Practical Distributed Control Synthesis*

Doron Peled
Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel

Sven Schewe
Department of Computer Science
University of Liverpool
Liverpool, UK

Classic distributed control problems have an interesting dichotomy: they are either trivial or undecidable. If we allow the controllers to fully synchronize, then synthesis is trivial. In this case, controllers can effectively act as a single controller with complete information, resulting in a trivial control problem. But when we eliminate communication and restrict the supervisors to locally available information, the problem becomes undecidable. In this paper we argue in favor of a middle way. Communication is, in most applications, expensive, and should hence be minimized. We therefore study a solution that tries to communicate only scarcely and, while allowing communication in order to make joint decision, favors local decisions over joint decisions that require communication.

1 Introduction

Synthesizing code directly from a formal specification is highly intractable. Although automated synthesis is an attractive concept, neither is the practice of programming currently under threat of extinction, nor is automatic synthesis close to become a major factor in code generation. Still, some small critical tasks or protocols may be quite tricky for a programmer to produce and can greatly benefit from either fully automatic synthesis or a computer assisted development methodology. Prominent representatives of such tasks are concurrency control protocols that guarantee mutual exclusion, locking, or efficient memory access. The most challenging programming problems are often concurrent in nature, and, alas, synthesis of concurrent algorithms is undecidable [20].

This undecidability result on synthesizing concurrent code provides an important information about how *not* to attack the synthesis problem: through a general catch-all algorithmic method. One common practice to deal with an undecidable result is to restrict the generality of the problem. This can be done by limiting the architecture of the system [20, 13, 14, 5, 24, 25]. Positive results, however, are restricted to very limited architectures, such as pipelines, rings, or assumption about the hierarchy of memory access.

Another approach is to use a heuristic method, accepting that it may not succeed in all cases. A genetic search among the space of syntactically limited programs, which mutates existing candidates and progresses based on ranking provided by model checking, is described in [7]. Instead of using a direct synthesis algorithm, this technique generates candidate solutions, evaluates their quality (the model checking is generalized to a fitness function that estimates the distance from a solution), and adjusts them to fitter solutions. This method is successful in automatically finding solutions to mutual exclusion [7] and leader election problems [8] and was even used to detect and correct an error in a complicated communication protocol [9]. In principle, such heuristic search techniques can be fully automatic, though they require human interaction, through setting the parameters or adjusting them after an unsuccessful run, to be efficient.

We concentrate on synthesizing distributed control [22, 23, 29]. Synthesis is achieved in an incremental way: an already existing distributed system is modified to satisfy an additional property. In our

*The research was funded by Israeli Science Foundation (ISF) grant 1252/09 and by the EPSRC grant EP/H046623/1.

F. Yu and C. Wang: The 13th International Workshop on Verification of Infinite State Systems (INFINITY'11)
EPTCS 73, 2011, pp. 2–17, doi:10.4204/EPTCS.73.2

case, an invariant. Controlling the system is done by selectively blocking transitions. Ideally, local decisions can be taken by the processes themselves, or equivalently, by supervisors (one per process) that control the processes and synchronize with them. It turns out that the controllability problem (whether such distributed control exists) is also undecidable [27, 28], even for simple safety properties such as execution according to priorities [6].

To challenge this undecidability result, we relax the problem and allow additional temporary interactions between processes in order to allow them to acquire sufficient information to decide together on allowing (the converse of blocking) a transition. Formally, this coordination is mapped to a supervisor. A variant of this method is to partition the processes into groups of communicating processes, or, likewise, to introduce regional supervisors and assign each process to one of them. These (regional) supervisors collect enough process information to make control decisions. Under this assumption, all processes may, at the limit, interact to decide globally on the execution of each transition. This reduces the problem, in the limit, to a sequential control problem, which is trivial for finite state systems. The efficiency of this method depends on the amount of additional synchronization needed to enforce the desired invariant.

The method we use to enforce control is based on *knowledge* [4, 16]. Intuitively, in a distributed system, the knowledge of a process includes all properties that globally hold in all states consistent with the local view of the process. It reflects limited visibility of processes about the situation in other processes. The definition of knowledge is quite subtle, as it involves some assumptions about the view of a process. Indeed, in order to make a distributed control decision, a process (or a supervisor process synchronized with it) must make a choice that is good for all possible global states that are consistent with its local view. As process knowledge may not be sufficient, interaction between processes may be used to acquire the joint knowledge of several processes. Furthermore, knowledge can be refined based on the history of an execution. In this way, the number of possible global states that are consistent with the local view may be reduced, based on different histories. On the other hand, using this kind of knowledge requires the support of an expensive program transformation. We will discuss at length the use of knowledge in constructing control for distributed systems.

The knowledge based control synthesis [16, 1, 2, 6] restricts the executions of the system. The information gathered during the model checking stage is used as a basis for a program transformation that controls the execution of the system by adding constraints on the enabledness of transitions. This does not produce new program executions or deadlocks and, consequently, preserves all stuttering closed [18] linear temporal logic properties of the system [15] when no fairness is assumed.

2 Preliminaries

We chose Petri Nets as our model because of the intuitive and concise representation offered by them. But the method and algorithms developed extend to other models, such as transition systems, communicating automata, etc.

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

- *P is a finite set of* places,
- *the* states *are defined as $S = 2^P$ where $s_0 \in S$ is the* initial state,
- *T is a finite set of* transitions, *and*
- *$E \subseteq (P \times T) \cup (T \times P)$ is a bipartite relation between the places and the transitions.*

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

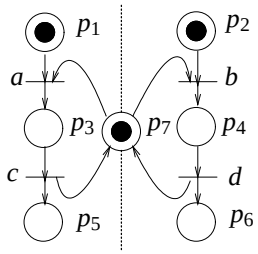

Figure 1: A Petri Net

Definition 2. *A transition t is* enabled *in a state s, denoted $s[t\rangle$, if ${}^\bullet t \subseteq s$ and $t^\bullet \cap s \subseteq {}^\bullet t$. A state s is in* deadlock *if there is no enabled transition from it.*

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

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

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

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

For convenience, we sometimes use as executions just the sequence of states, or just the sequence of transitions, as will be clear from the context. 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. As usual, we write $s \models p_i$ iff $p_i \in s$ and extend this in the standard way to Boolean combinations on state predicates. For a state s, we denote by φ_s the formula that is a conjunction of the places in s and the negated places not in s. Thus, φ_s is satisfied exactly by the state s. For the Petri Net in Figure 1, the initial state s_0 satisfies $\varphi_{s_0} = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge p_7$. For a set of states $Q \subseteq S$, let $\varphi_Q = \bigvee_{s \in Q} \varphi_s$, or any logically equivalent propositional formula, be a *characterizing formula* of Q. As usual in logic, when φ_Q and $\varphi_{Q'}$ characterize sets of states Q and Q', respectively, then $Q \subseteq Q'$ exactly when $\varphi_Q \rightarrow \varphi_{Q'}$.

An invariant [3] of N is a subset of the states $Q \subseteq 2^S$; a net N satisfies the invariant Q if $reach(N) \subseteq Q$. A *generalized invariant* of N is a set of pairs $I \subseteq S \times T$; a net N satisfies I if, whenever $s[t\rangle$ for a reachable s, then $(s,t) \in I$. This covers the above simple case of an invariant by pairing up every state that appears in Q with *all* transitions T.

Definition 6. *An execution of a Petri Net N* restricted *with respect to a set $I \subseteq S \times T$, denoted $exec_I(N)$, is a maximal set of executions $s_0 t_1 s_1 t_2 s_2 \ldots \in exec(N)$ such that, s_0 is the initial state, for each states s_i in the sequence, $s_i[t_{i+1}\rangle s_{i+1}$, and furthermore $(s_i, t_{i+1}) \in I$. The set of states reachable in $exec_I(N)$ is denoted $reach_I(N)$.*

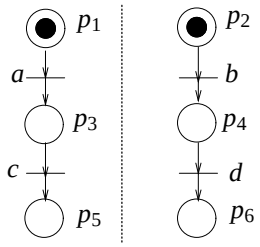

Figure 2: A Petri Nets with priorities $a \ll d$ and $b \ll c$

Definition 7. *For a set of executions X, let $pref(X)$ be the set of prefixes (including full executions) of X.*

Denote the last state of a finite prefix h of an execution by $last(h)$.

Lemma 1. *$reach_I(N) \subseteq reach(N)$ and $exec_I(N) \subseteq pref(exec(N))$.*

As an example of a property we may want to enforce, consider prioritized executions.

Definition 8. *A* Petri Net with priorities *is a pair $(N, \ll)$ with N a Petri Net and $\ll$ a partial order relation among the transitions T of N.*

Let $I_\ll = \{(s,t) \mid s[t\rangle \text{ and } \forall t' \in T\, s[t'\rangle \rightarrow t' \ll t\}$. The set of *prioritized executions* $exec_{I_\ll}(N)$ of $(N, \ll)$ is the set of executions restricted to $I_\ll$. The executions of the Petri Net M in Figure 2 (when the priorities $a \ll d$ and $b \ll c$ are *not* taken into account) include $abcd, acbd, bacd, badc$, etc. However, the prioritized executions of $(M, \ll)$ are the same as the executions of the Net N in Figure 1.

Definition 9. *A* process π *of a Petri Net N is a subset of the transitions T.*

We will represent the separation of transitions of a Petri Net into processes using dotted lines. We assume a given set of processes $\mathcal{C}$ that *covers* all transitions of the net, i.e., $\bigcup_{\pi \in \mathcal{C}} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. Let $proc(t) = \{\pi \mid t \in \pi\}$ be the set of processes to which t belongs. For the Petri Net in Figure 1, there are two executions: $acbd$ and $bdac$. There are two processes: the *left* process $\pi_l = \{a, c\}$ and the *right* process $\pi_r = \{b, d\}$.

The *neighborhood* of a set of processes Π includes all places that are either inputs or outputs to transitions of Π.

Definition 10. *The* neighborhood $ngb(\pi)$ *of a process π is the set of places $\bigcup_{t \in \pi}({}^\bullet t \cup t^\bullet)$. For a set of processes $\Pi \subseteq \mathcal{C}$, $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$.*

A set of processes Π *owns* the places in their neighborhood that can gain or lose a token by a transition t only if t is *exclusively* in Π.

Definition 11. *The set of places* owned *by a set of processes (including a singleton process) Π, denoted $own(\Pi)$, is $ngb(\Pi) \setminus ngb(\mathcal{C} \setminus \Pi)$.*

When a notation refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π, e.g., we write $own(\pi)$. Note that $ngb(\Pi_1) \cup ngb(\Pi_2) = ngb(\Pi_1 \cup \Pi_2)$, while $own(\Pi_1) \cup own(\Pi_2) \subseteq own(\Pi_1 \cup \Pi_2)$. The neighborhood of process π_l in the Petri Net of Figure 1 is $\{p_1, p_3, p_5, p_7\}$. Place p_7 is neither owned by π_l, nor by π_r, but it is owned by $\{\pi_l, \pi_r\}$. It belongs to the neighborhood of both processes and acts as a semaphore. It can be captured by the execution of a or of b, guaranteeing that $\neg(p_3 \wedge p_4)$ is an invariant of the system.

Our goal is to control the system to satisfy a generalized invariant by restricting some of its transitions from some of the states. The setting of the control problem may impose that only part of the transitions,

$ct(T) \subseteq T$, called *controllable* transitions, can be selectively supported by the processors that contain it. (It blocks if no processor supports it.) The other transitions, $uc(T) = T \setminus ct(T)$, are *uncontrollable*. Note that we may be at some state where either some uncontrollable transitions, or all enabled transitions, violate the generalized invariant. Being in such states is therefore "too late"; part of the controlling task is to avoid reaching such states.

In control theory, the transformation that takes a system and allows blocking some transitions adds a supervisor process [21], which is usually an automaton that runs *synchronously* with the controlled system. This (finite state) automaton observes the controlled system, progresses according to the transitions it observes, and blocks some of the enabled transitions, depending on its current state. In a similar way, in distributed control [29, 23, 22], for each process we assign such a supervisor, which changes its states each time the process it supervises makes a transition, or when a visible transition of another process (e.g., through the change of shared variables) is executed. Based on its states, the supervisor allows (supports) transitions of the controlled process. In a disjunctive control architecture [29], if no supervisor suports an, otherwise enabled, transition, it cannot execute and is thus blocked. Such a supervisor can be amalgamated, through a transformation, into the code of the controlled process. In order to capture this for Petri Nets, without a complicated transition splitting transformation, we use an extended model, as defined below. In particular, it allows adding enabling conditions and variable transformation to capture the encoding of the local supervision of the processes. It would also allow encoding additional asynchronous supervision in our solution.

Definition 12. *An* extended Petri Net *[12] is a Petri Net with a finite set of variables V_π over a finite domain per each process $\pi \in \Pi$. In addition, a transition t can be augmented with a predicate en_t on the variables $V_t = \cup_{\pi \in proc(t)} V_\pi$ and a transformation function $f_t(V_t)$. In order for t to fire, en_t must hold in addition to the basic Petri Net enabling condition on the input and output places of t. When t fires, in addition to the usual changes to the tokens, the variables V_t are updated according to the transformation f_t.*

We transform a Petri Net N and a generalized invariant I into an extended Petri Net N' that allows only the executions of N controlled to satisfy I.

Definition 13. *A* controlling *transformation obeys the following conditions:*

- *New transitions and places can be added.*
- *The input and output places of the new transitions are disjoint from the existing places.*
- *Variables, conditions and transformations can be added to existing transitions.*
- *Existing transitions will remain with the same input and output places.*
- *It is not possible to fire from some point an infinite sequence consisting of only added transitions.*

Added transitions are grouped into new (supervisory) processes. Added variables will represent some knowledge-dependent finite memory for controlling the system, and some interprocess communication media between the original processes and the added ones. Processes from the original net will have disjoint sets of variables from one another. The independence between the original transitions is preserved by the transformation, although some coordination may be enforced indirectly through the interaction with the new supervisory processes.

Definition 14. *Let $s\lceil_C$ map a state s of the transformed version N' into the places of the original version N by projecting out additional variables and places that N' may have on top of the places of N. This definition is also extended to executions (as sequences of states).*

This projection allows us to relate the sets of states of the original and transformed version. Firing of a transitions added by the controlling transformation does not change $s\lceil_{\mathcal{C}}$ and is not considered to violate I (the requirement that (s_i, t_{i+1}) in Definition 6 is imposed only when t_{i+1} is from the original net N). Note that our restrictions on the transformation implies that the sets $ngb(\Pi)$ and $own(\Pi)$ for $\Pi \subseteq \mathcal{C}$ are not affected by the transformation. Furthermore, albeit the rich structure of extended Petri Nets, our control transformation will allow a finite state control for a finite state system.

Definition 15. *Two executions σ and σ', viewed as sequences of states, are equivalent up to stuttering [18] when, by replacing any finite adjacent repetition of the same state by a single occurrence in both σ or σ', we obtain the same sequence. Let $stutcl(\Gamma)$ be the stuttering closure of a set Γ of sequences, i.e., all sequences that are stuttering equivalent to some sequences in Γ.*

Lemma 2. *A controlling transformation produces an extended Petri Net N' from N such that $exec(N')\lceil_{\mathcal{C}} \subseteq pref(stutcl(exec(N)))$.*

The controlling transformation may introduce new deadlocks, hence the lemma above asserts about the prefixes of the original executions. Of course, this is not a desirable outcome of the control transformation, and the solutions that will be given to the distributed control problem will circumvent it.

3 Process Knowledge and Joint Process Knowledge

The knowledge of a process at a given execution point consists of facts that hold in all global states that are consistent with the current local view of this process. The current local view represents the limited ability of a process to observe the global state of the system. A process may be aware of its own local variables and shared variables in its neighborhood. Similarly, we can define the joint knowledge of several processes, by considering their joint local view.

According to the limited observability of the processes Π, we can define an equivalence relation $\equiv_\Pi \subseteq S \times S$ (when the set of processes Π is a singleton, we can write $\equiv_\pi$) among the states S of the system; if the current state is $s \in S$, then the processes Π cannot distinguish, given their joint local view, between s and any state equivalent to it according to $\equiv_\Pi$. Such an equivalence relation is the basis of the definition of knowledge [4].

Definition 16. *The processes Π (jointly)* know *a property ψ in a state s, denoted $s \models K_\Pi \psi$, if, for all s' such that $s \equiv_\Pi s'$, we have that $s' \models \psi$.*

In the Petri Nets model, the equivalence relation $\equiv_\Pi$ can be defined by restricting first each state to a part of a state. Then, states that share the same part are considered equivalent. There are several possibilities to restrict the part of a state that is associated with a subset of the processes Π. We will give two possibilities for such a restriction. The first one is that of *local information*, which takes the part of the state that includes the neighborhood of the processes Π. This Petri Nets definition corresponds, in general systems, to the variables that can be read or written by the processes Π. The second such restriction is that of *local state* (different names were chosen only to make a distinction), based on restricting states to the places that the processes Π own. This corresponds, in general systems, to the variables that only the processes Π, and no other processes, can change (write).

Definition 17. *The* local information *of a set of processes Π of a Petri Net N in a state s is $s\lceil_\Pi = s \cap nbg(\Pi)$.*

In the Petri Net in Figure 1, the local information of π_l in any state s consists of the restriction of s to the places $\{p_1, p_3, p_5, p_7\}$. In the depicted initial state, the local information is $\{p_1, p_7\}$.

Definition 18. *The* local state *of a set of processes* Π *of a Petri Net N in a state s is* $s\lfloor_\Pi = s \cap own(\Pi)$.

It is always the case that $s\lfloor_\Pi \subseteq s\lceil_\Pi$. The local state of π_l in the initial state of Figure 1 is $\{p_1\}$.

Lemma 3. *If* $\pi \notin \Pi$ *then* $s\lfloor_{\Pi\cup\{\pi\}}$ *is the (disjoint) union of* $s\lfloor_\Pi$ *and* $s\lceil_\pi \cap own(\Pi \cup \{\pi\})$.

In the following definitions, we can often use either the local information or the local state. When this is the case, we will use $s|_\Pi$ instead of either $s\lceil_\Pi$ or $s\lfloor_\Pi$.

Definition 19. *Let* $\Pi \subseteq \mathcal{C}$ *be a set of processes. Define an equivalence relation* $\equiv_\Pi \subseteq reach(N) \times reach(N)$ *such that* $s \equiv_\Pi s'$ *when* $s|_\Pi = s'|_\Pi$.

As $s|_\Pi$ can stand for either $s\lceil_\Pi$ or $s\lfloor_\Pi$, this gives two different equivalence relations. When it is important to distinguish between them, we denote the one based on "$\lceil$" as $\equiv^w_\Pi$ (weak equivalence) and the one based on "$\lfloor$" as $\equiv^s_\Pi$ (strong equivalence).

Lemma 4. *If* $t \in \pi$ *and* $s \equiv^w_\pi s'$ *then* $s[t\rangle$ *if, and only if,* $s'[t\rangle$.

That is, the enabledness of a transition depends only on the local information of a process that contains it. This does not hold when we replace $\equiv^w_\pi$ by $\equiv^s_\pi$. In the Prioritized Petri Net in Figure 1, e.g., we have that $\{p_1, p_2, p_7\} \equiv^w_{\pi_l} \{p_1, p_4, p_7\}$, since π_l has the same local information $\{p_1, p_7\}$ in both states. The state $\{p_1, p_4\}$ is not equivalent to either of these states. On the other hand, these three states are equivalent according to $\equiv^s_{\pi_l}$ (p_7 is not in $own(\pi_l)$).

Corresponding with the two equivalence relations of Definition 19, we distinguish between knowledge based on strong equivalence $\equiv^s_\Pi$ (and hence on local states), denoted $K^s_\Pi\varphi$ and knowledge based on weak equivalence $\equiv^w_\Pi$ (and hence local information), denoted $K^w_\Pi\varphi$. The knowledge based on the local state (resp. local information) is called *s*trong (resp. *w*eak) knowledge. Since the local information determines the local state (while multiple local states may have the same local information), we have $K^s_\Pi\varphi \to K^w_\Pi\varphi$. Consequently, we may *know* more under weak knowledge.

The motivation for the different definitions of equivalence and, subsequently, the different definitions of knowledge is as follows. In order to make choices (to support or block a transition) that take into account knowledge based on local information, a process, or a set of processes, needs to have some guarantee that the local information will not be changed by other processes while it is collecting information from the processes or making the decision. For a single process, this may be achieved by the underlying hardware. But it is unreasonable to require such a guarantee for a set of processes that either temporary interact (interactions take time and other processes may meanwhile progress) or send their current local view to some supervisor process that collects views from several processes. Thus, for decisions involving a set of processes, strong knowledge, based on the joint local state, is used instead.

The classical definition of knowledge is based on relations $\equiv_\Pi$ over the reachable states $reach_I(N)$. However, when using knowledge to control a system to satisfy a generalized invariant, one may calculate the equivalences and the knowledge based on the states $reach_I(N)$ that appear in the executions of the original system that satisfy this generalized invariant I. This (cyclic looking) claim is proved [2] by induction on the progress of the execution in the controlled system: for a state already on such an execution (by the inductive assumption) the controlled system allows firing only transitions that preserve the generalized invariant, hence is also in $reach_I(N)$. We may need to restrict the generalized invariant I, in order not to introduce new deadlocks. This means even fewer reachable states, which can consequently increase the knowledge further.

One of the main challenges of using knowledge for controlling systems is that it is not always possible to decide, based on the local (or joint) knowledge, whether or not allowing a transition will guarantee the desired generalized invariant. One tool that can be used in this case is to allow additional interactions between processes, or knowledge accumulation by additional asynchronous supervisors. This will be explained later. However, before progressing to such an expensive solution, we may also try to improve

the knowledge by refining the equivalence relation that is used in its definition.

The definitions of knowledge that we used assumes that the processes do not maintain a log with their history. The use of knowledge with such a log, called *knowledge with perfect recall* [16], is discussed in [1]. Consider an equivalence $\approx_\pi$ between histories that seem undistinguishable to the process π. Two finite prefixes h, h' of Petri Net executions will be considered equivalent for $\approx_\pi$ if the projection of h on transitions visible to π are the same in both h and h'. Specifically for Petri Nets, we can define the transitions $vis(\pi) = \{t | (^\bullet t \cup t^\bullet) \cap ngb(\pi) \neq \emptyset\}$ (t is dependent on some transitions in π). In this case, the last states $last(h)$ and $last(h')$ of h and h', respectively, are equivalent under $\equiv^w$ (and hence also under $\equiv^s$). This can be shown by induction over the length of the prefixes, based on the fact that only the transitions in $vis(\pi)$ affect $ngb(\pi) \supseteq own(\pi)$.

Definition 20. *Let $h \models \psi$ exactly when $last(h) \models \psi$. Then we define* past knowledge, *where $h \models K^p_\pi \psi$ if, for all $h' \approx_\pi h$, $h \models \psi$.*

In particular for properties ψ that depend only on the last state of h, the use of the history refines the weak equivalence between states: $h \approx_\pi h'$ implies $last(h) \equiv^w_\Pi last(h')$. To take advantage of the refined definition of knowledge, we need somehow to distinguish local states that have non equivalent histories. On the face of it, this seems to require unbounded memory. However, looking deeper into the new definition of knowledge, one can observe that the following finite construction will work [16, 1].

Definition 21. *Let $\triangle_\pi$ be the set of finite sequences of transitions that do not change the neighborhood of π (i.e., independent with the transitions in π).*

Definition 22. *Let $\mathcal{A} = (S, s_0, T)$ be a finite automaton representing the global states S of a Petri Net N, including the initial state $s_0 \in S$ and the transitions T between them. For each process π, we construct an automaton $\mathcal{A}_\pi$ representing the set of states of $\mathcal{A}$ where the Petri Net N can be after a given local history. The automaton $\mathcal{A}_\pi$ has the following components:*

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

Then, one may use $K^p_\pi \psi$ instead of K^w_π for locally supporting transitions. (Note that $K^w_\pi \rightarrow K^p_\pi$.) However, the size of each such automaton (one per process π) can be exponential in the size of the global state space. Knowledge of perfect recall can be implemented by adding a synchronized supervisor with memory (basically implementing the automaton $\mathcal{A}_\pi$). It is natural to ask whether one can make an even finer distinction between states than with knowledge of perfect recall. This is indeed possible, but at the cost of a more involved program transformation. We may augment in our transformation the context of the interprocess communication between processes with additional transformation, that would implement the support for additional knowledge. Such a transformation can, e.g., be based on Gossip Automata [17], providing the most recent past local view of any other process.

We henceforth use knowledge formulas combined with Boolean operators and propositions. For a detailed syntactic and semantic description of logics with knowledge one can refer, e.g., to [4]. Once $s \models K_\Pi \psi$ is defined, ψ can also be a knowledge property, hence $s \models K_{\Pi'} K_\Pi \psi$ (knowledge about knowledge) is also defined, though the finite-state representation described above only applies to past knowledge used in outermost knowledge operators.

Lemma 5. *If $s \models K_\Pi \varphi$ and $s \equiv_\Pi s'$, then $s' \models K_\Pi \varphi$.*

Lemma 6. *Knowledge is monotonic with respect to the set of observing processes: if $\Pi' \subseteq \Pi$ then $K_{\Pi'}\varphi \rightarrow K_\Pi \varphi$.*

Lemma 7. *Given that $s \models K_\Pi \varphi$ in some basic Petri Net N, then $s \models K_\Pi \varphi$ also in a transformed version N'.*

Enforcing prioritized executions in a completely distributed way may be impossible. In Figure 2, a and c belong to the left process π_l, and b and d belong to the right process π_r, with no interaction between the processes. The left process π_l, upon having a token in p_1, cannot locally decide whether to execute a; the priorities dictate that a can be executed if d is not enabled, since a has a lower priority than d. But cannot distinguish between the cases where π_r has a token in p_2, p_4, or p_6.

In the Prioritized Petri Net in Figure 2, e.g., we have that $\{p_1, p_2\} \equiv^w_{\pi_l} \{p_1, p_4\}$, since in both states π_l has the same local information $\{p_1\}$. In the state $\{p_1, p_2\}$, a is a maximal priority enabled transition (incomparable with b), while in $\{p_1, p_4\}$, a is not maximal anymore, as we have that $a \ll d$, and both a and d are now enabled. In the initial state the local information (and also the local state) of π_l is $\{p_1\}$. Thus, π_l does not have enough knowledge to support any transition since $\{p_1, p_2\} \equiv^w_{\pi_l} \{p_2, p_3\}$). Similarly, the local information of π_r is $\{p_2\}$, which also is not sufficient to support any transition. After they both hang on a supervisor, it has enough information to support a or b.

4 A Globally Controlled System

Before providing a solution to the distributed control problem we need to provide a solution to the related global control problem. Some reachable states are not allowed according to the generalized invariant. In order not to reach these states, resulting in an immediately deadlock, we may need to avoid some transitions that lead to such states from previous states. This is done using game theoretical search.

The game is played between a *constructor*, who wants to preserve the generalized invariant I indefinitely (or reach a state that is already a deadlock in the original system N), and a *spoiler*, who has the opposite goal. The game is played on the states S of a net. It starts from the initial state s_0 and ends if a deadlock state is reached (and may go on forever). In each round, the constructor player chooses a nonempty subset of enabled transitions that must include all enabled uncontrollable transitions. Subsequently, the spoiler chooses a transition from this set, which is then executed. The spoiler wins as soon as she can choose a transition that violates I, i.e., $(s,t) \notin I$, while the constructor wins if this condition never holds (on an infinite run or a finite run that ends in a deadlock).

We can define an "attractor" $attr(A)$ that contains all states in A and all states that the spoiler can force to A in a single transition. A state s is in $attr(A)$ if one of the following conditions holds:

- $s \in A$,
- there exists an uncontrollable transition $t \in uc(T)$ enabled in s with $s[t\rangle s'$ and either $s' \in A$, or $(s,t) \notin I$, or
- s is not a deadlock state in the Petri Net N and, for all transitions t enabled in s, such that $s[t\rangle s'$ and $(s,t) \in I$, it holds that $s' \in A$.

As usual, we define $attr^{n+1}(A) = attr(attr^n(A))$, where $attr^0(A) = A$. Because of the monotonicity of the $attr(A)$ operator (with respect to set inclusion) and the finiteness of the state space, there is a least fixpoint $attr^*(A)$, which is $attr^n(A) = attr^{n+1}(A)$ for some (smallest) n.

Now, let $I_G = \{(s,t) \in I \mid s[t\rangle s' \text{ and } s' \notin attr^*(\emptyset)\}$. Let $G = reach_{I_G}(N)$ if $s_0 \notin attr^*(\emptyset)$, otherwise $G = \emptyset$. These are the "good" reachable states in the sense that they are allowed by I *and* the system can be controlled to henceforth adhere to I.

Definition 23. *Let* $R = \{(s,t) \in I \mid \exists s' \, s[t\rangle s' \wedge s, s' \in G\}$ *be the* safe transition relation.

If the initial state is good ($s_0 \in G$), then the constructor can win by playing according to R. If, on the other hand, s_0 is in the attractor $attr^*(\emptyset)$ of the bad states, then s_0 is in $attr^n(\emptyset)$ for some $n \leq |S|$. By the definition of $attr^n(\emptyset)$, the spoiler can force the game to $attr^{n-1}(\emptyset)$ in the next step, then to $attr^{n-2}(\emptyset)$, and so forth, and thus make sure the bad states are reached within at most n steps.

Lemma 8. *The constructor can force a win if, and only if,* $s_0 \in G$.

This game can obviously be evaluated quickly on the *explicit* game graph, and hence in time exponentially in the number of places. EXPTIME completeness can be demonstrated by a simple reduction from the PEEK-G_5 [26] game [10]. Deciding if the constructor can force a win is PSPACE complete for Petri Nets with only controllable transitions [10].

Model Checking

We will use the following propositional formulas, with propositions that are the places of the Petri Net:

- The good states G: φ_G.
- The states where a transition t is enabled: $\varphi_{en(t)}$.
- At least one transition is enabled, i.e., there is no deadlock: $\varphi_{df} = \bigvee_{t \in T} \varphi_{en(t)}$.
- Transition t is allowed from the current state by the safe transition relation R: $\varphi_{good(t)}$
- The local information (resp. local state) of processes Π at state s: $\varphi_{s\lceil_\Pi}$ (resp. $\varphi_{s\lfloor_\Pi}$).

The corresponding sets of states can easily be computed by model checking and stored in a compact way, e.g., using BDDs. Given a Petri Net, one can perform model checking in order to calculate whether $s \models K_\pi \psi$. The processes Π know ψ at state s exactly when $(\varphi_G \wedge \varphi_{s\lfloor_\Pi}) \rightarrow \psi$ is a propositional tautology. We can also check properties that include nested knowledge by simply checking first the innermost knowledge properties and marking the states with additional propositions for these innermost properties.

Model checking knowledge using BDDs is *not* the most space efficient way of checking knowledge properties, since φ_G can be exponentially big in the size of the Petri Net. In a (polynomial) space efficient check (which has a higher *time* complexity), we enumerate all states s' such that $s \equiv_\pi s'$, check reachability of s' using binary search, and, if reachable, check whether $s' \models \psi$. This can also be applied to nested knowledge formulas, where inner knowledge properties are recursively reevaluated each time they are needed. The PSPACE complexity is subsumed by the EXPTIME complexity in the general case algorithm for the safe transition relation R.

5 Control Using Knowledge Accumulation

According to the knowledge based approach to distributed control [1, 6, 2, 22], model checking of knowledge properties is used at a preliminary stage to determine when, depending the local information, an enabled transition can safely be fired. In our case, this means checking $s \models K^w_\pi \varphi_{good(t)}$ (by Lemma 5, the satisfaction only depends on $s\lceil_\pi$). At runtime, a process *supports* a transition in every local information where this holds. The following *support policy* uses this information at runtime:

> A transition t can be fired (is enabled) in a state when, in addition to its original enabledness condition, at least one of the processes in $proc(t)$ supports it.

Enabled uncontrolled transitions can always be supported, as a consequence of the following Lemma.

Lemma 9. *If $t \in \pi \cap uc(T)$ and $(s,t) \in R$, then $s \models K^w_\pi \varphi_{good(t)}$.*

This follows from the observation that the safe transition relation does not restrict the uncontrolled transition.

It is possible that, in some (non deadlock) states of G, no process has enough local knowledge to support an enabled transition and, furthermore, no uncontrollable transitions are enabled. We may need to synchronize several processes or collect the joint knowledge of several processes through the use of asynchronous supervisors. A process can decide, based on its current (lack of) knowledge, whether it *hangs* on such supervisor by sending it its local state. A supervisor $\mathcal{T}$ can make a decision, based on accumulated joined knowledge of several hung processes, that one of them can support an enabled transition. A process hangs on a supervisor, when the following property *does not* hold:

$$\kappa^\pi = \bigvee_{t \in \pi} K^p_\pi \varphi_{good(t)} \vee K^p_\pi \bigvee_{\pi' \neq \pi} \bigvee_{t \in \pi'} K^w_{\pi'} \varphi_{good(t)}$$

That is, a process does neither hang on the supervisor when it has enough knowledge to support a transition, nor if it knows that some other process has such knowledge. In the latter case, it does not actually need to be able to determine which process has that knowledge.

To avoid the overhead of computing past knowledge, it is often cheaper (and more appropriate) to use weak knowledge instead. In case nested knowledge calculation is too expensive as well, we may use the simplified knowledge formula $\bigvee_{t \in \pi} K^w_\pi \varphi_{good(t)}$ instead, at the expense of making more processes hang.

The supervisor $\mathcal{T}$ keeps the updated joint local state of the hung processes Π. When a process π hangs, it updates this view by transmitting to $\mathcal{T}$ its local information $s\lceil_\pi$, from which $\mathcal{T}$ keeps (according to Lemma 3) $s\lceil_\pi \cap own(\Pi \cup \{\pi\})$. Since all processes in $\Pi' = \Pi \cup \{\pi\}$ are now hung, no other process can change these places. Then the joint knowledge $K^s_{\Pi'} \varphi_{good(t)}$ can be used to support a transition t. Recall that knowledge based decisions of a single process use weak knowledge (based on the local information), while multiple processes use strong knowledge (i.e., based on the joint local state).

In the following cases,

1. after the decision of a process π to hang on $\mathcal{T}$, other processes make changes to π's local information that allow it to support some transition t,
2. when a transition t with $\{\pi, \pi'\} \subseteq proc(t)$ is supported by π' while π is hung, or
3. when an uncontrollable transition executed (which is enabled even if it belongs to a hung process),

we allow π to notify $\mathcal{T}$ that it has decided not to hang on it anymore. Moreover, $\mathcal{T}$, which acquired information about the hung processes Π, will have to forget the information about the places $own(\Pi) \setminus own(\Pi \setminus \{\pi\})$. The ability of processes to hang on a supervisor but also to progress independently before the supervisor has made any supporting choice requires some protocol between the processes and the supervisor.

Instead of having a single supervisor $\mathcal{T}$, we can use several supervisors $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$, where each supervisor $\mathcal{T}_i$ takes care of a set of processes $proc(\mathcal{T}_i)$. These sets are pairwise disjoint and do not necessarily cover all processes.

An effectively checkable criterion to determinte if at least one process or supervisor will be able to provide a progress from any nondeadlock state in G is as follows:

$$(\varphi_G \wedge \varphi_{df}) \rightarrow \big(\bigvee_{t \in \pi \in \mathcal{C}} K^w_\pi \varphi_{good(t)} \vee \bigvee_{i \in 1 \ldots k} \bigvee_{t \in \pi \in proc(\mathcal{T}_i)} K^s_{proc(\mathcal{T}_i)} \varphi_{good(t)} \big)$$

Lemma 10. *Under our transformation from a Petri Net N to an extended Petri Net N', $exec(N')\lceil_{\mathcal{C}} \subseteq stutcl(exec_I(N))$ holds.*

This is proved by induction on prefixes of the execution and using Lemma 2.

Lemma 11. *N' satisfies all stuttering invariant temporal properties of N.*

Implementing the Supervisors

Processes hang on a supervisor in some arbitrary order. The supervisor needs to decide, based on the part of the global state that it sees, whether or not there is enough information to support some transition.

Definition 24. *Let $L = \{s\lfloor_\Pi \times \Pi \mid s \in G, \Pi \subseteq \mathcal{C}\}$ denote the set of* joint local states*, each paired up with the set of relevant processes (then $G \times \mathcal{C} \subseteq L$). We define $\sqsubseteq \subseteq L \times L$ (and, symmetrically, $\sqsupseteq$) as follows: $q \sqsubseteq q'$ if $q = (s\lfloor_{\Pi_1}, \Pi_1), q' = (s\lfloor_{\Pi_2}, \Pi_2)$ (i.e., both are part of the same global state s) and $\Pi_1 \subseteq \Pi_2$. We say that q'* subsumes *q.*

Definition 25. *The support function $supp : L \to 2^T$ returns, for each $q \in L$, the transitions that are allowed by R from all states that subsume q. Formally, $supp(q) = \cap_{(s,\mathcal{C}) \sqsupseteq q}\{t \mid t \in T, (s,t) \in R\}$.*

That is, for $q = (s\lfloor_\Pi, \Pi)$, $t \in supp(q)$ iff $s \models K^s_\Pi \varphi_{good(t)}$. If $t \in supp(q) \cap ct(T)$, then the supervisor can select a process in $proc(t)$ to support t. Obviously, when $q \sqsubseteq q'$, $supp(q) \subseteq supp(q')$. There is no need for a supervisor to store in the domain of *supp* elements $q = (s\lfloor_\Pi, \Pi)$ where $|\Pi| < 2$: when $supp(q) \neq \emptyset$, the process with this local state can locally support a transition without the help of a supervisor.

Definition 26. *Let $\rightsquigarrow \subseteq L \times L$ be such that $q \rightsquigarrow q'$ if $q = (s\lfloor_\Pi, \Pi)$ and $q' = (s\lfloor_{\Pi \cup \{\pi\}}, \Pi \cup \{\pi\})$, where $\pi \notin \Pi$ (i.e., q' extends q according to exactly one process).*

The supervisor updates its view about the joint local state of the processes according to the relation $\rightsquigarrow$: when moving from q to q' by acquiring the relevant information about a new processor π; consequently, its knowledge grows and it can decide to support one of the transitions in $supp(q')$.

Definition 27. *A joint local state q is* minimal supporting *if $supp(q) \neq \emptyset$ and, for each q' such that $q' \rightsquigarrow q$, $supp(q') = \emptyset$.*

Definition 28. *The* upward closure *$\uparrow U$ of a subset of the joint local states $U \subseteq L$ is $\{q \in L \mid \exists q' \in U\, q' \sqsubseteq q\}$.*

Lemma 12. *A sufficient condition for restricting the domain $U \subseteq L$ of supp for a supervisor, without introducing new deadlocks, is that $G \times \{\mathcal{C}\} \subseteq \uparrow U$.*

Thus, there is no need to calculate and store *all* the cases of the function *supp*. This suggests the following algorithm for calculating the representation table for *supp*: perform DFS such that if $q \rightsquigarrow q'$, then q is searched before q'; backtrack when visiting q again, or when $supp(q) \neq \emptyset$. This algorithm can be used also for multiple supervisors, when restricting the search to the joint local states of $\Pi \subseteq proc(\mathcal{T}_i)$ for each $\mathcal{T}_i$.

In order to reduce the set of local states that a supervisor needs to keep in the support table, one may decide that a supervisor will not always support transitions as soon as the joint local state of the hung processes allows that. This introduces further delays in decisions, where the supervisor waits for more processes to hang even when it can already support some transitions. On the other hand, the set of supported transitions may be larger in this case, allowing more nondeterminism.

The size of the global state space of a Petri Net is in $O(2^{|P|})$. Since we need to keep also the joint local states, the size of the support table that we store in a supervisor, is in $O(2^{|P|+|\mathcal{C}|})$ (which is the size of L). However, by Lemma 12, the representation may be much more succinct. In theory, when there are no uncontrollable transitions, a (particularly slow) supervisor can avoid storing the support table, and perform the PSPACE binary search each time it needs to make a decision on a joint local state.

Control Through Temporary Interaction

The control solution suggested here makes use of (semi-)global supervisor(s) to accumulate the joint local states of several processes, when these processes cannot locally support transitions based on their weak (or past) knowledge. In [6], a solution based on temporary synchronization between the processes was suggested. Preference is given to supporting transitions locally. However, when the local knowledge is not enough to support a transition based on the local information (including the case where it is known that some other process currently has the knowledge), i.e., κ^π does not hold, the process tries to synchronize with other processes in order to achieve joint knowledge.

In order to put the solution in [6] in the context of the construction here, each process is, upon reaching a state with local information where κ^π does not hold, willing to be involved in interactions according to U. In order to implement this, each process maintains, for each local state (or, when using past knowledge, for each history), the set of joint local states that contain its local state, and where *supp* supports at least one transition τ. Upon reaching that local state, the process is willing to participate in interactions consisting of such joint local states. A successful interaction will allow firing transitions according to *supp*.

The coordination is facilitated through a protocol such as the α-core. The α-core protocol, as described in [19] contains a small error, which was automatically corrected using a genetic programming tool in [9]. Each interaction consists of exchanging of some messages, to request interaction, to allow it, to confirm the interaction or to cancel it, etc. Obviously, there is quite a lot of overhead involved.

There are advantages and disadvantages to both approaches: using a (semi-)global supervisor and using temporary syncrhonization. In particular, the latter is more flexible, as several interactions may be performed in parallel, and there is no need to commit on the distribution of processes to the semiglobal supervisors. On the other hand, it seems to require more overhead.

6 Reducing Process Hanging and Passing Responsibility

The introduction of a partial order $\succ$ on the set of processes leads to a situation, where a smaller process w.r.t. $\succ$ can avoid hanging on its supervisor if the bigger processes together can progress. Besides the advantage of reducing the number of calls to supervisors, it also allows for providing a preference to important processes, giving them an advanced access to supervisor support while reducing supervisor interaction for lesser processes significantly.

This makes use of nested knowledge, a generalization of the property κ^π to a set of processes $\kappa^\Pi \bigvee_{t\in\cup\Pi} K^s_\Pi \varphi_{good(t)}$.

The intuition is that a process can check whether it knows that the joint knowledge of the other processes, besides itself, is sufficient to support a transition, i.e., $K^w_\pi \kappa^{\mathcal{C}\setminus\{\pi\}}$. In this case, a process may decide not to hang, but to rather let the others provide the joint local state needed for making the progress decision. However, this solution makes it possible that too many processes will decide to delegate responsibility to others, without informing them. This can lead to the introduction of a deadlock.

The use of the partial order $\succ$ circumvents this problem. For a supervisor $\mathcal{T}_i$ we use $\Pi_i = proc(\mathcal{T}_i)$ to denote the processes it supervises. For a process π, we denote with $\Pi_i^{\succ\pi} = \{\pi' \in \Pi_i \mid \pi' \succ \pi\}$ the processes of Π_i that are strictly greater than π with respect to the partial order $\succ$. Naturally, a supervisor $\mathcal{T}_i$ would support some transition based on the knowledge of the processes in $\Pi_i^{\succ\pi}$ if $\kappa^{\Pi_i^{\succ\pi}}$ holds. A process π can thus idle if it knows $K^w_\pi \bigvee_{\Pi_i\in\mathcal{S}} \kappa^{\Pi_i^{\succ\pi}}$. This is used to reduce the states in which a process hangs on its supervisor.

The control strategy of the supervisors is not affected. The *ordered control strategy* is as follows:

1. If a process π knows that a transition is good, then it supports it.
2. Otherwise, if a process π knows that, for some transition $t \in \pi$, a different process knows that t is good, then π idles.
3. Otherwise, if a process π knows that, for some supervisor $\mathcal{T}_i$, the joint knowledge of $\Pi_i^{\succ\pi}$ is that some $t \in \Pi_i^{\succ\pi}$ is good, then π idles.
4. Otherwise, π hangs on its supervisor.

Ordered control does not introduce new deadlocks.

7 Conclusions

We presented simple and effective algorithms for synthesizing distributed control. The resulting control strategy uses communication and knowledge collection without blocking the processes unnecessarily. One strength of our approach is that it is complete in the sense that, provided a centralized solution exists, it finds a solution. However, this does not come at the cost of centralizing the control completely. To the contrary, the system can progress without the support of a global or regional supervisor as soon as the local information suffices to do so.

Our solution for the distributed control of systems uses knowledge to construct a distributed controller for a global constraint. In [1, 2], it is demonstrated that the local knowledge may be insufficient to construct a controller. Knowledge of perfect recall [16], which depends not only on the local state (information), but on the gathered visible history, can alleviate some, but not all, of these situations. The use of interprocess communication to obtain joint knowledge is suggested in [22]; however, no systematic algorithm for collecting such knowledge, or for evaluating when enough knowledge has been collected, was provided there. In [6], joint knowledge is calculated through temporary multiprocess synchronization. However, such synchronization is expensive, and multiple interactions (including different interactions of the same set of processes) may require a separate synchronizing process. We presented here a practical solution, based on [1, 2, 6, 10, 11] for distribute control where a small number of (or even a single) supervisor(s) run(s) concurrently with the controlled system.

While the classical synthesis problems for concurrent control of distributed systems are undecidability [20, 24, 27, 28], we relax the synthesis assumption to allow additional interactions, when needed. We believe that this makes a practical basis for synthesizing control for distributed systems. These methods were implemented [6, 10, 11]. There are various tradeoffs in the approaches presented, which calls for further experiments and tuning.

References

[1] A. Basu, S. Bensalem, D. Peled, J. Sifakis, Priority Scheduling of distributed Systems Based on Model Checking, CAV 2009, Grenoble, France, Lecture Notes in Computer Science 5643, Springer, 79-93; DOI: 10.1007/978-3-642-02658-4_10.

[2] S. Bensalem, M. Bozga, S. Graf, D. Peled, S. Quinton, ATVA 2010, Lecture Notes in Computer Science 6252, Springer, Singapore, 52-66; DOI: 10.1007/978-3-642-15643-4_6.

[3] E. M. Clarke, Synthesis of Resource Invariants for Concurrent Programs, ACM Transactions on Programming Languages and Systems 2(3), 338-358 (1980).

[4] R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, Cambridge MA, 1995.

[5] B. Finkbeiner, S. Schewe, Uniform distributed synthesis, LICS 2005, Chicago, IL, 321–330; DOI: 10.1109/LICS.2005.53.

[6] S. Graf, D. Peled, S. Quinton, Achieving Distributed Control Through Model Checking, CAV 2010, Lecture Notes in Computer Science 6174, Springer, Edinburgh, Scotland, 396-409; DOI: 10.1007/978-3-642-14295-6_35.

[7] G. Katz, D. Peled, Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms. Lecture Notes in Computer Science 5311, Springer, ATVA 2008, Seoul, Korea, 33-47; DOI: 10.1007/978-3-540-88387-6_5.

[8] G. Katz, D. Peled, Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming, Lecture Notes in Computer Science 6405, Springer, Haifa Verification Conference 2009, Haifa, Israel, 117-132; DOI: 10.1007/978-3-642-19237-1_13.

[9] G. Katz, D. Peled, Code Mutation in Verification and Automatic Code Correction, TACAS 2010, Lecture Notes in Computer Science 6015, Springer, Paphos, Cyprus, 435-450; DOI: 10.1007/978-3-642-12002-2_36.

[10] G. Katz, D. Peled, S. Schewe, Synthesis of Distributed Control through Knowledge Accumulation. CAV 2011, Lecture Notes in Computer Science 6806, Springer, Snow Bird, Utah, 510-525; DOI: 10.1007/978-3-642-22110-1_41.

[11] G. Katz, D. Peled, S. Schewe, The Buck Stops Here: Order, Chance, and Coordination in Distributed Control, ATVA 2011, Taipei, Taiwan; DOI: 10.1007/978-3-642-24372-1_31.

[12] R. M. Keller, Formal Verification of Parallel Programs, Communications of the ACM, 19, 1976, 371-384.

[13] O. Kupferman, M. Y. Vardi, Synthesizing Distributed Systems, LICS 2001, Boston, MA, 389–398.

[14] P. Madhusudan, P. S. Thaigarajan, Distributed Controller Synthesis for Local Specifications, ICALP 2001, Lecture Notes in Computer Science 2076, Springeri, Crete, Greece, 396-407; DOI: 10.1007/3-540-48224-5_33.

[15] Z. Manna, A. Pnueli, How to Cook a Temporal Proof System for Your Pet Language, POPL 1983, Austin, TX, 141–154.

[16] R. van der Meyden, Common Knowledge and Update in Finite Environment, Information and Computation, 140, 1980, 115-157.

[17] M. Mukund, M. A. Sohoni, Keeping Track of the Latest Gossip in a Distributed System. Distributed Computing 10(3): 137-148 (1997).

[18] D. Peled, Th. Wilke, Stutter-Invariant Temporal Properties are Expressible without the Text Time Operator, Information Processing Letters 63, 1997, 243–246.

[19] J. A. Pérez, R. Corchuelo, M. Toro, An Order-based Algorithm for Implementing Multiparty Synchronization, Concurrency - Practice and Experience 16(12), 2004, 1173-1206; DOI: 10.1002/cpe.903.

[20] A. Pnueli, R. Rosner, Distributed Reactive Systems are Hard to Synthesize, FOCS 1990, St. Louis, Missouri, 746-757.

[21] P. J. Ramadge, W. M. Wonham, Supervisory Control of a Class of Discrete Event Processes, SIAM journal on control and optimization, 25(1), 1987, 206–230.

[22] K. Rudie, S. L. Ricker, Know Means No: Incorporating Knowledge into Discrete-Event control systems, IEEE Transactions on Automatic Control, 45(9):1656–1668, 2000.

[23] K. Rudie, W. M. Wonham, Think Globally, Act Locally: Descentralized Supervisory Control, IEEE Transactions on Automatic Control, 37(11):1692–1708, 1992.

[24] S. Schewe, B. Finkbeiner, Synthesis of Asynchronous Systems, LOPSTR 2006, Lecture Notes in Computer Science 4407, Springer, Venice, Italy, 127–142; DOI: 10.1007/978-3-540-71410-1_10.

[25] S. Schewe, B. Finkbeiner, Distributed Synthesis for Alternating-Time Logics, ATVA 2007, Lecture Notes in Computer Science 4762, Springer, Tokyo, Japan, 268–283; DOI: 10.1007/978-3-540-75596-8_20.

[26] L. J. Stockmeyer, A. K. Chandra, Provably Difficult Combinatorial Games, SIAM Journal of Computing, 8, 1979, 151-174.

[27] J. G. Thistle, Undecidability in Descentralized Supervision, Systems and control letters 54, 503-509, 2005; DOI: 10.1016/j.sysconle.2004.10.002.

[28] S. Tripakis, Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters, 90(1):21–28, 2004; DOI: 10.1016/j.ipl.2004.01.004.

[29] T. S. Yoo, S. Lafortune, A General Architecture for Decentralized Supervisory Control of Discrete-Event Systems, Discrete event dynamic systems, theory & applications, 12(3) 2002, 335-377; DOI: 10.1023/A:1015625600613.