Scheduling of Distributed Systems with Priorities Based on Knowledge

Saddek Bensalem[1], Doron Peled[2], and Joseph Sifakis[1]

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

Abstract. Priorities are used to control the execution of systems to meet given requirements for optimal use of resources, e.g., by using scheduling policies. For distributed systems, it is hard to find efficient implementations for priorities; because they express constraints on global states, their implementation may incur considerable overhead.
Our method is based on performing model checking for knowledge properties. It allows identifying where the local information of a process is sufficient to schedule the execution of a high priority transition. As a result of the model checking, the program is transformed to react upon the knowledge it has at each point. The transformed version has no priorities, and uses the gathered information and its knowledge to limit the enabledness of transitions so that it matches or approximates the original specification of priorities.

1 Introduction

Executing transitions according to a priority policy is complicated when each process has a limited view of the situation of the rest of the system. Such limited local information can be described as the *knowledge* that processes have at each point of the execution [3, 4]. Separating the design of the system into a transition system and a set of priorities can be a very powerful tool [6], yet quite challenging to implement [1]. Our solution for implementing priorities is based on model checking [2, 10] of knowledge properties [9]. This analysis checks which processes possess knowledge about having a maximal priority transition enabled at the current state.

The information gathered during the model checking stage is used as a basis for a program transformation. It produces a new program without priorities, which implements or at least approximates the prioritized behaviors of the old program. At runtime, processes consult some table, constructed based upon the apriory model checking analysis, that tells them, depending on the current local information, whether a current enabled transition has a maximal priority and thus can be immediately executed. This transformation only blocks some of the transitions, based on the precalculated table. Thus, it does not introduce any new executions or deadlocks, and consequently preserves all the linear temporal logic properties [8] of the system.

For states where no process can locally know about having a maximal priority transition, we suggest several options. One solution is to put some semi-global observers that can observe the combined situation of several processes, obtaining in this way more knowledge regarding which process has a transition with maximal priority. Another possibility is to relax the priority policy, and allow a good approximation. The priorities discussed in this paper are inspired by the BIP system (Behavior Interaction Priority) [6].

2 Preliminaries

The model used in this paper is Petri Nets. This model has a visual representation that is helpful in presenting our examples. In addition, this model is very close to the BIP model. The method and algorithms developed here can equally apply to other models, e.g., transition systems, communicating automata, etc.

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

- P *is a finite set of* places. *The* states *are defined as* $S = 2^P$.
- 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.*
- $s_0 \subseteq P$ *is the* initial state *(hence* $s_0 \in S$*).*

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* $t^\bullet \cap s = \emptyset$.

A state s is in *deadlock* if there is no enabled transition from it. We denote the fact that t is enabled from s by $s[t\rangle$.

Definition 3. *A transition* t *can be* fired *(or* executed*) from state* s *to state* s'*, which is denoted by* $s[t\rangle s'$*, when* t *is enabled at* s*. Then,* $s' = (s \backslash {}^\bullet t) \cup t^\bullet$*. 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* $t_1 t_2 \dots t_k$.

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.*

Visually, transitions are represented as lines, places as circles, and the relation E is represented using arrows. In Figure 1, there are places p_1, p_2, ..., p_7 and transitions t_1, t_2, t_3, t_4. We depict a state by putting full circles, called *tokens*, inside the places of that state. In the example in Figure 1, the initial state s_0 is $\{p_1, p_2, p_7\}$. If we fire transition t_1 from the initial state, the tokens from p_1 and p_7 will be removed, and a token will be placed in p_3. The transitions that are enabled from the initial state are t_1 and t_2. In the Petri Net in Figure 1, all the transitions are dependent on each other, since they all involve the place p_7. Removing p_7, as in Figure 2, makes both t_1 and t_3 become independent on both t_2 and t_4.

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 $reach(N)$.

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 exactly by the state s and no other state. For the Petri Net in Figure 1 we have that the initial state s satisfies $\varphi_s = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge p_7$. For a set of states $Q \subseteq S$, we can write a *characterizing formula* $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or use any equivalent propositional formula. We say that a predicate φ is an *invariant* of a Petri Net N if $s \models \varphi$ for each $s \in reach(N)$. As usual in logic, when a formula φ_Q characterizes a set of states Q and a formula $\varphi_{Q'}$ characterizes a set of states Q', then $Q \subseteq Q'$ if and only if $\varphi_Q \rightarrow \varphi_{Q'}$.

Definition 6. *A* process *of a Petri Net N is a subset of the transitions $\pi \subseteq T$ satisfying that for each $t_1, t_2 \in \pi$, 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 will sometimes denote the separation of transitions of a Petri Net in a figure to different processes using dotted lines. We assume a given set of processes $\mathcal{S}$ that covers all the transitions of the net, i.e., $\bigcup_{\pi \in \mathcal{S}} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. Let $proc(t)$ be the set of processes to which t belongs, i.e., $proc(t) = \{\pi | t \in \pi\}$. Note that there can be multiple ways to define a set of processes for the same Petri Net.

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

In the rest of this paper, when a formula refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π instead. For the Petri Net in Figure 1, there are two executions: *acbd* and *bdac*. There are two processes: the *left* process $\pi_l = \{a, c\}$ and the *right* process $\pi_r = \{b, d\}$. The neighborhood of process π_l is $\{p_1, p_3, p_5, p_7\}$. The place p_7, belonging to the neighborhood of both processes, 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.

Definition 8. *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.*

Definition 9. *A transition t has a* maximal *priority in a state s if $s[t\rangle$ and, furthermore, there is no transition r with $s[r\rangle$ such that $t \ll r$.*

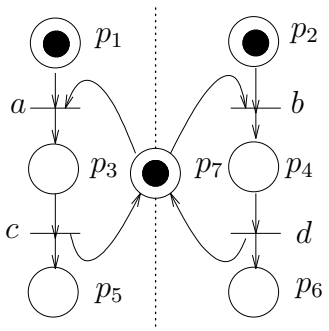

Fig. 1. A Petri Net

Definition 10. *An execution of a Petri Net with priorities is a maximal alternating sequence of states and transitions $s_0 t_1 s_1 t_2 s_2 t_3 \ldots$ with s_0 the initial state of the Petri Net. Furthermore, for each state s_i in the sequence it holds that $s_i[t_{i+1}\rangle s_{i+1}$ for t_{i+1} having maximal priority in s_i.*

To emphasize that the executions take into account the priorities, we sometimes call them *prioritized executions*. We denote the executions of a Prioritized Petri Net $(N, \ll)$ by $priorE(N, \ll)$. The set of states that appear on $priorE(N, \ll)$ will be denoted by $reach(N, \ll)$. The following is a direct consequence of the definitions:

Lemma 1. *$reach(N, \ll) \subseteq reach(N)$ and $priorE(N, \ll) \subseteq exec(N)$.*

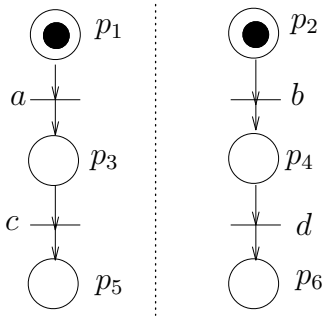

Fig. 2. A Petri Net with Priorities $a \ll d, b \ll c$.

The executions of the Petri Net M in Figure 2, when the priorities $a \ll d$ and $b \ll c$ are *not* taken into account, include $abcd, acbd, bacd, badc$, etc. However,

when taking the priorities into account, the prioritized executions of M are the same as the executions of the Net N in Figure 1.

Unfortunately, enforcing prioritized executions in a completely distributed way may incur high synchronization overhead [1] or even be impossible. In Figure 2, a and c belong to one (left) process π_l, and b and d belong to another (right) process π_r, with no interaction between the processes. Then, the left process π_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 be locally available to the left process, which cannot distinguish between the cases where the right process has a token in p_2, p_4 or p_6.

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

That is, the local information of Π at a given state consists of the restriction of the state to the neighborhood of the transitions of Π. The local information of a process π in a state s plays the role of a *local state* of π in s. We prefer to use the term "local information" since neighborhoods of different processes may overlap on some common places rather than partitioning the global states. In the Petri Net in Figure 1, the local information of the left process in any state s consists of restriction of s to the places $\{p_1, p_3, p_5, p_7\}$. In the depicted initial state, the local information is $\{p_1, p_7\}$.

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. Consider again the Petri Net in Figure 1. The places p_1, p_3 and p_5 may represent the location counter in the left process. When there is a token in p_1 or p_3, it is reasonable to assume that the existence of a token in place p_7 (the semaphore) is known to the left process. However, it is implementation dependent whether the left process is aware of the value of the semaphore when the token is at place p_5 or not. This is because at this point, the semaphore may affect the enabledness of the right process (if it has a token in p_2) but would not have an effect on the left process. Thus, a subtly different definition (and corresponding implementation) of local information can be used instead. For simplicity, we will continue with the simpler definition above.

Definition 12. *Let $\Pi \subseteq \mathcal{S}$ be a set of processes. Define an equivalence relation $\equiv_\Pi \subseteq reach(N) \times reach(N)$ such that $s \equiv_\Pi s'$ when $s|_\pi = s'|_\pi$ for each $\pi \in \Pi$.*

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 2. *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 only the prioritized executions in a Prioritized Petri Net $(N, \ll)$. It is possible that there are two 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. This can be demonstrated on the Prioritized Petri Net in Figure 2. There, we have that for π_l, $\{p_1, p_2\} \equiv_{\pi_l} \{p_1, p_4\}$. In the state $\{p_1, p_2\}$, a is a maximal priority enabled transition, while in $\{p_1, p_4\}$, a is not anymore maximal, as we have that $a \ll d$, and both a and d are now enabled.

In the following we will use predicates, with propositions that are the place of the Petri Net, to explain the approach and the implementation:

All the reachable states: $\varphi_{reach(N)}$.
The states where transition t is enabled: $\varphi_{en(t)}$.
At least one transition is enabled, i.e., there is no deadlock: $\varphi_{df} = \bigvee_{t \in T} \varphi_{en(t)}$.
The transition t has a maximal priority among all the enabled transitions of the system: $\varphi_{max(t)} = \varphi_{en(t)} \wedge \bigwedge_{t \ll r} \neg \varphi_{en(r)}$.
The local information of processes Π at state s: $\varphi_{s|_\Pi}$.

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

3 Knowledge Based Approach for Priority Scheduling

The problem we want to solve is the following: given a Petri Net with priorities $(N, \ll)$, we want to obtain a Petri Net N' without priorities, such that $exec(N') \subseteq priorE(N, \ll)$. Moreover, $reach(N')$ must not introduce new deadlock states that are not in $reach(N, \ll)$.

In control theory, the transformation that takes a system and allows blocking some transitions adds a supervisor process [11], 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. Some of the transitions may be defined as *uncontrollable*, meaning that the controller cannot block them. Of course, the definition of uncontrollable transitions must be consistent with the priorities; if a transition is uncontrollable and is enabled in some state together with a higher priority transition, then no correct controller can be constructed. A distributed controller sets up such a supervisor per each process. In a *conjunctive supervisor* [15], in order to execute an enabled transition t that belongs to several processes, all the corresponding supervisors must agree to fire it. In a *disjunctive supervisor*, it is sufficient that at least one of the supervisors allows (supports) t

Instead of constructing supervisors, one per processfor single or sets of processes, we transform the processes, represented as sets of transitions in a Petri Net. For simplicity of the transformation, we allow Extended Petri Nets, where processes may have local variables, and each transition has an enabling condition and a transformation.

Definition 13. *An* Extended Petri Net *has, in addition to the Petri Net components, also finite variables V_π for each process $\pi \in \Pi$. 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.*

As we saw in the previous section, we may not be able to decide, based on the local information of a process or a set of processes, whether some enabled transition is maximal with respect to priority. We can exploit some model checking based analysis to identify the cases where such local control decisions can be made. Our approach for a local or semi-local decision on firing transitions is based on the *knowledge* of processes [3], or of sets of processes. Basically, the knowledge of a process at a given state is the possible combination of reachable states that are consistent with the local information of that process.

Definition 14. *The processes Π (jointly)* know *a (Boolean) property ψ in a state s, denoted $s \models K_\Pi \psi$, exactly when for each s' such that $s \equiv_\Pi s'$, we have that $s' \models \psi$.*

At the moment, the definition of knowledge assumes that the processes do not maintain a log with their history. We henceforth use knowledge formulas combined with using Boolean operators and propositions. For a detailed syntactic and semantic description one can refer, e.g., to [3]. In this paper We neither define nor use the nesting of knowledge operators, e.g., $K_{\Pi_1}(K_{\Pi_2}(\varphi))$, nor the notion of "common" knowledge $C_\Pi \varphi$.

The following lemmas follow immediate from the definitions:

Lemma 3. *If $s \models K_\Pi \varphi$ and $s \equiv_\Pi s'$, then $s' \models K_\Pi \varphi$.*

Lemma 4. *The processes Π know ψ at state s exactly when $(\varphi_{reach(N)} \wedge \varphi_{s|_\Pi}) \rightarrow \psi$ is a propositional tautology.*

Now, given a Petri Net with priorities, one can perform *model checking* in order to calculate whether $s \models K_\pi \psi$. Note that implementing Lemma 4, say with BDDs, is *not* the most space efficient way of checking knowledge properties, since $\varphi_{reach(N)}$ can be exponentially big in the size of the description of the Petri Net. In a (polynomial) space efficient check, we enumerate all the states s' such that $s \equiv_\pi s'$, check reachability of s' using binary search and, if reachable, check whether $s' \models \psi$.

4 The Supporting Process Policy

The *supporting process policy*, described below, transforms a Prioritized Petri Net $(N, \ll)$ into a priorityless Extended Petri Net N' that implements or at least approximates the priorities of the original net. This transformation augments the

states with additional information, and adds conditions for firing the transitions. This is related to the problem of supervisory control [11], where a supervisor is imposed on a system, restricting transitions from being fired at some of the states. We can map the states of the transformed version N' into the states of the original version N by projecting out additional variables that N' may have. In this way, we will be able to related the sets of states of the original and transformed version.

The supporting process policy can be classified as having a *disjunctive architecture for decentralized control* [15]. Although the details of the transformation are not given here, they should be clear from the theoretical explanation.

> At a state s, a transition t is *supported by a process π containing t* only if π knows in s about t having a maximal priority (among all the currently enabled transitions of the system), i.e., $s \models K_\pi \varphi_{max(t)}$; 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.

Based on the definition of knowledge, we have the following monotonicity property of knowledge:

Theorem 1. *Given that $s \models K_\Pi \varphi$ in the original program N, (when not taking the priorities into account) then $s \models K_\Pi \varphi$ also in the transformed version N'.*

This property is important to ensure the maximality of the priority of a transition after the transformation. The knowledge about maximality will be calculated *before* the transformation, and will be used to control the execution of the transitions. Then, we can conclude that the maximality remains also *after* the transformation.

We consider three levels of knowledge of processes related to having a maximal enabled transition:

- φ_1 Each process knows about all of its enabled transitions that have maximal priorities (among all enabled transitions).
  That is, $\varphi_1 = \bigwedge_{\pi \in \mathcal{S}} \bigwedge_{t \in \pi} (\varphi_{max(t)} \rightarrow K_\pi \varphi_{\max(t)})$.
- φ_2 For each process π, when one of its transitions has a maximal priority, the process knows about at least *one* such transition.
  $\varphi_2 = \bigwedge_{\pi \in \mathcal{S}} ((\bigvee_{t \in \pi} \varphi_{max(t)}) \rightarrow (\bigvee_{t \in \pi} K_\pi \varphi_{\max(t)}))$.
  Note that when all the transitions of each process π are totally ordered, then $\varphi_1 = \varphi_2$.
- φ_3 For each state where the system is not in a deadlock, *at least one process* can identify *one* of its transitions that has maximal priority.
  $\varphi_3 = \varphi_{df} \rightarrow (\bigvee_{\pi \in \mathcal{S}} \bigvee_{t \in \pi} K_\pi \varphi_{\max(t)})$.

We Denote the fact that φ is an invariant (i.e., holds in every reachable state) by using the usual temporal logic notation $\Box\varphi$ (see [8]). Notice that $\varphi_1 \rightarrow \varphi_2$ and $\varphi_2 \rightarrow \varphi_3$ hold, hence also $\Box\varphi_1 \rightarrow \Box\varphi_2$ and $\Box\varphi_2 \rightarrow \Box\varphi_3$. Processes have less knowledge according to φ_2 than according to φ_1, and then even less knowledge if only φ_3 holds.

Definition 15. *Let priorS(N, φ_i) be the set of executions when transitions are fired according to the supporting process policy when $\Box\varphi_i$ holds.*

That is, when $\Box\varphi_1$ holds, the processes support all of their maximal enabled transitions. When $\Box\varphi_2$ holds, the processes support at least one of their maximal enabled transitions, but not necessarily all of them. When $\Box\varphi_3$ holds, at least one enabled transition will be supported by some process, at each state, preventing deadlocks that did not exist in the prioritized net.

Lemma 5. *priorS(N, φ_1) = priorE$(N, \ll)$. Furthermore, for $i = 2$ or $i = 3$, priorS$(N, \varphi_i) \subseteq$ priorE$(N, \ll)$.*

This is because when $\Box\varphi_2$ or $\Box\varphi_3$ hold, but $\Box\varphi_1$ does not hold, then some maximally enabled transitions are supported, but some others may not. On the other hand, if $\Box\varphi_1$ holds, the supporting process policy does not limit the firing of maximal enabled transitions.

Implementing the local support policy: the support table

We first create a *support table* as follows: We check for each process π, reachable state $s \in reach(N)$ and transition $t \in \pi$, whether $s \models K_\pi \varphi_{\max(t)}$. 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, according to Lemma 3, it is sufficient to check this for a single representative state containing $s|_\pi$ out of each equivalence class of '$\equiv_\pi$'.

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. It is easy to see from the definition of φ_3 that checking $\Box\varphi_3$ is equivalent to checking the validity of the following Boolean implication:

$$\varphi_{df} \rightarrow \bigvee_{\pi \in \mathcal{S}} \varphi_{support(\pi)} \tag{1}$$

This means that at every reachable and non deadlock state, at least one process knows (and hence supports) at least one of its maximal enabled transitions.

Now, if at least $\Box\varphi_3$ holds, the support table we constructed for checking it can be consulted by the transformed program for implementing the supporting process policy. Each process π is equipped with the entries of this table of the form $s|_\pi$ for reachable s. Before making a transition, a process π consults the entry $s|_\pi$ that corresponds to its current local information, and supports only the transitions that appear in that entry. The transformed program can be represented as an Extended Petri Net. The construction is simple. The size of the support table is limited to the number of different local informations of the process and not to the (sometimes exponentially bigger) size of the state space.

Priority approximation

It is typical that there will be many states where φ_3 does not hold. In the Petri Net in Figure 3, when s includes p_3 but neither p_6 nor p_7 (which are both in the

neighborhood of π_l because of the joint transition c), we do not know whether we can support e: it has a lower priority than j, and π_l does not know whether j is currently enabled, has terminated, or even if the nondeterministic selection at p_6 has picked up transition d, and j is not executing thereafter. Thus, π_l does not support e. For similar reasons, π_l does not support f because of the possibility that transition h, with the higher priority, might be enabled simultaneously.

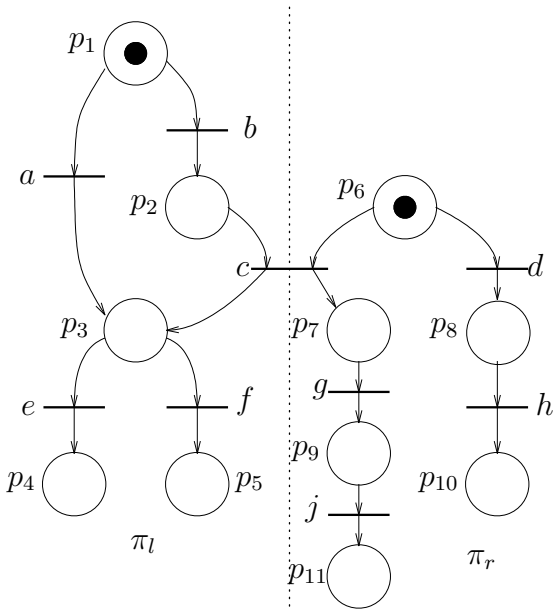

Fig. 3. Petri Net with priorities $e \ll j$ and $f \ll h$

When $\Box\varphi_3$ does not hold, one can provide various suboptimal solutions, which try to approximate the priority selection, meaning that not at all times the executed transition will be maximal. Consider a nondeadlock state s where $s \not\models \varphi_3$. In this case, the entries $s|_\pi$ are empty for each process π, and thus in state s, no transition will be supported. Hence none will be fired, resulting in a deadlock.

A *pessimistic approach* to fix this situation, without guaranteeing completely prioritized behavior, is to add to each empty entry $s|_\pi$ *at least one* of the transitions that are maximal among the enabled transitions of π. Another possibility, which adds less arbitrary transitions to the support table, but requires more intensive computation, is based on an iterative approach. Select an empty entry $s|_\pi$ in the support table where some transition $t \in \pi$ is enabled and is maximal among the enabled transitions of π. Put t into entry $s|_\pi$ of the support table. Update the formula (1), by adding the disjunct $\varphi_{s|_\pi}$ to $\varphi_{support(\pi)}$. Then recheck

Formula (1). Repeat adding transitions to empty entries in the support table until (1) holds. When it holds, it means that for each reachable state, there is a supported enabled transition, preventing new deadlocks.

Synchronizing Processes Approach

When Formula (1) does not hold, and thus also $\Box\varphi_3$, we can combine the knowledge of several processes to make decisions. This can be done by putting a supervisor that checks the combined local information of multiple processes. We then arrange a support table based on the joint local information of several processes $s|_{\Pi}$ rather than the local information of single processes $s|_{\pi}$. This corresponds to replacing π with Π in the formulas φ_1, φ_2 and φ_3. Such supervisors may reduce concurrency. However, this is not a problem if the controlled processes are threads, residing already in the same processor. It is not clear apriory on which sets of processes we want to put a supervisor in order to make their combined knowledge help in deciding the highest priority transition. Model checking under different groupings of processes, controlled and observed together, is then repeated until $\Box\varphi_1$ (or $\Box\varphi_2$ or $\Box\varphi_3$) holds.

Another possibility is the transfer of additional information via messages from one process to another. This also reduces concurrency and increases overhead.

Using Knowledge with Perfect Recall

Knowledge with perfect recall [9] assumes that a process π may keep its own local history, i.e., the sequence of local information sequence (sequence of local states) occurred so far. This may separate different occurrences of the same local information, when they appear at the end of different local histories. This allows the processes to decide on supporting a transition even in some cases where it was not possible under the previous knowledge definition.

Knowledge with perfect recall is defined so that a process *knows some property φ at some state s and given some local history σ*, if φ holds for each execution when reaching a state with the same local history σ. In our case, since the system is asynchronous, the processes are not always aware of other processes making moves, unless these moves can affect their own neighborhood (hence their local information). Hence the local history includes only moves by transitions that have some common input or output place with $ngb(\pi)$.

Definition 16. *Let ρ be a sequence of transitions of a Petri Net N and π a process of N. Then $\rho|_{\pi}$ is obtained from ρ by erasing the transitions that are independent of all the transitions in π.*

Observe that $\rho|_{\pi}$ includes exactly the transitions of ρ that change the neighborhood of π, including the transitions of π itself.

Definition 17. *Let $\sigma = s_1 t_1 s_2 t_2 \dots t_n s_{n+1}$ be a prefix of an execution of a Petri Net N and π a process. Then $\sigma|_{\pi}$, the* local history of π according to σ, *is an alternating sequence of local informations and transitions $l_{i_1} t_{i_1} l_{i_2} t_{i_2} \dots l_{i_k}$, where $t_{i_1} t_{i_2} \dots t_{i_{k-1}} = t_1 t_2 \dots t_n|_{\pi}$ and for each index i_j we have that $g_{i_j} = s_{i_j}|_{\pi}$.*

Thus, $\sigma|_\pi$ keeps from σ the transitions that change the neighborhood of σ, according to their order of appearance, and the local information of π just before and after each such transition. Since the system is asynchronous, π is not aware of the occurrence of any number of transitions that do not change its history. Recall that if a transition t that does not change the neighborhood of π is executed from state s, resulting in state s', then $s|_\pi = s'|_\pi$.

Now we can extend the definition of $\models$ in order to define knowledge with perfect recall.

Definition 18. *If σ is a finite prefix of an execution of a Petri Net N ending with a state s, then $\sigma \models \varphi$ exactly when $s \models \varphi$.*

We can also define an equivalence relation between finite prefixes:

Definition 19. *Let σ, σ' be two finite prefixes of a Petri Net N. Then $\sigma \equiv_\pi \sigma'$ when $\sigma|_\pi = \sigma'|_\pi$.*

This means that π observes the same alternating sequence of transitions and local information in both σ and σ'. We are ready now to define knowledge with perfect recall.

Definition 20. *Let σ be a finite prefix of a Petri Net N. Then a process π* knows with perfect recall *ψ after σ, if for each σ' such that $\sigma \equiv_\pi \sigma'$, $\sigma' \models \psi$.*

These definitions can be generalized to sets of processes by replacing π with Π. The properties φ_1, φ_2 and φ_3 can be checked where the knowledge operators refer to knowledge with perfect recall.

An algorithm for model checking knowledge with perfect recall was shown in [9], and our algorithm can be seen as a simplified version of it.

Definition 21. *Let $indseq_\pi$ be the set of finite sequences of transitions that do not change the neighborhood of π.*

Let $\mathcal{A} = (S, s_0, T)$ be a finite automaton representing the global states S of a Petri Net N, including the initial state $s_0 \in S$ and the transitions T between them. For each process π, we construct an automaton $\mathcal{A}_\pi$ representing the set of states of $\mathcal{A}$ where the Petri Net N can be after a given local history.

- The states of $\mathcal{A}_\pi$ are 2^S.
- The initial state Γ_0 of $\mathcal{A}_\pi$ is the set of states $\{s | \exists \mu \in indseq_\pi \, s.t. \, s_0[\mu\rangle s\}$. That is, the initial state of this automaton contains all the states obtained from s_0 by executing a finite number of transitions independent of (i.e., invisible of) π.
- The transition relation is $\Gamma \xrightarrow{t} \Gamma'$ between two states Γ, $\Gamma' \in 2^S$ and a transition $t \in T$ as follows: $\Gamma' = \{s' | \exists s \in \Gamma \, \exists \mu \in indseq_\pi \, s.t., s[t\mu\rangle s'\}$. That is, a move from Γ to Γ' corresponds to the execution of any transition t that change the neighborhood of π followed by transitions independent of π.

Model checking is possible even though the local histories may be unbounded because the number of such subsets Γ is bounded, and the successor relation between such different subsets, upon firing a transition t, as described above, is fixed.

Instead of the support table, for each process π we have a *support automaton*, representing the determinization of the above automaton. At runtime, the execution of each transition visible by π, i.e., one that can change π's neighborhood, will cause a move of this automaton (this means access to the support automaton of π with the execution of these transitions, even when they are not in π). If currently the state of the support automaton corresponds to a set of states Γ where in *all of them* the transition $t \in \pi$ is maximally enabled (checking this for the states in Γ was precalculated at the time of performing the transformation), then π currently supports t.

Unfortunately, the size of the support automaton, for each process, can be exponential in the size of the global state space (corresponding to a subset of the states where the current execution can be, given the local history). This gives quite a high overhead to such a transformation. Note that the local histories of the transformed net is a subset of the local histories of the original, priorityless net. Thus, Theorem 1 still holds when relativized to knowledge with perfect recall.

Returning to the example in Figure 3, knowledge with perfect recall can separate at a state where p_3 has the token but neither p_6 nor p_7 have a token, the case where a was executed from the case where c was executed. If a was executed, then π_l can safely support e, whose priority is comparable only with j, which is never enabled. Conversely, if c was executed, process π_l can support f, which is comparable only with h.

5 Discussion

We will now put our knowledge-based approach in the context of supervisor control synthesis. First, consider the general case where we have a concurrent system, described as a Petri Net (or a finite state transition system), on top of which we want to impose some property. The property imposed restricts the system to a particular set of states, and possibly a set of transitions allowed from any given state. This kind of restriction covers invariants and priorities, but not temporal properties. The transitions of the Petri Net are partitioned into processes, as in Definition 6, and each process can be controlled by a supervisor with local memory. Such a supervisor can observe the transitions that change the neighborhood of the process. Indeed, in control theory, such transitions are said to be *observable* by π (and the rest are thus unobservable by π). A process π can control a subset of the transitions that belong to that process. These are called the *controllable* transitions of π. A process can decide to support or not to support an enabled controllable transition. In a conjunctive controller, a transition must be supported by all the processes that are involved with it,

while in a disjunctive controller, it is enough that at least one of the involved processes supports an enabled transition in order for it to fire.

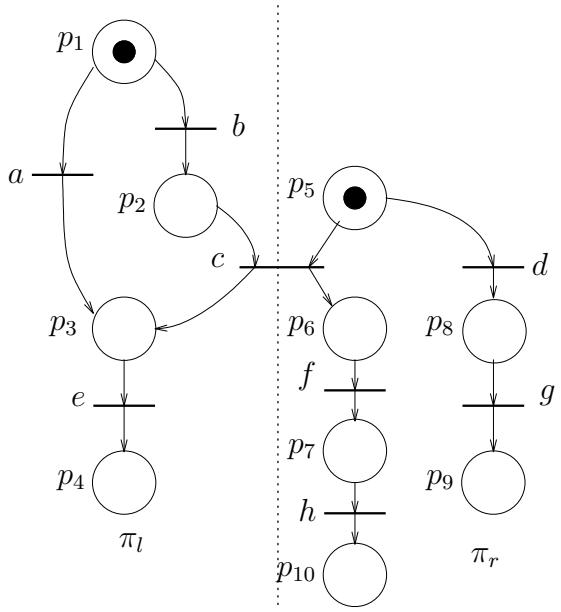

Fig. 4. Net with Priorities $e \ll h$ and $b \ll g$

Now consider the Net in Figure 4. According to the knowledge approach, when the local information of π_l includes p_1 neither p_5 nor p_6 (both of these latter places are in the neighborhood of π_l), process π_l does not know if g is enabled or not (the right process can have a token at p_7, p_8, p_9 or p_{10}). So π_l cannot support the transition b. Furthermore, when the left process is in state p_3, we cannot support transition e, since our knowledge does not distinguish between the case where h is enabled or not. If we use knowledge of perfect recall, then when we arrive at state p_3, we know whether previously a was executed or b. If a was executed, then it is safe to execute e. But if b happened (and subsequently c), then executing e may not be safe and adding the perfect recall does not help. Thus, our knowledge approach does not help us to construct a distributed supervisor.

Still, abandoning the knowledge approach, one can construct a distributed supervisor. This is done by having the left process π_l deciding to support only a from any local information with token in p_1. In this case, e will follow. In the right process π_r, the interaction c is not possible, hence d will be executed and thereafter g. In this case, there is no problem with priorities, as e is only ordered with respect to h. One may argue whether abandoning b in favor of a, both of

which have maximal priority, should be allowed here. But note that the support process policy, under conditions φ_2 and φ_3 may not know enough to support *all* the maximal priority transitions.

We have thus shown that the knowledge approach gives us an algorithm for constructing distributed controllers for priorities that may fail even when a controller exists. It is shown in [7] that the problem of deciding whether a distributed controller exists is, in general, an undecidable problem.

Knowledge was suggested as a tool for constructing a distributed supervisor in [12]. There, knowledge-controllability (termed *Kripke observability*) is studied as a basis for constructing a distributed supervisor. The requirement there is that *for each* transition, if it is enabled by the controlled system but must be blocked according to the additional constraint, then at least one process knows that fact and is thus able to prevent its execution. The construction here is different. We require that *at least* one process knows that the occurrence of *some* of its enabled transitions preserve the correctness of the imposed constraint, hence supporting its execution. The approach of [12] requires sufficient knowledge to allow *any* enabled transition that preserves the imposed constraint. Our approach preserves the correctness of the supervisor even when knowledge about other such transitions is limited, at the expense of restricting the choice of transitions.

6 Conclusions

Developing concurrent systems is an intricate task. One methodology, which lies behind the BIP system, is to define first the architecture and transitions, and at a later stage add priorities among the transitions. This methodology allows a convenient separation of the design effort. We presented in this paper the idea of using model checking analysis to calculate the local knowledge of the concurrent processes of the system about currently having a maximal priority transition. Model checking is used to transform the system into a priorityless version that implements the priorities. There are different versions of knowledge, related to the different ways we are allowed to transform the system. For example, the knowledge of each process, at a given time, may depend on including information about the history of computation.

After the analysis, we sometimes identify states where no process has enough information about having a maximal priority transition. In such cases, synchronizing between different processes, reducing the concurrency, is possible; semiglobal observers can coordinate several processes, obtaining joint knowledge of several processes. Another possible solution (not further elaborated here) involves adding coordination messages.

More generally, we suggest a programming methodology, based on a basic design (in this case, the architecture and the transitions) with added constraints (in this case, priorities). Model checking of knowledge properties is used to lift these added constraints by means of a program transformation. The resulted program behaves in an equivalent way, or approximates the behavior of the basic design with the constraints.

References

1. 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.
2. E. A. Emerson, E. M. Clarke, Characterizing Correctness Properties of Parallel Programs using Fixpoints, ICALP 1980, LNCS 85, Springer-Verlag, 169–181.
3. R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, Cambridge MA, 1995.
4. J. Y. Halpern, L. D. Zuck, A little knowledge goes a long way: knowledge based derivation and correctness proof for a family of protocols, Journal of the ACM, 39(3), 1992, 449–478.
5. C. A. R. Hoare, Communicating Sequential Processes, Communication of the ACM 21, 1978, 666-677.
6. G. Gößler, J. Sifakis, Priority Systems, Formal Methods for Components and Objects (FMCO 2003), Lecture Notes in Computer Science 3188, Springer-Verlag, Leiden, The Netherlands, 2004, 443-466.
7. S. Graf, D. Peled, S. Quinton, Achieving Distributed Control Through Model Checking, CAV 2010, Springer Verlag, to appear.
8. Z. Manna, A. Pnueli, How to Cook a Temporal Proof System for Your Pet Language, POPL 1983, Austin, TX, 141–154.
9. R. van der Meyden, Common Knowledge and Update in Finite Environment, Information and Computation, 140, 1980, 115-157.
10. J. P. Quielle, J. Sifakis, Specification and Verification of Concurrent Systems in CESAR, 5th International Symposium on Programming, 1981, 337–350.
11. 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.
12. 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.
13. K. Rudie, W. Murray Wonham, Think globally, act locally: decentralized supervisory control, IEEE Transactions on Automatic Control, 37(11):1692–1708, 1992.
14. Stavros Tripakis, Undecidable problems of decentralized observation and control on regular languages. Information Processing Letters, 90(1):21–28, 2004.
15. 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.