Achieving Distributed Control Through Model Checking

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

[1] VERIMAG, Centre Equation, Avenue de Vignate, 38610 Gières, France
[2]Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. We apply model checking of knowledge properties to the design of distributed controllers that enforce global constraints on concurrent systems. The problem of synthesizing a distributed controller is, in general, undecidable, and the local knowledge of the processes may not directly suffice to control them to achieve the global constraint. We calculate when processes can decide, autonomously, to take or block an action so that the global constraint will not be violated. When the separate processes cannot make this decision alone, it may be possible to temporarily coordinate several processes in order to achieve sufficient knowledge jointly and make combined decisions. Since the overhead induced by such coordinations is important, we strive to minimize their number, again using model checking . We show how this framework is applied to the design of controllers that guarantee a priority policy among transitions.

1 Introduction

Consider a concurrent system, where some global safety constraint, say of prioritizing transitions, needs to be imposed. A completely global coordinator can control this system and allow any of the maximal priority actions to progress in each state. However, the situation at hand is that of a distributed control [7, 12]; controllers, one per process or set of processes, may restrict the execution of some of the transitions if their occurrence may violate the imposed constraint. Due to the distributed nature of the system, each controller has a limited view of the entire system. Each controller may keep some limited amount of memory that is updated according to the history it can observe.

The *knowledge* of a process at any particular local state includes the properties that are common to all reachable (global) states containing it. There are several definitions for knowledge, depending on how much of the local history may be encoded in the local state. Knowledge was suggested as a tool for constructing a controller in [6, 1]. There, controlling a distributed system was achieved by first precalculating the *knowledge* of a process. Based on its precalculated knowledge, reflecting all the possible current situations of the other processes, a *controller* for a process may decide at runtime whether an action of the controlled process can be executed without violating the imposed constraint. Sometimes, however, the process knowledge is not sufficient. Then, the joint knowledge of several processes (sometimes called *distributed knowledge*) may be monitored using fixed controllers for sets of processes. Unfortunately, this approach causes the loss of actual concurrency among the processes that are jointly monitored.

Instead of permanent synchronizations via fixed process groups, we suggest in this paper a method for constructing distributed controllers that synchronize processes temporarily. We use model-checking techniques to precalculate a minimal set of synchronization points, where joint knowledge can be achieved during short coordination phases. An additional goal is synchronizing a minimal number of processes as rarely as possible. After each synchronization, the participating processes can again progress independently until a further synchronization is called for.

In [6], knowledge-controllability (termed *Kripke observability*) is studied as a basis for constructing a distributed controller. There, if a transition 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. This approach requires sufficient knowledge to allow any enabled transition that preserves the imposed constraint. The construction in [1] is different: it requires that *at least* one process knows that the occurrence of *some* enabled transition preserves the correctness of the imposed constraint, hence supporting its execution. This approach preserves the correctness of the controller even when knowledge about other such transitions is limited, at the expense of restricting the choice of transitions.

The approach suggested here extends the knowledge-based approach of [1]. We use a coordinator algorithm, such as the α-core [5], which achieves temporary multiprocess coordinations using asynchronous message passing. Such coordinations can be used to achieve a precalculated joint knowledge, i.e., knowledge common to several processes. Such interactions are still expensive as they incur additional overhead. Therefore, an important part of our task is to minimize the number of interactions and the number of processes involved in such interactions.

2 Preliminaries and Related Work

Definition 1 (Distributed Transition systems). *A distributed transition system $\mathcal{A}$ is a fivetuple $\langle \mathcal{P}, V, S, \iota, T \rangle$:*

- $\mathcal{P}$ *is a finite set of* processes*.*
- V *is a finite set of* variables*, each ranging over some finite domain. A process $p \in \mathcal{P}$ can access and change variables in V_p. Thus, $V = \cup_{p \in \mathcal{P}} V_p$. We do not require the sets V_p to be disjoint.*
- S *is the set of* global states*. Each state assigns a value to each variable in V according to its domain.*
- $\iota \in S$ *is the* initial state*.*
- T *is a finite set of* transitions*. A transition $\tau \in T$ consists of an* enabling condition *en_τ, which is a quantifier-free first order predicate, and a* state transformation *f_τ. The transitions $T_p \subseteq T$ are associated with process p.*

Thus, $T = \cup_{p \in \mathcal{P}} T_p$*. A transition* τ *may belong to multiple processes* $P_\tau = \{p | \tau \in T_p\}$*. Both enabling condition and transformation are over the variables* $\cup_{p \in P_\tau} V_p$*.*

Definition 2. *A* local state $s|_p$ *of a process* $p \in \mathcal{P}$ *is the restriction of a global state* s *to the variables in* V_p*. Similarly, the* joint local state $s|_P$ *of a set of processes* $P \subseteq \mathcal{P}$ *is the restriction of a global state to the variables in* $\cup_{p \in P} V_p$*.*

For a set of states S of a transition system, we denote the set of local states of process p by $S|_p$, and, respectively, the set of joint local states for set of processes $P \in \mathcal{P}$ by $S|_P$. A transition τ is *enabled* in a state s when $s \models en_\tau$ (i.e., s satisfies en_τ). If τ is enabled in s and τ is executed, a new state $s' = f_\tau(s)$ is reached. We denote this by $s \xrightarrow{\tau} s'$.

Definition 3. *An* execution *of a distributed system* $\mathcal{A}$ *is a maximal sequence* $s_0\, s_1\, s_2 \ldots$ *such that* $s_0 = \iota$*, and for each* $i \geq 0$*,* $s_i \xrightarrow{\tau_i} s_{i+1}$ *for some* τ_i*. A global state is called* reachable *if it appears in some execution sequence.*

Definition 4. *Given a system* $\mathcal{A}$*, a set of processes* $P \subseteq \mathcal{P}$ knows *in a state* s *some property* φ *over* V*, if* $s' \models \varphi$ *for each* reachable *global state* s' *with* $s'|_P = s|_P$*. We denote this by* $s \models K_P \varphi$ *.*

When P is a singleton, we often write p for the set $\{p\}$ as in $K_p \varphi$. It is easy to see that if $s \models K_P \varphi$ and $s|_P = s'|_P$ then also $s' \models K_P \varphi$.

Definition 5. Comment: **S: don't we have to say a controller for ψ ? and then that such a controller is correct if it ensures that each controlled execution satisfies ψ?**

Moreover, we say somewhere that we don't consider uncontrollable transitions, and here they are. This is confusing for the definition of controlled executions

A finite state distributed disjunctive controller *[7, 12] for a system* $\mathcal{A} = \langle \mathcal{P}, V, S, \iota, T \rangle$ *is a set of automata* $C_p = (L_p, l_p, T_p^o, T_p^c, \rightarrow_p, E_p)$*, one per process* p *in* $\mathcal{P}$*, where:*

- L_p *is the set of* states *of* C_p*, i.e., its finite memory.*
- $l_p \in L_p$ *is the initial state of* C_P*.*
- T_p^o *is the set of* transitions observable *by process* p*, satisfying* $T_p \subseteq T_p^o \subseteq T$*. This means that* p *is aware of the execution of transitions from* T_p^o *and thereupon the controller* C_p *can change its states.*
- T_p^c *is the set of* controllable *transitions, where* $T_p^c \subseteq T_p$*. We require consistency between processes regarding controllability: if* τ *is involved with several processes, then it is either controllable by all of them or by none of them.*
- $\rightarrow_p$: $L_p \times T_p \mapsto L_p$ *is the* transition function *of* C_p*.*
- $E_p : S|_p \times L_p \mapsto 2^{T_p^c}$ *is the* support function*, which in each local state returns the set of transitions of process* p *that* C_p *supports, i.e., allows to proceed.*

A controller is designed to impose some constraint ψ on a given system $\mathcal{A}$, while not to introduce any new deadlocks.

Definition 6. *A* controlled execution *of a distributed system* $\mathcal{A}$ *with controllers* C_p *for* $p \in \mathcal{P}$ *is defined over a set of* controlled states $G \subseteq S \times \Pi_{p\in\mathcal{P}} L_p$. *Each* controlled state $g \in G$ *contains some global state* $s \in S$, *and a state* $\rho_p \in L_p$ *for each controller* C_p. *An execution* $g_0\, g_1\, g_2 \ldots$ *is a maximal sequence of controlled states, satisfying that* g_0 *is the controlled state containing the initial states* ι *of* $\mathcal{A}$ *and* l_p *for each* C_p. *Furthermore, for each adjacent pair of controlled states* g_i *and* g_{i+1} *there exists a transition* τ *such that the following holds:*

1. $s \xrightarrow{\tau} s'$ *— where* $s \in S$ *is the state component of the controlled state* g_i *and* $s' \in S$ *the one of* g_{i+1}.
2. $\tau \in T_p \setminus T_p^c$ *is non controllable* Comment: **S: ?!**, *or* $\tau \in T_p^c$ *and* $(s|_p, \rho_p) \models en_\tau$ *for at least one process* p, *where* $s|_p$ *is the local state of* p *in* g_i, *and* ρ_p *the state of* C_p *in* g_i; *that is, at least one local controller* supports τ *(which should mean that* τ *preserves* ψ*)*
3. *For the states* ρ_i *and* ρ_{i+1} *of controller* C_p *of* g_i *and* g_{i+1}, *respectively, if* $\tau \in T_p^o$, *then* $\rho_i \xrightarrow{\tau}_p \rho_{i+1}$. *Otherwise,* $\rho_i = \rho_{i+1}$. *That is,* C_p *changes its internal state when an observable transition occurs.*

We denote by $\mathcal{A}_c$ *the transformation of* $\mathcal{A}$ *that includes its controllers. The* goal *of a controller is a set of pairs* $\psi \subseteq S \times T$ *such that for each transition* $s \xrightarrow{\tau} s'$ *(as in bullet 1. above) it holds that* $(s, \tau) \in \psi$.

Note that the goal of the controller is to satisfy an invariant that is not just over the states (of the original system $\mathcal{A}$), but may also include the immediate transition out of that state.

The definition of a controller allows the use of some finite memory that is updated with the execution of observable transitions. This can be useful, e.g., in constructing a controller based on knowledge with perfect recall [11]. However, for a controller based on simple knowledge, as in Definition 4, there is no need to exercise this capability, and L_p can thus consist of a single state. As in [1], we fix as a running example a particular property that we want to synthesize: that of enforcing some priority policy on the distributed system.

Definition 7 (Priority policy). *A* priority policy $Pr = (T, \ll)$ *for a system* $\mathcal{A}$ *is defined as a partial order relation* $\ll$ *on the set of transitions* T.

Among the transitions enabled in state s, we can identify those with *maximal priority*, i.e., enabled transitions such that for any other transition τ' enabled in s, either $\tau' \ll \tau$ or τ and τ' are incomparable. Let $\max_\tau$ be a predicate that holds in a state s, i.e., $s \models \max_\tau$, when the transition τ has a maximal priority among the transitions enabled in s.

Definition 8. *A* prioritized *execution of a system* $\mathcal{A}$ *according to a given priority policy* Pr *satisfies, in addition to the conditions of Definition 3, that when* $s_i \xrightarrow{\tau_i} s_{i+1}$, *then also* $s_i \models \max_{\tau_i}$.

The goal is then to construct a distributed controller for $\mathcal{A}$ such that, when running $\mathcal{A}$ together with its controller, only correctly prioritized executions occur. We assume that all the transitions are controllable. *Comment:* **S: can't we say that this is possible without loss of generality as long we don't forbid non observable non determinsism in $\mathcal{A}$?**

Definition 9. *Each local state $s|_p$ of process p satisfies one of the following properties k_i^p based on the knowledge of P at that state.*

- $k_1^p = \bigvee_{\tau \in T_p} K_p \max_\tau$*: process p can identify a transition τ such that it knows that τ is enabled with maximal priority.*
- $k_2^p = \neg k_1^p \wedge K_p \bigvee_{q \neq p} k_1^q$*: process p does not know whether it has a transition with maximal priority, but in all the global states s' with $s'|_p = s|_p$ some other process q is in a local state where k_1^q holds. This allows p to remain inactive without risk of introducing a deadlock.*
- $k_3^p = \neg k_1^p \wedge \neg k_2^p$*: p does not know whether or not there is an enabled transition with maximal priority.*

k_1^p can be extended to sets of processes: $k_1^P = \bigvee_{\tau \in \cup_{p \in P} T_p} K_P \max_\tau$.

When the constraint ψ to be imposed by the controller is different from the priority policy, the formula k_1^p needs to be changed accordingly; instead of $\max_\tau$, it must reflect the property that executing τ does not invalidate ψ. If ψ is a state property, then $\max_\tau$ can be replaced by the state predicate $wp_\tau(\psi)$ (for "weakest precondition"), which reflects the state property that holds when τ is enabled and ψ holds after its execution.

The construction in [1] checks whether $\bigvee_{p \in \mathcal{P}} k_1^p$ holds in all reachable states of the original system that are not deadlock (or termination). If so, it is sufficient that each process supports a transition when it knows that it is maximal in order to enforce the additional constraint ψ (in that case, priority) without introducing any additional deadlock. When this check fails to hold, it was suggested to monitor and control several processes together, or to use the more expensive knowledge of perfect recall (or to use both).

3 A Synchronization-Based Approach

In this paper, we suggest a new solution to the distributed control problem, which consists of synthesizing distributed controllers that allow processes to *temporarily synchronize* in order to obtain joint knowledge in those (local) states in which it is needed. The synchronization is achieved by using an algorithm like α-core [5]. This algorithm allows processes to notify, using asynchronous message passing, a set of coordinators about their wish to be involved in a joint action; the coordinators may accept or decline the requested action. We treat the synchronizations provided by the α-core, or any similar algorithm, as transitions that are joint between several participating processes. At a lower level, such synchronizations are achieved using asynchronous message passing. We assume that the correctness of such an algorithm guarantees the atomic-like behavior of such coordinations, allowing us to reason at this level of abstraction.

A joint local state $s|_P$ satisfying k_1^P indicates that the set of processes P know how to act in this state by selecting some transition with maximal priority. Our construction calculates using model checking for knowledge properties, which synchronizations are actually needed.

The *essential* condition for the construction of a synchronizing controller is the following: in a completely synchronous implementation, which monitors global states, given a non deadlock reachable global state (i.e., a state that has at least one enabled transition), at least one enabled transition must preserve the property ψ that we want to enforce. It follows from the fact that for the set of processes $\mathcal{P}$, it holds that $s \models K_{\mathcal{P}}\psi$ exactly when $s \models \psi$. The essential condition is a *sufficient* condition for obtaining a synchronizing global controller; however, it is not a necessary condition: it is possible that by imposing some control, some of the global states of the original system where the above property does not hold, would not be reached anyway.

An exact check for the existence of a global (completely synchronized) controller can be based on game theory. Accordingly, one may present the problem as implementing a strategy for the following two player game. One player, the environment, can always choose between the enabled uncontrollable *Comment:* **S: !?** transitions, while the other player can choose between the enabled controllable ones. The goal of the controller is that the safety property ψ is satisfied by the jointly selected execution. This can be solved using algorithms based on safety games [9].

Our algorithm proceeds in three steps: a first step calculates the local states and joint local states (synchronizations) providing sufficient knowledge to guarantee that in every global state at least one process supports some transition. We refer to this set of (joint) local states as the *knowledge table* Δ for $\mathcal{A}$.

The second step defines, with the help of the knowledge table, a distributed controller in the form of a distributed transition system which can then be implemented in a protocol by introducing a set of coordinators realizing the required synchronizations in Δ. In a third step, we propose to obtain a more efficient controller by minimizing the set of coordinations.

3.1 A set of synchronizations providing sufficient knowledge

In this first step we calculate the required *knowledge table* Δ. The construction of Δ is performed iteratively, starting with local states, then pairs of local states, triples etc. At each stage of the construction, Δ contains a set of (joint) local states $s|_P$ satisfying k_1^P.

Definition 10. *A set of (joint) local states Δ is an* invariant *of a system if each non-deadlock state of the system contains at least one (joint) local state from Δ.*

The first iteration includes in Δ, for all $p \in \mathcal{P}$, the singleton local states satisfying k_1^p, i.e. states in which progress of p is guaranteed. For each such local state $s|_p$ we *associate* the actual transition τ that makes k_1^p hold.

If Δ is not an invariant, we first calculate for each local state not satisfying k_1^p whether it satisfies k_2^p. Let U_p be the set of local states of process p satisfying $\neg(k_1^p \vee k_2^p)$. Now, in a second iteration, we add to Δ pairs $(s_p, s_q) \in U_p \times U_q$ for $p \neq q$ if there exists a reachable state s such that $s|_p = s_p$ and $s|q = s_q$, and if s_p, s_q satisfy $k_1^{\{p,q\}}$. Again, we associate with that entry of the table Δ the actual transition τ that witnesses the satisfaction of $k_1^{\{p,q\}}$ for that entry. The second iteration terminates as soon as Δ is an invariant or if all such pairs of local states have been classified. In a third iteration, we consider triples of local states from $U_p \times U_q \times U_r$ such that no subtuple is in Δ, and so forth.

3.2 A distributed controller imposing the global property

In the second step of the algorithm, we transform the system $\mathcal{A}$ into a controlled transition system $\mathcal{A}_c$ allowing only prioritized execution. We implement $\mathcal{A}_c$ using a set of coordinators realizing the required synchronization of Δ by an algorithm such as the α-core.

We want to achieve the joint local knowledge promised by the precalculation of Δ using synchronizations amongst the processes involved. Our construction guarantees that each time the transition associated with a tuple $(s|_{p_1} \dots s|_{p_k})$ from Δ is executed from a state that includes these local components, the property ψ we want to impose is preserved. We transform the system $\mathcal{A}$ such that only transitions associated with entries in Δ can be executed.

If a transition τ is associated with a singleton element $s|_p$ in Δ, then the controller for p, at the local state $s|_p$, will support τ. Otherwise, τ is associated with a tuple of local states in Δ; when reaching any of these local states, the corresponding processes $p_1 \dots p_k$ try to achieve a synchronization, which consequently allows τ to execute. This is done according to the protocol of the synchronization algorithm that is used. Upon reaching the synchronization, the associated transition τ is then supported by any of its participating processes. Formally, for each transition τ associated with a tuple of local states $(s|_{p_1} \dots s|_{p_k})$, we execute a transition, enabled exactly at the joint local state with the above components, and performing the original transformation of τ.

3.3 Minimizing the number of coordinators

It is wasteful to include a coordination for each joint local state involving at least two processes in Δ. We now show how to minimize the number of coordinators for pairs of the form $(s|_p, r|_q)$ in Δ. The general version of this method for larger tuples is analogous. We denote by $\Delta_{p,q}$ the set of pairs of Δ containing components from processes p and q.

A naive implementation may use a coordination for every pair in Δ. Nevertheless, the large number of messages needed to implement coordination by an algorithm like α-core suggests that we minimize the number of coordinations. A completely opposite extreme would be to use a unique coordination between processes p and q. Accordingly, when process p identifies that it may have a

q-partner in $\Delta_{p,q}$, then coordination starts. When coordination succeeds, the joint event checks whether the local states of p and q actually appear in $\Delta_{p,q}$. If they do, it provides the appropriate behavior; otherwise, the coordination is abandoned. In this way, many (expensive) coordinations may be made just to be abandoned, not even guaranteeing eventual progress.

Consider now a set of pairs $\Gamma \subseteq \Delta_{p,q}$ such that if $(s,r),(s',r') \in \Gamma$, then $(s,r'),(s',r) \in \Gamma$ (s and s' do not have to be disjoint, and neither do r and r'). This means that Γ is a complete bipartite subgraph of $\Delta_{p,q}$. We certainly can generate one coordination for all the pairs in Γ, and, upon success of the coordination, the precalculated table $\Delta_{p,q}$ will be consulted about which transition to allow, depending on the components $s|_p$ and $s|_q$. Thus, according to this strategy, a sufficient number of interactions is formed by finding a covering partition $\Gamma_1, \ldots, \Gamma_m$ of complete bipartite subgraphs of $\Delta_{p,q}$. That is, each pair $(s|_p, r|_q) \in \Delta_{p,q}$ must be in some set Γ_i. However, the minimization problem for such a partition turns out to be in NP-Complete.

Property 1. [4] Given a bipartite graph $G = (N, E)$ and a positive integer $K \leq |E|$, finding whether there exists a set of subsets $N_1, \ldots N_k$ for $k \leq K$ of complete bipartite subgraphs of G such that each edge (u, v) is in some N_i is in NP-Complete.

We use the following notation: when Γ is a set of pairs of local states, one from p and one from q, we denote by $\Gamma|_p$ and by $\Gamma|_q$ the p and the q components in these pairs, respectively. We apply the following heuristics to calculate a (not necessarily minimal) set of complete bipartite subsets $\Gamma_i \subseteq \Delta_{p,q}$ covering $\Delta_{p,q}$. We start with a first partition $\Gamma_1^0, \ldots, \Gamma_{m_0}^0$, and refine it until we obtain a fixpoint $\Gamma_1^k, \ldots, \Gamma_{m_k}^k$. We decide to start with process p if $|\Delta_{p,q}|_p| < |\Delta_{p,q}|_q|$, i.e., the number of elements paired up in $\Delta_{p,q}$ is smaller for p than for q. Otherwise, we symmetrically start with q. Let the elements of $\Delta_{p,q}|_p$ be $x_1, \ldots x_{m_0}$, and Γ_i^0 be the pairs in $\Delta_{p,q}$ containing x_i. Now, we repeatedly alternate between the q side and the p side the following step: we check for each two sets Γ_i^l and Γ_j^l whether $\Gamma_i^l|_q = \Gamma_j^l|_q$. If it is the case, we combine them into a single set $\Gamma_i^l \cup \Gamma_j^l$ (on even steps, we replace q with p). This is done as long as we can unify new subsets in this way. The whole process is performed in time cubic in the size of $\Delta_{p,q}$.

Figure 1 shows the result for an example. The left-hand side represents the coordinators induced by $\Delta_{p,q}$ and the right-side the minimal set of coordinators. Each Γ_i contains a single state of q. And indeed, if we start the procedure with q, the initial partition is already the solution.

4 The Difference between Distributed Controllers and Knowledge-based Controllers

We now show some connections between the classical controller synthesis problem (see, e.g., [7]) and knowledge-based control. In particular, we show that the

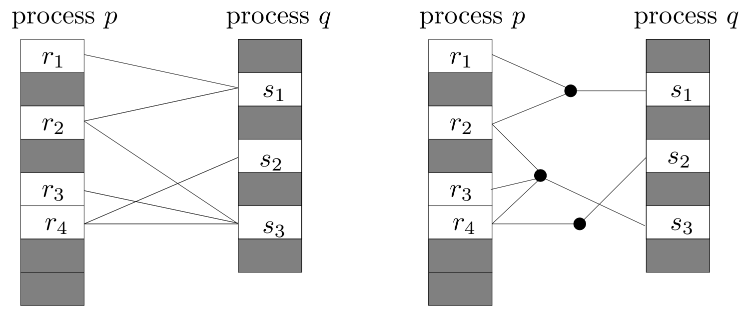

Fig. 1. Minimizing the number of coordinators

problem of synthesizing *Comment:* **S: distributed?** controllers for enforcing a priority policy is undecidable.

The knowledge approach to control in [6] requires that there is sufficient knowledge to allow *any* transition of the controlled system that does not violate the enforced property ψ. In [1], which we extend here, this requirement is relaxed; the knowledge must suffice to execute *at least one* enabled transition not violating ψ when such a transition exists. In the more general case of distributed controller design, one may want to block some enabled transitions even if their execution does not immediately violate the enforced property. This is required to prevent the transformed system from reaching deadlocked states, where the controlled system originally had a way to progress (thus, introducing new deadlocks).

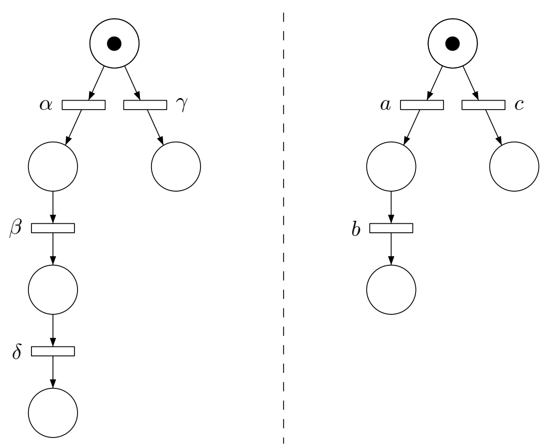

Fig. 2. A system that cannot be controlled

Consider a concurrent system, as in Figure 2, with two processes π_l (left) and π_r (right), each one of them having initially a nondeterministic choice. *Comment:*

this example is perhaps a bit confusing: we want to show that it might be impossible to know whether such a distributed controller exists or not. But in this example, we just emphasize that (a) it may be hard find one when it exists, and (b) there is not necessarily a unique least restrictive solution. The proof is however correct because it uses the argument that one cannot always decide whether non observable non determinsim exists in the system or not — which means that $\mathcal{A}$ is not finite The priorities in this system are $\delta \ll b \ll \beta$. Each process can observe only its own transitions. In the initial state, all four enabled transitions α, γ, a, c are unordered by priorities, and thus are all maximal. If α is fired and subsequently a (or vice versa), we reach a global state where process π_r does not have any enabled transition with maximal priority. Process π_l does, and it can execute β. Thereafter, since $\delta \ll b$, process π_l cannot execute δ and must wait for process π_r to execute b. Now, with its limited observability, π_l cannot distinguish between the situation before or after b was executed by π_r. Thus π_l lacks the capability, and the corresponding knowledge, of deciding whether to execute δ.

When a controller may block transitions even when their execution does not immediately lead to violation of the property to be preserved, the situation can be recovered. In the example above, we may choose either to block α in favor of γ, or to block a in favor of c. Blocking both α and a is not necessary. This example also shows that there is no *unique maximal* solution to the control problem that blocks the *smallest* number of transitions. Note that an alternative solution to blocking α or a can be achieved using a temporary interaction between the processes, as shown earlier in this paper.

It was shown in [10, 8] that the problem of synthesizing a distributed controller is, in general, undecidable. We show here that even when restricting the synthesis problem to priority policies, the problem remains undecidable. The proof for that appears is given below. Notice that when we have the flexibility of allowing additional coordination, as done in this paper, the problem, in the limit, becomes a sequential control problem, which is decidable.

Theorem 1. *Constructing a distributed controller that enforces a priority policy, is undecidable. Furthermore, this holds even in the case where all the transitions are controllable by the processes that include them.*

Proof. Following [10], the proof is by reduction from the post correspondence problem (PCP). In PCP, there is a finite set of pairs $\{(l_1, r_1), \ldots, (l_n, r_n)\}$, where the components l_i, r_i are words over a common alphabet Σ, and one needs to decide whether one can concatenate separately a *left word* from the left components and a *right word* from the right components according to a mutual nonempty sequence of indexes $i_1 i_2 \ldots i_k$, such that $l_{i_1} l_{i_2} \ldots l_{i_k} = r_{i_1} r_{i_2} \ldots r_{i_k}$.

Now, let $i \in \{1..n\}$, $\hat{l}_i$ be the word $l_i i$, i.e., the i^{th} left component concatenated with the index i. Similarly, let $\hat{r}_i$ be $r_i i$. We consider two regular languages: $L = (\hat{l}_1 + \hat{l}_2 + \ldots + \hat{l}_k)^+$ and $R = (\hat{r}_1 + \hat{r}_2 + \ldots + \hat{r}_k)^+$. Now suppose the process π_p executes according to the regular expression $l.L.x.a.b + r.R.x.c.d$. Assume for the moment that choice between l and r is uncontrollable. Suppose also that

π_p coordinates (through shared transitions) the alphabet letters from Σ with a process π_{q_1}, and the indexes letters from Σ with another process π_{q_2}. After observing x from π_p, then π_{q_1} and π_{q_2} are allowed to interact with each other. Specifically, π_{q_2} sends π_{q_1} the sequences of indexes it has observed. Suppose that now π_{q_1} has a nondeterministic choice between two transitions: α or β. The priorities are set as $b \ll \alpha \ll a$ and $d \ll \beta \ll c$. All other pairs of transitions are unordered according to $\ll$. If π_{q_1} selects α and r was executed, or π_{q_1} selects β and l was executed, then there is no problem, as α is unordered with respect to c and d, and also β is unordered with respect to a and b, respectively. Otherwise, there is no way to control the system so that it executes the sequence $a.\alpha.b$ or $c.\beta.d$ allowed by the priorities.

When there is no solution to the induced PCP problem, there is a controller for the priority policy problem: sending the sequence of executed indexes from π_{q_1} to π_{q_2} allows to match the sequence of words and indexes and check whether it is a left word or a right word exclusively. When the answer to the PCP problem is positive, then some left and right words are identical and with the same indexes. Process π_{q_1} cannot make a decision: given the information that π_{q_1} observed and later received from π_{q_2} is the same in both cases for the mutual left and right word. Thus, it cannot anticipate whether $c.d$ or $a.b$ will happen and cannot control the choice between α and β accordingly. This means that deciding the existence of a controller for this system would solve the corresponding PCP problem. It is thus undecidable.

The proof was done so far assuming that transitions l and r are uncontrollable. We would like to remove this assumption. To do that, we can implement that choice by a shared communication with two concurrent processes, π_{s_l} providing l and π_{s_r} providing r. After L we will allow the communication r, and after R we will allow the communication l. Thus, both communications must be consumed. Furthermore, the system terminates only by a single communication of π_{q_l} with the process that sends l and of π_{q_r} with the process that sends r. Thus, if our contoller had decided to block either l or r, then a deadlock would eventually occur. ∎

Note that in this proof we do not ensure a finite memory controller, even when one exists. Indeed a finite controller may not exist. To see this, assume a PCP problem with one word $\{(a, aa)\}$. To check whether we have observed a left or a right word, we may just compare the number of a's that p has observed with the number of indexes that q has observed.

5 Implementation and Experimental Results

We have implemented a prototype for experimenting with this approach. In our tool, we use Petri nets to represent distributed transition systems.

This tool first builds the set of reachable states and the corresponding local knowledge of each process. Then, it checks whether local knowledge is sufficient to ensure correct distributed execution of the system under study. Let $\mathcal{U}$-states be global states in which all corresponding local states satisfy $\neg(k_1^p \vee k_2^p)$. In fact, since $\neg k_1^p$ holds for each process p implies that k_2^p also holds. The existence of a $\mathcal{U}$ state means that Δ is not an invariant without adding some tuples for synchronization. We allow simulating the system while counting the number of synchronizations and $\mathcal{U}$-states encountered during execution according to different strategies as a measurement to the amount of additional synchronization required.

The example that we used in our experiments is a variant of the dining philosophers where philosophers may arbitrarily take first either the fork that is on their left or right. In addition, a philosopher may hand over a fork to one of his neighbors when his second fork is not available and the neighbor is looking for a second fork as well. Such an exchange (labeled *ex*) is a way to avoid the well-known deadlocks when all philosophers take first the fork on the same side. This example is partially represented by the Petri net of Figure 5.

In our example, places (concerning philosopher β) are defined as follows:

- $fork^i$: the i-th fork is on the table or not.
- $0fork_\beta$ (resp. $2forks_\beta$): philosopher β has no fork (resp. 2 forks) in his hands.
- $1fork_\beta^l$ (resp. $1fork_\beta^r$): philosopher β holds his left (resp. right) fork.

Transitions (concerning philosopher β) play the following role:

- get_β^{kl} (resp. get_β^{kr}), $k = 1, 2$: philosopher β takes the fork on his left (resp. on his right). This is his k-th fork.
- $eat\ and\ return_\beta$: philosopher β eats and puts both forks back on the table.
- $ex_{\alpha,\beta}$: philosopher α gives his right fork to philosopher β.
- $ex_{\beta,\alpha}$: philosopher β gives his left fork to philosopher α .

Processes correspond to philosophers. The transitions defining a process β have a β in their name, including the four exchange transitions $ex_{\alpha,\beta}$, $ex_{\beta,\alpha}$, $ex_{\beta,\gamma}$ and $ex_{\gamma,\beta}$. In Figure 5, transitions related only to philosopher β are drawn with full lines. Transitions in dashed lines are shared between β and one of his neighbors (α on the left, γ on the right).

Not controlling such exchanges at all allows non progress cycles. To avoid them we add priorities which allow exchange actions only when a "blocking situation" has been reached within some degree of locality.

First variant. We use a priority rule stating that an exchange between philosophers α and β has lower priority than α or β taking a fork. This leads to the following priorities for each α and β such that α is βs left neighbor:

- $ex_{\alpha,\beta} \ll get_\alpha^{2l}$: if α can pick up a left fork, he won't give his right fork to β.
- $ex_{\beta,\alpha} \ll get_\beta^{2r}$: symmetrically if β can pick up a right fork.

In this variant, local knowledge is sufficient. Indeed, when a philosopher α and both his neighbors are blocked in a state where they all have a left (resp. a right) fork, then philosopher α has enough knowledge to support an exchange with his left (resp. right) neighbor. For any number of philosophers, there is no $\mathcal{U}$-state. Thus, no extra synchronization is needed.

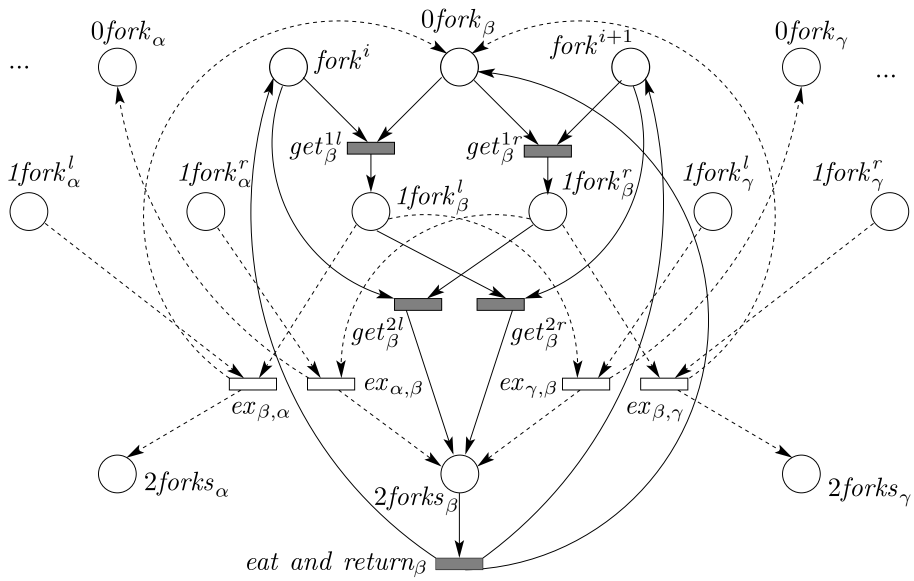

Fig. 3. A partial representation of the dining philosophers (philosopher β)

Second variant. Now, to further reduce the number of exchanges, one may decide that philosopher β may give his left fork to his left neighbor α only if (1) α is blocked (2) β is blocked and (3) βs right neighbor γ is also blocked (similarly for exchanges of right forks). This translates into adding the following priorities:

- $ex_{\alpha,\beta} \ll get^{2l}_{\delta}$, *eat and return*$_\delta$ (with δ the left neighbor of philosopher α)
- $ex_{\beta,\alpha} \ll get^{2r}_{\gamma}$, *eat and return*$_\gamma$ (with γ the right neighbor of philosopher β)

Local knowledge alone cannot ensure here correct distributed execution. However, binary synchronizations are sufficient in this example to ensure that the system is always able to move on, and this for any number of philosophers.

In Table 1, we show results for the second variant with 6, 8 and 10 philosophers. There are two $\mathcal{U}$-states which correspond to the situation where all philosophers hold their left fork, or they all hold their right fork. For computing the number of synchronizations, we used each time 100 runs of a length of 10,000 steps (i.e. transitions). Note that the number of exchange transitions is identical to the number of synchronizations.

philosophers	6	8	10
reachable states	729	6561	59049
synchronizations	354	285	237
$\mathcal{U}$-states encountered	253	149	100

Table 1. Results for 100 executions of 10,000 steps for the second variant

At the current stage, the minimization of the set of coordinators has not been implemented (we use one coordinator per synchronization pair in Δ) and our tool handles only joint local states consisting of two states.

6 Conclusion

Imposing a global constraint upon a distributed system by imposing blocking transitions is, in general, undecidable, as can be seen in control theory [10]. One practical approach for this problem was to use model checking of knowledge properties [1]. If we allow additional synchronization, the problem becomes decidable: at the limit, everything becomes synchronized, although this, of course, is highly undesirable. The method presented there provided a (disjunctive) controller. The problem with that approach is that in many cases the local knowledge of the separate processes does not suffice. A suggested remedy was to monitor several processes together, achieving this way an increased level of knowledge.

In the current work we look at the situation where we are allowed to coordinate between several processes, but only temporarily. First, we calculate whether the constraint we want to impose is feasible, when all processes are combined together. If this is the case, we check if we can control the system based on the local knowledge of processes or temporary interactions between processes. Of course, our goal is to minimize the number of interactions, and moreover, the number of processes involved in each interaction.

As an implementation, one can use a multiparty synchronization algorithm such as the α-core algorithm [5]. Based on that, we provide an algorithm using model checking to calculate at which local states synchronizations are needed. The synchronizing processes, successfully coordinating, are then able to use the knowledge table calculated by model checking, which dictates to them which transition can be executed. Some small corrections to the original presentation of the α-core algorithm appear in [3].

The framework suggested in this paper can be used as a distributed implementation for the Verimag BIP system [2]. BIP is based on a clear separation between the behavior of atomic components and the interaction between such components, which is represented using (potentially hierarchical) connectors. Priorities offer a mechanism to enforce scheduling policies by filtering the set of interactions that can be fired. So far, implementing BIP systems in a distributed setting remained a challenging task.

References

1. Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis. Priority scheduling of distributed systems based on model checking. In *CAV*, volume 5643 of *LNCS*, pages 79–93. Springer, 2009.
2. Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM*, pages 3–12. IEEE Computer Society, 2006.

3. Gal Katz and Doron Peled. Code mutation in verification and automatic code generation. In *TACAS 2010*, volume to appear of *LNCS*. Springer, 2010.
4. James B. Orlin. Contentment in graph theory: covering graphs with cliques, 1977.
5. José Antonio Pérez, Rafael Corchuelo, and Miguel Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency - Practice and Experience*, 16(12):1173–1206, 2004.
6. Karen Rudie and S. Laurie Ricker. Know means no: Incorporating knowledge into discrete-event control systems. *Transactions on Automatic Control*, 45(9):1656–1668, 2000.
7. Karen Rudie and W. Murray Wonham. Think globally, act locally: decentralized supervisory control. *Transactions on Automatic Control*, 37(11):1692–1708, 1992.
8. John G. Thistle. Undecidability in descentralized supervision. *System and Control Letters*, 54:503–509, 2005.
9. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13. Springer-Verlag, 1995. LNCS 900.
10. Stavros Tripakis. Undecidable problems of decentralized observation and control on regular languages. *Inf. Process. Lett.*, 90(1):21–28, 2004.
11. Ron van der Meyden. Common knowledge and update in finite environments. *Inf. Comput.*, 140(2):115–157, 1998.
12. Tae-Sic Yoo and Stéphane Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 12(3):335–377, 2002.