Monitoring Distributed Systems Using Knowledge

Susanne Graf, Doron Peled, Sophie Quinton

Monitoring Distributed Systems using Knowledge

Susanne Graf[1] and Doron Peled[2] and Sophie Quinton[3]

[1] VERIMAG, Centre Equation, Avenue de Vignate, 38610 Gières, France
[2]Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
[3] Institute of Computer and Network Engineering, 38106 Braunschweig, Germany

Abstract. In this paper, we use knowledge-based control theory to monitor global properties in a distributed system. We control the system to enforce that if a given global property is violated, at least one process knows this fact, and therefore may report it. Our approach uses knowledge properties that are precalculated based on model checking. As local knowledge is not always sufficient to monitor a global property in a concurrent system, we allow adding temporary synchronizations between two or more processes to achieve sufficient knowledge. Since synchronizations are expensive, we aim at minimizing their number using the knowledge analysis.

1 Introduction

The goal of this paper is to transform a distributed system such that it can detect and report violations of invariants. Such properties can be described by a predicate ψ on global states in distributed systems. This may express for example that the overall power in the system is below a certain threshold. When a violation is detected, some activity to adjust the situation may be triggered. There is no global observer that can decide whether a global state violating the given property ψ has been reached. On the other hand, the processes may not have locally enough information to decide this and thus need sometimes to communicate with each other to obtain more information.

Our solution for controlling a system to detect global failure is based on precalculating *knowledge* properties of the distributed system [5, 12]. We first calculate in which local states a process has enough information to identify that ψ is violated: in each reachable global state in which ψ becomes false, at least one process must detect this situation. This process may then react, e.g. by informing the other processes or by launching some repair action. Furthermore, we do not want false alarms. Due to the distribute nature of the system, there can be states where firing a transition t would lead to a state in which no process knows (alone) whether ψ has been violated. In that case, additional knowledge is necessary to fire t. We achieve this by adding temporary synchronizations that allow combining the knowledge of a set of processes. To realize at runtime the temporary synchronizations needed to achieve such combined knowledge, as

precalculated using the knowledge analysis at compile time, a synchronization algorithm (such as α-core [14]) is used. To reduce the communication overhead, it is desirable to minimize both the number of additional synchronizations and the number of participants in each synchronization.

This work is related to the knowledge based control method of [2, 7]. There, knowledge obtained by model checking is used to control the system in order to enforce some property, which may be a state invariant ψ. Here, we want to control the system to enforce that there is always at least one process with knowledge to detect a violation of such a property as soon as it happens. Controlling the system to *avoid* property violation is a different task from controlling the system to *detect* it. In some cases, controlling for avoidance may lead to restricting the system behavior much more severely than controlling for detection. Note also that for an application such as runtime verification, prevention is not needed while detection is required (e.g., it is acceptable that the temperature raises above its maximal expected level, but whenever this happens, some specific intervention is required).

Monitoring is a simpler task than controlling as it is *nonblocking*. Attempting to enforce a property ψ may result in being blocked in a state where any continuation will violate ψ. This may require strengthening ψ in order not to reach such states, through an expensive global state search, which may consequently imply a severe reduction in nondeterministic choice. This situation does not happen in monitoring; at worst, this may lead to a synchronization between processes.

As an alternative to monitoring one may use *snapshot algorithms* such as those of Chandy and Lamport [4] or Apt and Francez [1]. However, snapshot algorithms only report about some sampled global states. If the property ψ is not stable, that is, if $\psi \Rightarrow \Box\psi$ is not guaranteed, then the fact that ψ has been true at some global state may go undetected.

2 Preliminaries

We represent distributed systems as Petri nets, but the method and algorithms developed here can equally apply to other models, e.g., communicating automata or transition systems.

Definition 1. *A (safe)* 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).*

Definition 2. *For a transition* $t \in T$*, we define the set of* input places $^\bullet t$ *as* $\{p \in P | (p, t) \in E\}$*, and the set of* output places $t^\bullet$ *as* $\{p \in P | (t, p) \in E\}$*.*

Definition 3. *A transition* t *is* enabled *in a state* s *if* $^\bullet t \subseteq s$ *and* $(t^\bullet \setminus {}^\bullet t) \cap s = \emptyset$*. 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 4. *The* execution *(*firing*) of a transition t leads from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled in s and $s' = (s \setminus {}^\bullet t) \cup t^\bullet$.*

We use the Petri net of Figure 1 as a running example. As usual, transitions are represented as blocks, places as circles, and the relation E as arrows from transitions to places and from places to transitions. The Petri net of Figure 1 has places named $p_1, p_2, \ldots, p_8$ and transitions $a, b, \ldots, e$. We represent a state s by putting *tokens* inside the places of s. In the example of Figure 1, the depicted initial state s_0 is $\{p_1, p_4\}$. The transitions enabled in s_0 are a and b. Firing a from s_0 means removing the token from p_1 and adding one to p_3.

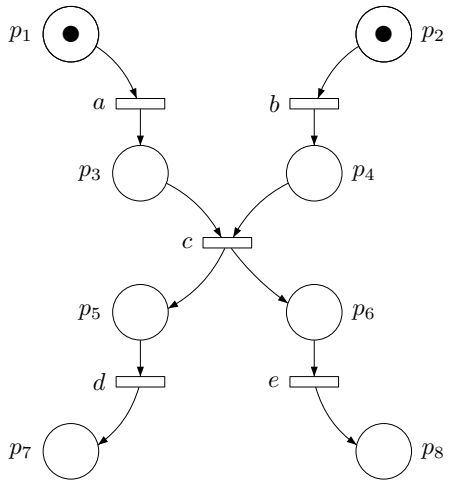

Fig. 1. A Petri net with initial state $\{p_1, p_2\}$

Definition 5. *An* execution *is a maximal (i.e., it cannot be extended) alternating sequence of states and transitions $s_0 t_1 s_1 t_2 s_2 \ldots$ with s_0 the initial state of the Petri net, such that for each state s_i in the sequence with $i > 0$, it holds that $s_{i-1}[t_i\rangle s_i$.*

We denote the set of executions of a Petri net N by $exec(N)$. The set of prefixes of the executions in a set X is denoted by $pref(X)$. A state is *reachable* in N if it appears in at least one execution of N. We denote the set of reachable states of N by $reach(N)$.

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

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

We assume a given set of processes Π_N that covers all the transitions of N, i.e., $\bigcup_{\pi \in \Pi_N} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. The set of processes to which t belongs is denoted $proc(t)$.

In this section, all the notions and notations related to processes extend naturally to sets of processes. Thus, we usually provide definitions directly for sets of processes. Then, when a formula refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π instead. The neighborhood of a process π describes the places of the system whose state π can observe.

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

Definition 8. *The* local state *of a set of processes* Π *in a (global) state* $s \in S$ *is defined as* $s|_\Pi = s \cap ngb(\Pi)$*. A local state* s_Π *of* Π *is* part of *a global state* $s \in S$ *if and only if* $s|_\Pi = s_\Pi$*.*

That is, the local state of a process π in a global state s consists of the restriction of s to the neighborhood of π. It describes what π can see of s based on its limited view. In particular, according to this definition, any process π can see whether one of its transitions is enabled. The local state of a set of processes Π containing more than one process is called a *joint* local state.

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

Thus, if $t \in \bigcup_{\pi \in \Pi} \pi$ and $s \equiv_\Pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$.

Figure 2 represents one possible distribution of our running example. We represent processes by drawing dashed lines between them. Here, the left process π_l consists of transitions a, c and d while the right process π_r consists of transitions b, c and e. The neighborhood of π_l contains all the places of the Petri net except p_2 and p_8. The local state $s_0|_{\pi_l}$, part of the initial state $s_0 = \{p_1, p_2\}$ is $\{p_1\}$. Note that the local state $s|_{\pi_l}$, part of $s = \{p_1, p_4\}$ is also $\{p_1\}$, hence $s_0 \equiv_{\pi_l} s$.

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

Our approach for achieving a local or semi-local decision on which transitions may be fired, while preserving observability, is based on the *knowledge* of processes [5] or sets of processes.

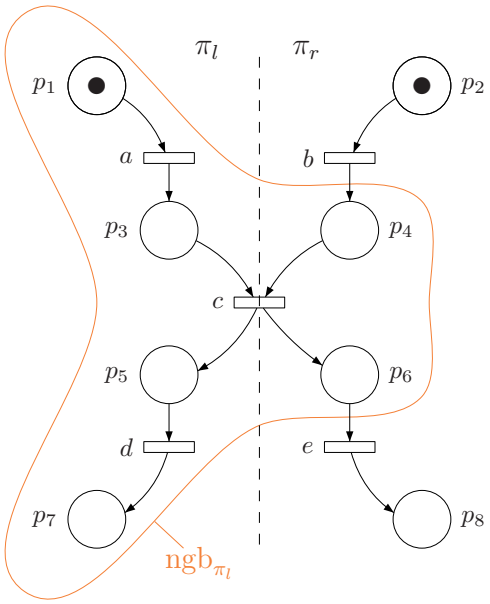

Fig. 2. A distributed Petri net with two processes π_l and π_r

Definition 10. *Given a set of processes* Π *and a property* φ*, we define the property* $K_\Pi\varphi$ *as the set of global states* s *such that for each* reachable s' *with* $s \equiv_\Pi s'$*,* $s' \models \varphi$*. Whenever* $s \models K_\Pi\varphi$ *for some state* s*, we say that* Π *(jointly)* knows φ in s*.*

We easily obtain that if $s \models K_\Pi\varphi$ and $s \equiv_\Pi s'$, then $s' \models K_\Pi\varphi$. Hence we can write $s|_\Pi \models K_\Pi\varphi$ rather than $s \models K_\Pi\varphi$ to emphasize that this knowledge property is calculated based on the local state of Π. Given a Petri net and a property φ, one can perform model checking in order to decide whether $s \models K_\Pi\varphi$ for some state s. If Π contains more than one process, we call it *joint* knowledge.

Note that when a process π needs to know and distinguish whether η or μ holds, we write $K_\pi\eta \vee K_\pi\mu$. When we do not need to distinguish between these cases but only need to know whether at least one of them holds, we use the weaker $K_\pi(\eta \vee \mu)$.

3 Knowledge Properties for Monitoring

Our goal is to control the system, i.e., restrict its possible choice of firing transitions, in order to enforce that if a given property ψ becomes false, at least one process knows it. This should interfere minimally with the execution of the system in order to monitor when ψ is violated. To detect the occurrence of a failure, we need the following notion of a "weakest precondition":

Definition 11. *For a given property φ, $wp_t(\varphi)$ is the property such that for any state s, $s \models wp_t(\varphi)$ if and only if $s[t\rangle$ and $s' \models \varphi$ where $s[t\rangle s'$.*

Remember that in a Petri net, there is exactly one state s' such that $s[t\rangle s'$ for a given state s and transition t.

We take into account, with an increasing degree of complication:

- Whether it is allowed to report the same violation of ψ multiple times.
- Whether there exists one or several types of property violation. That is, $\neg\psi$ may be of the form $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$, where each φ_i represents a certain kind of failure to satisfy ψ. Then, whenever ψ is violated, we may need to identify and report which failure occurred.
- Whether a single transition may cause several failures to occur at the same time (and each one of them needs to be identified).

First, we assume that ψ consists of only one type of failure, and furthermore, that there is no harm in reporting it several times; it is the responsibility of the recovery algorithm to take care of resolving the situation and ignore duplicate reports. For a transition $t \in T$ and a set of processes $\Pi \subseteq proc(t)$, we define a property $\delta(\Pi, t)$ as follows: if t is fired and ψ is falsified by the execution of t, Π will jointly know it. Formally:

$$\delta_1(\Pi, t) = K_\Pi wp_t(\neg\psi) \vee K_\Pi(\neg\psi \vee wp_t(\psi))$$

In other words, $\delta_1(\Pi, t)$ holds if and only if the processes in Π either jointly know that after firing t property ψ will be violated; or they know that firing t cannot make ψ *become* false: either it is already false or it will be true after firing t. Note that the knowledge operators separate the case where $\neg\psi$ will hold in the next state from the other two cases. There is no need to distinguish between the latter two cases, hence we could use the weaker requirement, where both of them appear inside a single knowledge operator $K_\Pi(\neg\psi \vee wp_t(\psi))$.

Now, suppose that we should not report that ψ is violated again, when it was already violated before the execution of t, and therefore has already been or will be reported. This requires strengthening the knowledge:

$$\delta_2(\Pi, t) = K_\Pi(\psi \wedge wp_t(\neg\psi)) \vee K_\Pi(\neg\psi \vee wp_t(\psi))$$

For the case where failure of ψ means one out of several failures $\varphi_1, \ldots, \varphi_n$, but firing one transition cannot cause more than only one particular failure, we need to consider stronger knowledge:

$$\delta_3(\Pi, t) = \bigvee_{i \leq n} K_\Pi(\neg\varphi_i \wedge wp_t(\varphi_i)) \vee \bigwedge_{i \leq n} K_\Pi(\varphi_i \vee wp_t(\neg\varphi_i))$$

Finally, if one single transition may cause multiple failures, we need even stronger knowledge:

$$\delta_4(\Pi, t) = \bigwedge_{i \leq n} (K_\Pi(\neg\varphi_i \wedge wp_t(\varphi_i)) \vee K_\Pi(\varphi_i \vee wp_t(\neg\varphi_i)))$$

Note that in $\delta_j(\Pi, t)$, for $1 \leq j \leq 4$, the left disjunct, when holding, is responsible for identifying the occurrence of the failure, and also for $j \in \{3, 4\}$, identifying its type. In the following, $\delta(\Pi, t)$ stands for $\delta_j(\Pi, t)$, where $1 \leq j \leq 4$.

Definition 12. *A* knowledgeable step *for a set of processes $\Pi \subseteq \Pi_N$ is a pair $(s|_\Pi, t)$ such that $s|_\Pi \models \delta(\Pi, t)$ and there is at least one process $\pi \in \Pi$ with $t \in \pi$.*

Note that if all processes synchronize at every step, a violation of Ψ can always be detected as soon as it happens. Of course, the additional synchronizations required to achieve joint knowledge induce some communication overhead, which we have to minimize. We explain how we do this in the next section.

4 Building the Knowledge Table

We use model checking to identify knowledgeable steps following a method similar to [7]. The basic principle of our monitoring policy is the following: a transition t may be fired in a state s if and only if, in addition to its original enabledness condition, $(s|_\pi, t)$ is a knowledgeable step for at least one process π containing t. However, there may be some state in which no individual process has enough knowledge to take a knowledgeable step. In that case, we consider knowledgeable steps for pairs of processes, then triples etc. until we can prove that no deadlock is introduced by the monitoring policy.

The monitoring policy is based on a *knowledge table* Δ which indicates, for a process or a set of processes Π in a given reachable (joint) local state $s|_\Pi$, whether there exists a knowledgeable step $(s|_\Pi, t)$, and then which transition t may thus be safely fired. When building such a table, two issues must be considered: first, the monitoring policy should not introduce deadlocks with respect to the original Petri net N. This means that we have to check that for every reachable non-deadlock global state of N, there is at least one corresponding knowledgeable step in Δ. Second, achieving joint knowledge requires additional synchronization, which induces some communication overhead, as will be explained in the next section. Therefore we must add as few knowledgeable steps involving several processes as possible.

Definition 13. *For a given Petri net N, a* knowledge table *Δ is a set of knowledgeable steps.*

To avoid introduce new deadlocks, we require that the table Δ contains enough joint local states to cover all reachable global states. This is done by requiring the following invariant.

Definition 14. *A knowledge table Δ is an* invariant *if for each reachable non-deadlock state s of N, there is at least one (joint) local state in Δ that is part of s.*

Given a Petri net N and a property ψ, the corresponding knowledge table Δ is built iteratively as follows:

The first iteration includes in Δ, for every process $\pi \in \Pi_N$, all knowledgeable steps $(s|_\pi, t)$ where $s|_\pi$ is a *reachable* local state of π, i.e., it is part of some reachable global state of N. If Δ is an invariant after the first iteration, then taking only knowledgeable steps appearing in Δ does not introduce deadlocks. If Δ is not an invariant, we proceed to a second iteration. Let U be the set of reachable non-deadlock global states s of N for which there is no (joint) local state in Δ that is part of s.

In a second iteration, we add to Δ knowledgeable steps $(s|_{\{\pi,\rho\}}, t)$ such that $s|_{\{\pi,\rho\}}$ is part of some global state in U. For a given local state $s|_{\{\pi,\rho\}}$, all corresponding knowledgeable steps are added together to the knowledge table. The second iteration terminates as soon as Δ becomes an invariant or if all knowledgeable steps for pairs of processes not strictly including knowledgeable steps consisting of single processes have been added to the table.

If Δ is still not an invariant, then we perform further iterations where we consider knowledgeable steps for triples of processes, and so forth. Eventually, Δ becomes an invariant, in the worst case by adding knowledgeable steps involving all processes.

5 Monitoring using a Knowledge Table

As in [7], we use the knowledge table Δ to control (restrict) the executions of N so as to allow only knowledgeable steps. Formally, this can be represented as an extended Petri net [6, 8] N^Δ where processes may have local variables, and transitions have an enabling condition and a data transformation.

Definition 15. *An* extended *Petri net N' consists of (1) a Petri net N (2) a finite set of variables V with given initial values and (3) for each transition $t \in T$, an enabling condition en_t and a transformation predicate f_t on variables in V. In order to fire t, en_t must hold in addition to the usual Petri net enabling condition on the input and output places of t. When t is executed, in addition to the usual changes to the tokens, the variables in V are updated according to f_t.*

A Petri net N' *extends* N if N' is an extended Petri net obtained from N according to Definition 15. The comparison between the original Petri net N and N' extending it is based only on places and transitions. That is, we project out the additional variables.

Lemma 1. *For a given Petri net N and an extension N' of N, we have: $exec(N') \subseteq pref(exec(N))$.*

Proof. The extended Petri net N' strengthens the enabling conditions, thus it can only restrict the executions. However, these restrictions may result in new deadlocks. □

Furthermore, we have the following monotonicity property.

Theorem 1. *Let N be a Petri net and N' an extension of N according to Definition 15 and φ a state predicate for N. If $s \models K_\pi \varphi$ in N, then $s \models K_\pi \varphi$ also in N'.*

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

The latter lemma and theorem show that the additional variables used to extend a Petri net N define a controller for N.

Definition 16. *Given a Petri net N and a property Ψ, from which a knowledge table Δ has been precalculated, the extended Petri net N^Δ is obtained as follows:*

- *Encode in a set of Boolean variables en_t^Π for $\Pi \subseteq \Pi_N$ and $t \in T$ the knowledge properties calculated in Δ such that en_t^Π is true if and only if $(s|_\Pi, t)$ is a knowledgeable step, where $s|_\Pi$ is the current local state of Π.*
- *Encode in each f_t the update of variables as t is fired and local states modified.*
- *Define each en_t as $\bigvee_{\Pi \subseteq \Pi_N} en_t^\Pi$. That is, t can be fired if at least one set of processes knows that it is part of a knowledgeable step*[1].

In practice, we obtain joint knowledge by adding synchronizations amongst the processes involved. Such synchronizations are achieved by using an algorithm like α-core [14], which allows processes to notify, using asynchronous message passing, a set of coordinators about their wish to be involved in a joint action. This is encoded into the extended Petri net. Once a coordinator has been notified by all the participants in the synchronization it is in charge of, it checks whether some conflicting synchronization is already under way (a process may have notified several coordinators but may not be part of several synchronizations at the same time). If this is not the case, the synchronization takes place. The correctness of the algorithm guarantees the atomic-like behavior of the coordination process, allowing us to reason at a higher level of abstraction where we treat the synchronizations provided by α-core (or any similar algorithm) as transitions that are joint between several participating processes.

Specifically, each process π of N is equipped with a local table Δ_π containing the knowledgeable steps $(s|_\pi, t)$ that appear in the knowledge table Δ and the knowledgeable steps $(s|_\Pi, t)$ such that $\pi \in \Pi$. Before firing a transition in a given local state $s|_\pi$, process π consults its local table Δ_π. It Δ_π contains a knowledgeable step $(s|_\pi, t)$, then π notifies α-core about its wish to *initiate* t, so that the coordinator algorithm will handle potential conflicts with other knowledgeable steps. If Δ_π contains a knowledgeable step $(s|_\Pi, t)$ such that $s|_\pi$ is part of $s|_\Pi$, then π notifies α-core about its wish to achieve joint knowledge through synchronization with the other processes in Π. If the synchronization takes place, then any process in Π and containing t may initiate it. The processes in Π remain synchronized until t has been fired or disabled by some other knowledgeable step.

[1] Note that this condition comes in conjunction with the usual Petri net firing rule based on the places in the neighborhood of t, as in Definition 3

6 Implementation and Experimental Results

In this section we apply our approach to a concrete example that was implemented in a modified version of the prototype presented in [7]. We have implemented properties δ_1 to δ_3 as defined in Section 3. Property δ_4 is not relevant here because a single transition may never cause multiple failures. The prototype implementation computes the knowledge table Δ based on the local knowledge of processes, as described in Section 4.

The example presented here is a Petri net representing the following scenario: trains enter and exit a train station such as the one represented in Figure 3 (trains that are outside the train station are not represented), evolving between track segments (numbered from 1 to 12). A track segment can accept at most one train at a time, therefore there must be some mechanism to detect and resolve conflicts amongst trains trying to access the same track segment. Trains enter and exit the station on the left, i.e. entry segments are tracks 1 to 4. After entering the station, a train moves from left to right until it reaches one of the platforms (tracks 9 to 12); then it starts moving from right to left until it exits the station on one of the entry segment. A train leaving the station on a given track segment may reenter the station only on this segment.

We monitor a property Ψ which we call absence of *partial gridlock*. A partial gridlock is a situation where some trains are blocking each other at a given intersection. These trains cannot follow their normal schedule and must inform a supervisor process that initiates some specific repair action, e.g. requesting some trains to backtrack. For each intersection where n track segments meet, a partial gridlock is reached when there is one train on each of these segments that is moving toward the intersection. A global state satisfies Ψ if and only if it does not contain any partial gridlock.

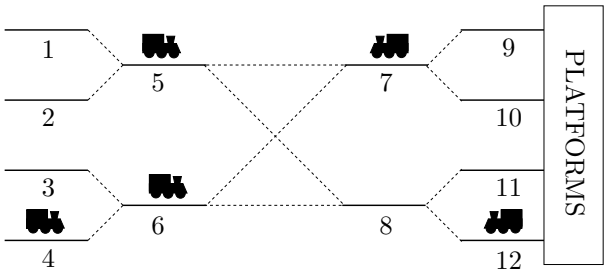

Fig. 3. Train station TS_1

Transitions describe how trains can enter, exit and move within the station. Processes correspond to track segments. That is, the process associated with a segment σ, denoted π_σ, consists of all the transitions involving σ (a train arriving

on σ or leaving it). In particular, this means that transitions corresponding to a train moving from one segment σ_1 to another segment σ_2 belong to π_{σ_1} and π_{σ_2} while transitions representing a train entering or exiting the train station belong to exactly one process namely the entry segment on which the train is entering or leaving. Furthermore, according to the definition of neighborhood, a process π_σ knows if there is a train on segment σ and also on the *neighbors* of σ, i.e., the track segments from which a train may reach σ.

Example 1. Let us first focus on train station TS_1 of Figure 3. A train entering on segment 1 can progress to segment 5 and then segment 7 or 8. From there, it can reach either platform 9 or 10, or platform 11 or 12, respectively. Then, it moves from right to left, until it exits the train station through one of the segments 1 to 4.

Two patterns of partial gridlocks are represented in Figure 4. All possible partial gridlocks of train station TS_1 can be represented by a set of similar patterns. If we consider 6 trains, there are 820,368 reachable global states in this example, of which 11,830 contain a partial gridlock and 48 are global deadlock states.

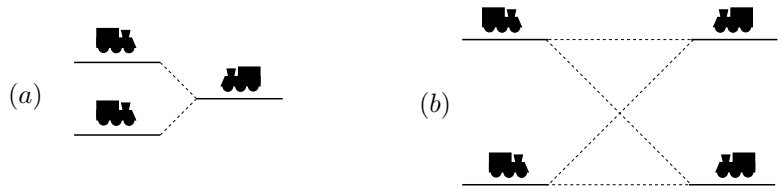

Fig. 4. Two possible partial gridlocks, i.e., violations of Ψ

Interestingly, no additional synchronization is needed to enforce that only knowledgeable steps are taken in this example, independently of the choice of the knowledge property δ_i. The reason for this is twofold. First, partial gridlock (a) of Figure 4 is always detected by the track segment on the right, which knows that there is a train on all three segments. Second, partial gridlock (b) of Figure 4 is not reachable, as we further explain. No additional synchronization is needed. Intuitively, partial gridlock (b) is not reachable for the following reason: whenever train station TS_1 reaches a global state similar to that represented in Figure 3, segment 8 cannot be entered by the train on segment 12. Indeed, this move is a knowledgeable step neither for process π_{12} nor for process π_8, since none of them knows whether this move would introduce a partial gridlock or not. Remember that π_8 does not know whether there is a train or not on segment 7. However, moves from the trains on segments 5 and 6 to 8 are knowledgeable steps for π_8, as π_8 knows they do not introduce any partial gridlock.

Table 1 presents some results about the influence of our monitoring policy on the behavior of the system. The notation NR indicates that some information is irrelevant for the uncontrolled system. A first observation is that the behavior

system controlled according to	δ_1	δ_2	δ_3	uncontrolled
states actually reachable	820,096	820,026	820,096	820,368
transitions inhibited	46	99	46	NR (none)
transitions supported	6,856	7,656	6,913	NR (all)
steps to first partial gridlock	117	454	121	56

Table 1. Results for 1000 executions of 10,000 steps

of the system is not dramatically affected by the monitoring policy: whatever the knowledge property δ_j used to define the monitoring policy, very few global states become unreachable compared to the uncontrolled system. Also, the ratio of supported transitions to inhibited transitions is around 150:1 for δ_1 and δ_3, and 75:1 for δ_2. Finally, the fact that a partial gridlock is reached on average after 56 steps in the uncontrolled system shows that monitoring the system does not drive it, in this example, into states in which the property under study Ψ is violated.

Furthermore, note that δ_1 and δ_3 yield similar results in contrast with δ_2. This is due to the fact that every process knows exactly whether it is creating a given partial gridlock, but it does not always know whether it is creating the first partial gridlock in the system. As a result, the ratio of transitions *inhibited*, that is, transitions enabled but not part of a knowledgeable step, is higher when the system is monitored according to δ_2 than when it is monitored according to δ_1 and δ_3.

Example 2. Figure 5 shows another train station TS_2 and a global reachable non-deadlock state in which there is no knowledgeable step for a single process. As a result, an additional synchronization must be added.

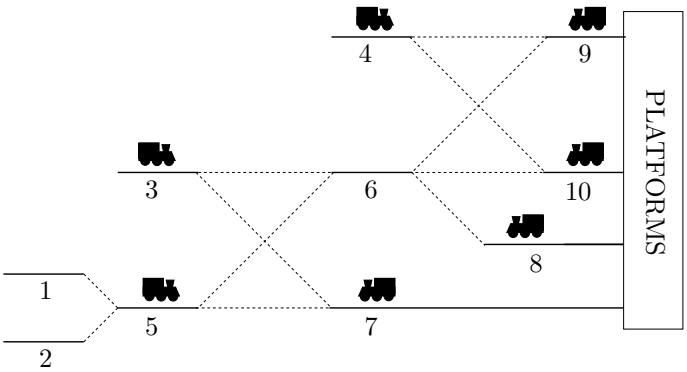

Fig. 5. Train station TS_2

Our experiments on train station TS_2 consider 7 trains. There are 1,173,822 reachable global states in the corresponding Petri net, among which 9,302 contain a partial gridlock. Besides, there are 27 global deadlock states. However, there is only one global reachable non-deadlock state for which there is no corresponding knowledgeable step for a process alone. This state, which we denote s_{dl}, is represented in Figure 5. A synchronization between processes π_3 and π_6 is sufficient to ensure that no deadlock is introduced by the monitoring policy, as there are three transitions t such that (s_{dl}, t) is a knowledgeable step for $\{\pi_3, \pi_6\}$.

We have also performed some experiments to evaluate the number of synchronizations added by the monitoring policy as well as the number of transitions inhibited at runtime. All δ_j yield similar results, so we present them together. Interestingly, the number of synchronizations due to the monitoring policy is very low and although the only progress property that we preserve is deadlock-freedom, few transitions are inhibited in this example. Besides, only 1,972 global states are not actually reachable in the controlled system. Thus, in this example, controlling the system in order to preserve the knowledge about absence of partial gridlock hardly induces any communication overhead and does not alter significantly the behavior of the global system.

system controlled according to	δ_j for $j \in \{1, 2, 3\}$
states actually reachable	1,171,850
synchronizations	0.01
transitions inhibited	62
transitions supported	18,456

Table 2. Results for 100 executions of 10,000 steps

Note that it is sufficient here to add one temporary synchronization between two processes in order to detect that a partial gridlock occurred, whereas knowledge about *absence* of a partial gridlock would require an almost global synchronization. Besides, controlling the system in order to enforce absence of partial gridlocks (instead of monitoring it) would require that processes avoid states in which every possible move leads (inevitably) to a partial gridlock. That is, it requires look-ahead.

7 Conclusion

In this paper, we have proposed an alternative approach to distributed runtime monitoring that guarantees by a combination of monitoring and control (property enforcement) that the fact that some property ψ becomes false is always detected instantaneously when the corresponding transition is fired. Furthermore, there are no "false alarms", that is whenever ψ is detected, it does hold

at least in the state reached at that instant. In other words, we use control as introduced in [2, 7, 3] to enforce a strong form of local monitorability of ψ, rather than to enforce ψ itself.

We use synchronizations amongst a set of processes, which are realized by a coordinator algorithm such as α-core [14], in order to reliably detect that: either ψ will be false after the transition or the transition does not change the status of ψ. We use model checking to calculate whether (joint) local states have the required knowledge to fire a given transition. We control the system by allowing only such *knowledgeable* steps and we add as many synchronizations as necessary to enforce absence of global deadlocks which do not already appear in the original system.

We have applied this approach to a nontrivial example, showing the interest of enforcing some knowledge about the property instead of the property itself.

References

1. K. Apt, N. Francez, Modeling the Distributed Termination Convention of CSP, ACM Trans. Program. Lang. Syst., 6(3): 370–379, 1984.
2. 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.
3. S. Bensalem, M. Bozga, S. Graf, D. Peled, S. Quinton, Methods for Knowledge Based Controlling of Distributed Systems, ATVA 2010, Singapore, LNCS 6252, Springer, 52–66.
4. K. Chandy, L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Trans. Comput. Syst., 3(1): 63–75, 1985.
5. R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, Cambridge MA, 1995.
6. H. J. Genrich, K. Lautenbauch, System Modeling with High-level Petri Nets, Theoretical Computer Science 13, 1981, 109-135.
7. S. Graf, D. Peled, S. Quinton, Achieving Distributed Control through Model Checking, CAV 2010, Edinburgh, UK, LNCS 6174, Springer, 396–409.
8. R.M. Keller, Formal Verification of Parallel Programs, Communications of the ACM, 19, 1976, 371–384.
9. Klaus Havelund, Grigore Rosu, Monitoring Java Programs with Java PathExplorer. Electr. Notes Theor. Comput. Sci. 55(2): (2001)
10. Klaus Havelund, Grigore Rosu, Efficient monitoring of safety properties. STTT 6(2): 158-173 (2004)
11. G. Katz, D. Peled, Code Mutation in Verification and Automatic Code Generation, TACAS 2010, Paphos, Cyprus, LNCS 6015, Springer, 435–450.
12. R. van der Meyden, Common Knowledge and Update in Finite Environment, Information and Computation, 140(2): 115–157, 1998.
13. J. Orlin, Contentment in Graph Theory: Covering Graphs with Cliques, Indagationes Mathematicae, 80(5): 406–424, 1977.
14. J. Pérez, R. Corchuelo, M. Toro, An Order-based Algorithm for Implementing Multiparty Synchronization, Concurrency - Practice and Experience, 16(12): 1173–1206, 2004.

15. J. G. Thistle, Undecidability in Decentralized Supervision, System and Control Letters volume 54, 2005, 503-509.
16. W. Thomas, On the Synthesis of Strategies in Infinite Games, STACS 1995, LNCS 900, Springer, 1–13.
17. S. Tripakis, Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters, 90(1):21–28, 2004.