Distributed Controller Synthesis for Deadlock Avoidance

Hugo Gimbert
Université de Bordeaux, CNRS, France

Corto Mascle
Université de Bordeaux, France

Anca Muscholl
Université de Bordeaux, France

Igor Walukiewicz
Université de Bordeaux, CNRS, France

Abstract

We consider the distributed control synthesis problem for systems with locks. The goal is to find local controllers so that the global system does not deadlock. With no restriction this problem is undecidable even for three processes each using a fixed number of locks. We propose two restrictions that make distributed control decidable. The first one is to allow each process to use at most two locks. The problem then becomes complete for the second level of the polynomial time hierarchy, and even in PTIME under some additional assumptions. The dining philosophers problem satisfies these assumptions. The second restriction is a nested usage of locks. In this case the synthesis problem is NEXPTIME-complete. The drinking philosophers problem falls in this case.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Distributed Synthesis, Concurrency, Lock Synchronisation, Deadlock Avoidance

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.125

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version *Full Version*: `https://arxiv.org/abs/2204.12409`

Funding Work partially funded by ANR project FREDDA (ANR-17-CE40-0013).

1 Introduction

Synthesis of distributed systems has a big potential since such systems are difficult to write, test, or verify. The state space and the number of different behaviors grow exponentially with the number of processes. This is where distributed synthesis can be more useful than centralized synthesis, because an equivalent, sequential system may be very big. The other important point is that distributed synthesis produces by definition a distributed system, while a synthesized sequential system may not be implementable on a given distributed architecture. Unfortunately, very few settings are known for which distributed synthesis is decidable, and those that we know require at least exponential time.

The problem was first formulated by Pnueli and Rosner [27]. Subsequent research showed that, essentially, the only decidable architectures are pipelines, where each process can send messages only to the next process in the pipeline [20, 23, 11]. In addition, the complexity is non-elementary in the size of the pipeline. These negative results motivated the study of distributed synthesis for asynchronous automata, and in particular synthesis with so called causal information. In this setting the problem becomes decidable for co-graph action alphabets [12], and for tree architectures of processes [14, 25]. Yet the complexity is again non-elementary, this time w.r.t. the depth of the tree. Worse, it has been recently established that distributed synthesis with causal information is undecidable for unconstrained architectures [17]. Distributed synthesis for (safe) Petri nets [10] has encountered a similar line of limited advances, and due to [17], is also undecidable in the general case, since it is inter-reducible to distributed synthesis for asynchronous automata [3]. This situation raised the question if there is any setting for distributed synthesis that covers some standard examples of distributed systems, and is manageable algorithmically.

In this work we consider distributed systems with locks; each process can take or release a lock from a pool of locks. Locks are one of the most classical concepts in distributed systems. They are also probably the most frequently used synchronization mechanism in concurrent programs. We formulate our results in a control setting rather than synthesis – this avoids the need for a specification formalism. The objective is to find a local strategy for each process so that the global system does not get stuck. For unrestricted systems with locks we hit again an undecidability barrier, as for the models discussed above. Yet, we find quite interesting restrictions making distributed control synthesis for systems with locks decidable, and even algorithmically manageable.

The first restriction we consider is to limit the number of locks available to each process. The classical example are dining philosophers, where each philosopher has two locks corresponding to the left and the right fork. Observe that we do not limit the total number of processes, or the total number of locks in the system. We show that the complexity of this synthesis problem is at the second level of the polynomial hierarchy. The problem gets even simpler when we restrict it to strategies that cannot block a process when all locks are available. We call them *locally live strategies*. We obtain an NP-algorithm for locally live strategies, and even a PTIME algorithm when the access to locks is *exclusive*. This means that once a process tries to acquire a lock it cannot switch to some other action before getting the lock.

The second restriction is nested lock usage. This is a very common restriction in the literature [19], simply saying that acquiring and releasing locks should follow a stack discipline. Drinking philosophers [4] are an example of a system of this kind. We show that in this case distributed synthesis is NEXPTIME-complete, where the exponent in the algorithm depends only on the number of locks.

We formalize the distributed synthesis problem as a control problem [28]. A process is given as a transition graph where transitions can be local actions, or acquire/release of a lock. Some transitions are controllable, and some are not. A controller for a process decides which controllable transitions to allow, depending on the local history. In particular, the controller of a process does not see the states of other processes. Our techniques are based on analyzing patterns of taking and releasing locks. In decidable cases there are finite sets of patterns characterizing potential deadlocks.

The notion of patterns resembles locking disciplines [7], a tool frequently used to prevent deadlocks. An example of a locking discipline is "take the left fork before the right one" in the dining philosophers problem. Our results allow to check if a given locking discipline may result in a deadlock, and in some cases even list all deadlock-avoiding locking disciplines.

In summary, the main results of this work are:

- Σ_2^P-completeness of the deadlock avoidance control problem for systems where each process has access to at most 2 locks.
- An NP algorithm when additionally strategies need to be locally live.
- A PTIME algorithm when moreover lock access is exclusive.
- A NEXPTIME algorithm and the matching lower bound for the nested lock usage case.
- Undecidability of the deadlock avoidance control problem for systems with unrestricted access to locks.

Related work

Distributed synthesis is an old idea motivated by the Church synthesis problem [5]. Actually, the logic CTL has been proposed with distributive synthesis in mind [6]. Given this long history, there are relatively few results on distributed synthesis. Three main frameworks have been considered: synchronous networks of input/output automata, asynchronous automata, Petri games.

The synchronous network model has been proposed by Pnueli and Rosner [27]. They established that controller synthesis is decidable for pipeline architectures and undecidable in general. The undecidability result holds for very simple architectures with only two processes. Subsequent work has shown that in terms of network shape pipelines are essentially the only decidable case [20, 23, 11]. Several ways to circumvent undecidability have been considered. One was to restrict to local specifications, specifying the desired behavior of each automaton in the network separately. Unfortunately, this does not extend the class of decidable architectures substantially [23]. A further-going proposal was to consider only input-output specifications. A characterization, still very restrictive, of decidable architectures for this case is given in [13].

The asynchronous (Zielonka automata) model was proposed as a reaction to these negative results [12]. The main hope was that causal memory helps to prevent undecidability arising from partial information, since the synchronization of processes in this model makes them share information. Causal memory indeed allowed to get new decidable cases: co-graph action alphabets [12], connectedly communicating systems [24], and tree architectures [14, 25]. There is also a weaker condition covering these three cases [16]. This line of research suffered however from a very recent result showing undecidability in the general case [17].

Distributed synthesis in the Petri net model, Petri games, has been proposed recently in [10]. The idea is that some tokens are controlled by the system and some by the environment. Once again causal memory is used. Without restrictions this model is inter-reducible with the asynchronous automata model [3], hence the undecidability result [17] applies. The problem is EXPTIME-complete for one environment token and arbitrary many system tokens [10]. This case stays decidable even for global safety specifications, such as deadlock, but undecidable in general [9]. As a way to circumvent the undecidability, bounded synthesis has been considered in [8, 18], where the bound on the size of the resulting controller is fixed in advance. The approach is implemented in the tool ADAMSYNT [15].

The control formulation of the synthesis problem comes from the control theory community [28]. It does not require to talk about a specification formalism, while retaining most useful aspects of the problem. A frequently considered control objective is avoidance of undesirable states. In the distributed context, deadlock avoidance looks like an obvious candidate, since it is one of the most basic desirable properties. The survey [32] discusses the relation between the distributed control problem and Church synthesis. Some distributed versions of the control problem have been considered, also hitting the undecidability barrier very quickly [29, 31, 30, 1].

We would like to mention two further results that do not fit into the main threads outlined above. In [33] the authors consider a different synthesis problem for distributed systems: they construct a centralized controller for a scheduler that would guarantee absence of deadlocks. This is a very different approach to deadlock avoidance. Another recent work [2] adds a new dimension to distributed synthesis by considering communication errors in a model with synchronous processes that can exchange their causal memory. The authors show decidability of the synthesis problem for 2 processes.

Outline of the paper

In the next section we define systems with locks, strategies, and the control problem. We introduce locally live strategies as well as the 2-lock, exclusive, and nested locking restrictions. This permits to state the main results of the paper. The following three sections consider systems with the 2-lock restriction. First, we briefly give intuitions behind the Σ_2^p-completeness in the general case. Section 4 presents an NP algorithm for the distributed synthesis problem for locally live strategies. Section 5 gives a Ptime algorithm under the exclusive restriction. Next, we consider the nested locking case, and show that the problem is Nexptime-complete. Finally, we prove that without any restrictions the synthesis problem for systems with locks is undecidable. The full version can be found at `https://arxiv.org/4272787`.

2 Main definitions and results

A *lock-sharing system* is a distributed system with components (processes) synchronizing over locks. Processes do not communicate, but they synchronize using locks from a global pool. Some transitions of processes are uncontrollable, intuitively the environment decides if such a transition is taken. The goal is to find a local strategy for each process so that the entire system never deadlocks. The strategy can observe only local transitions – it does not see transitions performed by other processes, nor states other processes are in. While the system is finite state, the challenge comes from the locality of strategies. Indeed, the unrestricted problem is undecidable. The main contribution of this work are restrictions that make the problem decidable, and even solvable in Ptime.

In this section we define lock-sharing systems, strategies, and the deadlock avoidance control problem, that is the topic of this paper. We then introduce restrictions on the general problem and state the main decidability and complexity results.

A finite-state *process* p is an automaton $\mathcal{A}_p = (S_p, \Sigma_p, T_p, \delta_p, init_p)$ with a set of locks T_p that it can acquire or release. The transition function $\delta_p : S_p \times \Sigma_p \dot{\to} Op(T_p) \times S_p$ associates with a state from S_p and an action from Σ_p an operation on some lock and a new state; it is a partial function. The lock operations are acquire ($\texttt{acq}_t$) or release ($\texttt{rel}_t$) some lock t from T_p, or do nothing: $Op(T_p) = \{\texttt{acq}_t, \texttt{rel}_t \mid t \in T_p\} \cup \{nop\}$. Figure 1 gives an example.

A *local configuration* of process p is a state from S_p together with the locks p currently owns: $(s, B) \in S_p \times 2^{T_p}$. The initial configuration of p is $(init_p, \emptyset)$, namely the initial state with no locks. A transition between configurations $(s, B) \xrightarrow{a,op} (s', B')$ exists when $\delta_p(s, a) = (op, s')$ and one of the following holds:

- $op = nop$ and $B = B'$;
- $op = \texttt{acq}_t$, $t \notin B$ and $B' = B \cup \{t\}$;
- $op = \texttt{rel}_t$, $t \in B$, and $B' = B \setminus \{t\}$.

A *local run* $(a_1, op_1)(a_2, op_2) \cdots$ of $\mathcal{A}_p$ is a finite or infinite sequence over $\Sigma_p \times Op(T_p)$ such that there exists a sequence of configurations $(init_p, \emptyset) = (s_0, B_0) \xrightarrow{(a_1,op_1)}_p (s_1, B_1) \xrightarrow{(a_2,op_2)}_p \cdots$ While the run is determined by the sequence of actions, we prefer to make lock operations explicit. We write $Runs_p$ for the set of runs of $\mathcal{A}_p$.

A *lock-sharing system* $\mathcal{S} = ((\mathcal{A}_p)_{p\in Proc}, \Sigma^s, \Sigma^e, T)$ is a set of processes together with a partition of actions between *controllable* and *uncontrollable* actions, and a set T of locks. We have $T = \bigcup_{p\in Proc} T_p$, for the set of all locks. Controllable and uncontrollable actions belong to the system and to the environment, respectively. We write $\Sigma = \bigcup_{p\in Proc} \Sigma_p$ for the set of actions of all processes and require that (Σ^s, Σ^e) partitions Σ. The sets of states and action alphabets of processes should be disjoint: $S_p \cap S_q = \emptyset$ and $\Sigma_p \cap \Sigma_q = \emptyset$ for $p \neq q$. The sets of locks are not disjoint, in general, since processes may share locks.

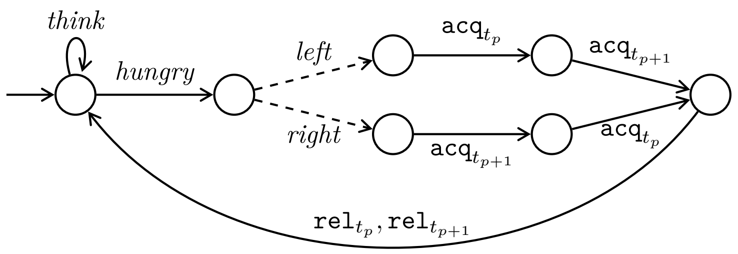

Figure 1 A dining philosopher p. Dashed transitions are controllable.

▶ **Example 1.** The dining philosophers problem can be formulated as control problem for a lock-sharing system $\mathcal{S} = ((\mathcal{A}_p)_{p\in Proc}, \Sigma^s, \Sigma^e, T)$. We set $Proc = \{1, \dots, n\}$ and $T = \{t_1, \dots, t_n\}$ as the set of locks. For every process $p \in Proc$, process $\mathcal{A}_p$ is as in Figure 1, with the convention that $t_{n+1} = t_1$. Actions in Σ^s are marked by dashed arrows. These are controllable actions. The remaining actions are in Σ^e. Once the environment makes a philosopher p hungry, she has to get both the left (t_p) and the right (t_{p+1}) fork to eat. She may however choose the order in which she takes them; actions *left* and *right* are controllable.

A *global configuration* of $\mathcal{S}$ is a tuple of local configurations $C = (s_p, B_p)_{p\in Proc}$ provided the sets B_p are pairwise disjoint: $B_p \cap B_q = \emptyset$ for $p \neq q$. This is because a lock can be taken by at most one process at a time. The initial configuration is the tuple of initial configurations of all processes.

Such systems are *asynchronous*, with transitions between two configurations done by a single process: $C \xrightarrow{(p,a,op)} C'$ if $(s_p, B_p) \xrightarrow{(a,op)}_p (s'_p, B'_p)$ and $(s_q, B_q) = (s'_q, B'_q)$ for every $q \neq p$. A global run is a sequence of transitions between global configurations. Since our systems are deterministic we usually identify a global run by the sequence of transition labels. A global run w *determines a local run* of each process: $w|_p$ is the subsequence of p's actions in w.

A *control strategy* for a lock-sharing system is a tuple of local strategies, one for each process: $\sigma = (\sigma_p)_{p\in Proc}$. A *local strategy* σ_p says which actions p can take depending on a local run so far: $\sigma_p : Runs_p \to 2^{\Sigma_p}$, provided $\Sigma^e \cap \Sigma_p \subseteq \sigma_p(u)$, for every $u \in Runs_p$. This requirement says that a strategy cannot block environment actions.

A local run u of a system *respects* σ_p if for every non-empty prefix $v(a, op)$ of u, we have $a \in \sigma_p(v)$. Observe that local runs are affected only by the local strategy. A global run w respects σ if for every process p, the local run $w|_p$ respects σ_p. We often say just σ-run, instead of “run respecting σ”.

As an example consider the system for two philosophers from Example 1. Suppose that both local strategies always say to take the *left* transition. So $\mathit{hungry}^1, \mathit{left}^1, \texttt{acq}^1_{t_1}, \texttt{acq}^1_{t_2}$ is a local run of process 1 respecting the strategy; similarly $\mathit{hungry}^2, \mathit{left}^2, \texttt{acq}^2_{t_2}, \texttt{acq}^2_{t_1}$ for process 2. (We use superscripts to indicate the process doing an action.) The global run $\mathit{hungry}^1, \mathit{hungry}^2, \mathit{left}^1, \mathit{left}^2, \texttt{acq}^1_{t_1}, \texttt{acq}^2_{t_2}$ respects the strategy and blocks, since each philosopher needs a lock the other one owns.

▶ **Definition 2** (Deadlock avoidance control problem). *A σ-run w* leads to a deadlock in σ *if w cannot be prolonged to a σ-run. A control strategy σ is* winning *if no σ-run leads to a deadlock in σ. The* deadlock avoidance control problem *is to decide if for a given system there is some winning control strategy.*

In this work we consider several variants of the deadlock avoidance control problem. Maybe surprisingly, in order to get more efficient algorithms we need to exclude strategies that can block a process by itself:

▶ **Definition 3** (Locally live strategy). *A local strategy σ_p for process p is* locally live *if every local σ_p-run u can be prolonged to a σ_p-run: there is some $b \in \Sigma_p$ such that ub is a local run respecting σ_p. A strategy σ is* locally live *if every local strategy is so.*

In other words, a locally live strategy guarantees that a process does not block if it runs alone. Coming back to Example 1: a strategy always offering one of the *left* or *right* actions is locally live. A strategy that offers none of the two is not. Observe that blocking one process after the hungry action is a very efficient strategy to avoid a deadlock, but it is not the intended one. This is why we consider locally live to be a desirable property rather than a restriction.

Note that being locally live is not exactly equivalent to a strategy always proposing at least one outgoing transition. In our semantics, a process blocks if it tries to acquire a lock that it already owns, or to release a lock it does not own. But it becomes equivalent thanks to the following remark:

▶ Remark 4. We can assume that each process keeps track in its state which locks it owns. Note that this assumption does not compromise the complexity results when the number of locks a process can access is fixed. We will not use this assumption in Section 6, where a process can access arbitrarily many locks (in nested fashion).

Without any restrictions our synthesis problem is undecidable.

▶ **Theorem 5.** *The deadlock avoidance control problem for lock-sharing systems is undecidable. It remains so when restricted to locally live strategies.*

We propose two cases when the control problem becomes decidable. The two are defined by restricting the usage of locks.

▶ **Definition 6** (2LSS). *A process $\mathcal{A}_p = (S_p, \Sigma_p, T_p, \delta_p, init_p)$* uses two locks *if $|T_p| = 2$. A system $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$ is 2LSS if every process uses two locks.*

Note that in the above definition we do not bound the total number of locks in the system, just the number of locks per process. The process from Figure 1 is 2LSS. Our first main result says that the control problem is decidable for 2LSS.

▶ **Theorem 7.** *The deadlock avoidance control problem for 2LSS is Σ_2^P-complete.*

For the lower bound we use strategies that take a lock and then block. This does not look like a very desired behavior, and this is the reason for introducing the concept of locally live strategies. The second main result says that restricting to locally live strategies helps.

▶ **Theorem 8.** *The deadlock avoidance control problem for 2LSS is in* NP *when strategies are required to be locally live.*

We do not know if the above problem is in PTIME. We can get a PTIME algorithm under one more assumption.

▶ **Definition 9** (Exclusive systems). *A process p is* exclusive *if for every state $s \in S_p$: if s has an outgoing transition with some $\mathtt{acq}_t$ operation then all outgoing transitions of s have the same $\mathtt{acq}_t$ operation. A system is exclusive if all its processes are.*

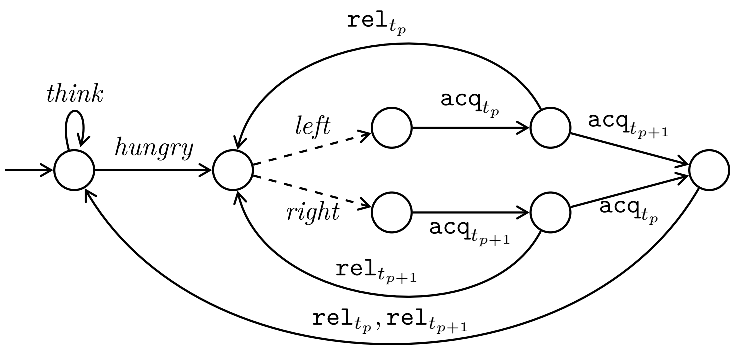

Figure 2 A flexible philosopher p. She can release a fork if the other fork is not available.

▶ **Example 10.** The process from Figure 1 is exclusive, while the one from Figure 2 is not. The latter has a state with one $\mathtt{acq}_{t_{p+1}}$ and one $\mathtt{rel}_{t_p}$ outgoing transition. Observe that in this state the process cannot block, and has the possibility to take a lock at the same time. Exclusive systems do not have such a possibility, so their analysis is much easier.

▶ **Theorem 11.** *The deadlock avoidance control problem for exclusive 2LSS is in* PTIME*, when strategies are required to be locally live.*

Without local liveness, the problem stays Σ_2^p-hard for exclusive 2LSS. Our last result uses a classical restriction on the usage of locks:

▶ **Definition 12** (Nested-locking). *A local run is* nested-locking *if the order of acquiring and releasing locks in the run respects a stack discipline, i.e., the only lock a process can release is the last one it acquired. A local strategy is nested-locking if all local runs respecting the strategy are nested-locking. A strategy is nested-locking if all local strategies are nested-locking.*

The process from Figure 1 is nested-locking, while the one from Figure 2 is not.

▶ **Theorem 13.** *The deadlock avoidance control problem is* NEXPTIME*-complete when strategies are required to be nested-locking.*

3 Two locks per process

We give some intuitions as to why the deadlock avoidance problem for 2LSS is Σ_2^p-complete (Theorem 7).

When every process uses only two locks there are only few patterns of local lock usage that are relevant for deadlocks. A finite local run u of process p using locks t_1, t_2 can be of one of the following four types:

- p owns both locks at the end of u;
- p owns no lock at the end of u;
- p owns only one lock, say t_1, at the end of u, and the last lock operation of u is $\mathtt{acq}_{t_1}$;
- p owns only one lock, say t_1, at the end of u, and the last lock operation of u is $\mathtt{rel}_{t_2}$.

A *pattern* of a run is its type, and the set of available actions at the end. If a run reaches a deadlock then the only available actions are to acquire locks owned by other processes.

We fix a 2LSS $(\{\mathcal{A}_p\}_{p\in Proc}, \Sigma^s, \Sigma^e, T)$ over the set of processes *Proc*. We assume that it satisfies Remark 4.

Given a strategy $\sigma = (\sigma_p)_{p\in Proc}$, we call a local σ-run *risky* if it ends in a state from which every outgoing action allowed by σ acquires some lock (this includes states with no outgoing transition). A local σ-run is *neutral* if it ends in a configuration (s, B) with $B = \emptyset$.

▶ **Definition 14.** *We define the* pattern *of a* ***risky*** *σ_p-run u_p as follows. Let T_{owns} be the set of locks that p owns after executing u_p and T_{blocks} the set of locks that outgoing transitions allowed by σ_p after u_p need to acquire.*

The pattern of u_p is the tuple $(T_{owns}, T_{blocks}, ord)$ with:

- *If u_p is of the form $u_1(a, \mathtt{acq}_{t_1})u_2(b, \mathtt{rel}_{t_2})u_3$ with no action on t_1 in u_2 and no action on either t_1 or t_2 in u_3 then $ord = (t_1, t_2)$.*
- *Otherwise $ord = \perp$.*

Note that in light of Remark 4, T_{owns} and T_{blocks} are necessarily disjoint. Furthermore if ord is of the form (t_1, t_2) then $T_{owns} = \{t_1\}$, and either $T_{blocks} = \emptyset$ or $T_{blocks} = \{t_2\}$.

A strategy $\sigma = (\sigma_p)_{p\in Proc}$ respects *a family of sets of patterns $(Patt_p)_{p\in Proc}$ if for all $p \in Proc$, the patterns of all risky σ_p-runs belong to $Patt_p$.*

In this definition, T_{owns} and T_{blocks} serve as witnesses of deadlock configurations, in which all required locks are owned by another process, and no lock is owned by two different processes. Further, the *ord* component indicates the fourth case described before the definition.

Our key result in this part is Lemma 15. It gives simple, necessary and sufficient, conditions on the family of patterns of local σ-runs $(u_p)_{p\in Proc}$ that lead to a deadlock under a suitable scheduling. The difficulty is to verify if there exists a global run which is a combination of those local runs. For that, all processes must own disjoint sets of locks at the end. The rest can be inferred from the types of runs listed above.

We describe how to schedule local runs into a global one depending on the four types listed before Definition 14.

- In the first case we can assume that p's run is scheduled at the end of the global run, as it ends up keeping both locks anyway, so no other process will use them after p.
- In the second case, we can assume that p's run is scheduled at the beginning of the global run, as it is neutral.
- In the third case, we can split p's run in two parts: a first, neutral part which can be scheduled at the beginning, and a second part in which p acquires t_1 and there is no lock operation afterwards. The second part can be scheduled at the end, because no other process will use t_1 after p.
- In the final case, p acquires t_1, never releases it but later uses t_2. This can be a problem if for instance another process does the same with t_1 and t_2 reversed. The first process that takes its first lock would prevent the other from finishing its local run. We express these constraints by requiring the existence of a global order in which process take locks without releasing them.

▶ **Lemma 15.** *Let $\sigma = (\sigma_p)_{p\in Proc}$ be a control strategy. For all p let $Patt_p$ be the set of patterns of local risky σ_p-runs of p. The control strategy σ is* ***not*** *winning if and only if there exists for each p a pattern $(T^p_{owns}, T^p_{blocks}, ord_p) \in Patt_p$ such that:*

- *$\bigcup_{p\in Proc} T^p_{blocks} \subseteq \bigcup_{p\in Proc} T^p_{owns}$,*
- *the sets T^p_{owns} are pairwise disjoint,*
- *there exists a total order $\leq$ on T such that for all p, if $ord_p = (t, t')$ then $t \leq t'$.*

Proof. Suppose σ is not winning, let u be a run ending in a deadlock. For each process p let u_p be the corresponding local run. The local run u_p is risky, as otherwise u_p could be extended in a longer run consistent with σ. Thus u_p has a pattern $(T^p_{owns}, T^p_{blocks}, ord_p) \in Patt_p$.

We check that those patterns $(u_p)_{p\in Proc}$ meet the requirements of the lemma. Clearly as we are in a deadlock, all locks that some process wants are taken, hence the first condition is satisfied. Furthermore, no two processes can own the same lock, implying the second condition. Finally, let $\leq$ be a total order on locks given by the order of the last operations on each lock in u: we set $t \leq t'$ iff the last operation on t in u is before the last one on t'. Let p be a process, and suppose ord_p is (t, t'). Then u_p has the form $u_1(a, \mathtt{acq}_t)u_2(b, \mathtt{rel}_{t'})u_3$ with no action on t in u_2 or u_3. Hence, $t \leq t'$.

The other direction is a bit more complicated. Suppose that for each p there is a pattern $(T^p_{owns}, T^p_{blocks}, ord_p) \in Patt_p$ such that those patterns satisfy all three conditions of the lemma. Let $\leq$ be a suitable total order on locks for the third condition, and let $<$ be its strict part. For every p there exists a risky local run u_p yielding the chosen pattern for p.

We start by executing all neutral runs u_p one by one in some order. All locks are free after these executions.

For all p such that $T^p_{owns} = \{t\}$ and $ord_p = \perp$, we can decompose u_p as $u_1(a, \mathtt{acq}_t)u_2$ with no action on locks in u_2. We execute all runs u_1, which are neutral and thus leave all locks free after execution.

Finally, we execute all u_p such that $ord_p \neq \perp$ in increasing order on the first component of ord_p according to $\leq$. For all such p, let $(t, t') = ord_p$, so we have $T^p_{owns} = \{t\}$ and $t < t'$. As all T^p_{owns} are disjoint, before executing u_p all locks greater or equal to t according to $\leq$ are free. In particular, t and t' are free, thus we can execute u_p. In the end all locks are free except the ones belonging to T^p_{owns} for those processes p.

Now we execute the remaining part of the u_p with $T^p_{owns} = \{t\}$ and $ord_p = \perp$ (referred to as $(a, \mathtt{acq}_t)u_2$ before). Those runs do not contain any action on locks besides the first acquire. As all T^p_{owns} are disjoint, the locks they acquire are free, hence all those runs can be executed.

The remaining runs are the ones such that $T^p_{owns} = \{t, t'\}$. As all T^p_{owns} are disjoint, both these locks are free, hence u_p can be executed as p can only use these two locks.

We have combined all local runs into one global run reaching a configuration where all processes have to acquire a lock from $\bigcup_{p\in Proc} T^p_{blocks}$ to keep running, and all locks in $\bigcup_{p\in Proc} T^p_{owns}$ are taken. As $\bigcup_{p\in Proc} T^p_{blocks} \subseteq \bigcup_{p\in Proc} T^p_{owns}$, we have reached a deadlock. ◀

The algorithm for Theorem 7 proceeds in four phases:

- guess a set of patterns $Patt_p$, one for each process p,
- check that there are local strategies σ_p such that the patterns of all runs belong to $Patt_p$,
- let the adversary guess a pattern in each $Patt_p$,
- check whether those patterns satisfy the conditions of Lemma 15.

The alternation between guessing and adversarial guessing yields a Σ^p_2 algorithm.

The lower bound is obtained by a reduction from ∃∀-SAT. The system controls existential variables, the environment controls universal ones. There are two locks for each variable, acquiring one of them is interpreted as choosing the value of the variable. Note that this construction relies on processes that take a lock and then block on their own in states with no outgoing transitions. In the following section we will forbid such unnatural behavior by considering only locally live strategies.

We use some extra processes to enforce that the system wins if and only if the valuation given by the choices of the two players satisfies the SAT formula. The interesting part is that even though it looks like the guessing values of variables is done concurrently by the system and the environment, the whole setting enforces a ∃∀ dependency.

4 Two locks per process with locally live strategies

We describe how to solve the control problem for 2LSS and locally live strategies in NP, as stated in Theorem 8.

We fix a 2LSS satisfying the assumption discussed in Remark 4. We will show that the relevant information about a strategy σ can be formalized as a finite lock graph G_σ and a lockset family Locks_σ; the latter is a family of sets of sets of locks (see definitions below). This information is very similar to the one described by patterns in the previous section. As we work with locally live strategies, the set of possible patterns of local runs is more restricted and we can view this more conveniently as a graph.

Our algorithm first guesses an abstract lock graph G and lockset family Locks. Then it performs two checks:

Step 1 check if there is some strategy σ with $G = G_\sigma$ and $\mathit{Locks} = \mathit{Locks}_\sigma$, and
Step 2 check if there is no deadlock scheme for G and Locks (see Definition 21 below).
A deadlock scheme is some kind of forbidden situation. It is easy to get a co-NP algorithm for the second step: just guess the scheme and check that it has the right shape. The challenge is to do this in PTIME. This is necessary if we want to get an NP algorithm.

We introduce now some notions in order to define G_σ and Locks_σ conveniently. Consider a local run u of a process p:

$$(\mathit{init}_p, \emptyset) = (s_0, B_0) \xrightarrow{(a_1, op_1)}_p (s_1, B_1) \cdots \xrightarrow{(a_i, op_i)}_p (s_i, B_i) \ .$$

We say that u *has set of locks* B if $B = B_i$. A σ_p-run u is B*-locked* by the local strategy σ_p if every transition in $\sigma_p(u)$ has as operation $\mathtt{acq}_t$ for some $t \in B$. Process p is B*-lockable* by σ_p if it has a neutral, B-locked σ_p-run.

The intuition is that in order to get a deadlock, a B-lockable process can be scheduled first. It can do a run leading to a state where it requires some of the locks in B without holding any locks. So, the process will be blocked if we ensure that all locks in B are already taken. For example, consider the process in Figure 1. The run *hungry*, *left* is $\{t_p\}$-locked, as the unique next action is $\mathtt{acq}_{t_p}$. The process is $\{t_p\}$-lockable by σ_p if e.g. σ_p always chooses the *left* action. Indeed, in this case the run *hungry*, *left* is a neutral σ_p-run, which is $\{t_p\}$-locked. Process p is not $\{t_{p+1}\}$-lockable by a strategy σ_p choosing always the *left* action, as there is no neutral σ_p-run leading to $\mathtt{acq}_{t_{p+1}}$.

▶ **Definition 16** (Lockset family Locks_σ). *A* lockset *for a local strategy σ_p is a set $L_p \subseteq 2^{T_p}$ of sets B such that p is B-lockable by σ_p. A* lockset family *for σ is $\mathit{Locks}_\sigma = (L_p)_{p \in \mathit{Proc}}$.*

▶ **Definition 17** (Lock graph G_σ). *For a strategy σ, a* lock graph *$G_\sigma = \langle T, E_\sigma \rangle$ has an edge $t_1 \xrightarrow{p} t_2$ whenever there is some σ_p-run u of p that has $\{t_1\}$ and is $\{t_2\}$-locked. If there is such a run u where the last lock operation in u is $\mathtt{acq}_{t_1}$ then the edge is called* green*, and otherwise it is called* blue.

We will say that σ allows *a blue edge $t_1 \overset{p}{\hookrightarrow} t_2$ or a green edge $t_1 \overset{p}{\mapsto} t_2$. We write $t_1 \xrightarrow{p} t_2$ when the color of the edge is irrelevant.*

For example, a strategy choosing the *left* action in Figure 1 yields the green edge $t_p \overset{p}{\mapsto} t_{p+1}$. Lockset families say on which sets of locks each process can block while not holding any lock. An edge $t_1 \xrightarrow{p} t_2$ in the lock graph corresponds to a run of p where P owns lock t_1 (the source of the edge) and waits for the other lock t_2 (the target of the edge).

A lockset represents a run of the second type in the previous section, a green edge a run of the third type, and a blue edge a run of the fourth type with no similar run of the third type. The first type cannot appear in a deadlock when strategies are locally live, as processes always have an available action.

Since we have assumed nothing about how strategies are given, it is not clear how to compute G_σ. Instead of restricting to, say, finite memory strategies, we will work with arbitrary lock graphs and lockset families. This is possible thanks to Lemma 19 below, that allows to check if a graph is the lock graph of some strategy. For this we need to define lockset families and lock graphs abstractly. Notice that the size of both these objects is bounded, as the set of locks per process is fixed for 2LSS.

▶ **Definition 18.** *A* lockset family *is a tuple of sets of locks indexed by processes* $(L_p)_{p\in Proc}$*, with* $L_p \subseteq 2^{T_p}$*. A* lock graph *is an edge-labeled graph* $G = \langle T, E \subseteq T \times Proc \times \{blue, green\} \times T\rangle$ *where nodes are locks from the set* T *and every edge is labeled by a process and a color. A cycle in* G *is called* proper *if all its edges are labeled by different processes. It is denoted as* green *if it contains at least* ***one*** *green edge; otherwise, so if* ***all*** *edges are blue, it is denoted* blue*.*

At this point we have enough notions to carry out the first step on page 10.

▶ **Lemma 19.** *Given a lock graph* G *and a lockset family* $Locks$*, it is decidable in* PTIME *if there is a locally live strategy* σ *such that* $G = G_\sigma$ *and* $Locks_\sigma = Locks$*.*

The proof is by reduction to model-checking a fixed-size MSOL formula over a given regular tree. For every process p we need to check if there is a local strategy σ_p satisfying the conditions imposed by G and $Locks = (L_p)_{p\in Proc}$. Consider the regular tree of all local runs of process p. The formula says that there is a strategy tree inside this regular tree such that L_p contains exactly those sets B such that the subtree has some neutral, B-locked path; and for every edge in G labelled by p there is a path of the required shape in the subtree. This can be expressed by an MSOL formula of constant size, as the process uses only 2 locks. From the MSOL formula we get a tree automaton of constant size. The emptiness check of its product with the tree automaton accepting the unfolding of the automaton $\mathcal{A}_p$ can be done in PTIME.

In the rest of the section we discuss the second step. We first define a Z-deadlock scheme for some set Z of locks. Intuitively, this is a situation showing that there is a run blocking all locks in Z. Then a deadlock scheme is a Z-deadlock scheme for some Z big enough to block all processes.

▶ **Definition 20** (Z-deadlock scheme)**.** *Let* $G = \langle T, E\rangle$ *be a lock graph,* $Locks = (L_p)_{p\in Proc}$ *a lockset family, and* Z *a set of locks. We define* $Proc_Z$ *as the set of processes whose both accessible locks are in* Z*,* $Proc_Z = \{p \in Proc : T_p \subseteq Z\}$*. A* Z-deadlock scheme *is a function* $ds_Z : Proc_Z \to E \cup \{\perp\}$ *such that:*

- *For all* $p \in Proc_Z$*, if* $ds_Z(p) \neq \perp$ *then* $ds_Z(p)$ *is an edge of* G *labeled by* p*.*
- *If* $p \in Proc_Z$ *and* $L_p = \emptyset$ *then* $ds_Z(p) \neq \perp$*.*
- *For all* $t \in Z$ *there exists a unique* $p \in Proc_Z$ *such that* $ds_Z(p)$ *is an outgoing edge from* t*.*
- *The subgraph of* G*, restricted to* $ds_Z(Proc_Z)$ *does not contain any blue cycle.*

The main point of this definition is that for every lock in Z there is an outgoing edge in ds_Z. Intuitively, it means that we have a run where every lock from Z is taken, and every process in $Proc_Z$ requires a lock from Z.

▶ **Definition 21** (Deadlock scheme)**.** *A* deadlock scheme *for* G *and* $Locks = (L_p)_{p\in Proc}$ *is a* Z*-deadlock scheme such that for every process* $p \in Proc \setminus Proc_Z$ *there is* $B \in L_p$ *with* $B \subseteq Z$*.*

Thus a deadlock scheme represents a situation where all processes are blocked, since every process not in $Proc_Z$ can be brought into a state where it needs a lock from Z, but all these locks are taken.

The next lemma says that the absence of deadlock schemes characterizes winning strategies. We could reuse the patterns defined above to obtain a shorter proof but we prefer to give a slightly longer but elementary one.

▶ **Lemma 22.** *A locally live control strategy σ is winning if and only if there is no deadlock scheme for its lock graph G_σ and its lockset family $Locks_\sigma$.*

Proof. Suppose σ is not winning. Then there exists a global σ-run u leading to a deadlock. As a consequence, in the deadlock configuration all processes must be trying to acquire some lock that is already taken.

We then construct a deadlock scheme (BT, ds) as follows. Let BT be the set of locks taken in the deadlock configuration, and for all $p \in Proc$, define $ds(p)$ as:

- $\bot$ if p does not own any lock in the deadlock configuration,
- $t_1 \xrightarrow{p} t_2$ if p owns t_1 and is trying to acquire t_2 in the deadlock configuration (the color of the edge is determined by the run, it is irrelevant for the argument).

Clearly for all $p \in Proc$ the value $ds(p)$ is either $\bot$ or a p-labeled edge of the lock graph G_σ.

Suppose $ds(p) = \bot$, and let t_1, t_2 be the two locks accessible by p. As the final configuration is a deadlock, all actions allowed by σ_p are necessarily $\mathtt{acq}_{t_1}$ or $\mathtt{acq}_{t_2}$. So p is $\{t_1, t_2\}$-lockable. Furthermore, as we are in a deadlock, the lock(s) blocking p are in BT (if they were free, p would be able to advance), therefore p is BT-lockable.

For every $t \in BT$, there is a process p holding t in the final configuration. As we are in a deadlock, p is trying to acquire its other accessible lock t' (recall that the definition of control strategy demands that at least one action be available to each process at all times). Thus $ds(p)$ is an edge from t to t'. Furthermore t' cannot be free as we are in a deadlock, thus $t' \in BT$. There are no other outgoing edges from t as no other process can hold t while p does.

Finally let $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1}$ be a cycle with $t_1 = t_{k+1}$ in the subgraph of G_σ restricted to BT and $ds(Proc)$. One of the locks t_i was the last lock taken in the run u (say by process p_i). We show now by contradiction that the edge $t_i \xrightarrow{p_i} t_{i+1}$ is green. If p_i would have released t_{i+1} after the last $\mathtt{acq}_{t_i}$ in u, then p_{i+1} would have done its last $\mathtt{acq}_{t_{i+1}}$ later, a contradiction. The subgraph of G_σ restricted to BT and $ds(Proc)$ has therefore no blue cycles, therefore (BT, ds) is a deadlock scheme.

For the other direction, suppose we have a deadlock scheme (BT, ds) for the lock graph G_σ. As $(BT, ds(Proc))$ does not contain a blue cycle, we can pick a total order $\leq$ on locks such that for all blue edges $t_1 \overset{p}{\hookrightarrow} t_2 \in ds(Proc)$, we have $t_1 \leq t_2$.

By definition of the lock graph, for each process $p \in Proc$ we can take a local run u_p of $\mathcal{A}_p$ respecting σ with the following properties.

- If $ds(p) = \bot$ then p is BT-lockable. So there exists a neutral run u_p leading to a state where all outgoing transitions require locks from BT.
- If $ds(p) = t_p^1 \xrightarrow{p} t_p^2$ then there is u_p of the form $u_p^1(a, \mathtt{acq}_{t_p^1})u_p^2(a', \mathtt{acq}_{t_p^2})$ without $\mathtt{rel}_{t_p^1}$ transition in u_p^2. Moreover if $ds(p)$ is green then we know that there is no $\mathtt{rel}_{t_p^2}$ transition in u_p^2.

We now combine these runs to get a run respecting σ ending in a deadlock configuration. For each process p such that $ds(p) = \bot$, execute the local run u_p. Since u_p is neutral, all locks are available after executing it. The only possible actions of p after this run are to acquire some locks from BT.

Next, for every process p such that $ds(p)$ is a green edge, execute the local run u_p^1. This is also a neutral run. After this run p is in a state where σ_p allows to take lock t_p^1, but p does not own any lock.

Next, in increasing order according to $\leq$, for every lock t with an outgoing blue edge $ds(p) = t \overset{p}{\hookrightarrow} t'$ execute the run u_p, except for the last $\mathtt{acq}_{t'}$ action. After this run lock t is taken by p, and all actions allowed by σ_p are $\mathtt{acq}_{t'}$ actions. Since there is only one outgoing edge from every lock, and since we are respecting the order $\leq$, both t and t' are free before executing that run. Hence it is possible to execute this run.

Finally, we come back to processes p such that $ds(p)$ is a green edge. For every such process we execute $\mathtt{acq}_{t_p^1}$ followed by u_p^2. This is possible because t_p^1 is free as there is a unique outgoing edge from t_p^1. After executing these runs every process p with $ds(p) \neq \bot$ is in a state when the only possible action is $\mathtt{acq}_{t_p^2}$.

At this stage all locks that are sources of edges from $ds(Proc)$ are taken. Since every lock in BT is a source of an edge, all locks from BT are taken. Thus no process p with $ds(p) = \bot$ can move as it needs some lock from BT. Similarly, no process p with $ds(p) \neq \bot$ can move, as they need locks pointed by targets of the edges $ds(p)$, and these are in BT too. So we have constructed a run respecting σ and reaching a deadlock. ◀

From now on we concentrate on deciding if there is some deadlock scheme for a given graph G along with a lockset family *Locks*. Our approach will be to repeatedly eliminate edges from G or add locks to Z, and construct a deadlock scheme on Z at the same time.

As a preparatory step we observe that we can almost ignore the lockset family. Examining the definition of Z-deadlock scheme we see that the only information about *Locks* it uses is whether $L_p = \emptyset$ or not. Hence we call a process *solid* if $L_p = \emptyset$, and *fragile* otherwise. The second condition in the definition of Z-deadlock scheme becomes: if $p \in Proc_Z$ is solid then $ds_Z(p) \neq \bot$.

The next lemma gives an important composition principle for deadlock schemes. Suppose we already have a set of "kernel" locks Z on which we know how to construct a Z-deadlock scheme. Then the lemma says that in order to get a deadlock scheme for G it is enough to consider the remaining part $G \setminus Z$.

▶ **Lemma 23.** *Let $Z \subseteq T$ be such that there is no edge labeled by a solid process from a lock of Z to a lock of $T \setminus Z$ in G. Suppose $ds_Z : Proc_Z \to E \cup \{\bot\}$ is a Z-deadlock scheme. Then there is a deadlock scheme for G if and only if there is one equal to ds_Z over $Proc_Z$.*

The rest of the proof is a sequence of stages. We start with $H = G$ and $Z = \emptyset$. At each stage we remove some edges in H or extend Z. This process continues till some obstacle to the existence of a deadlock scheme is found, or till Z is big enough to be a deadlock scheme. We use three invariants:

▶ **Invariant 1.** *G admits a deadlock scheme if and only if H does.*

▶ **Invariant 2.** *There are no edges labeled by a solid process from Z to $T \setminus Z$ in H.*

▶ **Invariant 3.** *There exists a Z-deadlock scheme.*

▶ **Proposition 24.** *There is a polynomial time algorithm to decide if a lock graph G and a lockset family Locks have a deadlock scheme.*

The final argument behind Theorem 8 is as follows. We start by non-deterministically guessing G and *Locks*. These are of polynomial size with respect to the size of the 2LSS. We can check in polynomial time that there exists a strategy σ giving G and *Locks* (Lemma 19). If that is not the case, we reject the input. Otherwise we check if G and *Locks* admit a deadlock scheme (Proposition 24). By Lemma 22, the strategy σ is winning if and only if the check says that there is no deadlock scheme in G and *Locks*.

5 Solving the exclusive case in Ptime

In this section we study exclusive 2LSS. We have shown an NP algorithm for the deadlock avoidance control problem when restricting to locally live strategies. Here we show that the problem is in Ptime if the 2LSS is exclusive (Definition 9). This is possible because the exclusive assumption simplifies the structure of lock graphs, and makes the lockset family unnecessary.

Throughout this section we fix an exclusive 2LSS, call it $\mathcal{S}$. The exclusive property prohibits situations as in Figure 2 where a state has one outgoing $\mathtt{acq}_{t_{p+1}}$ transition, and one $\mathtt{rel}_{t_p}$ transition. Compared to the previous section we do not need to make a difference between solid and fragile processes. We can even ignore colors on the arrows. This is a consequence of the following two lemmas.

▶ **Lemma 25.** *Let σ be a locally live control strategy and G_σ its lock graph. For all $t_1, t_2 \in T$, if G_σ has a blue edge $t_1 \overset{p}{\hookrightarrow} t_2$ then it has a green edge $t_2 \overset{p}{\mapsto} t_1$.*

▶ **Lemma 26.** *Let σ be a locally live control strategy and G_σ its lock graph. For every edge $t_1 \overset{p}{\to} t_2$ in G, process p is $\{t_1, t_2\}$-lockable.*

Thanks to these simplifications there is a much more direct way of checking if a strategy is winning. Take a locally live strategy σ. Consider a decomposition of G_σ into strongly connected components (SCC). We say that an SCC is a *direct deadlock* if it contains at least two nodes, and:

- either it has an edge that is not a double edge: $t_1 \xrightarrow{p} t_2$ but not $t_1 \xleftarrow{p} t_2$, for some p;
- or all edges in the component are double edges and there is a proper cycle, i.e., all edges are labeled by different processes.

A *deadlock* SCC is a direct deadlock SCC or an SCC that can reach some direct deadlock SCC. Let BT_σ be the set of all the locks appearing in some deadlock SCC. We obtain a simple characterization of winning strategies.

▶ **Proposition 27.** *A strategy σ is winning if and only if there exists a process that is not BT_σ-lockable.*

Building on this result we can give a method to decide if there is a winning strategy in the system $\mathcal{S}$. For every process p and every set of edges between two locks of p we check if there is a local strategy inducing exactly these edges. This can be done in a similar way as Lemma 19. We say that an edge la belled by p is *unavoidable* if all the local strategies σ_p induce this edge. Let $G_{\mathcal{S}}$ be the graph whose nodes are locks and edges are unavoidable edges.

We calculate a set $BT_{\mathcal{S}}$ in a similar way as BT_σ in the previous proposition except that we use slightly more general basic SCCs of $G_{\mathcal{S}}$. A *direct semi-deadlock SCC* is either a direct deadlock SCC or an SCC containing at least two nodes, only double edges, and two locks t_1 and t_2 such that for some process p not inducing a double edge between t_1, t_2 in $G_{\mathcal{S}}$: every strategy for p induces at least one edge between t_1 and t_2. Then a *semi-deadlock SCC* is an SCC that can reach some direct semi-deadlock SCC, or is itself a direct semi-deadlock SCC.

Let $BT_{\mathcal{S}}$ be the set of locks appearing in semi-deadlock SCCs of $G_{\mathcal{S}}$. Theorem 11 follows from the next proposition.

▶ **Proposition 28.** *Let $\mathcal{S}$ be an exclusive 2LSS. There is a winning locally live strategy for the system if and only if there exists a locally live strategy σ_p for some process p preventing it from acquiring any lock from $BT_{\mathcal{S}}$.*

The algorithm computes $BT_{\mathcal{S}}$, and then checks if for some process p the condition from the proposition holds. This check amounts to solving a safety game on a finite graph – the transition graph of process p.

6 Nested-locking strategies

We switch to another decidable case, where we require that locks are acquired and released in stack-like manner. Our goal is Theorem 13 saying that the deadlock avoidance control problem is NEXPTIME-complete when restricted to nested-locking strategies (cf. Definition 12).

In the context of this section we cannot assume that a process knows which locks it has (cf. Remark 4). In consequence, it is not realistic to require that a strategy is locally live. Yet, the lower bound works also for locally live strategies.

We will use some notions about local runs as defined on page 10.

▶ **Definition 29.** *A* stair decomposition *of a local run u is*

$$u = u_1 \mathtt{acq}_{t_1} u_2 \mathtt{acq}_{t_2} \dots u_k \mathtt{acq}_{t_k} u_{k+1}$$

where in the configuration reached by $u_1 \mathtt{acq}_{t_1} u_2 \mathtt{acq}_{t_2} \dots u_i$ the set of locks held by the process is $\{t_1, \dots, t_{i-1}\}$ for every $i > 0$, and there is no operation on t_i in $u_{i+1} \dots u_{k+1}$. (We omit the actions associated with each operation as they are irrelevant here).

Every nested-locking run has a unique stair decomposition.

Without the locally live assumption we may have runs simply ending because there are no outgoing actions. Recall that given a strategy σ, a *risky σ-run* is a local σ-run ending in a state from which every outgoing action allowed by σ acquires some lock. We define patterns of risky local runs that will serve as witnesses of reachable deadlocks.

▶ **Definition 30.** *Consider a stair decomposition $u_1 \mathtt{acq}_{t_1} u_2 \mathtt{acq}_{t_2} \cdots u_k \mathtt{acq}_{t_k} u_{k+1}$ of a risky σ-run u of a process p. Suppose the run is T_{blocks}-blocked, and let $T_{owns} = \{t_1, \dots, t_k\}$. We associate with u a* stair pattern *$(T_{owns}, T_{blocks}, \preceq)$, where $\preceq$ is the smallest partial order on the set T_p of locks of p satisfying: for all i, for all $t \in T_p$, if the last operation on t in the run is after the last $\mathtt{acq}_{t_i}$ then $t_i \preceq t$. A* behavior of σ *is a family of sets of stair patterns $(\mathcal{P}_p)_{p \in Proc}$, where $\mathcal{P}_p$ is the set of stair patterns of local risky σ-runs of p.*

Similarly to Lemma 22 we can show that the family of patterns for a strategy determines if it is winning.

▶ **Lemma 31.** *A nested-locking control strategy σ with behavior $(\mathcal{P}_p)_{p \in Proc}$ is **not** winning if and only if for every $p \in Proc$ there is a stair pattern $(T^p_{owns}, T^p_{blocks}, \preceq^p) \in \mathcal{P}_p$ such that:*
- *$\bigcup_{p \in Proc} T^p_{blocks} \subseteq \bigcup_{p \in Proc} T^p_{owns}$,*
- *the sets T^p_{owns} are pairwise disjoint,*
- *there exists a total order $\preceq$, on the set of all locks T, compatible with all $\preceq^p$.*

Similarly to Lemma 19 we can check if there is a strategy whose set of patterns has only patterns from a given family. Observe that the depth of nesting is bounded by the number of locks.

▶ **Lemma 32.** *Given a lock-sharing system $((\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$, a process $p \in Proc$ and a set of patterns $\mathcal{P}_p$, we can check in polynomial time in $|\mathcal{A}_p|$ and $2^{|T|}$ whether there exists a nested-locking local strategy σ_p with set of patterns included in $\mathcal{P}_p$.*

▶ **Proposition 33.** *The deadlock avoidance control problem is decidable for lock-sharing systems with nested-locking strategies in non-deterministic exponential time.*

Proof. The decision procedure guesses a set of patterns $\mathcal{P}_p$ for each process p, of size at most $2^{2|T|}|T|! \leq 2^{O(|T|\log(|T|)}$. Then it checks if there exist local strategies yielding subsets of those sets of patterns. This takes exponential time by Lemma 32. If the result is negative then the procedure rejects. Otherwise, it checks if some condition from Lemma 31 does not hold. It it finds one then it accepts, otherwise it rejects.

Clearly, if there is a winning nested-locking strategy then the procedure can accept by guessing the family of patterns corresponding to this strategy. For this family the check from Lemma 32 does not fail, and one of the conditions of Lemma 31 must be violated.

Conversely, if the decision procedure concludes that there exists a winning strategy, then let $(\mathcal{P}_p)_{p\in Proc}$ be the guessed family of sets of patterns. We know that there exists a strategy σ with behaviors $(\mathcal{P}'_p)_{p\in Proc}$ such that $\mathcal{P}'_p \subseteq \mathcal{P}_p$ for all $p \in Proc$. Furthermore, as there are no patterns in $(\mathcal{P}_p)_{p\in Proc}$ satisfying the requirements of Lemma 31, there cannot be any in the $\mathcal{P}'_p$ either. Hence σ is a winning strategy. ◀

7 Undecidability for unrestricted lock-sharing systems

In this section we show that the deadlock avoidance control problem for lock-sharing systems is undecidable for three processes with a fixed number of locks. Three locks used in non-nested fashion allow to synchronize two processes in lock-step manner. This is an essential ingredient for the undecidability proof.

We have defined lock-sharing systems so that initially all locks are free. First we show the undecidability result supposing that we are allowed to start with a designated distribution of locks. Later we describe how to implement initial lock distributions using extra locks.

▶ **Lemma 34.** *The control problem for lock-sharing systems with 3 processes, fixed initial configuration and fixed number of locks per process is undecidable.*

The proof uses the usual recipe for the undecidability of distributed synthesis [26, 27]. Two processes P and $\overline{P}$ synchronize with a third process C over a stream of bits chosen by their strategy. The process C is partially controlled by the environment, which selects non-deterministically an interleaving of the two streams and parses the interleaving with a finite automaton. This is enough to get undecidability by a reduction from an infinite Post Correspondence Problem (PCP).

Consider an instance $(\alpha_i, \beta_i)_{i\in I}$ of PCP on the alphabet $\{0, 1\}$. A solution is an infinite sequence $i_1 i_2 \ldots \in I^\omega$ such that $\alpha_{i_1}\alpha_{i_2}\ldots = \beta_{i_1}\beta_{i_2}\ldots$. The two streams sent by P and $\overline{P}$ to C, are $\alpha = \alpha_{i_1} i_1 \alpha_{i_2} i_2 \ldots$ and $\beta = \beta_{j_1} j_1 \beta_{j_2} j_2 \ldots$, resp. With finite memory C can check equality of the two words $(\alpha_{i_1}\alpha_{i_2}\cdots = \beta_{j_1}\beta_{j_2}\ldots)$ or equality of the two index sequences $(i_1 i_2 \ldots = j_1 j_2 \ldots)$. Since P and $\overline{P}$ are not aware of what C does, the streams are fixed by the strategies and do not depend on what C is checking.

The locks used in the proof are $\{c, s_0, s_1, p, \overline{c}, \overline{s}_0, \overline{s}_1, \overline{p}\}$. Process C and P use locks from $\{c, s_0, s_1, p\}$ to synchronize and similarly for C, $\overline{P}$ and $\{\overline{c}, \overline{s}_0, \overline{s}_1, \overline{p}\}$.

It remains to explain the synchronization mechanism. The two processes P and C synchronize over a bit of information, say bit 0, by executing specific finite runs using the locks $\{s_0, c, p\}$ in non-nested fashion. Initially, C owns $\{s_0, c\}$ and P owns $\{p\}$. First, C releases lock s_0 and P acquires it, which we denote as $C \xrightarrow{s_0} P$. Here, P is waiting for C to release s_0, and the two actions $\mathtt{rel}_{s_0}$ of C and $\mathtt{acq}_{s_0}$ of P are ordered. The rest of the run follows a similar pattern: at each step, one of the processes is waiting to take a lock released by the other process. With the same notation, the run proceeds with $P \xrightarrow{p} C$, and continues until each process owns the same locks it owned at the start: each lock is sent twice, from its initial owner to the other process, and back. To sum up, the exchange of bit 0 between C and P corresponds to $C \xrightarrow{s_0} P \xrightarrow{p} C \xrightarrow{c} P \xrightarrow{s_0} C \xrightarrow{p} P \xrightarrow{c} C$. In other words, processes C and P respectively perform two local runs:

$$C: \ \mathtt{rel}_{s_0}\, \mathtt{acq}_p\, \mathtt{rel}_c\, \mathtt{acq}_{s_0} \mathtt{rel}_p\, \mathtt{acq}_c \qquad\qquad P: \ \mathtt{acq}_{s_0} \mathtt{rel}_p\, \mathtt{acq}_c\, \mathtt{rel}_{s_0} \mathtt{acq}_p \mathtt{rel}_c$$

Observe that P and C need to execute these sequences in lock-step manner, as one of the two processes waits for a lock from the other.

In order to synchronize over bit 1, the two processes perform a similar synchronization, using s_1 instead of s_0. The communication between C and $\overline{P}$ is identical, except that it uses locks from $\{\overline{c}, \overline{s}_0, \overline{s}_1, \overline{p}\}$.

In each round, P and C must agree beforehand on a bit they are going to synchronize on, either s_0 or s_1. Otherwise the two processes get blocked, and $\overline{P}$ will get blocked too, as it needs locks held by C. A bit stream between C and P is encoded as a concatenation of such runs, and similarly for $C, \overline{P}$. The content of the two bit streams is chosen by the strategies of $P, \overline{P}, C$. Since the strategy has infinite memory, there is no upper bound on the complexity of the streams. Interestingly, two locks are not enough for two processes to synchronize over a bit stream.

▶ **Lemma 35.** *There is a polynomial-time reduction from the control problem for lock-sharing systems with initial configuration to the control problem where all locks are initially free. The reduction adds $|Proc|$ new locks.*

We sketch the proof idea. Assume that we have pairwise disjoint sets $(I_p)_{p \in Proc}$ of locks, and a lock-sharing system $\mathcal{S}$ in which each process p initially owns exactly the locks in I_p. We build another lock-sharing systems $\mathcal{S}_\emptyset$ that starts with all locks initially free, makes every process acquire all locks in I_p, and then simulates $\mathcal{S}$.

It is important that the initialization phase of $\mathcal{S}_\emptyset$ does not interfere with the simulation of $\mathcal{S}$. We ensure this by using one additional lock k_p per process, called the "key" of p.

For process p, the initialization sequence consists of three steps.

1. First, p takes one by one (in a fixed arbitrary order), all its initial locks in I_p.
2. Second, p takes and releases, one by one (in a fixed arbitrary order) all the keys of the other processes $(k_q)_{q \neq p}$.
3. Finally, p acquires its key k_p and keeps it forever.

After acquiring k_p process p reaches the initial state in $\mathcal{S}$.

In order to prevent the initialization phase to create extra deadlocks, there is a local *nop* loop on every state of the initialization sequence. This way, a deadlock may only occur if all processes have finally completed their initialization sequences. Note that the initialization phase does not interfere with the simulation of $\mathcal{S}$. This is because the exchange of keys guarantees that up to the moment where a process p has completed the initialization in $\mathcal{S}_\emptyset$, no other process has used any lock from I_p.

8 Conclusions

Motivated by a recent undecidability result for distributed control synthesis [17] we have considered a model for which the problem has not been investigated yet. With hindsight it is strange that the well-studied model of lock synchronization has not been considered in the context of distributed synthesis. One reason may be the "non-monotone" nature of the synthesis problem. It is not the case that for a less expressive class of systems the problem is necessarily easier because the controllers get less powerful, too.

The two decidable classes of lock-sharing systems presented here are rather promising. Especially because the low complexity results cover already non-trivial problems. All our algorithms are based on analyzing lock patterns. While in this paper we consider only finite state processes, the same method applies to more complex systems, as long as solving the centralized control problem in the style of Lemma 19 is decidable. This is for example the case for pushdown systems.

There are numerous directions that need to be investigated further. We have focused on deadlock avoidance because this is a central property, and deadlocks are difficult to discover by means of testing or verification. Another option is partial deadlock, where some, but not all, processes are blocked. The concept of Z-deadlock scheme from Definition 20 should help here, but the complexity results may be different. Reachability, and repeated reachability properties need to be investigated, too.

We do not know if the upper bound from Theorem 8 is tight. The algorithm for verifying if there is a deadlock in a given strategy graph, Proposition 24, is already quite complicated, and it is not clear how to proceed when a strategy is not given.

Another research direction is to consider probabilistic controllers. It is well known that there are no symmetric solutions to the dining philosophers problem but there is a randomized one [21, 22]. Symmetric solutions are quite important for resilience issues as it is preferable that every process runs the same code. The Lehmann-Rabin algorithm is essentially the system presented in Figure 2 where the choice between *left* and *right* is made randomly. This is one of the examples where randomized strategies are essential. Distributed synthesis has a potential here because it is even more difficult to construct distributed randomized systems and prove them correct.

References

1 A. Arnold and I. Walukiewicz. Nondeterministic controllers of nondeterministic processes. In J. Flum, E. Grädel, and T. Wilke, editors, *Logic and Automata*, volume 2 of *Texts in Logic and Games*, pages 29–52. Amsterdam University Press, 2007.

2 B. Bérard, B. Bollig, P. Bouyer, M. Függer, and N. Sznajder. Synthesis in presence of dynamic links. In J-F. Raskin and D. Bresolin, editors, *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020*, volume 326 of *EPTCS*, pages 33–49, 2020. To appear in Information and Computation. doi:10.4204/EPTCS.326.3.

3 R. Beutner, B. Finkbeiner, and J. Hecking-Harbusch. Translating asynchronous games for distributed synthesis. In W. J. Fokkink and R. van Glabbeek, editors, *30th International Conference on Concurrency Theory, CONCUR 2019*, volume 140 of *LIPIcs*, pages 26:1–26:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CONCUR.2019.26.

4 K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984. doi:10.1145/1780.1804.

5 A. Church. Applications of recursive arithmetic to the problem of cricuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, volume I, pages 3–50. Cornell Univ., Ithaca, N.Y., 1957.

6 E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

7 M. D. Ernst, A. Lovato, D. Macedonio, F. Spoto, and J. Thaine. Locking discipline inference and checking. In L. K. Dillon, W. Visser, and L. A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 1133–1144. ACM, 2016. doi:10.1145/2884781.2884882.

8 B. Finkbeiner. Bounded synthesis for Petri games. In R. Meyer, A. Platzer, and H. Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday. Proceedings*, volume 9360 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2015. `doi:10.1007/978-3-319-23506-6_15`.

9 B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog. Global winning conditions in synthesis of distributed systems with causal memory. In F. Manea and A. Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, Virtual Conference*, volume 216 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CSL.2022.20`.

10 B. Finkbeiner and E.-R. Olderog. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.*, 253:181–203, 2017.

11 B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *20th IEEE Symposium on Logic in Computer Science (LICS) 2005, Proceedings*, pages 321–330. IEEE Computer Society, 2005. `doi:10.1109/LICS.2005.53`.

12 P. Gastin, B. Lerman, and M. Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In K. Lodaya and M. Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2004. `doi:10.1007/978-3-540-30538-5_23`.

13 P. Gastin, N. Sznajder, and M. Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, June 2009.

14 B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Asynchronous games over tree architectures. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2013. `doi:10.1007/978-3-642-39212-2_26`.

15 M. Gieseking, J. Hecking-Harbusch, and A. Yanich. A web interface for Petri nets with transits and Petri games. In J. F. Groote and K. G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 381–388. Springer, 2021. `doi:10.1007/978-3-030-72013-1_22`.

16 H. Gimbert. On the control of asynchronous automata. In S. V. Lokam and R. Ramanujam, editors, *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 30:1–30:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.FSTTCS.2017.30`.

17 H. Gimbert. Distributed asynchronous games with causal memory are undecidable. *CoRR*, abs/2110.14768, 2021. Submitted. `arXiv:2110.14768`.

18 J. Hecking-Harbusch and N. O. Metzger. Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In Y. F. Chen, C. H. Cheng, and J. Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 369–386. Springer, 2019. `doi:10.1007/978-3-030-31784-3_22`.

19 V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), Proceedings*, pages 101–110. IEEE Computer Society, 2006. `doi:10.1109/LICS.2006.11`.

20 O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *16th Annual IEEE Symposium on Logic in Computer Science, Proceedings*, pages 389–398. IEEE Computer Society, 2001. `doi:10.1109/LICS.2001.932514`.

21 D. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In J. White, R. J. Lipton, and P. C. Goldberg, editors, *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 133–138. ACM Press, 1981. `doi:10.1145/567532.567547`.

22 N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

23 P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2001. doi:10.1007/3-540-48224-5_33.

24 P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In R. Ramanujam and S. Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference*, volume 3821 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 2005. doi:10.1007/11590156_16.

25 A. Muscholl and I. Walukiewicz. Distributed synthesis for acyclic architectures. In V. Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014*, volume 29 of *LIPIcs*, pages 639–651. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.639.

26 G. L. Peterson and J. H. Reif. Multiple-person alternation. In *20th Annual Symposium on Foundations of Computer Science*, pages 348–363. IEEE Computer Society, 1979. doi:10.1109/SFCS.1979.25.

27 A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *31st Annual Symposium on Foundations of Computer Science*, pages 746–757. IEEE Computer Society, 1990. doi:10.1109/FSCS.1990.89597.

28 P. J.G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989.

29 K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Trans. on Automat. Control*, 37(11):1692–1708, 1992.

30 J. G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005. doi:10.1016/j.sysconle.2004.10.002.

31 S. Tripakis. Undecidable problems in decentralized observation and control for regular languages. *Information Processing Letters*, 90(1):21–28, 2004.

32 I. Walukiewicz. Synthesis with finite automata. In J.-É. Pin, editor, *Handbook of Automata Theory*, pages 1217–1260. European Mathematical Society Publishing House, Zürich, Switzerland, 2021. doi:10.4171/Automata-2/11.

33 Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 252–263. ACM, 2009. doi:10.1145/1480881.1480913.