Priority Scheduling of Distributed Systems Based on Model Checking

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

[1] Centre Equation - VERIMAG, 2 Avenue de Vignate, Gieres, 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

In a distributed system, it can be quite nontrivial to implement distributed communication; for example, once one process decides that it is willing to communicate with a second process, that communication might not be available anymore, as the second process has meanwhile communicated with a third process. For this reason, concurrent programming languages may restrict the choice of communication. For example, Hoare [7] has initially restricted his programming language CSP to commit to a single output, where choice is still allowed between inputs; overriding this restriction may require some nontrivial algorithms [2,5]. A further implementation complication can, orthogonally, result from imposing priorities between transitions. Separating the design of the system into a transition system and a set of priorities can be a very powerful tool [8], yet quite challenging [1].

Executing transitions according to a priority policy is complicated due to the fact that 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 [6,4]. Our solution for implementing priorities is based on running an analysis of the system before it is executed using model checking [3,11]. Specifically, we use model checking for knowledge properties, similar to the algorithms suggested by Van der Meyden in [10]. This analysis checks which processes possess "knowledge" about having a maximal priority transition enabled at the current state.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 79–93, 2009.

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 model checking analysis, that tells them, depending on the current local state, whether a current enabled transition has a maximal priority and thus can be immediately executed. This transformation does not introduce any new executions or deadlocks, and is intended to preserve the linear temporal logic properties [9] of the net.

For states where no process can locally know about having a maximal priority transition, we suggest several options; this includes putting some semi-global observers that can observe the combined situation of several processes, passing additional synchronization messages, or using global variables, to inform the different processes about the situation of their neighbors. 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) [8].

2 Preliminaries

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 transition.*
- $s_0 \subseteq P$ *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 $\{p \in P | (t, p) \in E\}$. 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 $s[t\rangle$. A transition t is *fired* (or *executed*) from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled at s, and, furthermore, $s' = (s \setminus {}^\bullet t) \cup t^\bullet$. We extend our notation and denote by $s[t_1 t_2 \dots t_n\rangle s'$ the fact that there is a sequence of states $s = r_0, r_1, \dots r_n = s'$ such that $r_i[t_{i+1}\rangle r_{i+1}$.

Definition 2. *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.*

Graphically, 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 depicted 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 places in p_3. The transitions that are enabled from the initial state are t_1 and t_2. In the Petri Net of 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, both t_1 and t_3 become independent on both t_2 and t_4.

Definition 3. *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 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 of 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 4. *A* process *of a Petri Net N is a subset of the transitions $\pi \subseteq T$ satisfying that for each $t_1, t_2 \in T$, $(t_1, t_2) \notin I$.*

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. Note that there can be multiple ways to define a set of processes for the same Petri Net.

Definition 5. *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 Π, $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$.*

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: $t_1 t_3 t_2 t_4$ and $t_2 t_4 t_1 t_3$. There are two processes: the *left* process $\pi_l = \{t_1, t_3\}$ and the *right* process $\pi_r = \{t_2, t_4\}$. The neighborhood of process π_l is $\{p_1, p_3, p_5, p_7\}$. The place p_7 acts as a semaphore. It can be captured by the execution of t_1 or of t_2, guaranteeing that $\neg(p_3 \wedge p_4)$ is an invariant of the system.

Definition 6. *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 7. *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 $t \ll r$.*

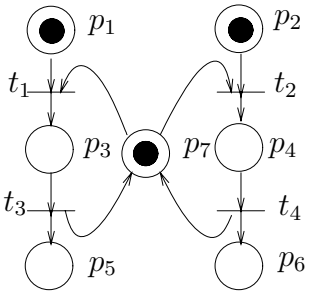

Fig. 1. A (safe) Petri Net

Definition 8. *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 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. $priorE(N, \ll) \subseteq exec(N)$ *and* $reach(N, \ll) \subseteq reach(N)$*.*

The executions of the Petri Net M in Figure 2, when the priorities $t_1 \ll t_4$ and $t_2 \ll t_3$ are not taken into account, include $t_1 t_2 t_3 t_4, t_1 t_3 t_2 t_4, t_2 t_1 t_3 t_4, t_2 t_1 t_4 t_3$, etc. However, when taking the priorities into account, the prioritized executions of M are the same as the executions of the Net N of Figure 1.

Unfortunately, enforcing prioritized executions in a completely distributed way may incur high synchronization overhead [1] or even be impossible. In our example, consider the case where t_1 and t_3 belong to one (left) process π_l, and

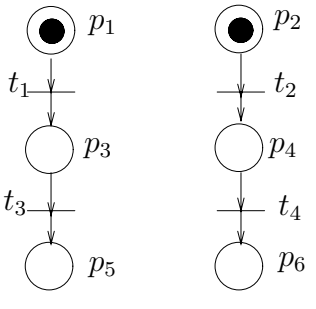

Fig. 2. A Petri Net with Priorities $t_1 \ll t_4, t_2 \ll t_3$

t_2 and t_4 belong to another (right) process π_r, with no interaction between them. Then, the left process π_l, upon having a token in p_1, cannot locally decide whether to execute t_1; the priorities dictate that t_1 can be executed if t_4 is not enabled, since t_1 has a lower priority than t_4. But this information may 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 9. *The* local information *of a set of process Π of a Petri Net N in a state s is $s|_\Pi = s \cap nbg(\Pi)$.*

That is, the local information of Π 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. However, 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. Thus, in the Petri Net in Figure 1, the local information of the left process in a state s consists of restriction of s to the places $\{p_1, p_3, p_5, p_7\}$. In the initial state, these are $\{p_1, p_7\}$.

Our definition of local information is only one among possible definitions that can be used for modeling the part of the state to which the system is aware at any moment. Consider again the Petri Net of 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 may or may not be true that the left process is aware of the value of the semaphore when the token is at place p_5. 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 subtle difference in the definition of local information can be used instead. For simplicity, we will continue with the simpler definition above.

Definition 10. *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.

Lemma 2. *If $t \in \pi$ and $s \equiv_\pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$.*

We cannot 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 $t \notin \pi$ is a maximal enabled transition in s, but in s' this transition is either not enabled or not maximal among the enabled transitions. This can be easily 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 $\{p_1, p_2\}$, t_1 is a maximal priority enabled transition (and so is p_2), while in $\{p_1, p_4\}$, t_1 is not anymore maximal, as we have that $t_1 \ll t_4$, and t_4 is enabled. Since the history of the execution for process π_l is the same in

both of the above states, recording the local history, is sufficient for distinguishing between these two cases.

3 Knowledge Based Approach

Although, 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 may be able to exploit some model checking based analysis of the system.

Our first approach for a local or semi-local decision on firing transitions is based on the *knowledge* of processes [4], 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.

Since the set of states is limited to $S = 2^P$ (and the reachable states are a subset of that), we can write formulas characterizing (i.e., satisfied *exactly* by) the states that satisfy various properties as follows, where the propositions of the formula are the places of the Petri Net.

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: $\varphi_{s|_\Pi}$.

We can perform model checking in order to calculate these formulas, and store them in a compact way, e.g., using BDDs.

Definition 11. *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 $s' \models \psi$.*

That is, the processes Π "jointly know" at state s any property that holds for all the reachable states with the same local information that Π have at s. 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 using Boolean operators with propositions. For a detailed syntactic and semantic description one can refer, e.g., to [4]. We do not define, nor use in this paper the nesting of knowledge operators, e.g., $K_{\Pi_1}(K_{\Pi_2}(\varphi))$, nor the notion of "common" knowledge $C_\Pi \varphi$.

Consider the Petri Net in Figure 3, with priorities $t_3 \ll t_5 \ll t_4 \ll t_7$. This Net is an augmentation of the Net in Figure 1. Process π_l now has the additional transition t_6. Process π_r has the same transitions as before (but a bigger neighborhood, as it now includes p_8). We also have a third process $\pi_3 = \{t_5, t_7\}$. Then at any state s with $s|_{\pi_l} = \{p_3\}$ (take, e.g., $s = \{p_2, p_3, p_8\}$), it is impossible that p_4 has a token, because of the mutual exclusion provided by the place p_7. Thus, $s \models K_{\pi_l} \neg p_4$. On the other hand, it is not the case that

$s \models K_{\pi_l} \neg p_{10}$. This follows from the fact that there are two different states $\{p_2, p_3, p_8\}$ and $\{p_2, p_3, p_8, p_{10}\}$, both of them with $s|_{\pi_l} = \{p_3\}$; in the former state p_{10} does not have a token (and t_5 is disabled), while in the latter state, p_{10} has a token (and t_5 is enabled).

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 which runs on a distributed architecture, one can perform *model checking* in order to calculate whether $s \models K_\pi \psi$. Note that this 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 can 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 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 [12], where a controller is imposed on a system, restricting transitions from being fired at some of the states. We will identify the states of the transformed version N' with the states of the original version N, since the mapping will only *add* some information to the states; information that will not be addressed by our Boolean formulas. In this way, we will be able to compare between the sets of states of the original and transformed version. In particular, the restriction will imply the following:

$$reach(N') \subseteq reach(N). \tag{1}$$

Note that $reach(N)$ are the reachable states when not taking into account the priorities. We will later relate also the executions of these nets. In particular, the supporting process policy can be classified as having a *disjunctive architecture for decentralized control* [13]. Although the details of the transformation are not given here, it 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 previous enabledness condition, at least one of the processes containing it supports it.

Based on Equation (1) and 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 maximally enabled transitions:

φ_1 Each process knows which of its enabled transitions 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.
$\varphi_3 = \varphi_{df} \rightarrow (\bigvee_{\pi \in \mathcal{S}} \bigvee_{t \in \pi} K_\pi \varphi_{\max(t)})$.

Denote the fact that φ is an invariant (i.e., holds in every reachable state) using the usual temporal logic notation $\Box\varphi$ (see [9]). 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 12. *Let $priorS(N, \varphi_i)$ 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 transition. When $\Box\varphi_2$, the processes support at least one of their maximal enabled transition, 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. *$priorS(N, \varphi_1) = priorE(N, \ll)$. 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 maximally enabled transitions.

Advanced note. When $\Box\varphi_1$ holds, our transformation preserves transition fairness (transition justice, respectively) [9]. This is because when a (transformed)

transition satisfies its original enabledness condition infinite often (continuously from some point, respectively), then it is also supported infinitely often (continuously from some point, respectively). For a similar argument, $\Box\varphi_2$ guarantees to preserve process fairness (precess justice, respectively); in this case, not all the transitions enabled according to the original enabledness condition are also supported, but at least one per process. Finally, $\Box\varphi_3$ guarantees that no new deadlocks are added.

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 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 not empty 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:

$$\varphi_{df} \rightarrow \bigvee_{\pi \in \Pi} \varphi_{support(\pi)} \tag{2}$$

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 $\Box\varphi_3$ 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 again as a Petri Net. The construction is simple and the details are not given here for space constraints. The size of the support table is directly proportional to the number of different local information combinations and not to the (sometimes exponentially bigger) size of the state space.

Technically, for Petri Nets, strengthening the enabledness condition means adding input places. Since in Petri Nets the enabledness is a conjunction of places and the added condition for enabledness is disjunctive, we may want to extend the Petri Nets model. Alternatively, we may split transitions, taking into account of the mapping between the new and the original transitions.

Priority Approximation

It is typical that there will be many states where even φ_3 does not hold. That is, processes will not know locally whether one of their transitions is of highest priority.

In the Petri Net of Figure 3, Given the analysis described above, when t_3 is enabled, process π_l knows that t_4 is not enabled. If it also knew that t_5

is not enabled, then, given the above priorities, t_3 would be maximal enabled transitions and will be supported by π_l. However, as shown above, there are two different states were t_3 is enabled, in one of which t_5 is not enabled (this is the first time when t_3 is enabled), and in the other t_5 is enabled (this is the second time). Thus, t_3 cannot be supported by π_l in both states. In the state $\{p_2, p_3, p_8\}$, no transition is supported.

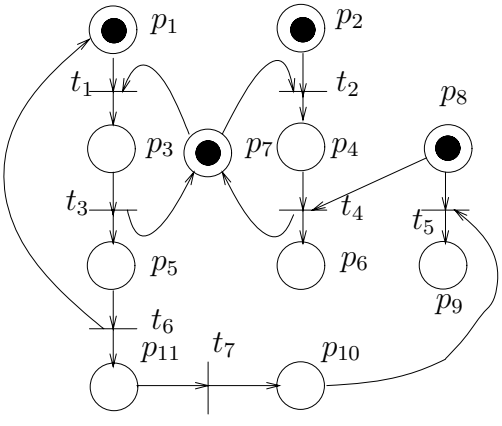

Fig. 3. Petri Net with priorities $t_3 \ll t_5 \ll t_4 \ll t_7$

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. Under the support policy, in state s, no transition will be supported, hence none will be fired, resulting in a deadlock. A *pessimistic approach* to fix this situation 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. Add t to that entry is sufficient for checking enabledness). Update the formula (2), by adding the disjunct $\varphi_{s|_\pi}$ to $\varphi_{support(\pi)}$. Then recheck Formula (2). Repeat adding transitions to empty entries in the support table until (2) 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 (2) 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 controller 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 controllers may reduce concurrency. However, this is not a problem if the controlled processes are threads, residing anyway in the same processor. It is not clear apriory on which sets of processes we want to put a controller. 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 [10] 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, allowing to decide on executing a transition even when it was not possible under the previous knowledge definition. For lack of space, we describe the solution based on knowledge with perfect recall briefly and somewhat informally.

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 the $ngb(\pi)$. By definition 2, these are the transition that are dependent on some transition in π (this includes all the transitions of 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 [10], and our algorithm can be seen as a simplified version of it. For each process π, we construct an automaton representing the entire state space of the system. We keep track of all the possible (global) states Γ that correspond to having some local history. A move from Γ to Γ', corresponds to the execution of any transition t that is dependent on some transition in π. To construct Γ' from Γ, we first take all the states reached from states in Γ by executing t, when t is enabled. Then, we add to this set the states that are obtained by executing any sequence of transitions that are independent of all those in π. The initial state of this automaton contains any state obtained from s_0 by executing a finite number of 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 subset construction, i.e., the determinization, of the above automaton. At runtime, the execution of each transition dependent on π 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 a transition $t \in \pi$ is maximally enabled (checking this for the states in Γ was done 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 state space (reflecting 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. The local histories of the transformed net is a subset of the local histories of the original, priorityless net, providing a result, similar to Theorem 1, that is expressed in terms of knowledge with perfect recall.

Returning to the example in Figure 3, the knowledge with perfect recall can separate the first time when t_3 is enabled from the second time. On the first time, t_5 is not enabled, hence π_l knows that t_3 is a maximal enabled transition, supporting it. On the second time, t_5 is enabled (or first t_7 is enabled and then, due to its priority, will execute and t_5 will become enabled), and π_l does not support t_3. Thus, given knowledge with perfect recall, there is no need to group processes together in order to obtain the knowledge about the maximalty of t_5.

5 Implementation and Experimental Results

We have implemented a prototype engine which performs model checking, creates the process support tables, determines whether the invariant φ_3 holds for the distributed processes, and finally allows for execution of the processes based on their support tables.

For experiments, we used the following example: a train station uses a network of tracks to divert the trains from the point where they enter the station and into an empty platform (we assume that is no preassignment of trains to segments). There are some trains and some track segments. Some of the segments are initial, i.e., where the train first appears when it wishes to enter the station, and some of them final, consisting of actual platforms. When moving from segment r to segment r', a controller for segment r' checks that the segment is empty, and then allows the train to enter. The highest priority trains are TGV trains (Train Grande Vitesse, in our example there is a single TGV train), then there are local trains (two local in our example), and, with least priority, freight trains (a single freight train). There are trains of three speeds. Our Petri Net has the following places:

- $p_{empty-r}$ has a token when the track segment r is empty.
- $p_{k@r}$ train k is at segment r.
- $p_{outside-k}$ train k is not in the vicinity of the station.

There are three types of transitions:

- Progress by train k from segment r to segment r'. Such a transition $t_{r \to r',k}$, has inputs $p_{k@r}$ and $p_{empty-r'}$, and outputs $p_{k@r'}$ and $p_{empty-r}$.
- A new train k is entering the system at segment r. Such a transition$t_{k\ enters-r}$ has inputs $p_{outside-k}$ and $p_{empty-r}$ and one output $p_{k@r}$.
- A train k leaves the station after being on segment (track) r. the input is $p_{k@r}$ and the outputs are $p_{outside-k}$ and $p_{empty-r}$.

Each process π can be a collection of transitions that are related to the same segment. Thus, a transition $t_{r \to r',k}$ is shared between the processes r and r'. Priorities can be assigned according to the train speed (transitions of TGV trains higher than transitions of local trains, and these, in turn have higher transitions than those involving freight trains.

The actual segments structure we used in the experiments is shown in Figure 4

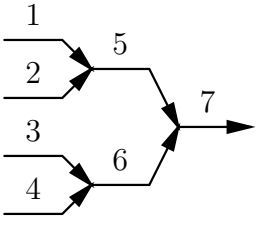

Fig. 4. The actual segments used in the experiments

In order to measure the effect of the approximations, we have considered a penalty, which measures the deviation of the measured behavior from an ideal prioritized behavior. This measure sums up the deviation from selecting a maximal enabled transition at each execution step. If a maximal transition is indeed executed, we add 0. If a second-best transition is selected, e.g., involving a local train when a TGV train is also enabled, we add 1. If all transitions for all types of trains are enabled and a fright train (i.e., with lowest priority) moves, then we add 2. All results are based on random traces of length 100,000 steps (i.e., transitions), and taking the average 10 random traces. We have measured the penalty for the different distributions of the processes. For a worst case estimate of the penalty, we executed the system without considering the priorities. That gave a penalty of 34332, which is considerably higher than the penalties measured when applying approximations, as shown in the results below.

The results obtained for different distribution of the processes are shown in the following table. The global state space of the system consists of 1961 states. This was run on a Pentium-4 PC (Dell GX520) with 1GB RAM, running GNU-linux.

process groups	table size	% table empty	φ_3 holds	penalty
1	1961	0	yes	0
2	1118	0	yes	0
3	219	5.5	no	387
7	126	14.4	no	3891

The "process groups" tells us how many processes are grouped together for calculating the knowledge. When there is only one group, all the processes are grouped together, so the global state is known to all. With 7 groups, each group is a singleton, containing one process; thus the knowledge of multiple processes is not combined. The "table size" shows the number of entries in the knowledge table of the processes. In case of a single process (all transitions in one process), the number of entries in the knowledge table is the same as the total number of states in the system, and the process has complete information with respect to the maximal transitions to be taken from a state (no empty entries). As a result, φ_3 holds, and there is no penalty.

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. The result of the model checking is integrated into the program via a data structure that helps the processes to select prioritized transition at each point. Thus, 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 situations where a process, locally, does not have enough information about having a maximal priority transition. In this case, 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.

We experimented with an example of a train station, including trains entering the system via a net of segments that divert the trains until they arrive at an available platform.

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. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
2. Buckley, G.N., Silberschatz, A.: An Effective Implementation for the Generalized Input-Output Construct of CSP. ACM Transactions on Programming Language and Systems 5, 223–235 (1983)
3. Emerson, E.A., Clarke, E.M.: Characterizing Correctness Properties of Parallel Programs using Fixpoints. In: ICALP 1980. LNCS, vol. 85, pp. 169–181. Springer, Heidelberg (1980)
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
5. Francez, N., Rodeh, M.: A Distributed Abstract Data Type Implemented by a Probabilistic Communication Scheme. In: 21st Annual Symposium on Foundations of Computer Science (FOCS), Syracuse, New York, pp. 373–379 (1980)
6. Halpern, J.Y., Zuck, L.D.: 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)
7. Hoare, C.A.R.: Communicating Sequential Processes. Communication of the ACM 21, 666–677 (1978)
8. Gößler, G., Sifakis, J.: Priority Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 443–466. Springer, Heidelberg (2004)
9. Manna, Z., Pnueli, A.: How to Cook a Temporal Proof System for Your Pet Language. In: POPL 1983, Austin, TX, pp. 141–154 (1983)
10. van der Meyden, R.: Common Knowledge and Update in Finite Environment. Information and Computation 140, 115–157 (1980)
11. Quielle, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: 5th International Symposium on Programming, pp. 337–350 (1981)
12. 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)
13. 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)