# Synthesis of Distributed Control through Knowledge Accumulation*

Gal Katz[1], Doron Peled[1], and Sven Schewe[2]

[1] Department of Computer Science,
Bar Ilan University, Ramat Gan 52900, Israel
[2] Department of Computer Science,
University of Liverpool, Liverpool, UK

**Abstract.** In distributed systems, local controllers often need to impose global guarantees. A solution that will not impose additional synchronization may not be feasible due to the lack of ability of one process to know the current situation at another. On the other hand, a completely centralized solution will eliminate all concurrency. A good solution is usually a compromise between these extremes, where synchronization is allowed for in principle, but avoided whenever possible. In a quest for practicable solutions to the distributed control problem, one can constrain the executions of a system based on the pre-calculation of knowledge properties and allow for temporary interprocess synchronization in order to combine the knowledge needed to control the system. This type of control, however, may incur a heavy communication overhead. We introduce the use of simple *supervisor* processes that *accumulate* information about processes until *sufficient knowledge* is collected to allow for safe progression. We combine the knowledge approach with a game theoretic search that prevents progressing to states from which there is no way to guarantee the imposed constraints.

## 1 Introduction

Designing concurrent systems such that they satisfy a given specification is a highly difficult task that requires a lot of creativity. Automatic synthesis is very desirable, but unfortunately undecidable [16,10,11,5,20]. Separating the design of the system such that global constraints can be later imposed on a distributed system can be a very powerful tool. The hardness of imposing distributed control is derived from the fact that the processes can only rely on their individual observations of the system state. Such limited local information can be described as *knowledge* that processes have at each point of the execution [4,7].

In the classical approach to the synthesis of distributed control [24,18,19], memory is added to each process. This memory is updated based on the observation made by the respective process and, based on the state of this memory, the processes can support transitions or block them. Checking whether a distributed system can be controlled in

* The research of the 1st author was funded by Israeli Science Foundation (ISF) grant 1252/09. The research of the 2nd author was funded by Royal Society Grant TG100660 'Synthesising Permissive Monitors', The research of the 3rd author was funded by Engeering and Physical Science Research Council (EPSRC), grant EP/H046623/1.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 510–525, 2011.

this way to satisfy a given property is undecidable [22,23], even when the imposed property is an invariant on states and subsequent transitions [6].

In this paper, we study the problem of controlling distributed systems to satisfy an invariant property by adding supervisory processes that can communicate asynchronously with several processes, helping them to make a safe progress decision. We show that in this case controllability is decidable; it is EXPTIME complete, and PSPACE complete for systems with or without uncontrollable transitions, respectively.

In order to decide controllability and determine a control strategy, we first perform a global *strategy search* in order to avoid being trapped into a state where the invariant cannot be satisfied. Then, the obtained reduced state space is redistributed according to the original processes based on model checking the processes *knowledge* [13,1,2,6]. The information gathered during the model checking stage is used as a basis for a program transformation that controls the execution of the system by adding constraints on the enabledness of transitions. This does not produce new program executions or deadlocks, and consequently preserves all stuttering closed [14] linear temporal logic properties of the system [12].

When a process does not have enough local knowledge to progress safely, it can *hang* on a *supervisor* process. Supervisors accumulate information about multiple processes that are hung on them until sufficient knowledge is acquired. In a series of solutions, we show strategies to minimize the amount of overhead in terms of additional communication and synchronization, and of blocking of concurrent occurrence of transitions. There are several contributions in this paper:

- While most of the results on distributed synthesis are negative, we provide a constructive and efficient solution. We introduce supervisor processes that, while running asynchronously with the controlled system, accumulate the knowledge from several processes until a decision to safely execute a transition can be made.
- Knowledge has been shown to be a useful tool to impose distributed control [1,2,6,18]. Synchronization [6] or message passing [18] can be used to prevent blocking (deadlock) due to lack of knowledge. However, the controlled system may still block by reaching a state where all the possible enabled transitions would violate the imposed property. We solve this by calculating knowledge based on the result of a game theoretic strategy search.
- We introduce a detailed and realistic concurrency model that distinguishes the part of the global state that a process can observe from the smaller part it can control.

## 2 Preliminaries

The model used in this paper is Petri Nets. It was chosen due to its visual representation. The method and algorithms developed here can equally apply to other models, e.g., transition systems, communicating automata, etc.

**Definition 1.** *A* (1-safe) Petri Net $N$ *is a tuple* $(P,T,E,s_0)$ *where*

- $P$ *is a finite set of* places*,*
- *the* states *are defined as* $S = 2^P$ *where* $s_0 \in S$ *is the* initial state*,*

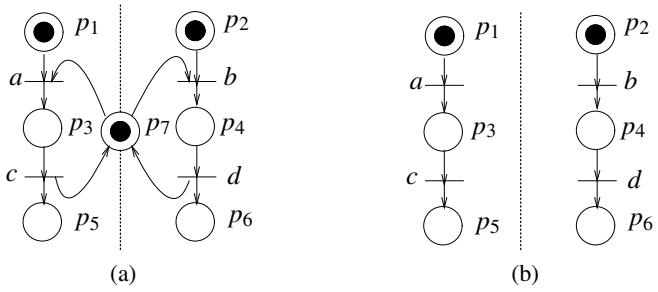


**Fig. 1.** Petri Nets: (a) without priorities and (b) with priorities $a \ll d$ and $b \ll c$

- *$T$ is a finite set of* transitions*, and*
- *$E \subseteq (P \times T) \cup (T \times P)$ is a bipartite relation between the places and the transitions.*

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

**Definition 2.** *A transition $t$ is* enabled *in a state $s$, denoted $s[t\rangle$, if $^\bullet t \subseteq s$ and $t^\bullet \cap s \subseteq {}^\bullet t$ (we allow self loops). A state $s$ is in* deadlock *if there is no enabled transition from it.*

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

**Definition 4.** *Two transitions $t_1$ and $t_2$ are* dependent *if $({}^\bullet t_1 \cup t_1{}^\bullet) \cap ({}^\bullet t_2 \cup t_2{}^\bullet) \neq \emptyset$. Let $D \subseteq T \times T$ be the* dependence *relation. Two transitions are* independent *if they are not dependent.*

Transitions are visualized as lines, places as circles, and the relation $E$ is represented using arrows. In Figure 1(a), there are places $p_1, p_2, \ldots, p_7$ and transitions $a, b, c, d$. We depict a state by putting full circles, called *tokens*, inside the places of that state. In the example in Figure 1(a), the initial state $s_0$ is $\{p_1, p_2, p_7\}$. The transitions that are enabled from the initial state are $a$ and $b$. If we fire transition $a$ from the initial state, the tokens from $p_1$ and $p_7$ will be removed, and a token will be placed in $p_3$. In this Petri Net, all transitions are dependent on each other, since they all involve the place $p_7$. Removing $p_7$, see Figure 1(b), makes both $a$ and $c$ become independent from both $b$ and $d$.

**Definition 5.** *An* execution *of a Petri Net $N$ is a maximal (i.e., it cannot be extended) alternating sequence of states $s_0 t_1 s_1 t_2 s_2 \ldots$, where $s_0$ is the initial state, such that, for each states $s_i$ in the sequence, $s_i[t_{i+1}\rangle s_{i+1}$. We denote these executions by $exec(N)$.*

For convenience, we sometimes use as executions just the sequence of states, or just the sequence of transitions, as will be clear from the context. A state is *reachable* in a Petri Net if it appears on at least one of its executions. We denote the reachable states of a Petri Net $N$ by $reach(N)$. In order to simplify the presentation and not distinguish between finite and infinite maximal executions, a special new element $\varepsilon$ is added to $T$; it

has no input and output places and can be fired exactly in states $s$ with deadlocks (where no other transition is enabled), thence $s[\varepsilon\rangle s$. Yet, these states are still distinguished as deadlocks.

We use places also as state predicates. As usual, we write $s \models p_i$ iff $p_i \in s$ and extend this in the standard way to Boolean combinations on state predicates. For a state $s$, we denote by $\varphi_s$ the formula that is a conjunction of the places in $s$ and the negated places not in $s$. Thus, $\varphi_s$ is satisfied exactly by the state $s$. For the Petri Net in Figure 1(a), the initial state $s_0$ satisfies $\varphi_{s_0} = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge p_7$. For a set of states $Q \subseteq S$, we call $\bigvee_{s \in Q} \varphi_s$, or any logically equivalent propositional formula $\varphi_Q$, a *characterizing formula* of $Q$. As usual in logic, when $\varphi_Q$ and $\varphi_{Q'}$ characterize sets of states $Q$ and $Q'$, respectively, then $Q \subseteq Q'$ exactly when $\varphi_Q \rightarrow \varphi_{Q'}$.

An invariant [3] of $N$ is a subset of the states $Q \subseteq 2^S$; a net $N$ satisfies the invariant $Q$ if $reach(N) \subseteq Q$. A *generalized invariant* of $N$ is a set of pairs $I \subseteq S \times T$; a net $N$ satisfies $I$ if, whenever $s[t\rangle$ for a reachable $s$, then $(s,t) \in I$. This covers the above simple case of an invariant when pairing up every state that appears in $Q$ with *all* transitions $T$.

**Definition 6.** *An execution of a Petri Net $N$* restricted *with respect to a set $I \subseteq S \times T$, denoted $exec_I(N)$, is the set of executions $s_0 t_1 s_1 t_2 s_2 \ldots \in exec(N)$ such that, for each pair $s_i t_{i+1}$ in the sequence, $(s_i, t_{i+1}) \in I$ holds. The set of states reachable in $exec_I(N)$ is denoted $reach_I(N)$.*

**Lemma 1.** *$reach_I(N) \subseteq reach(N)$ and $exec_I(N) \subseteq exec(N)$.*

**Definition 7.** *A* process $\pi$ *of a Petri Net $N$ is a subset of the transitions $T$ satisfying that, for each pair $t_1, t_2 \in \pi$ of independent (i.e., $(t_1, t_2) \notin D$) transitions in $\pi$, there is no reachable state $s$ in which both $t_1$ and $t_2$ are enabled.*

We will represent the separation of transitions of a Petri Net into processes using dotted lines. We assume a given set of processes $\mathcal{C}$ that *covers* all transitions (except $\varepsilon$) of the net, i.e., $\bigcup_{\pi \in \mathcal{C}} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. Let $proc(t) = \{\pi \mid t \in \pi\}$ be the set of processes to which $t$ belongs. For the Petri Net in Figure 1(a), there are two executions: *acbd* and *bdac*. There are two processes: the *left* process $\pi_l = \{a, c\}$ and the *right* process $\pi_r = \{b, d\}$. We use the same partitioning of transitions to processes in Figure 1(b).

The *neighborhood* of a set of processes $\Pi$ includes all places that are either inputs or outputs to transitions of $\Pi$.

**Definition 8.** *The* neighborhood *$ngb(\pi)$ of a process $\pi$ is the set of places $\bigcup_{t \in \pi} ({}^\bullet t \cup t^\bullet)$. For a set of processes $\Pi \subseteq \mathcal{C}$, $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$.*

A set of processes $\Pi$ *owns* the places in their neighborhood that cannot gain or loose a token by a transition that is not exclusively in $\Pi$.

**Definition 9.** *The places* owned *by a set of processes (including a singleton process) $\Pi$, denoted $own(\Pi)$, is $ngb(\Pi) \setminus ngb(\mathcal{C} \setminus \Pi)$.*

When a notation refers to a set of processes $\Pi$, we will often replace writing the singleton process set $\{\pi\}$ by writing $\pi$, e.g., we write $own(\pi)$. Note that $ngb(\Pi_1) \cup ngb(\Pi_2) = ngb(\Pi_1 \cup \Pi_2)$, while $own(\Pi_1) \cup own(\Pi_2) \subseteq own(\Pi_1 \cup \Pi_2)$. The neighborhood of

process $\pi_l$ is $\{p_1, p_3, p_5, p_7\}$. Place $p_7$ in Figure 1(a) is neither owned by $\pi_l$, nor by $\pi_r$, but it is owned by $\{\pi_l, \pi_r\}$. It belongs to the neighborhood of both processes and acts as a semaphore. It can be captured by the execution of $a$ or of $b$, guaranteeing that $\neg(p_3 \wedge p_4)$ is an invariant of the system.

Our goal is to control the system to satisfy a generalized invariant by restricting some of its transitions from some of the states. We may be allowed to control only part of the transitions $ct(T) \subseteq T$, called *controllable* transitions. The other transitions, $uc(T) = T \setminus ct(T)$, are *uncontrollable*. Note that we may be at some state where either some uncontrollable transitions, or all enabled transitions, violate the generalized invariant. Being in such states is therefore "too late"; part of the controlling task is to avoid reaching such states.

As a running example for a generalized invariant we use a property of *priorities*.

**Definition 10.** *A* Petri Net with priorities *is a pair $(N, \ll)$, where $N$ is a Petri Net and $\ll$ is a partial order relation among the transitions $T$ of $N$. For the set $I_\ll = \{(s,t) \mid t$ is maximal among the enabled transitions in $s$ w.r.t. $\ll\}$, the set of* prioritized executions *$exec_{I_\ll}(N)$ of $(N, \ll)$ is the set of executions restricted to $I_\ll$.*

The executions of the Petri Net $M$ in Figure 1(b) (when the priorities $a \ll d$ and $b \ll c$ are *not* taken into account) include $abcd, acbd, bacd, badc$, etc. However, the prioritized executions of $(M, \ll)$ are the same as the executions of the Net $N$ in Figure 1(a).

Enforcing prioritized executions in a completely distributed way may be impossible. In Figure 1(b), $a$ and $c$ belong to the left process $\pi_l$, and $b$ and $d$ belong to the right process $\pi_r$, with no interaction between the processes. The left process $\pi_l$, upon having a token in $p_1$, cannot locally decide whether to execute $a$; the priorities dictate that $a$ can be executed if $d$ is not enabled, since $a$ has a lower priority than $d$. But this information is not locally available to $\pi_l$, which cannot distinguish between the cases where $\pi_r$ has a token in $p_2$, $p_4$, or $p_6$.

The local information of $\Pi$ at a given state consists of the restriction of the state to the neighborhood of the transitions of $\Pi$.

**Definition 11.** *The* local information *of a set of processes $\Pi$ of a Petri Net $N$ in a state $s$ is $s\lceil_\Pi = s \cap nbg(\Pi)$.*

In the Petri Net in Figure 1(a), the local information of $\pi_l$ in any state $s$ consists of the restriction of $s$ to the places $\{p_1, p_3, p_5, p_7\}$. In the depicted initial state, the local information is $\{p_1, p_7\}$. The local state restricts the places to the ones changeable only by a process (or set of processes).

**Definition 12.** *The* local state *of a set of processes $\Pi$ of a Petri Net $N$ in a state $s$ is $s\lfloor_\Pi = s \cap own(\Pi)$.*

The local state of $\pi_l$ in Figure 1(a) is $\{p_1\}$. It is always the case that $s\lfloor_\Pi \subseteq s\lceil_\Pi$.

**Lemma 2.** *If $\pi \notin \Pi$ then $s\lfloor_{\Pi \cup \{\pi\}}$ is the (disjoint) union of $s\lfloor_\Pi$ and $s\lceil_\pi \cap own(\Pi \cup \{\pi\})$.*

## 3 A Globally Controlled System

Before providing a solution to the distributed control problem we need to provide a solution to a related global control problem. Some reachable states are not allowed

according to the generalized invariant. In order not to reach these states, we may need to avoid some transitions that lead to such states from previous states. This is done using game theoretical search.

The game is played between a *constructor*, who wants to preserve the generalized invariant $I$ indefinitely, and a *spoiler*, who has the opposite goal. The game is played on the states $S$ of a net, starting from the initial state $s_0$. In each round, the constructor player chooses a nonempty subset of enabled transitions that must include all enabled uncontrollable transitions. Subsequently, the spoiler chooses a transition from this set, which is then executed. The spoiler wins as soon as she can choose a transition that violates $I$, i.e., $(s,t_s) \notin I$, while the constructor wins if he can avoid this for ever.

Let $B$ (for "bad") be the reachable states that we do not want to reach, namely,

$$B = \{s \mid s \in \mathit{reach}(N) \wedge (\exists t \in \mathit{uc}(T)\,(s[t\rangle \wedge (s,t) \notin I) \vee \forall t \in T\, s[t\rangle \rightarrow (s,t) \notin I\})$$

We may be forced into $B$ in a single step from every state that has an uncontrollable transition into $B$ or for which all enabled transitions allowed according to $I$ lead to states in $B$.

Let $\mathit{attr}(A)$ be states $s$ that satisfy that one of the following conditions:

- $s \in A$, or
- there exists an uncontrollable transition $t$ enabled[1] in $s$ with $s[t\rangle s'$ and $s' \in A$, or
- for all transitions $t$ enabled in $s$, such that $s[t\rangle s'$ and $(s,t) \in I$, it holds that $s' \in A$.

As usual, we define $\mathit{attr}^n(A) = \mathit{attr}(\mathit{attr}^{n-1}(A))$, where $\mathit{attr}^1(A) = \mathit{attr}(A)$. Because of the monotonicity of the $\mathit{attr}(A)$ operator (with respect to set inclusion) and the finiteness of the state space, there is a least fixpoint $\mathit{attr}^*(A)$, which is $\mathit{attr}^n(A) = \mathit{attr}^{n+1}(A)$ for some (smallest) $n$.

Now, let $G = \mathit{reach}(N) \setminus \mathit{attr}^*(B)$. These are the "good" reachable states in the sense that they are allowed by $I$ *and* the system can be controlled to henceforth adhere to $I$.

**Definition 13.** *Let* $R = \{(s,t) \in I \mid \exists s'\, s[t\rangle s' \wedge s,s' \in G\}$ *be the* safe transition relation.

If the initial state is good (or, equivalently, $G$ is nonempty), then the constructor can win by playing according to $R$. If, on the other hand, $s_0$ is in the attractor $\mathit{attr}^*(B)$ of the bad states, then $s_0$ is in $\mathit{attr}^n(B)$ for some $n \leq |S|$. But by the definition of $\mathit{attr}^n(B)$, the spoiler can force the game to $\mathit{attr}^{n-1}(B)$ in the next step, then to $\mathit{attr}^{n-2}(B)$, and so forth, and thus make sure the bad states are reached within $n$ steps.

**Lemma 3.** *The constructor can force a win if, and only if,* $s_0 \in G$.

This game can obviously be evaluated quickly on the *explicit* game graph, and hence in time exponentially in the number of places. EXPTIME completeness can be demonstrated by a simple reduction from the PEEK-$G_5$ [21] game. An instance of PEEK-$G_5$ consists of two disjoint sets of Boolean variables, $X$ (owned by a safety player) and $Y$ (owned by a reachability player), a subset $Z \subseteq X \cup Y$ of them that are initially *true*, and a Boolean formula $\varphi$ over $X \cup Y$ that the reachability player wants to become *true* eventually. The game is played in turns between the safety and the reachability player

[1] If $s[t\rangle$ and $s \notin A$ then $(s,t) \in I$.

(say, with reachability player moving first), who in their turn can change the truth value of up to one of their variables.

To reduce the PEEK-$G_5$ game to the question if the initial state $s_0$ of a Petri Net is good, we introduce two places for each variable in $X \cup Y$, where a token is always in exactly one of them, one for true and one for false. Initially, the tokens are placed in accordance with $Z$. We also have introduce two control places, indicating that it is the reachability and safety player's move, respectively. (We again always have a token, called the control token, in exactly one of these places.) The invariant $I$ represents the formula $\neg\varphi$.

For each variable $v \in X$ (resp. $v \in Y$), we have a transition that takes the control token from the safety (resp. reachability) place and the token from the *true/false* place of $v$, and puts the control token to the reachability (resp. safety) place and toggles the value of $v$ by putting a token into the *false/true* place of $v$. We also transitions that only take the control token from the safety (resp. reachability) place and put it into the reachability (resp. safety) place. (In total, we have $2|X \cup Y| + 2$ transitions.) Exactly the $2|Y| + 1$ transitions that move the control token from the reachability to the safety place are uncontrollable. The initial state is the state encoding $Z$ with the control token in the reachability place.

The constructor can play the role of the safety player in the PEEK-$G_5$ game by choosing a singleton set of transitions when the control pebble is in the safety place, while the spoiler can play the role of the reachability player by choosing among the enabled (uncontrollable) transitions when the control pebble is in the reachability place.

**Lemma 4.** *Deciding if the constructor can force a win is EXPTIME complete.*

In case all transitions are controllable, checking whether or not $s \in G$ (or $(s,t) \in R$) can be done in PSPACE using a binary search for an ultimately periodic cycle in $reach_I(N)$ that includes the state $s$ (followed immediately by the transition $t$, respectively). For hardness, it is easy to reduce the halting problem of a deterministic space-bounded Turing machine to this decision problem by representing the tape explicitly.

**Lemma 5.** *Deciding if the constructor can force a win is PSPACE complete for Petri Nets with only controllable transitions.*

## 4 Knowledge Based Distributed Control

In control theory, the transformation that takes a system and allows blocking some transitions adds a supervisor process [17,24,19], which is usually an automaton that runs synchronously with the controlled system. This (finite state) automaton observes the controlled system, progresses according to the transitions it observes, and blocks some of the enabled transitions, depending on its current state [24]. This is often insufficient for obtaining distributed control [18]. We propose a control mechanism with supervisors that run asynchronously with the controlled processes.

In the following definitions, we can often use either the local information or the local state. When this is the case, we will use $s|_\Pi$ instead of either $s\lceil_\Pi$ or $s\lfloor_\Pi$.

**Definition 14.** *Let $\Pi \subseteq \mathcal{C}$ be a set of processes. Define an equivalence relation $\equiv_\Pi \subseteq reach(N) \times reach(N)$ such that $s \equiv_\Pi s'$ when $s|_\Pi = s'|_\Pi$.*

As $s|_\Pi$ can stand for either $s\lceil_\Pi$ or $s\lfloor_\Pi$, this gives two different equivalence relations. When it is important to distinguish between them, we denote the one based on "$\lceil$" as $\equiv^w_\Pi$ (weak equivalence) and the one based on "$\lfloor$" as $\equiv^s_\Pi$ (strong equivalence). It is easy to see that the enabledness of a transition depends only on the local information of a process that contains it, as stated in the following lemma.

**Lemma 6.** *If $t \in \pi$ and $s \equiv^w_\pi s'$ then $s[t\rangle$ if, and only if, $s'[t\rangle$.*

This lemma does not hold when we replace $\equiv^w_\pi$ by $\equiv^s_\pi$. In the Prioritized Petri Net in Figure 1(b), e.g., we have that $\{p_1, p_2\} \equiv^w_{\pi_l} \{p_1, p_4\}$, since in both states $\pi_l$ has the same local information $\{p_1\}$. In the state $\{p_1, p_2\}$, $a$ is a maximal priority enabled transition (incomparable with $b$), while in $\{p_1, p_4\}$, $a$ is not maximal anymore, as we have that $a \ll d$, and both $a$ and $d$ are now enabled.

**Definition 15.** *The processes $\Pi$ (jointly)* know *a property $\psi$ in a state $s$, denoted $s \models K_\Pi \psi$, if, for each $s'$ such that $s \equiv_\Pi s'$, we have that $s' \models \psi$.*

We distinguish knowledge based on strong equivalence $\equiv^s_\Pi$ (and hence on local states), denoted $K^s_\Pi \varphi$, from knowledge based on weak equivalence $\equiv^w_\Pi$ (and hence local information), denoted $K^w_\Pi \varphi$. Since there can be multiple local informations for a single local state, we have $K^s_\Pi \varphi \rightarrow K^w_\Pi \varphi$. Therefore the knowledge based on the local state (resp. local information) is called *s*trong (resp. *w*eak) knowledge.

In order to make choices (to execute a transition) that take into account knowledge based on local information, a process, or a set of processes, needs to have some guarantee that the local information will not be changed by other processes while it is making the decision. For a single process, this may be achieved by the underlying hardware. On the other hand, it is unreasonable to require such a guarantee for a set of processes. Thus, for decisions involving a set of processes, knowledge based on the joint local state is used instead.

The classical definition of knowledge is based on the relations $\equiv_\Pi$ over the reachable states. However, when using knowledge to control a system to satisfy a generalized invariant, one may calculate the equivalences and the knowledge based on the states that appear in the executions of the original system that satisfy this general invariant [2]. This (cyclic looking) claim is proved [2] by induction on the progress of the execution in the controlled system: the controlled system allows at any state to execute only transitions that preserve the generalized invariant. For the same argument, when our allowed transitions will be restricted according to the safe transition relation $R$, we can further restrict the states space used to calculate the knowledge to $G \subseteq reach(N)$.

The definition of knowledge that we use assumes that the processes do not maintain a log with their history. The use of knowledge with such a log, called *knowledge with perfect recall* [13], is discussed in [1]. It is shown there that updating such a log can require enormous complexity for each process. We henceforth use knowledge formulas combined with Boolean operators and propositions. For a detailed syntactic and semantic description of logics with knowledge one can refer, e.g., to [4]. Once $s \models K_\Pi \psi$ is defined, $\psi$ can also be a knowledge property, hence $s \models K_{\Pi'} K_\Pi \psi$ (knowledge about knowledge) is also defined.

**Lemma 7.** *If* $s \models K_\Pi \varphi$ *and* $s \equiv_\Pi s'$, *then* $s' \models K_\Pi \varphi$.

**Lemma 8.** *Knowledge is monotonic: if* $\Pi' \subseteq \Pi$ *then* $K_{\Pi'}\varphi \rightarrow K_\Pi \varphi$.

We will use the following propositional formulas, with propositions that are the places of the Petri Net, to explain the approach and the implementation:

- The good states $G$: $\varphi_G$.
- The states where a transition $t$ is enabled: $\varphi_{en(t)}$.
- At least one transition is enabled, i.e., there is no deadlock: $\varphi_{df} = \bigvee_{t \in T} \varphi_{en(t)}$.
- Transition $t$ is allowed from the current state by the safe transition relation $R$: $\varphi_{good(t)}$
- The local information (resp. local state) of processes $\Pi$ at state $s$: $\varphi_{s\lceil_\Pi}$ (resp. $\varphi_{s\lfloor_\Pi}$).

The corresponding sets of states can easily be computed by model checking and stored in a compact way, e.g., using BDDs.

Now, given a Petri Net, one can perform model checking in order to calculate whether $s \models K_\pi \psi$. The processes $\Pi$ know $\psi$ at state $s$ exactly when $(\varphi_G \wedge \varphi_{s\lfloor_\Pi}) \rightarrow \psi$ is a propositional tautology. We can also check properties that include nested knowledge by simply checking first the innermost knowledge properties and marking the states with additional propositions for these innermost properties.

Model checking knowledge using BDDs, is *not* the most space efficient way of checking knowledge properties, since $\varphi_G$ can be exponentially big in the size of the Petri Net. In a (polynomial) space efficient check (which has a higher *time* complexity), we enumerate all states $s'$ such that $s \equiv_\pi s'$, check reachability of $s'$ using binary search, and, if reachable, check whether $s' \models \psi$. This can apply also to nested knowledge formulas, where inner knowledge properties are recursively reevaluated each time they are needed. The PSPACE complexity is subsumed by the EXPTIME complexity in the general case algorithm for the safe transition relation $R$.

**Definition 16.** *An* extended Petri Net *[9] is a Petri Net with a finite set of variables* $V_\pi$ *over a finite domain per each process* $\pi \in \Pi$. *In addition, a transition* $t$ *can be augmented with a predicate* $en_t$ *on the variables* $V_t = \cup_{\pi \in proc(t)} V_\pi$ *and a transformation function* $f_t(V_t)$. *In order for* $t$ *to fire,* $en_t$ *must hold in addition to the basic Petri Net enabling condition on the input and output places of* $t$. *When* $t$ *fires, in addition to the usual changes to the tokens, the variables* $V_t$ *are updated according to the transformation* $f_t$.

We transform a Petri Net $N$ and a generalized invariant $I$ into an extended Petri Net $N'$ that allows only the executions of $N$ controlled according to $I$.

**Definition 17.** *A* controlling *transformation obeys the following conditions:*

- *New transitions and places can be added.*
- *The input and output places of the new transitions are disjoint from the existing places.*
- *Variables, conditions and transformations, as in Definition 16 can be added to existing transitions.*
- *Existing transitions will remain with the same input and output places.*
- *It is not possible to fire from some point an infinite sequence of added transitions.*

The added transitions are grouped into new (supervisory) processes. The added variables will represent some knowledge-dependent finite memory for controlling the system, and some interprocess communication between the original processes and the added ones. Processes from the original net will have disjoint sets of variables from one another; intuitively, the independence between the original transitions is preserved by the transformation, although some coordination may be enforced indirectly through the interaction with the new supervisory processes.

As a natural extension of Definition 11, $s\lceil_C$ maps a state $s$ of the transformed version $N'$ into the places of the original version $N$ by projecting out additional variables and places that $N'$ may have on top of the places of $N$. In this way, we will be able to relate the sets of states of the original and transformed version. Firing of a transitions added by the controlling transformation does not change $s\lceil_C$ and is not considered to violate $I$ (the requirement that $(s_i, t_{i+1})$ in Definition 6 is imposed only when $t_{i+1}$ is from the original net $N$). Note that our restrictions on the transformation implies that the sets $ngb(\Pi)$ and $own(\Pi)$ for $\Pi \subseteq \mathcal{C}$ are not affected by the transformation.

**Definition 18.** *Two sequences of states $\sigma$ and $\sigma'$ are equivalent up to stuttering [14] when, by replacing any finite adjacent repetition of the same state by a single occurrence in either $\sigma$ or $\sigma'$, we obtain the same sequence. Let $stutcl(\Gamma)$ be the stuttering closure of a set $\Gamma$ of sequences, i.e., all sequences that are stuttering equivalent to some sequences in $\Gamma$.*

**Lemma 9.** *A controlling transformation produces an extended Petri Net $N'$ from $N$ such that $exec(N')\lceil_C \subseteq stutcl(exec(N))$.*

**Lemma 10.** *Given that $s \models K_\Pi \varphi$ in some basic Petri Net $N$, then $s \models K_\Pi \varphi$ also in a transformed version $N'$.*

## 5 Control Using Knowledge Accumulating Supervisors

According to the knowledge based approach to distributed control [1,6,2,18], model checking of knowledge properties is used at a preliminary stage to find when, depending on the local information, an enabled transition can be safely fired. In our case, this means checking $s \models K^w_\pi \varphi_{good(t)}$ (by Lemma 7, the satisfaction only depends on $s\lceil_\pi$). At runtime, a process *supports* a transition in every local information where this holds. The following *support policy* uses this information at runtime:

> A transition $t$ can be fired (is enabled) in a state when, in addition to its original enabledness condition, at least one of the processes in $proc(t)$ supports it.

**Lemma 11.** *If $t \in \pi \cap uc(T)$ and $(s,t) \in R$, then $s \models K^w_\pi \varphi_{good(t)}$.*

This follows from the observation that the safe transition relation does not restrict the uncontrolled transition. This means that an uncontrolled transition can always be supported by any processes that contains it.

It is possible that, in some (non deadlock) states of $G$, no process has enough local knowledge to support an enabled transition and, furthermore, no uncontrollable transitions are enabled. We may count such global states as additional "bad states", and repeat the strategy calculation to refine the state space further, as done in Section 3.

Still, there are cases where the local knowledge will not be sufficient, as in the priority example in Figure 1(b). Then, we may need to synchronize several processes to collect more knowledge. Then, a process can decide, based on its current knowledge, whether it needs to hang on a supervisor and send it its local state; the supervisor can make a decision, based on accumulated joined knowledge of several hung processes, that one of them can support an enabled transition. For example, in Figure 1(b), in the initial state the local information (and also the local state) of $\pi_l$ is $\{p_1\}$. Thus, $\pi_l$ does not have enough knowledge to support any transition. Similarly, the local information of $\pi_r$ is $\{p_2\}$, which also is not sufficient to support any transition. After they both hang on a supervisor, it has enough information to support $a$ or $b$.

To simplify the presentation, we will describe the solution in several steps. Each solution is an improvement over the previous solutions.

**Solution 1. [Completely centralized; Waiting for all processes]**

First, we assume that there is a single supervisor process $\mathcal{T}$, which is responsible for all processes. At this point, assume that no uncontrollable transitions are allowed. Dealing with them is deferred until Solution 3. A process hangs on a supervisor, when the following property *does not* hold:

$$\kappa_1^\pi = \bigvee_{t \in \pi} K_\pi^w \varphi_{good(t)}$$

When $\mathcal{T}$ sees that *all* processes are hanging on it, then it can acquire the complete global state from the individual processes. Since all processes are hung, none of them can progress. Each process then reports its local information to the supervisor $\mathcal{T}$. Now, $\mathcal{T}$ has enough information to instruct some process to support an enabled transition, according to the safe transition relation $R$. When a transition is supported and fired, all processes are freed from being hung.

This solution prevents new deadlocks in the system that did not occur under the original Petri Net: if no process has the local knowledge that is required to progress, no process $\pi$ has the knowledge according to $\kappa_1^\pi$; eventually all processes will hang on $\mathcal{T}$, and $\mathcal{T}$ has the capability to break the deadlock.

**Solution 2. [Supervisor can decide based on a subset of processes]**

The supervisor $\mathcal{T}$ keeps the updated joint local state of the hung processes $\Pi$. When a process $\pi$ hangs, it updates this view by transmitting to $\mathcal{T}$ its local information $s\lceil_\pi$, from which $\mathcal{T}$ keeps (according to Lemma 2) $s\lceil_\pi \cap own(\Pi \cup \{\pi\})$. Since all processes in $\Pi' = \Pi \cup \{\pi\}$ are now hung, no other process can change these places. Then the knowledge $K^s_{\Pi'} \varphi_{good(t)}$ can be used to support a transition $t$. Recall that knowledge based decisions of a single process use weak knowledge (based on the local information), while multiple processes use strong knowledge (i.e., based on the joint local state).

The supervisor process $\mathcal{T}$ can identify (precalculated) joint local states where enough knowledge is available to decide that a transition $t$ is allowed by the safe transition relation $R$ and make one of the processes in $proc(t)$ support it. The supervisor $\mathcal{T}$ does not need to wait for all processes to hang on it to make such a decision. Once $t$ is fired, $\mathcal{T}$ frees all hung processes.

**Solution 3. [Hanging processes may still progress]**

Consider the following cases:

1. After the decision of a process $\pi$ to hang on $\mathcal{T}$, other processes make changes to $\pi$'s local information that allow it to support some transition $t$.
2. A transition $t$ with $\{\pi, \pi'\} \subseteq proc(t)$ is supported by $\pi'$ while $\pi$ is hung.
3. An uncontrollable transition can always be executed, even if it belongs to a hung process.

In all of these cases, we allow $\pi$ to notify $\mathcal{T}$ that it has decided not to hang on it anymore. Moreover, $\mathcal{T}$, which acquired information about the hung processes $\Pi$, will have to forget the information about the places $own(\Pi) \setminus own(\Pi \setminus \{\pi\})$.
We can now weaken the condition $\kappa_1^\pi$ for a process not to hang into:

$$\kappa_2^\pi = \bigvee_{t \in \pi} K_\pi^w \varphi_{good(t)} \vee K_\pi^w \bigvee_{\pi' \neq \pi} \bigvee_{t \in \pi'} K_{\pi'}^w \varphi_{good(t)}$$

That is, a process does neither hang on the supervisor when it has enough knowledge to support a transition, nor if it knows that some other process has such knowledge. In the latter case, it does not actually need to distinguish which process has that knowledge.

The ability of processes to hang on a supervisor but also to progress independently before the supervisor has made any supporting choice requires some protocol between the processes and the supervisor. The $\alpha$-core protocol [15,8] can be adopted here.

**Solution 4. [Multiple supervisors]**

Instead of having a single supervisor $\mathcal{T}$, we use several supervisors $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$. Each supervisor $\mathcal{T}_i$ takes care of a set of processes $proc(\mathcal{T}_i)$. These sets are pairwise disjoint and do not necessarily cover all the processes.

An effectively checkable criterion to determinte if at least one process or supervisor will be able to provide a progress from any nondeadlock state in $G$ is as follows:

$$(\varphi_G \wedge \varphi_{df}) \rightarrow \Big( \bigvee_{t \in \pi \in \mathcal{C}} K_\pi^w \varphi_{good(t)} \vee \bigvee_{i \in 1 \ldots k} \bigvee_{t \in \pi \in proc(\mathcal{T}_i)} K_{proc(\mathcal{T}_i)}^s \varphi_{good(t)} \Big)$$

It is interesting to compare Solution 3 with Solution 4. The former is more flexible in the sense that knowledge accumulated is not limited to a fixed partitioning of the processes. On the other hand, the latter solution allows true concurrency between transitions supported with the help of supervisors, whereas the former one interleaves the firing of such transitions. A solution that allows supervisors for a nondisjoint set of processes, which achieves the flexibility of Solution 3 *and* allows concurrent (independent) support of transitions by different supervisors as in Solution 4, is also possible. At first glance, this looks like gaining from both worlds; in practice, however, allowing processes to hang on *multiple* supervisors requires a large amount of overhead.

**Lemma 12.** *Under our transformations from a Petri Net N to an extended Petri Net N', $exec(N')\lceil_C \subseteq stutcl(exec_I(N))$ holds.*

This is proved by induction on prefixes of the execution and using Lemma 9.

**Lemma 13.** *$N'$ satisfies all stuttering invariant temporal properties of N.*

This follows from the fact that our transformation does not introduce new deadlocks.

## 6 Implementing Supervisors

Processes hang on a supervisor in some arbitrary order. The supervisor needs to decide, based on the part of the global state that it sees, whether there is enough information to support some transition. We consider first the case of a single supervisor.

**Definition 19.** *Let* $L = \{s\lfloor_\Pi \times \Pi \mid s \in G, \Pi \subseteq \mathcal{C}\}$ *denote the set of joint local states, each paired up with the set of relevant processes (then* $G \times \mathcal{C} \subseteq L$*). We define* $\sqsubseteq \subseteq L \times L$ *(and, symmetrically,* $\sqsupseteq$*) as follows:* $q \sqsubseteq q'$ *if* $q = (s\lfloor_{\Pi_1}, \Pi_1), q' = (s\lfloor_{\Pi_2}, \Pi_2)$ *(i.e., both are part of the same global state s) and* $\Pi_1 \subseteq \Pi_2$*. We say that* $q'$ subsumes $q$*.*

**Definition 20.** *The support function* $supp : L \to 2^T$ *returns, for each* $q \in L$*, the transitions that are allowed by R from all states that subsume q. Formally,* $supp(q) = \cap_{(s,C) \sqsupseteq q}\{t \mid t \in T, (s,t) \in R\}$.

That is, for $q = (s\lfloor_\Pi, \Pi)$, $t \in supp(q)$ iff $s \models K^s_\Pi \varphi_{good(t)}$. If $t \in supp(q) \cap ct(T)$, then the supervisor can select a process in $proc(t)$ to support $t$. Obviously, when $q \sqsubseteq q'$, $supp(q) \subseteq supp(q')$. There is no need for a supervisor to store in the domain of *supp* elements $q = (s\lfloor_\Pi, \Pi)$ where $|\Pi| < 2$; in this case, when $supp(q) \neq \emptyset$, the process with that local state can locally support a transition without the help of a supervisor.

**Definition 21.** *Let* $\rightsquigarrow \subseteq L \times L$ *be such that* $q \rightsquigarrow q'$ *if* $q = (s\lfloor_\Pi, \Pi)$ *and* $q' = (s\lfloor_{\Pi \cup \{\pi\}}, \Pi \cup \{\pi\})$*, where* $\pi \notin \Pi$ *(i.e.,* $q'$ *extends q according to exactly one process).*

In the example from Figure 1(b), we have that $(\{p_1\}, \{\pi_l\}) \rightsquigarrow (\{p_1, p_2\}, \{\pi_l, \pi_r\})$ and $(\{p_2\}, \{\pi_r\}) \rightsquigarrow (\{p_1, p_2\}, \{\pi_l, \pi_r\})$. The supervisor updates its view about the joint local state of the processes according to the relation $\rightsquigarrow$: when moving from $q$ to $q'$ by acquiring the relevant information about a new processor $\pi$; consequently, its knowledge grows and it can decide to support one of the transitions in $supp(q')$.

**Definition 22.** *A joint local state q is* minimal supporting *if* $supp(q) \neq \emptyset$ *and, for each* $q'$ *such that* $q' \rightsquigarrow q$*,* $supp(q') = \emptyset$*.*

**Definition 23.** *The* upward closure $\uparrow U$ *of a subset of the joint local states* $U \subseteq L$ *is* $\{q \in L \mid \exists q' \in U\ q' \sqsubseteq q\}$.

Because processes hang on a supervisor one by one, there is no need to calculate and store *all* the cases of the function *supp*.

**Lemma 14.** *A sufficient condition for restricting the domain* $U \subseteq L$ *of supp for a supervisor, without introducing new deadlocks, is that* $G \times \{\mathcal{C}\} \subseteq \uparrow U$*.*

This suggests the following algorithm for calculating the representation table for *supp*: perform DFS such that if $q \rightsquigarrow q'$, then $q$ is searched before $q'$; backtrack when visiting $q$ again, or when $supp(q) \neq \emptyset$. This algorithm can be used also for multiple supervisors, when restricting the search to the joint local states of $\Pi \subseteq proc(\mathcal{T}_i)$ for each $\mathcal{T}_i$.

In order to reduce the set of local states that a supervisor needs to keep in the support table, one may decide that a supervisor will not always support transitions as soon as the joint local state of the hung processes allows that. This introduces further delays

**Table 1.** Tests and Results

| Processes | States | Global table size | Avg. num. of hung procs. | Duration (sec) |
|---|---|---|---|---|
| 5 | 164 | 147 | 3.24 | 0.0002 |
| 8 | 3344 | 3296 | 4.06 | 0.2 |
| 9 | 9136 | 8449 | 4.37 | 1.3 |
| 10 | 24960 | 21371 | 4.65 | 6.8 |

in decisions, where the supervisor waits for more processes to hang even when it can already support some transitions. On the other hand, in this case the set of supported transitions may be larger, allowing more nondeterminism.

The size of the global state space of a Petri Net is $O(2^{|P|})$. Since we need to keep also the joint local states, the size of the support table that we store in a supervisor, is $O(2^{|P|+|C|})$ (which is the size of $L$). However, by Lemma 14, the representation may be much more succinct. In theory, when there are no uncontrollable transitions, a (particularly slow) supervisor can avoid storing the support table, and perform the PSPACE binary search each time it needs to make a decision on a joint local state.

## 7 Experimental Results

We implemented a prototype tool that allows to define Petri Nets with priorities and then simulate their executions according to Solution 3 described above. The tool uses model checking for building the local support tables for each process, as well as a central support table allowing the supervisor to base its support on accumulated knowledge. The tests were performed on a standard PC with Intel Core™2 Duo Processor and 2GB RAM.

In order to measure the efficiency of the suggested solution, the tests were repeated for several Petri Nets with various numbers of processes, places, and transitions. In each test, multiple random execution sequences of $10,000$ steps were generated, allowing to gather reliable performance statistics. Table 1 summarizes some tests and their results. The table presents, for each test, the number of global states, the size of the table used by the supervisor, the average number of hanged processes needed until the supervisor could support a transition, and the model checking duration.

## 8 Conclusions

We have developed a simple and effective algorithm for synthesizing distributed control. The resulting control strategy uses communication and knowledge collection without blocking the processes unnecessarily. One strength of our approach is that it is complete in the sense that, provided a centralized solution exists, it finds a solution. However, this does not come at the cost of centralizing the control completely. To the contrary, the system can progress without the support of a global or regional supervisor as soon as the local information suffices to do so.

Our solution for the distributed control of systems uses knowledge to construct a distributed controller for a global constraint. In [1,2], it is demonstrated that the local knowledge may be insufficient to construct a controller. Knowledge of perfect recall [13], which depends not only on the local state (information), but on the gathered visible history, can alleviate some, but not all, of these situations. The use of interprocess communication to obtain joint knowledge is suggested in [18]; however, no systematic algorithm for collecting such knowledge, or for evaluating when enough knowledge has been collected, was provided there. In [6], joint knowledge is calculated through temporary multiprocess synchronization. However, such synchronization is expensive, and multiple interactions (including different interactions of the same set of processes) may require a separate synchronizing process. We proposed here a practical solution for distribute control where a small number of (or even a single) supervisor(s) run(s) concurrently with the controlled system.

The algorithms in [1,2,6] guarantee a solution only under the assumption that the required constraint on a distributed system may not cause a deadlock (i.e., is nonblocking); accordingly, at the limit, one may synchronize all the processes and then make a decision how to progress given the global state. These solutions work, e.g., in the case of imposing priorities without uncontrollable transitions. However, when imposing the priorities when uncontrollable transitions are allowed, such control synthesis may fail: a state where the uncontrollable transitions have minimal priority among the enabled ones would be too late, and needs to be eliminated in advance. Our solution solves this problem by calculating the knowledge based on a reduced state space that avoids being blocked in such situations.

An interesting research direction is to deal with more general temporal properties. The difficulty lies with the need to use global memory for controlling the system and then distributing it according to the original architecture.

## References

1. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems Based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
2. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge based controlling of distributed systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 52–66. Springer, Heidelberg (2010)
3. Clarke, E.M.: Synthesis of Resource Invariants for Concurrent Programs. ACM Transactions on Programming Languages and Systems 2(3), 338–358 (1980)
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
5. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: LICS 2005, Chicago, IL, pp. 321–330 (2005)
6. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control through Model Checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)
7. Halpern, J.Y., Zuck, L.: A little knowledge goes a long way: knowledge based derivation and correctness proof for a family of protocols. Journal of the ACM 39(3), 449–478 (1992)

8. Katz, G., Peled, D.: Code Mutation in Verification and Automatic Code Correction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 435–450. Springer, Heidelberg (2010)
9. Keller, R.M.: Formal Verification of Parallel Programs. Communications of the ACM 19, 371–384 (1976)
10. Kupferman, O., Vardi, M.Y.: Synthesizing Distributed Systems. In: LICS 2001, Boston, MA (2001)
11. Madhusudan, P., Thiagarajan, P.S.: Distributed Controller Synthesis for Local Specifications. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 396–407. Springer, Heidelberg (2001)
12. Manna, Z., Pnueli, A.: How to Cook a Temporal Proof System for Your Pet Language. In: POPL 1983, Austin, TX, pp. 141–154 (1983)
13. van der Meyden, R.: Common Knowledge and Update in Finite Environment. Information and Computation 140, 115–157 (1980)
14. Peled, D., Wilke, T.: Stutter-Invariant Temporal Properties are Expressible without the Text Time Operator. Information Processing Letters 63, 243–246 (1997)
15. Pérez, J.A., Corchuelo, R., Toro, M.: An Order-based Algorithm for Implementing Multi-party Synchronization. Concurrency - Practice and Experience 16(12), 1173–1206 (2004)
16. Pnueli, A., Rosner, R.: Distributed Reactive Systems are Hard to Synthesize. In: FOCS 1990, St. Louis, Missouri, pp. 746–757 (1990)
17. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM Journal on Control and Optimization 25(1), 206–230 (1987)
18. 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)
19. Rudie, K., Wonham, W.M.: Think globally, act locally: descentralized supervisory control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
20. Schewe, S., Finkbeiner, B.: Synthesis of Asynchronous Systems. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 127–142. Springer, Heidelberg (2007)
21. Stockmeyer, L.J., Chandra, A.K.: Provably Difficult Combinatorial Games. SIAM Journal of Computing 8, 151–174 (1979)
22. Thistle, J.G.: Undecidability in Descentralized Supervision. Systems and Control Letters 54, 503–509 (2005)
23. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Information Processing Letters 90(1), 21–28 (2004)
24. Yoo, T.S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dynamic Systems, Theory & Applications 12(3), 335–377 (2002)