Methods for Knowledge Based Controlling of Distributed Systems

Saddek Bensalem, Marius Bozga, Susanne Graf, Doron Peled, Sophie Quinton

Methods for Knowledge based Controlling of Distributed Systems

Saddek Bensalem[1], Marius Bozga[1], Susanne Graf[1],
Doron Peled[2], and Sophie Quinton[1]

[1]Centre Equation - VERIMAG, 2 Avenue de Vignate, Gieres, France
[2]Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Controlling concurrent systems to impose some global invariant, is an undecidable problem. One can gain decidability at the expense of reducing concurrency. Even under this flexible design assumption, the synthesis problem remains highly intractable. One practical method for designing controllers is based on checking knowledge properties upon which the processes can make their decisions whether to allow or block transitions. A major deficiency of this synthesis method is in calculating the knowledge based on the given system that we want to control, and not on the resulted system. The original system has less knowledge, and as a result, we may introduce far more synchronization than needed. In this paper we show techniques to reduce this overhead.

1 Introduction

Model checking has provided algorithms for the automatic analysis of systems. Techniques for automating the process of system design, in order to obtain correct-by-construction systems, have been recently studied as well. The synthesis problem from LTL specification was shown by Pnueli and Rosner [12] to be 2EXPTIME hard for sequential reactive systems and undecidable for concurrent systems. A related problem is to control an already given system in order to force it to satisfy some additional property [14]. For distributed systems, this has also been shown to be undecidable [18, 17]. Under the assumption that a system is flexible to the addition of further synchronization, the control problem becomes decidable. A solution based on model checking of knowledge properties was suggested [1, 7].

In this paper, we look at the problem of reducing the need for additional synchronization in order to control distributed systems. We identify the main problem of the knowledge approach in using the controlled (source) system to calculate the knowledge. In fact, this is merely an approximation, as the actual knowledge needs to be satisfied by the (target) system after it is being controlled. After the control is applied, there are fewer executions, and fewer reachable states, hence the knowledge cannot decrease.

Our first observation is somewhat surprising: we prove that it is safe to calculate the knowledge based on the source system when considering for the analysis

only the executions of the source system that satisfy the desired constraint. This provides a smaller set of executions and reachable states, hence, also potentially more knowledge.

A second observation is that once we control a system according to its knowledge properties, we obtain again a system with fewer executions and reachable states: even if in the original system there are states where the system lacks the knowledge to continue, these states may, in fact, already be unreachable. Thus, one needs to make another round of checks on the obtained controlled system; this version may, in fact, have enough knowledge to implement the desired constraints.

These two observations can be used in conjunction with other methods for constructing distributed controllers based on knowledge:

- Using knowledge of perfect recall (proposed in [1]).
- Adding coordinations to combine knowledge (proposed in [7]).

We show here that all these techniques are independent of each other, hence can be combined.

2 Preliminaries

The model used in this paper is Petri Nets. The method and algorithms developed here can equally apply to other models, e.g., transition systems and communicating automata.

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 *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* $t^\bullet \cap s = \emptyset$. *We denote the fact that* t *is enabled from* s *by* $s[t\rangle$.

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 at* s. *Then,* $s' = (s \setminus {}^\bullet t) \cup t^\bullet$. *We extend this notation to* $s[t_1 t_2 \dots t_k\rangle s'$, *when there is a sequence* $s[t_1\rangle s_1[t_2\rangle s_2 \dots s_{k-1}[t_k\rangle s'$, *i.e., the system moves from* s *to* s' *by firing the sequence of transitions* $t_1 t_2 \dots t_k$.

The prefixes on the executions in a set X are denoted by $pref(X)$.

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

As usual, transitions are represented as lines, places as circles, and the relation E is represented by arrows from transitions to places and from places to transitions. We will use Petri Net N of Figure 1 as a running example. In N, there are places p_1, p_2, ..., p_5 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 of Figure 1, the depicted initial state s_0 is $\{p_1, p_4\}$. If we fire transition a from this initial state, the tokens from p_1 will be removed, and a token will be placed in p_2. The transitions enabled in s_0 are a and c. In this example, a and b are independent of c and d.

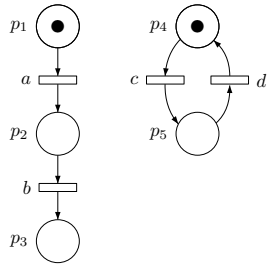


Fig. 1. A Petri Net N with priorities $a \ll \{c, d\} \ll b$

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

We denote the executions of a Petri Net N by $exec(N)$. 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 $reach(N)$. The reachable states of our running example N are $\{p_1, p_4\}$, $\{p_1, p_5\}$, $\{p_2, p_4\}$, $\{p_2, p_5\}$, $\{p_3, p_4\}$ and $\{p_3, p_5\}$.

We use places also as state predicates and denote $s \models p_i$ iff $p_i \in s$. This is extended to Boolean combinations on such predicates in a standard way. For a state s, we denote by φ_s the formula that is a conjunction of the places that are in s and the negated places that are not in s. Thus, φ_s is satisfied by the state s and by no other state. For the Petri Net in Figure 1, the initial state s is characterized by $\varphi_s = p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4 \wedge \neg p_5$. For a set of states $Q \subseteq S$, we can write a *characteristic formula* $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or use any equivalent propositional formula. We say that a formula φ is an *invariant* of a Petri Net N if $s \models \varphi$ for each $s \in reach(N)$, i.e., if φ holds in every reachable state.

Definition 6. *A* process *of a Petri Net N is a subset of the transitions $\pi \subseteq T$ satisfying that for each $t_1, t_2 \in T$, such that $(t_1, t_2) \in I$, there is no reachable state s in which both t_1 and t_2 are enabled.*

We assume a given set of processes Π that covers all the transitions of the net, i.e., $\bigcup_{\pi \in \Pi} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes.

Definition 7. *The* neighborhood $ngb(\pi)$ *of a process* π *is the set of places* $\bigcup_{t\in\pi}({}^\bullet t \cup t^\bullet)$.

We want to enforce global properties on Petri Nets in a distributed fashion. For a Petri Net N, we consider a property of the form $\Psi \subseteq S \times T$. That is, Ψ defines the allowed states, and, furthermore, which transition may be fired in each allowed global state. Note that as a private case Ψ can represent an *invariant* (when the transitions are not constrained).

Definition 8. *Let* N *be a Petri Net and* $\Psi \subseteq S \times T$. *We denote* (N, Ψ) *the pair made of* N *and the property* Ψ *that we want to enforce. A transition* t *of* N *is* enabled with respect to Ψ *in a state* s *if* $s[t\rangle$ *and, furthermore,* $(s,t) \in \Psi$. r *with* $s[r\rangle$ *such that* $t \ll r$. *An* execution *of* (N, Ψ) *is a maximal sequence* $s_0t_1s_1t_2s_2t_3\ldots$ *of* N *such that for each state* s_i *in the sequence, it holds that* $(s_i, t_{i+1}) \in \Psi$. *We denote the executions of* (N, Ψ) *by* $exec(N, \Psi)$, *and the set of states reachable these executions by* $reach(N, \Psi)$. *We assume that those sets are nonempty.*

Clearly, $reach(N, \Psi) \subseteq reach(N)$ and $exec(N, \Psi) \subseteq pref(exec(N))$; recall that restricting N according to Ψ may introduce deadlocks.

In particular, we are interested in enforcing priority policies. Indeed, a priority order $\ll$ is a partial order relation among the transitions T of N and thus defines a set Ψ in a straightforward manner. We use priorities as a running example. If Ψ is defined by a priority order $\ll$, then $(s,t) \in \Psi$ if $s[t\rangle$ is enabled in s and has a maximal priority among the transitions enabled in s. That is there is no other transition r with $s[r\rangle$ such that $t \ll r$. In the special case of a Petri Net with priorities $(N, \ll)$, we write $exec(N, \ll)$ and $reach(N, \ll)$ instead of $exec(N)$ and $reach(N)$, respectively. Note that priority orders do not introduce new deadlocks, and thus we have $exec(N, \ll) \subseteq exec(N)$.

Let us now consider the prioritized Petri Net N of Figure 1. The executions of N, when the priorities are *not* taken into account, include those with finite prefixes $abcd, acbd, bacd, badc$. However, when taking the priorities into account, the prioritized executions of N contain only alternations of c and d.

Definition 9. *The* local information *of a process* π *of a Petri Net* N *in a state* s *is* $s|_\pi = s \cap nbg(\pi)$.

That is, the local information of a process π in a given state consists of the restriction of the state to the neighborhood of the transitions of π. It plays the role of a *local state* of π in s. Our definition of local information is only one among possible definitions that can be used for modeling the part of the state that the system is aware of at any given moment.

Definition 10. *Define an equivalence relation* $\equiv_\pi \subseteq reach(N) \times reach(N)$ *such that* $s \equiv_\pi s'$ *when* $s|_\pi = s'|_\pi$.

It is easy to see that the enabledness of a transition depends only on the local information of a process that contains it, i.e., if $t \in \pi$ and $s \equiv_\pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$.

We cannot always make a local decision, based on the local information of processes (and sometimes sets of processes) that would guarantee that the global property Ψ is enforced. Indeed, in a Prioritized Petri Net $(N, \ll)$, there may exist different states $s, s' \in reach(N)$ such that $s \equiv_\pi s'$, a transition $t \in \pi$ is an enabled transition in s with maximal priority, but in s' this transition is not maximal among the enabled transitions.

To reason about properties, we will use predicates. We can easily construct the following formulas, representing state sets, using only propositions representing places of the Petri Net:

- $\varphi_{reach(N)}$: all the reachable states of N. Similarly, $\varphi_{reach(N,\Psi)}$ are the reachable states of the executions of $(N, \Psi$.
- $\varphi_{en(t)}$: the states where transition t is enabled.
- $\varphi_{\Psi(t)}$: the set of 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^{Ψ}_{df}: the states in which at least one transition is enabled w.r.t. Ψ, i.e., in which there is no deadlock in (N, Ψ). Formally: $\varphi^{\Psi}_{df} = \varphi_{reach(N,\Psi)} \wedge \bigvee_{t\in T} \varphi_{\Psi(t)}$.
- $\varphi_{s|_\pi}$: the states in which the local information of process π is $s|_\pi$.

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

3 Knowledge based Approach for Distributed Control

3.1 The support policy

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 $exec(N') \subseteq exec(N, \Psi)$. Moreover, $reach(N')$ must not introduce new deadlock states that are not already in $reach(N, \Psi)$ or be empty. In this case, we say that N' *implements* (N, Ψ).

For simplicity of the transformation, we consider extended Petri Nets [5], where processes may have local variables, and transitions have an enabling condition and a data transformation.

Definition 11. *An* Extended Petri Net *has, in addition to the Petri Net components, also finite variables V_π for each process $\pi \in \Pi$. These variables have some initial value according to some initial assignment $\mathcal{I}$. The enabling condition of each transition t is augmented to include also a predicate en_t on the variables $V_t = \cup_{\pi\in proc(t)} V_\pi$. In order for t to fire, 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 V_t are updated according to the transformation f_t that is also associated with t. A Petri Net N' extends *N, if N' is obtained from N by adding only variables, enabled conditions on the added variables and assignments to these variables by the transitions and by $\mathcal{I}$.*

In order to compare the execution of an original Petri Net and a Petri Net extending it as above, the comparison will be based only on the places and transitions of the original net. That is, we will ignore (or project out, for the sake of comparison) the additional variables.

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

Proof. The net N' can only restrict the executions $exec(N')$, while adding values to the variables added in N'. Some of these restrictions may result in an early deadlock. □

As we saw in the previous section, it is not possible in general to decide, based only on the local information of a process or a set of processes, whether some enabled transition is maximal with respect to a priority order, meaning that it is allowed by Ψ. We may, however, exploit some model checking based analysis of the system to identify the cases where such decision can be made.

Our approach for a local or semi-local decision on firing transitions is based on the knowledge of processes [4]. Basically, the knowledge of a process in a given (global) state is the set of reachable states that are consistent with the local information of that process.

Definition 12. *The process π* knows *a (Boolean) property φ in a state s, denoted $s \models K_\pi \varphi$, exactly when for each s' such that $s \equiv_\pi s'$, we have that $s' \models \varphi$.*

We obtain immediately from the definitions that if $s \models K_\pi \varphi$ and $s \equiv_\pi s'$, then $s' \models K_\pi \varphi$. Furthermore, the 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 Boolean property φ, one can perform model checking in order to decide whether $s \models K_\pi \varphi$. We have the following monotonicity property:

Theorem 1. *Let N be a Petri Net and N' an extension 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 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 in $s|_\pi$ can only increase. □

Monotonicity is important to ensure Ψ in N'. 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'. Monotonicity thus ensures the correctness of N'.

The transformation for the support policy will represent a disjunctive architecture for decentralized controller [19]. In control theory, one transforms the system to block some transitions in order to satisfy the given constraint. This

is done by adding a supervisor process [14], which is usually an automaton that runs synchronously with the controlled system. The supervisors are often (finite state) automata observing the controlled system, progressing according to the transitions they observe, and blocking some of the enabled transitions, depending on its current state. A distributed controller sets up a supervisor per each process. Some of the transitions may be defined as *uncontrollable*, meaning that the controller cannot block them.

The construction of a decentralized controller is based on a support table as introduced in [1, 7]. We do not formalize the details of the construction here, but the intuition provided here should be sufficient.

> At a state s, a transition t is *supported by a process* π *containing* t only if π knows in s (and thus in all s' which π cannot distinguish from s) about (s, t) respecting $\Psi s \models K_\pi \varphi_{\Psi(t)}$; a transition can be fired (is enabled) in a state only if, in addition to its original enabledness condition, at least one of the processes containing it *supports* it.

To implement the support policy, we first create a *support table* Δ as follows: we check for each process π, reachable state $s \in reach(N)$ and transition $t \in \pi$, whether $s \models K_\pi \varphi_{\Psi(t)}$. If it holds, we put in the support table at the entry $s|_\pi$ the transitions t that are responsible for satisfying this property. In fact, as $s \models K_\pi \varphi$ and $s \equiv_\pi s'$ implies that $s' \models K_\pi \varphi$, it is sufficient to check this for a single representative state containing $s|_\pi$ out of each equivalence class of '$\equiv_\pi$'.

We construct for a Petri Net N a support table Δ and use it to control (restrict) the executions of N to satisfy the property Ψ. Each process π in N is equipped with the entries of this table of the form $s|_\pi$ for s a reachable state. Before firing a transition, a process π of N consults the entry $s|_\pi$ that corresponds to its current local information, and supports only the transitions that appear in that entry. This can be represented as an extended Petri Net N^Δ.

The construction of the support table is simple and its size is limited to the number of different local informations of the process and not to the (sometimes exponentially larger) size of the state space.

3.2 Solutions when the support policy fails

Sometimes the knowledge based analysis does not provide an indication for a controller. Consider the Prioritized Petri Net $(N, \ll)$ of Figure 1. The right process π_r, upon having a token in p_4, does not support c; the priorities dictate that c can be executed if b is not enabled, since c has a lower priority than b. But this information is not locally available to the right process, which cannot distinguish between the cases where the right process has a token in p_1, p_2 or p_3. To tackle this issue, several suggestions have been made:

1. Use knowledge of perfect recall [11, 1]. This means that the knowledge is not based only on the local information, 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 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. Although in this way we extend our knowledge (by separating local informations according with different histories), this still does not guarantee that a distributed controller can be found.

2. Combine the knowledge of certain processes together by synchronizing them. The definition of knowledge can be based on equivalence classes of states that share the same local information of several processes. With the combined knowledge, one can achieve more situations where the maximal priority transition is known. However, to use this knowledge at run time, these sets of processes need to be able to access their joint local information. This means synchronizing between them, at the cost of losing concurrency. At the limit, all processes can be combined, and no actual concurrency remains.
3. Instead of the fixed synchronization between processes, one may use temporary synchronization [7]. Processes interact to achieve common knowledge. This does not reduce the concurrency as much as the previous method, but requires a lot of overhead in sending messages to achieve the temporary synchronization.

4 Support policy Based on the Controlled System

We propose here two additional techniques, which are orthogonal to the previous ones, to handle the case where the support policy fails, i.e. where Δ does not allow establishing that N^Δ implements a live controller. The first one is based on the following observation:

> Instead of calculating the knowledge with respect to *all* the executions of the original system, we may calculate it based on the executions of the original system that invariantly satisfy Ψ.

The set of global states on these executions are a subset of the reachable states, and, furthermore, for each local information, the set of global states containing it is contained in the corresponding set of the original Petri Net. Thus, our knowledge in each global configuration may not decrease but possibly grow. Still, we need to show that calculating knowledge using this set of executions produces a correct controller.

Theorem 2. *Let N be a Petri Net and Ψ a property to be enforced. Let Δ be the support table calculated for $reach(N, \Psi)$, and let N^Δ be the extended Petri Net constructed for Δ. Then $exec(N^\Delta) \subseteq pref(exec(N, \Psi))$ and $reach(N^\Delta) \subseteq reach(N, \Psi)$.*

Proof. When a transition of N^Δ is supported in some state s according to the support table Δ, then for some supporting process $\pi \in \Pi$, $s \models K_\pi \Psi$. By definition of the knowledge operator, this implies that $s \models \Psi$. Thus, each firing

of a transition of N^{Δ} preserves Ψ. However, it is possible that at some point, there is not enough knowledge to support any transition. □

The above proof does not guarantee that N^{Δ} implements (N, Ψ), because in some states not enough knowledge is available to support transitions.

Let $\varphi_{support(\pi)}$ denote the disjunction of the formulas $\varphi_{s|_\pi}$ such that the entry $s|_\pi$ is nonempty in the support table. A sufficient condition for N^{Δ} to correctly implement (N, Ψ) is

$$\varphi_{df}^{\Psi} \rightarrow \bigvee_{\pi \in \Pi} \varphi_{support(\pi)} \tag{1}$$

This condition requires that for each state $reach(N, \Psi)$, that is not a deadlock in (N, Ψ), at least one transition is supported. When this condition does not hold, we say that the support table is *incomplete*.

For the Prioritized Petri Net of Figure 1, the calculation of the knowledge based on the reduced number of executions provides a controller, whereas the original calculation did not. If we analyze the knowledge based on the constrained executions, then c and d are fired alternately, and p_2 is never reached, hence b is never enabled. In this case, our knowledge in p_4 and in p_5 allows us to execute c or d, respectively, and avoid the deadlock.

Let us look now at a more elaborate example. Consider Petri Net N_1 of Figure 2 with the given priority rules. The separation of transitions of N_1 according

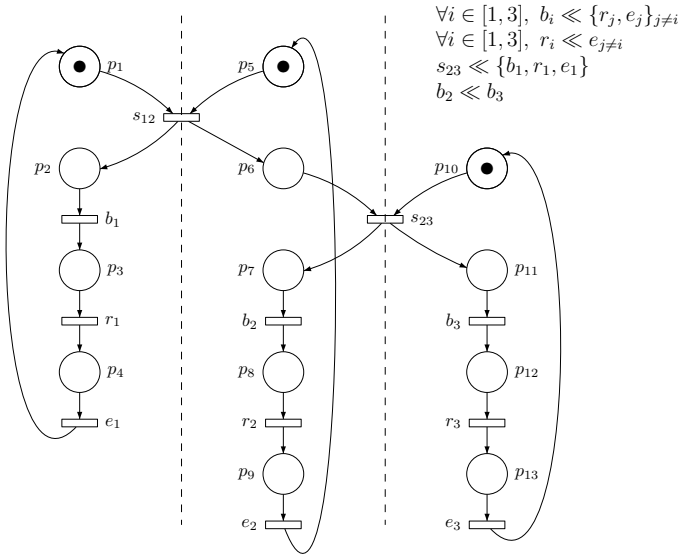


Fig. 2. A Petri Net N_1 with three processes π_1, π_2 and π_3

to processes is represented using dashed lines.

The example shows three processes π_1 (left), π_2 (in the middle), π_3 (right) that use binary synchronizations and priorities to enforce mutual exclusion for the execution of critical sections $(b_i r_i e_i)_{i=1,3}$. Intuitively, priority rules $b_i \ll \{r_j, e_j\}$ and $r_i \ll e_j$ give higher priority to transitions close to the end of the critical sections over the others. This enforces the mutual exclusion. Moreover, priority rules $s_{23} \ll \{b_1, r_1, e_1\}$ and $b_2 \ll b_3$ enforce a particular execution order of critical sections: repeatedly π_1 followed by π_3 and then by π_2.

Using the method of [1] described in section 3, no controller is found. Indeed, as all states are reachable, no process has enough knowledge to enter or progress in its critical section.

Now, if we calculate the support table on the prioritized executions, then we are able to construct a controller for N_1. Indeed, in the prioritized executions, there is always at most one process in its critical section. Thus, process π_1 always supports all its transitions as it can only enter the critical section in global states in which the other processes are blocked in front of a synchronization. Process π_3 supports all its transitions except s_{23}. Process π_2 supports transition s_{23} when π_1 is in p_1, transition b_2 when π_3 is in p_{10}, and transitions r_2 and e_2 in all cases.

5 Controllers based on an incomplete support table

In this section we show that even an incomplete support table Δ for (N, Ψ) may still define a controller N^{Δ} that correctly implements (N, Ψ). The reason is that states that are reachable in the executions of (N, Ψ) may be unreachable when applying the calculated support. The executions according to the support table may be a subset of the executions of (N, Ψ), and the problematic states may not occur.

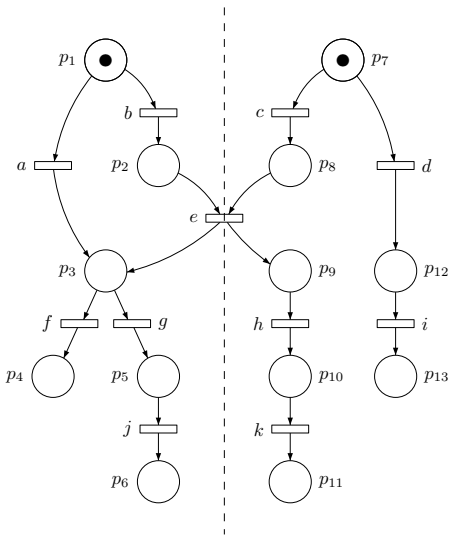
We illustrate this now on an example with priorities. Consider Petri Net N_2 of Figure 3. It represents two processes π_l (left) and π_r (right) with a single joint transition, which means that π_l can observe whether π_r is in one of the places p_8 and p_9. Similarly, π_l can observe whether π_r is in p_2 or in p_3. The table in Figure 3 shows the set of reachable states of N_2, including its termination (deadlock) states, denoted ■. Non-reachable states are in grey.

Suppose that the following set of priority rules must be enforced for the Petri Net N_2: $k \ll j$ and $c \ll b \ll i$.

The support table is calculated based on the knowledge of the original system. Table 1 presents a view of the global states of the Petri Net.

- non-reachable (grey), or
- in termination or deadlock (■) or
- reachable and non-deadlock.

In the latter case, the cell contains the transitions which are supported in this state by any of the processes (i.e., we have accumulated all the transitions supported by the local states that constitute together the global state). The blanks in this incomplete table represent states in which no process supports any transition. There are two such states, namely $\{p_4, p_{10}\}$ and $\{p_6, p_{10}\}$. The situation



$\pi_l = \{a, b, e, f, g, j\}$
$ngb(\pi_l) = \{p_1, \dots, p_6, p_8, p_9\}$

$\pi_r = \{c, d, e, h, i, k\}$
$ngb(\pi_r) = \{p_2, p_3, p_7, \dots, p_{13}\}$

$reach(N_2) =$

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1							
p_2							■
p_3							
p_4		■			■		■
p_5							
p_6		■			■		■

Fig. 3. A Petri Net N_2 with two processes π_l and π_r

in both states is the following: π_l has terminated and π_r could take transition k, but without an additional synchronization, there is no way for π_r to know that it may safely execute k.

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, d					a, i	a
p_2	c, d	e				i	■
p_3	f, g, c, d	f, g	f, g, h	f, g, k	f, g	f, g, i	f, g
p_4	d	■	h		■	i	■
p_5	j, d	j	j, h	j	j	j, i	j
p_6	d	■	h		■	i	■

Table 1. Support policy for N_2 with priorities $k \ll j$ and $c \ll b \ll i$

Note that in state $\{p_1, p_7\}$, process π_l supports a and process π_r supports d; it is impossible for π_l to know whether π_r is in p_{12} or not, and therefore b (which has lower priority than i) is not supported by π_l. Similarly, π_r does not support c (which has lower priority than b). While c is supported, e.g., in $\{p_2, p_7\}$, transition b is never supported, hence never fired in N_2^{Δ}, although it is allowed according to the priority rules in some states.

As a consequence, the set of states reachable in N_2^{Δ} may be smaller than $reach(N_2, \ll)$. Indeed, $reach(N_2^{\Delta})$ does not contain any state including the places p_2 together with p_9, p_{10} or p_{11}. This means in particular that the blanks in Table 1 are in fact not reachable, and thus N_2^{Δ} correctly implements $(N_2, \ll)$.

6 Comparison with history-based controllers

We show now that the use of perfect recall is independent of the methods proposed in this paper, meaning that in some cases only history is able to provide a controller, while in others it is still relevant to check whether an incomplete table provides a controller.

Consider the Petri Net N_2 of Figure 3, this time with the priorities $g \ll k$ and $f \ll i$. In this case, the set of reachable states is the same, regardless of the use or not of priorities. Consequently, there is no difference between the support policy based on the unrestricted system and the prioritized executions. Moreover, this support policy fails because there are two reachable global states where no process is supporting a transition, appearing as blanks in table 2: $\{p_3, p_{11}\}$ and $\{p_3, p_{13}\}$. Furthermore, these global states are also reachable in the controlled system, meaning that the heuristics applied in the previous example does not help either.

Nevertheless, this example may be controlled if perfect recall is used. If the left-process can remember the path it takes to reach p_3, it can distinguish between reaching p_3 directly after p_1 (by firing a) or respectively by passing through place p_2. Now, the set of reachable states contains enough information for the support policy to succeed.

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, b, c, d	a, b				a, b, i	a, b
p_2	c, d	e				i	■
p_3	c, d	f, g	f, g, h	k		i	
p_4	c, d	■	h	k	■	i	■
p_5	j, c, d	j	j, h	j, k	j	j, i	j
p_6	c, d	■	h	k	■	i	■

Table 2. Support policy for N_2 with priorities $g \ll k$ and $f \ll i$ without history

Our last example illustrates the combined use of perfect recall and an incomplete table to build a controller. Consider again Petri Net N_2 of Figure 3, now with priorities $g \ll k$, $f \ll \{i, k\}$ and $c \ll b \ll i$. On one hand, building the support table using the prioritized executions does not provide enough knowledge to control the system, and the incomplete support table does not provide a controller. On the other hand, the use of history as shown previously does not help either. Table 3 reflects the incomplete support table constructed using jointly the prioritized executions and perfect recall. Additional information related to the total recall is presented in the rows and columns of the table only when it is relevant for the support table. We can observe that in $\{p_3$ after $p_2, p_{11}\}$, no transition is supported by any process. However, the system can be controlled according to this table. Indeed, no extra-deadlock (blank) is actually reachable within the controlled system for a reason similar to the example in Section 5. This means that only the combination of several techniques leads to a controller.

	p_7	p_7 after p_3	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, d						a, i	a
p_2	c, d		e				i	■
p_3 after p_1		c, d, g	f, g				g, i	g
p_3 after p_2				f, g, h	k			
p_4		c, d	■	h	k	■	i	■
p_5		j, c, d	j	j, h	j, k	j	j, i	j
p_6		c, d	■	h	k	■	i	■

Table 3. Support policy for N_2 with $g \ll k$, $f \ll \{i, k\}$, $c \ll b \ll i$ and history

7 Implementation and Experimental Results

In [7], we implemented a prototype for experimenting with knowledge based controlling of distributed systems. We integrated now the two results of this paper. More precisely, the support table is now 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.

We present here some results illustrating the improvements thus obtained. Let us go back to the Petri Net of Figure 2. Using the approach of [7], several additional synchronizations were added in order to check maximality of transitions in the critical section. Out of 80 reachable states, 26 are global states in which no process can support an interaction. More precisely, 4 transitions out of 11 always require a synchronization to be fired: these transitions are b_1, r_1, b_3, r_3. As a result, an execution of 10,000 steps contains exactly 3636 ($= 10000 \times 4/11$) synchronizations. Using the prioritized executions to build the support table, we do not need any synchronization to build a controller.

Consider now a simplification of this example with two processes instead of three. In this case, interestingly, the method of [7] would not result in the execution of any additional synchronization. The reason for this is that the states requiring additional synchronizations are exactly the states in which no transition can be supported, meaning that synchronizations are added only when they are necessary. As these states are unreachable in the prioritized executions, no synchronization ever takes place. This emphasizes the fact that both approaches can be combined efficiently.

8 Conclusion

Calculating knowledge properties can be used for constructing a distributed controller that imposes some state property Ψ on a system. A transformation is used to impose a global property Ψ invariantly. To maintain the concurrent nature of the system, the decision on which transitions to permit needs to be made locally. Checking whether one can impose such a transformation without adding interaction is undecidable [18, 17, 7]. Knowledge can be used to help in constructing such a controller. When a process locally knows that executing a

transition will satisfy Ψ, then it is safe to support it. By combining the knowledge of different processes [7], if we are allowed to add synchronization, the synthesis becomes decidable, since at the limit, we may obtain a fully synchronized, i.e., a global system. Now, adding extensive synchronizations is undesirable.

We observe here that the knowledge approach for constructing a distributed controller is based on analyzing the original system in order to achieve the invariance of Ψ after the transformation. Thus, the use of knowledge calculated on the original version may be pessimistic in concluding when transitions can be supported. This brings us to two useful observations that can remove the need for some of the additional interactions 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 its executions that enforce Ψ. This gives fewer executions and fewer reachable states and enhances 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 result of the knowledge analysis may seem to lack the ability of supporting a way to continue from some states, this may not be the case. Analyzing the system after imposing the restriction forced by the analysis may result in a system that does not introduce new deadlock: the deadlocks appearing in states where no enabled transition is supported are in fact unreachable.

In this paper, we showed 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 interactions or fixed synchronizations between processes. We propose to use these knowledge based techniques as a practical way of synthesizing distributed controllers.

References

1. A. Basu, S. Bensalem, D.Peled, J. Sifakis, Priority Scheduling of Distributed Systems Based on Model Checking, CAV 2009, Grenoble, France, LNCS 5643, Springer, 79–93.
2. A. Basu, P. Bidinger, M. Bozga, J. Sifakis, Distributed Semantics and Implementation for Systems with Interaction and Priority, FORTE 2008, Tokyo, Japan, LNCS 5048, Springer, 116–133.
3. E. A. Emerson, E. M. Clarke, Characterizing Correctness Properties of Parallel Programs using Fixpoints, ICALP 1980, LNCS 85, Springer, 169–181.
4. R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, Cambridge MA, 1995.
5. H. J. Genrich, K. Lautenbauch, System Modeling with High-level Petri Nets, Theoretical Computer Science 13, 1981, 109-135.
6. G. Gößler, J. Sifakis, Priority Systems, FMCO 2003, LNCS 3188, Springer, Leiden, The Netherlands, 2004, 443-466.
7. S. Graf, D. Peled, S. Quinton, Achieving Distributed Control Through Model Checking, CAV 2010, Springer, to appear.

8. J. Y. Halpern, L. D. Zuck, A Little Knowledge Goes a Long Way: Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols, Journal of the ACM, 39(3), 1992, 449–478.
9. C. A. R. Hoare, Communicating Sequential Processes, Communication of the ACM 21, 1978, 666-677.
10. Z. Manna, A. Pnueli, How to Cook a Temporal Proof System for Your Pet Language, POPL 1983, Austin, TX, 141–154.
11. R. van der Meyden, Common Knowledge and Update in Finite Environment, Information and Computation, 140, 1980, 115-157.
12. A. Pnueli, R. Rosner, Distributed Reactive Systems Are Hard to Synthesize, FOCS 1990, St. Louis, USA, IEEE II, 746–757.
13. J. P. Queille, J. Sifakis, Specification and Verification of Concurrent Systems in CESAR, 5th International Symposium on Programming, 1981, 337–350.
14. P. J. Ramadge, W. M. Wonham, Supervisory Control of a Class of Discrete-Event Processes, SIAM journal on control and optimization, 25(1), 1987, 206–230.
15. K. Rudie, S. Laurie Ricker, Know Means No: Incorporating Knowledge into Discrete-Event Control Systems, IEEE Transactions on Automatic Control, 45(9):1656–1668, 2000.
16. K. Rudie, W. Murray Wonham, Think Globally, Act Locally: Decentralized Supervisory Control, IEEE Transactions on Automatic Control, 37(11):1692–1708, 1992.
17. J. G. Thistle, Undecidability in Descentralized Supervision, System and Control Letters volume 54, 2005, 503-509.
18. S. Tripakis, Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters, 90(1):21–28, 2004.
19. T. S. Yoo, S. Lafortune, A General Architecture for Decentralized Supervisory Control of Discrete-Event systems, Discrete event dynamic systems, theory & applications, 12(3) 2002, 335-377.