that witness the satisfaction of $k_1^{\{\pi,\rho\}}$. The second iteration terminates as soon as $\varDelta$ is an invariant or if all such pairs of local states have been classified — that is, added to $\varDelta$ or discarded.

In a third iteration, we consider triples of local states from $U_\pi \times U_\rho \times U_\sigma$ such that no subtuple is in $\varDelta$, and so forth. Eventually, $\varDelta$ becomes an invariant, in the worst case when complete synchronization states are added to $\varDelta$: indeed, synchronizing all the processes ensures that any transition enabled and allowed by $\varPsi$ in such a state will be supported. Our construction guarantees that each time the transition t associated with a tuple $(s|_{\pi_1} \dots s|_{\pi_k})$ from $\varDelta$ is executed from a state that includes these local components, t is indeed enabled with respect to the constraint $\varPsi$.

If we go back to our example, the support table presented in Figure 4 is not an invariant. It must be enriched with the entries for the joint local states given in Table 1. Note that in this example, there exist two synchronization states, namely $\{p_5, p_7\}$ and $\{p_7, p_8\}$, each corresponding to one entry in the support table. However, it is not always necessary to have as many entries in the table as there are synchronization states: if there are more than two processes in the system, a single synchronization may be sufficient to ensure progress in several synchronization states. Note also, that in an execution of our example according to the augmented table $\varDelta$, a synchronization may take place only when the system has actually reached a synchronization state. This again is not the case in general. Indeed, two processes may decide to synchronize because they both know that a synchronization state may have been reached, although this is actually not the case.

$\varDelta^{\pi_l, \pi_r}$

p_5, p_7	$\{b\}$
p_7, p_8	$\{\delta\}$

Table 1 Additional entries for the support table of Figure 4 to become an invariant

A distributed controller imposing the global property. We now have to explain how the joint local knowledge used to enforce the invariance of $\varDelta$ is achieved in practice. Indeed, the method proposed in the previous section for building the extended Petri net $N^\varDelta$ from N and $\varPsi$ does not apply directly. The reason is that joint knowledge cannot be expressed by disjoint sets of variables. We solve this by adding *synchronizations* amongst the processes involved.

Such synchronizations are achieved by using an algorithm like α-core [12], which allows processes to notify, using asynchronous message passing, a set of coordinators about their wish to be involved in a joint action. Once a coordinator has been notified by all the participants in the synchronization it is in charge of, it checks whether conflicting synchronizations are already under way (a process may have notified several coordinators). If this is not the case, coordination succeeds, and the synchronization can take place. We assume that the correctness of the algorithm guarantees the atomic-like behavior of the coordination process, allowing us to reason at a higher level of abstraction where we treat the synchronizations provided by α-core (or any similar algorithm) as transitions that are joint between several participating processes.

Thus, if a transition t is associated with a singleton element $s|_\pi$ in Δ, then the controller for π, in local state $s|_\pi$, supports t. Otherwise, t is associated with a tuple of local states in Δ; when reaching any of these local states, the corresponding processes $\pi_1 \ldots \pi_k$ try to achieve a synchronization using the coordination algorithm. If coordination succeeds, and the synchronization takes place, the associated transition t is then supported by all the participating processes (there may be several such transitions). Formally, for each transition t associated with a tuple of local states $(s|_{\pi_1} \ldots s|_{\pi_k})$, we execute a transition enabled exactly in the global states containing this tuple and performing the original transformation of t.

Minimizing the number of coordinators. It is wasteful to set up one coordination for each joint local state involving at least two processes in Δ. We now show how to minimize the number of coordinators for pairs of the form $(s|_\pi, r|_\rho)$ in Δ. The general version of this method for larger tuples is analogous. We denote by $\Delta_{\pi,\rho}$ the set of pairs of Δ made of a local state from process π and one from process ρ.

A naive implementation may use a coordination for every pair in Δ. Nevertheless, the large number of messages needed to implement coordination by an algorithm like α-core suggests that we minimize their number. The opposite extreme would be to use a unique coordination between every two processes π and ρ. However, as α-core does not offer guarded coordinations, success of a coordination does not imply in this case that the resulting synchronization will be useful. Thus, many (expensive) useless coordinations may be achieved, not even guaranteeing eventual progress.

We propose an intermediate solution. Consider now a set of pairs $\Gamma \subseteq \Delta_{\pi,\rho}$ such that if $(s, r), (s', r') \in \Gamma$, then also $(s, r'), (s', r) \in \Gamma$ (s and s' do not have to be disjoint, and neither do r and r'). This means that Γ is a complete bipartite subgraph of $\Delta_{\pi,\rho}$. It is sufficient to generate one coordination for all the pairs in Γ: upon success of the coordination, the precalculated table $\Delta_{\pi,\rho}$ will be consulted about which transition to allow, depending on $s|_\pi$ and $s|_\rho$. Thus, according to this strategy, a sufficient number of coordinations is formed by finding a covering partition $\Gamma_1, \ldots, \Gamma_m$ of complete bipartite subgraphs of $\Delta_{\pi,\rho}$. That is, each pair $(s|_\pi, r|_\rho) \in \Delta_{\pi,\rho}$ must be in some set Γ_i. However, the minimization problem for such a partition turns out to be in NP-Complete, as stated in the following theorem.

Property 1 [11] Given a bipartite graph $G = (V, E)$ and a positive integer $K \leq |E|$, finding whether there exists a set of subsets $V_1, \ldots V_k$ for $k \leq K$ of complete bipartite subgraphs of G such that each edge (u, v) is in some V_i is in NP-Complete.

We use the following notation: when Γ is a set of pairs of local states, one of π and the other of ρ, we denote by $\Gamma|_\pi$ and by $\Gamma|_\rho$ the π and the ρ components in these pairs, respectively. We suppose that $|\Delta_{\pi,\rho}|_\pi| \leq |\Delta_{\pi,\rho}|_\rho|$, i.e., the number of elements paired up in $\Delta_{\pi,\rho}$ is smaller for π than for ρ. If this is not the case, one simply has to replace π with ρ

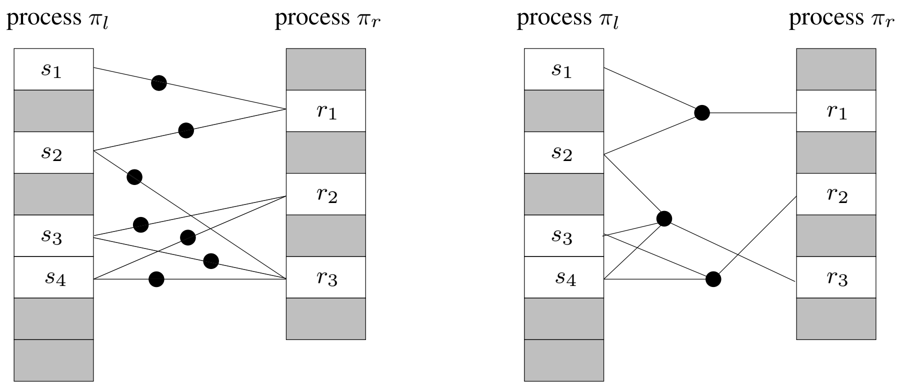


Fig. 5 Minimizing the number of coordinators

and vice versa in the sequel. We apply the following heuristics to calculate a (not necessarily minimal) set of complete bipartite subsets $\varGamma_i \subseteq \varDelta_{\pi,\rho}$ covering $\varDelta_{\pi,\rho}$.

Let the elements of $\varDelta_{\pi,\rho}|_\pi$ (that is, the π components of the pairs in $\varDelta_{\pi,\rho}$) be $x_1, \ldots, x_m$. We start with a first partition $\varGamma_1, \ldots, \varGamma_m$ where $\varGamma_i$ is the set of pairs in $\varDelta_{\pi,\rho}$ containing x_i, for any $i \in [1, m]$. These sets are obviously complete, and the partition is a covering.

Now, in order to refine this partition, we check for each two sets $\varGamma_i$ and $\varGamma_j$ whether $\varGamma_i|_\rho = \varGamma_j|_\rho$. If it is the case, we merge them into a single set $\varGamma_i \cup \varGamma_j$. The resulting set is complete because it contains a pair (x_i, y) if and only if it also contains (x_j, y), where $y \in \varDelta_{\pi,\rho}|_\rho$. Note that each x_i always appears in exactly one subgraph, thus we cannot repeat the process for π.

Figure 5 shows how this heuristics works for an example. Lines represent pairs of local states which must be synchronized; blacks dots represent coordinators. The left-hand side of the figure shows the coordinators induced by $\varDelta_{\pi,\rho}$ and the right-hand side the minimal set of coordinators that is obtained by our heuristics. We start with process π_r: each $\varGamma_i$ contains a single state of π_r. In this case, the initial partition turns out to be already the solution. Note that starting with process π_l would also result in a solution with three coordinators, because the coordinators for s_3 and s_4 can be merged.

5 Implementation and Experimental Results

We have implemented a prototype for experimenting with this approach. This tool first builds the set of reachable states and the corresponding local knowledge of each process. Then, it checks whether local knowledge is sufficient to ensure correct distributed execution of the system under study. We allow simulating the system while counting the number of synchronizations and synchronization states encountered during execution as a measurement of the amount of additional synchronization required.

One example that we used in our experiments is a variant of the dining philosophers where philosophers may arbitrarily take first either the fork that is on their left or right, provided it is on the table. In addition, a philosopher may hand over a fork to one of his neighbors when his second fork is not available and the neighbor is looking for a second fork as well. Such an exchange (labeled ex) is a way to avoid the well-known deadlocks when all philosophers hold one fork in their left (respectively right) hand: our philosophers are pragmatic enough to exchange forks when they have nothing better to do. This example is partially represented by the Petri net of Figure 6.

In our example, places (concerning philosopher β) are defined as follows:

- $fork^i$: the i-th fork is on the table.
- $0fork_\beta$ (respectively $2forks_\beta$): philosopher β has no fork (respectively 2 forks) in his hands.
- $1fork^l_\beta$ (respectively $1fork^r_\beta$): philosopher β holds his left (respectively right) fork.

Transitions (concerning philosopher β) play the following role:

- get^{kl}_β (respectively get^{kr}_β), $k = 1, 2$: philosopher β takes the fork on his left (respectively on his right). This is his k-th fork.
- *eat-and-return*$_\beta$: philosopher β eats and puts both forks back on the table.
- $ex_{\alpha,\beta}$: philosopher α gives his right fork to philosopher β.
- $ex_{\beta,\alpha}$: philosopher β gives his left fork to philosopher α .

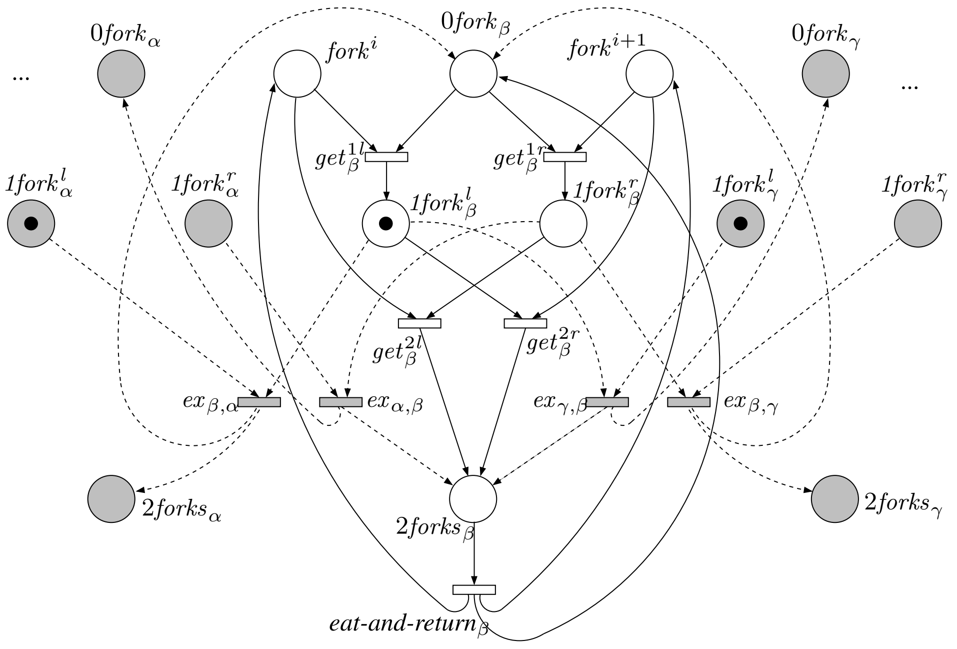


Fig. 6 A partial representation of the dining philosophers (philosopher β)

Processes correspond to philosophers. The transitions defining a process β are those with a β in their name, including the four exchange transitions $ex_{\alpha,\beta}$, $ex_{\beta,\alpha}$, $ex_{\beta,\gamma}$ and $ex_{\gamma,\beta}$. In Figure 6, transitions related only to philosopher β are in blue. Transitions in orange and green are shared between β and one of his neighbors (respectively α on the left and γ on the right).

Not controlling exchanges at all allows non-progress cycles, that is, philosophers exchanging forks without ever eating. To avoid this, we add priorities which allow exchange actions only when a blocking situation has been reached within some degree of locality.

First variant. We use a priority rule stating that an exchange between philosophers α and β has lower priority than α or β taking a fork. This leads to the following priorities for each α and β such that α is βs left neighbor:

– $ex_{\alpha,\beta} \ll get^{2l}_{\alpha}$: if α can pick up a left fork, he may not give his right fork to β.
– $ex_{\beta,\alpha} \ll get^{2r}_{\beta}$: symmetrically if β can pick up a right fork.

In this variant, local knowledge is sufficient. Indeed, when philosopher β and both his neighbors are blocked in a state where they all have a left (respectively a right) fork, then philosopher β has enough knowledge to support an exchange with his left (respectively right) neighbor, because he knows that he has nothing better to do. For any number of philosophers, there is no synchronization state. Thus, no extra synchronization is needed.

Second variant. Now, to further reduce the number of exchanges, one may decide that philosopher β may give his left fork to his left neighbor α only if (1) α is blocked (2) β is blocked and (3) βs right neighbor γ is also blocked (the exchange of right forks is similar).This translates into adding the following priorities:

– $ex_{\alpha,\beta} \ll get^{2l}_{\delta}$, *eat-and-return*$_{\delta}$ (with δ the left neighbor of philosopher α)
– $ex_{\beta,\alpha} \ll get^{2r}_{\gamma}$, *eat-and-return*$_{\gamma}$ (with γ the right neighbor of philosopher β)

Local knowledge alone cannot ensure here correct distributed execution. However, binary synchronizations are sufficient in this example to ensure that the system is always able to move on, for any number of philosophers.

In Table 2, we show results for the second variant with 6, 8 and 10 philosophers. In all cases, there are two synchronization states which correspond to the situation where all philosophers hold their left fork, or they all hold their right fork. For computing the number of synchronizations, we used each time 100 runs of a length of 10,000 steps (i.e. transitions). Note that the number of exchange transitions is identical to the number of synchronizations. We provide the average number of synchronizations over 100 executions of 10,000 steps according to two different strategies:

1. Synchronizations are allowed only when no other transition is supported.
2. Synchronizations and supported transitions have the same probability.

The first strategy (denoted *min* in Table 2) cannot be distributed and only aims at simulating how many synchronizations are needed to escape the synchronization states encountered during execution. The second strategy (denoted *average*) is implementable in a distributed setting. As one can see, this strategy increases the number of synchronizations taking place (because a synchronization can take place as soon as the philosophers involved in it all believe that a synchronization state may have been reached), but allows reaching synchronization states less often. Thus, the communication overhead induced by synchronizations which are unnecessary with respect to deadlock-freedom is compensated by the added degree of progress achieved by the system.

6 A Practical Solution to the Distributed Control Problem

We now show some connections between the classical controller synthesis problem (see, e.g., [14]) and knowledge-based control. We have provided a solution to the synthesis of distributed controllers, based on adding synchronizations in order to combine the knowledge

philosophers	6		8		10	
strategy	min	average	min	average	min	average
reachable states	729		6561		59049	
synchronizations	322	354	229	285	178	237
synchronization states encountered		253		149		100

Table 2 Results for 100 executions of 10,000 steps for the second variant

of individual processes. In this section, we want to put the knowledge-based solution in the context of the distributed control problem when adding synchronizations is *not* allowed.

The knowledge approach to control in [13] requires that there is sufficient knowledge to allow *any* transition of the controlled system that does not violate the constraint Ψ. In [1], which we extend here, this requirement is relaxed; the knowledge must suffice to execute *at least one* enabled transition not violating Ψ when such a transition exists. In the more general case of distributed controller design, one may want to block some enabled transitions even if their execution does not immediately violate the enforced property. This is required to prevent the transformed system from later reaching deadlock states, where the controlled system originally had a way to progress (thus, introducing new deadlocks). When a controller is allowed to block transitions even when their execution does not immediately lead to violation of the property to be preserved, the situation can be recovered.

Notice that the Petri net of Figure 4, where local knowledge is not sufficient for controlling the system, can be easily controlled by blocking transitions, even when they are known to be maximal: we may choose either to block α in favor of γ, or to block a in favor of c. Blocking both α and a is not necessary. This illustrates that distributed controllers are more general than knowledge-based controllers. This example also shows that there is no *unique maximal* solution to the control problem that blocks the *smallest* number of transitions. Note that an alternative solution to blocking α or a can be achieved using a temporary synchronization between the processes, as shown earlier.

Actually, even if we redefine our knowledge-based controllers to allow them to block transitions in order to avoid a deadlock later in the execution, they will still be less powerful than general distributed controllers. The reason for that is that local (knowledge-based) controllers lack the ability to agree *a priori* on transitions which should be taken or not. Consider a variant of the Petri net of Figure 4, where a situation similar to that between δ, b and β occurs in the right branch of the processes after γ and c are fired. A distributed controller can still decide that the left process will go left while the right process will go right. This is not feasible using knowledge-based controllers.

On the other hand, there is no algorithm that guarantees constructing distributed controllers. It was shown in [17, 15] that the problem of synthesizing a distributed controller is, in general, undecidable. We show here that even when restricting the synthesis problem to priority policies, the problem remains undecidable. The proof for that is given below. Notice that when we have the flexibility of allowing additional synchronizations, as proposed in Section 4, the problem, in the limit, becomes a sequential control problem, which is decidable.

Theorem 2 *How to construct a distributed controller that enforces a priority order is an undecidable problem.*

Proof Following [17], the proof is by reduction from the post correspondence problem (PCP). In PCP, there is a finite set of pairs $\{(l_1, r_1), \ldots, (l_n, r_n)\}$, where the components

l_i, r_i are words over a common alphabet Σ, and one needs to decide whether one can concatenate separately a *left word* from the left components and a *right word* from the right components according to a mutual nonempty sequence of indices $i_1 i_2 \ldots i_k$, such that $l_{i_1} l_{i_2} \ldots l_{i_k} = r_{i_1} r_{i_2} \ldots r_{i_k}$.

Let $i \in \{1..n\}$, $\hat{l}_i$ be the word $l_i\, i$, i.e., the i-th left component concatenated with the index i. Similarly, let $\hat{r}_i$ be $r_i\, i$. We consider two regular languages: $L = (\hat{l}_1 + \hat{l}_2 + \ldots + \hat{l}_k)^+$ and $R = (\hat{r}_1 + \hat{r}_2 + \ldots + \hat{r}_k)^+$. Now suppose a process π_p executes according to the regular expression $l.L.x.a.b + r.R.x.c.d$. The choice of π_p between l and r is uncontrollable. Suppose also that π_p coordinates (through shared transitions) the alphabet letters from Σ with a process π_{q_1}, and the indices letters from $\{1, \ldots, n\}$ with another process π_{q_2}. After that, π_{q_1} and π_{q_2} are allowed to interact with each other. Specifically, π_{q_2} sends π_{q_1} the sequences of indices it has observed. Suppose that now π_{q_1} has a nondeterministic choice between two transitions: α or β. The priorities are set as $b \ll \alpha \ll a$ and $d \ll \beta \ll c$. All other pairs of transitions are unordered according to $\ll$. If π_{q_1} selects α and r was executed, or π_{q_1} selects β and l was executed, then there is no problem, as α is unordered with respect to c and d, and also β is unordered with respect to a and b, respectively. Otherwise, there is no way to control the system so that it executes the sequence $a.\alpha.b$ or $c.\beta.d$ allowed by the priorities.

We show by contrapositive that if there is a controller, then the answer to the PCP problem is negative. Suppose the answer to the PCP problem is positive, i.e., some left and right words are identical and with the same indices. Then process π_{q_1} cannot make a decision: the information that π_{q_1} observed and later received from π_{q_2} is exactly the same in both cases for the mutual left and right word. Thus, π_{q_1} cannot anticipate whether $c.d$ or $a.b$ will happen and cannot make a safe choice between α and β accordingly.

Conversely, if there is no controller, it means that π_{q_1} cannot make a safe choice between α and β. This can only happen if π_{q_1} and π_{q_2} can observe exactly the same visible information for both an l and an r choice by π_p.

This means that deciding the existence of a controller for this system would solve the corresponding PCP problem. It is thus undecidable.

Note that in this proof we do not ensure a finite memory controller, even when one exists. Indeed, a finite controller may not exist. To see this, assume a PCP problem with one word $\{(a, aa)\}$. To check whether we have observed a left or a right word, we may just compare the number of a's that π_p has observed with the number of indices that π_q has observed.

We have shown that even our limited problem (and running example) of controlling a system according to priorities is already undecidable. This advocates that the construction of knowledge-based controllers, and furthermore, the use of additional synchronization, is a practical solution to the distributed control problem.

7 Conclusion and Perspectives

Imposing a global constraint upon a distributed system by blocking transitions is, in general, undecidable [17,15]. One practical approach for this problem is to use model checking of knowledge properties [1], where a precalculation is used to determine when processes can decide, autonomously, to take or block an action so that the global constraint will not be violated. If we allow additional synchronizations, the problem becomes decidable: at the limit, everything becomes synchronized, although this, of course, is highly undesirable. Since the overhead induced by such synchronizations is important, we strive to minimize their number, again using model checking. This framework applies in particular to the design of controllers that enforce a priority order among transitions.

For achieving a distributed implementation, one can use a multi-party synchronization algorithm such as the α-core algorithm [12]. Based on that, we presented an algorithm that uses model checking to calculate when synchronization between local states is needed. The synchronizing processes, successfully coordinating, are then able to use the support table calculated by model checking, which dictates to them which transition can be executed. Some small corrections to the original presentation of the α-core algorithm appear in [9].

Finally, we have proved that the distributed control problem is undecidable even for the special case of enforcing a priority order, which is the original motivation for this work. This advocates using knowledge-based controllers enriched with additional synchronizations as a practical solution to the distributed control problem.

Further work. In [3], we have observed that the knowledge used for constructing a distributed controller is computed based on the original (uncontrolled) system. Thus, it may be pessimistic in concluding when transitions can be supported. This has led us to two useful observations that can remove the need for some of the additional synchronizations used to control the system:

1. Although the analysis of the knowledge of the system is done with the original system, it is safe to use only the executions that satisfy the constraint. This results in fewer executions and fewer reachable states, thus enhancing the knowledge.
2. Blocking transitions (not supporting them) because of lack of knowledge has a propagating effect that can prevent reaching other states. Thus, even when the support policy may seem to fail without additional synchronization, this may not be the case. Indeed, analyzing the system when it is restricted according to the support table may be sufficient: the deadlocks corresponding to states where no enabled transition is supported are in fact unreachable.

We have shown that using these two observations is orthogonal to other tools used to force knowledge-based control such as using knowledge of perfect recall and adding temporary or fixed synchronizations between processes. More precisely, the support table is actually built directly from the set of reachable states in the prioritized executions. Then, if the table contains empty entries, we check the reachability of the states in which no transition can be supported before adding synchronization.

Perspectives. There are many interesting ways of refining the approach presented here. A key question is to find a compromise between progress and communication overhead. Deadlock-freedom is not sufficient in many contexts. Allowing for more progress implies adding communication overhead. Thus, we need to define other meaningful criteria to decide when a synchronization should be added. In particular, this requires to have a better understanding of the impact of synchronizations on the number of messages exchanged during the coordinator process.

Another improvement on our work would be to combine it with abstraction techniques. Indeed, knowledge of a process, defined as the set of global reachable states consistent with its local state, is well-suited for being obtained based on an abstraction of the rest of the system.

Finally, it would be meaningful to integrate this approach into the distributed implementation of BIP [2] — standing for Behavior, Interaction and Priority —which is currently under development. So far, only systems without priorities have been implemented [4,5].

The question of how to implement BIP systems in a distributed setting remains a challenging task.

References

1. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority scheduling of distributed systems based on model checking. In: Proceedings of CAV'09, *LNCS*, vol. 5643, pp. 79–93. Springer (2009)
2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proceedings of SEFM'06, pp. 3–12. IEEE Computer Society (2006)
3. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge-based controlling of distributed systems. In: Proceedings of ATVA'10, *LNCS*, vol. 6252, pp. 52–66. Springer (2010)
4. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: Automated conflict-free distributed implementation of component-based models. In: Proceedings of SIES'10, pp. 108–117. IEEE Computer Society (2010)
5. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: Proceedings of EMSOFT'10, pp. 209–218. ACM (2010)
6. Fagin, R., Halpern, J.Y., Vardi, M.Y., Moses, Y.: Reasoning about knowledge. MIT Press, Cambridge, MA, USA (1995)
7. Genrich, H.J., Lautenbach, K.: System modelling with high-level petri nets. Theor. Comput. Sci. **13**, 109–136 (1981)
8. Graf, S., Peled, D., Quinton, S.: Achieving distributed control through model checking. In: Proceedings of CAV'10, *LNCS*, vol. 6174, pp. 396–409. Springer (2010)
9. Katz, G., Peled, D.: Code mutation in verification and automatic code correction. In: Proceedings of TACAS'10, *LNCS*, vol. 6015, pp. 435–450. Springer (2010)
10. van der Meyden, R.: Common knowledge and update in finite environment. Information and Computation **140**(2), 115–157 (1998)
11. Orlin, J.B.: Contentment in graph theory: covering graphs with cliques. Indagationes Mathematicae **80**(5), 406–424 (1977)
12. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience **16**(12), 1173–1206 (2004)
13. Rudie, K., Ricker, S.L.: Know means no: Incorporating knowledge into discrete-event control systems. IEEE Transactions on Automatic Control **45**(9), 1656–1668 (2000)
14. Rudie, K., Wonham, W.M.: Think globally, act locally: decentralized supervisory control. IEEE Transactions on Automatic Control **37**(11), 1692–1708 (1992)
15. Thistle, J.G.: Undecidability in decentralized supervision. System and Control Letters **54**, 503–509 (2005)
16. Thomas, W.: On the synthesis of strategies in infinite games. In: Proceedings of STACS'95, *LNCS*, vol. 900, pp. 1–13. Springer (1995)
17. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Inf. Process. Lett. **90**(1), 21–28 (2004)
18. Yoo, T.S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dynamic Systems **12**(3), 335–377 (2002)