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 Σ_2^P-complete, 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, concurrent systems, lock synchronisation, deadlock avoidance

Digital Object Identifier 10.4230/LIPIcs...1

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 [28]. 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, 24, 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 [29]. 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, 28]. 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, 24, 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 [24]. 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 [23], 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 [29]. 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 [33] 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 [30, 32, 31, 1].

We would like to mention two further results that do not fit into the main threads outlined above. In [34] 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. Missing proofs are included in the appendix.

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 ($\mathtt{acq}_t$) or release ($\mathtt{rel}_t$) some lock t from T_p, or do nothing: $Op(T_p) = \{\mathtt{acq}_t, \mathtt{rel}_t \mid t \in T_p\} \cup \{nop\}$. 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 $B = B'$;
- $op = \mathtt{acq}_t$, $t \notin B$ and $B' = B \cup \{t\}$;
- $op = \mathtt{rel}_t$, $t \in B$, 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.

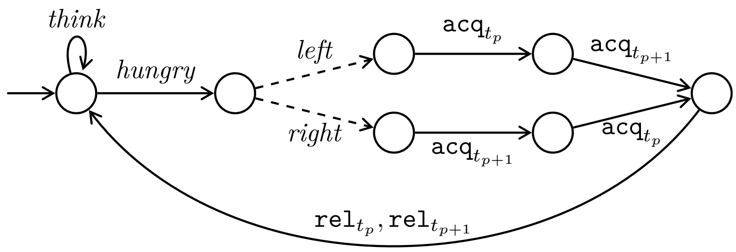

Figure 1 A dining philosopher p. Dashed transitions are controllable.

▶ **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 $hungry^1, left^1, \texttt{acq}^1_{t_1}, \texttt{acq}^1_{t_2}$ is a local run of process 1 respecting the strategy; similarly $hungry^2, left^2, \texttt{acq}^2_{t_2}, \texttt{acq}^2_{t_1}$ for process 2. (We use superscripts to indicate the process doing an action.) The global run $hungry^1, hungry^2, left^1, left^2, \texttt{acq}^1_{t_1}, \texttt{acq}^2_{t_2}$ 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.*

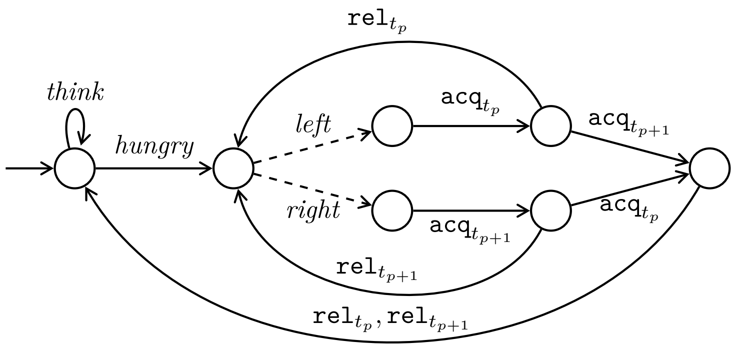

Figure 2 A flexible philosopher p. She can release a fork if the other fork is not available.

▶ **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), the details can be found in Appendix A.

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. The processes enforcing the choice are displayed in Figure 3 in the appendix. 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 $\exists\forall$ 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. The full proof is in Appendix B.

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 $Locks_\sigma$; 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 $Locks = Locks_\sigma$, 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 $Locks_\sigma$ conveniently. Consider a local run u of a process p:

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

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 $Locks_\sigma$). *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 $Locks_\sigma = (L_p)_{p \in Proc}$.*

▶ **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 $ds_Z : Proc_Z \to E \cup \{\bot\}$ such that:*

- *For all $p \in Proc_Z$, if $ds_Z(p) \neq \bot$ then $ds_Z(p)$ is an edge of G labeled by p.*
- *If $p \in Proc_Z$ and $L_p = \emptyset$ then $ds_Z(p) \neq \bot$.*
- *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:

- $\perp$ 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 $\perp$ or a p-labeled edge of the lock graph G_σ.

Suppose $ds(p) = \perp$, 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 $ds(p) = \perp$ 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 $ds(p) = \perp$, 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 $ds(p) \neq \perp$ 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 $ds(p) = \perp$ can move as it needs some lock from BT. Similarly, no process p with $ds(p) \neq \perp$ 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 $ds_Z(p) \neq \perp$.

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 $ds_Z : Proc_Z \to E \cup \{\perp\}$ 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.*

The relatively long proof of the following proposition is presented in Appendix B.2.

▶ **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 $t_1 \xrightarrow{p} t_2$ 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. The complete proof is presented in Appendix C.

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. ◀

The proof of the matching lower bound and the missing lemmas are in Appendix D.

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 $\texttt{rel}_{s_0}$ of C and $\texttt{acq}_{s_0}$ 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:

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

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.

The next lemma shows a generic reduction of the control problem with initial configuration to the one where all locks are initially free.

▶ **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.

Let us explain why the initialization phase does not interfere with the simulation of $\mathcal{S}$. 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. The details of the construction are presented in Appendix E.

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 Jörg Flum, Erich Grädel, and Thomas 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 Jean-François Raskin and Davide 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 *International Conference on Concurrency Theory (CONCUR'19)*, volume 140 of *LIPIcs*, pages 26:1–26:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
4. K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, oct 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 *ICSE 2016, Proceedings of the 38th International Conference on Software Engineering*, pages 1133–1144, Austin, TX, USA, May 2016.
8. B. Finkbeiner. Bounded synthesis for Petri games. In Roland Meyer, André Platzer, and Heike Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. 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 Florin Manea and Alex 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 *LICS'05*, pages 321–330. IEEE Computer Society, 2005.
12. P. Gastin, B. Lerman, and M. Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 275–286. Springer, 2004.
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 *International Colloquium on Automata, Languages and Programming (ICALP'13)*, volume 7966 of *LNCS*, pages 275–286. Springer, 2013.
15. M. Gieseking, J. Hecking-Harbusch, and A. Yanich. A web interface for Petri nets with transits and Petri games. In Jan Friso Groote and Kim Guldstrand 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 *FSTTCS'17*, volume 30 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

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

18 J. Hecking-Harbusch and N. O. Metzger. Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In Yu-Fang Chen, Chih-Hong Cheng, and Javier 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 *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 101–110, 2006. `doi:10.1109/LICS.2006.11`.

20 O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *LICS'01*, pages 389–398. IEEE, 2001.

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 John White, Richard J. Lipton, and Patricia C. Goldberg, editors, *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, pages 133–138. ACM Press, 1981. `doi:10.1145/567532.567547`.

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

23 P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 201–212. Springer, 2005.

24 P. Madhusudan and P.S. Thiagarajan. Distributed control and synthesis for local specifications. In *ICALP'01*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.

25 A. Muscholl and I. Walukiewicz. Distributed synthesis for acyclic architectures. In *FSTTCS'14*, volume 29 of *LIPIcs*, pages 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

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

27 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. ACM POPL*, pages 179–190, 1989.

28 A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS'90*, pages 746–757. IEEE Computer Society, 1990.

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

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

31 J. G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005.

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

33 I. Walukiewicz. Synthesis with finite automata. In J. E. Pin, editor, *Handbook of Automata Theory*, volume 2, pages 1215–1258. 2021. https://www.labri.fr/perso/igw/Papers/igw-synt-chapter.pdf.

34 Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In Zhong Shao and Benjamin 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`.

A Two locks per process: Σ_2^P-completeness

In this section we prove Theorem 7, as recalled below:

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

Thanks to Lemma 15, in order to decide if there is a winning strategy for a given system we need to come up with a set of patterns $Patt_p$ for each process p and show two things:

- these sets of patterns do not meet the conditions given in Lemma 15.
- there is a strategy σ whose local runs on each process p all match a pattern of $Patt_p$.

Note that in the second condition we only require an inclusion because it is clear from the previous lemma that the less patterns a strategy allows, the less chances there are it leads to a deadlock.

We start by showing that we can check the second condition in polynomial time.

▶ **Lemma 36.** *Given a family $(Patt_p)_{p \in Proc}$ of sets of patterns, it is decidable in* PTIME *whether there exists a strategy $\sigma = (\sigma_p)_{p \in Proc}$ such that for all p and all σ_p-runs u_p of p, the pattern of u_p is in $Patt_p$.*

Proof. First of all note that we only need to check for each p that there exists a local strategy σ_p that does not allow any risky runs whose pattern is not in $Patt_p$.

Let $p \in Proc$, let $\mathcal{A}_p$ be its transition system. We extend it in a similar way as in Remark 4, by adding some information in the states. We already assumed that the states contained the information of which locks are currently owned by p. We duplicate the states where p only owns one lock t_1, in order to add a bit of information saying whether p released its other lock t_2 since acquiring t_1 for the last time.

This way, the risky nature of local runs and their patterns depend only on the state in which they end and its outgoing transitions. For instance if a state has no outgoing transitions and is such that when reaching it p holds t_1 and released t_2 since acquiring it, then the pattern of runs ending there is $(\{t_1\}, \emptyset, (t_1, t_2))$. If this pattern is not in $Patt_p$ then we declare this state as bad.

Formally, we define as good all states such that there exists a set of outgoing transitions containing all environment transitions and such that

- either it contains a transition with no acquire operation
- or the set B of locks gotten by those transitions is such that runs ending in that state have a pattern in $Patt_p$.

We define the other states as bad. If all states of the system have that property then clearly there is a suitable strategy.

Otherwise a local control strategy cannot allow any run to reach a bad state without getting a pattern outside of the input set, hence the following algorithm. It resembles the usual algorithm for solving safety games, except that we do not simply want to avoid some states, but we want to avoid having to allow some sets of actions from some states.

We simply iteratively delete bad states and all their ingoing transitions. If one of those transitions is controlled by the environment, we declare its source state as bad (as reaching that state would allow the environment to take that transition, leading us to a bad state). Note that deleting transitions may create more bad states by reducing the choice of the system. If we end up deleting all states, we conclude that there is no suitable strategy. Otherwise the subsystem we obtain only has good states, allowing us to get a strategy matching the input set of patterns. ◀

▶ **Proposition 37.** *The deadlock avoidance control problem for 2LSS is decidable in Σ_2^P.*

Proof. We first non-deterministically guess a set of patterns $Patt_p$ for each process p (each set is of bounded size). By Lemma 36, we can then check in polynomial time if there exists a strategy σ respecting that set of patterns.

If it exists, then we have an adversarial selection of a pattern $pat_p \in Patt_p$ for each p, as well as an adversarial guess of a total order on locks $\leq$. It is then easy to check in polynomial time if these patterns meet the conditions of Lemma 15.

If they do, we reject the input, otherwise we accept it.

If there exists a winning strategy then we take the sets of patterns it allows, we conclude that the system wins by Lemma 15.

Conversely if we find sets of patterns not meeting the requirements of Lemma 15 and such that there is a strategy σ respecting them, then the sets of patterns allowed by σ are subsets of the ones we guessed and therefore they do not meet the conditions of Lemma 15 either. Hence σ is a winning strategy. ◀

We provide the matching lower bound.

▶ **Proposition 38.** *The deadlock avoidance control problem for 2LSS is Σ_2^P-hard, even when restricted to exclusive systems.*

Proof. We reduce from the $\exists\forall$-SAT problem. We are given a boolean formula in 3-disjunctive normal form $\bigvee_{i=1}^{k} \alpha_i$; each clause α_i is a conjunction of three literals $\ell_1^i \wedge \ell_2^i \wedge \ell_3^i$ over a set of variables $\{x_1, \dots, x_n, y_1, \dots, y_m\}$. The question is whether the formula $\exists x_1, \dots, x_n \forall y_1, \dots, y_m, \bigvee_{i=1}^{k} \alpha_i$ is true.

We construct a 2LSS for which there is a winning strategy iff the formula is true. The 2LSS will use locks:

$$\{t_i \mid 1 \leq i \leq k\} \cup \{x_i, \overline{x_i} \mid 1 \leq i \leq n\} \cup \{y_j, \overline{y_j} \mid 1 \leq j \leq m\} \ .$$

For all $1 \leq i \leq n$ we have a process p_i with four states, as depicted in Figure 3. In that process the system has to take both x_i and $\overline{x_i}$, and then may release one of them before being blocked in a state with no outgoing transitions. Similarly, for all $1 \leq j \leq m$ we have a process q_j, in which the environment has to take y_j or $\overline{y_j}$, and then is blocked.

For each clause α_i we also have a process $p(\alpha_i)$ which just has one transition acquiring lock t_i towards a state with a local loop on it. Hence to block all those processes the environment needs to have all t_i taken by other processes.

It can do that with our last kind of processes. For each clause α_i and each literal ℓ of α_i there is a process $p_i(\ell)$. There the environment has to acquire t_i and then ℓ before entering a state with a self-loop.

The only way to block $p_i(\ell)$ is to have either the t_i or the ℓ lock taken by another process. Intuitively, in the first case the environment accepts that the literal ℓ is true while in the second case the environment claims that the literal ℓ is false and has to prove his claim.

A strategy for the system amounts to choosing whether p_i should release x_i or $\overline{x}_i$, for each $i = 1, \dots, n$. It may also choose to release neither. Since the environment has a global view of the system, it can afterwards choose one of $y_j, \overline{y_j}$ in process q_j, for each $j = 1, \dots, m$. Those choices represent valuations, the free lock remaining being the satisfied literal. In order to win, the environment has to ensure that no lock t_j is free (otherwise some $p(C_j)$ would not be blocked), by choosing a literal ℓ_j whose corresponding lock is not free for every formula α_j and taking t_j in $p_j(\ell_j)$. This means that the environment wins if and only if for

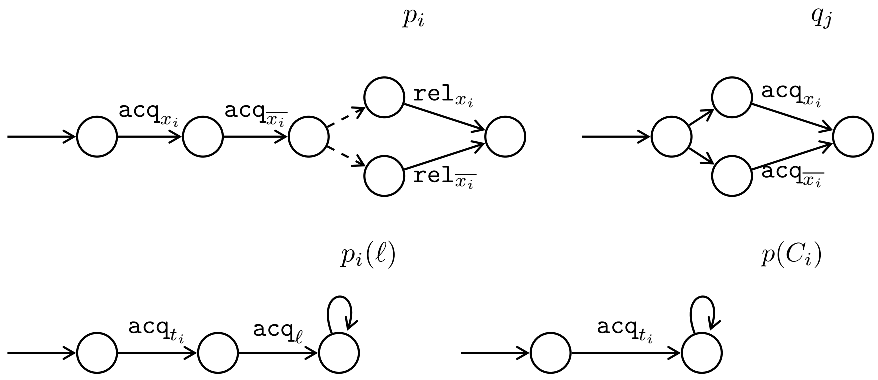

Figure 3 The processes used in the reduction in Proposition 38. Transitions of the system are dashed.

all formulas α_i, one of the literals of α_i is taken. It is therefore never in the interest of the system to release neither x_i nor $\overline{x_i}$. We can assume that it always releases one of them.

Equivalently, the system wins if and only if there exists a valuation of the x's such that for every valuation of the y's there is at least one α_j whose three literals are all satisfied. This concludes our reduction.

◀

B Two locks per process with locally live strategies

We provide missing proofs and constructions from Section 4.

B.1 Characterization of winning strategies (Lemma 22)

Lemma 22 states that the information given by the lock graph and lockset family of a locally live strategy is enough to determine if it is winning. Furthermore, it gives us a goal for the verification of such witnesses: We have to check whether there exists a deadlock scheme.

We prove the lemmas necessary to our polynomial-time algorithm checking the existence of a deadlock scheme.

▶ **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 $Proc_Z$.*

Proof. Suppose (BT, ds) is a deadlock scheme for G_σ. We construct another one ds' which is equal to ds_Z over $Proc_Z$. For every process $p \in Proc$, we define $ds'(p)$ as:

- $ds_Z(p)$ if $p \in Proc_Z$,
- $\bot$ if p labels an edge from Z to $T \setminus Z$,
- $ds(p)$ otherwise.

We assumed that edges from Z to $T \setminus Z$ could not be labeled by solid processes, thus all processes mapped to $\bot$ are fragile. Every lock $t \in BT$ has at most one outgoing edge in ds', since it can only come from ds_Z, if $t \in Z$, or from ds, if $t \in BT \setminus Z$. We verify that there is at least one outgoing edge. By definition of Z-deadlock scheme there is one outgoing edge from every lock in Z. A lock $t \in BT \setminus Z$ has exactly one outgoing edge in $ds(Proc)$, and this edge stays in $ds'(p)$.

Finally, there cannot be any blue cycle in $ds'(Proc)$ as there are none within Z or $BT \setminus Z$ and all edges between the two sets are towards Z. ◀

B.2 PTIME procedure to check the existence of a deadlock scheme

We recall the proposition we want to prove

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

We will describe several polynomial-time algorithms operating on graph $H = (T, GE)$ and a set Z of locks. Observe that H will always have all locks as nodes. Each of those algorithms will either eliminate some edges from H or extend Z, while maintaining Invariants 1–3, recalled below.

▶ **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.*

We start with H being the given G_σ and $Z = \emptyset$. The invariants are clearly satisfied.

There is a simplification we can make: for most of the algorithm we will not use all of $Locks_\sigma$ but simply distinguish between *solid* processes that are not B-lockable for any B (i.e., that will necessarily be mapped to an edge in a deadlock scheme) and the others (called *fragile*).

▶ **Definition 39.** *A process p is called* solid *if $L_p = \emptyset$ and* fragile *otherwise.*

▶ **Definition 40** (Double and solo solid edges)**.** *Consider a solid process p. We say that there is a* double solid edge *$t_1 \xleftrightarrow{p} t_2$ in H if both $t_1 \xrightarrow{p} t_2$ and $t_1 \xleftarrow{p} t_2$ exist in H. We say that $t_1 \xrightarrow{p} t_2$ in H is a* solo solid edge *if there is no $t_1 \xleftarrow{p} t_2$ in H.*

Our first algorithm looks for a solo solid edge $t_1 \xrightarrow{p} t_2$ and erases all other outgoing edges from t_1. It is correct as a deadlock scheme for H has to map p to the edge $t_1 \xrightarrow{p} t_2$ and there must be at most one outgoing edge from every lock.

Algorithm 1 Trimming the graph:

Find $t \in H \setminus Z$ with a solo solid edge $t \xrightarrow{p} t' \in EH$ and some other outgoing edges
If there is no such edge then stop and report success.
for every edge $t \xrightarrow{q} t'' \in HE$ from t with $q \neq p$ **do**
    if q is solid and $t \xleftarrow{q} t'' \notin HE$ **then**
        return "Fail: no H-deadlock scheme"
    else
        Erase $t \xrightarrow{q} t'$
    end if
end for

We repeat this algorithm till no edges are removed. If some call to of the algorithm fails then there is no full deadlock scheme for H. Otherwise the resulting H satisfies the property:

(Trim) if a lock t in $H \setminus Z$ has an outgoing solo solid edge then it has no other outgoing edges.

We call H *trimmed* if it satisfies property (Trim).

▶ **Lemma 41.** *Suppose (H, Z) satisfies Invariants 1–3. If Algorithm 1 fails then there is no H-deadlock scheme. After a successful execution of the algorithm all the invariants are still satisfied. If a successful execution does not remove an edge from H then H satisfies (Trim).*

Proof. Let H' be the graph after an execution of Algorithm 1. Observe that the algorithm does not change Z. If $H = H'$ then (Trim) holds. If the algorithm fails then there is a lock with two solo solid outgoing edges. Thus it is impossible to find a H-deadlock scheme.

Finally, if the algorithm succeeds but H' is smaller than H, we must show that all the invariants hold. Since the algorithm does not change Z, Invariants 2 and 3 continue to hold. For Invariant 1, suppose $t \xrightarrow{p} t'$ is the edge found by the algorithm. Observe that if ds_H is an H-deadlock scheme then $ds_H(p)$ must be this edge. So ds_H is also a deadlock scheme for H'. In the other direction, an H'-deadlock scheme is also an H-deadlock scheme as H' is a subgraph of H and $Proc_H = Proc_{H'}$. The latter holds because H' has the same locks as H. ◀

Our second algorithm searches for cycles formed by solid edges and eventually adds them to Z. If the found cycle has a green edge then it can be added to Z. If the cycle is blue, it may still be the case that its reversal is green. More precisely it may be the case that for every solid edge $t_i \xrightarrow{p_i} t_{i+1}$ in the cycle there is also a reverse edge $t_i \xleftarrow{p_i} t_{i+1}$ (and it is solid by definition). If the reversed cycle is also blue then there is no H-deadlock scheme. If it does, Z can be added to H. The result still satisfies the invariants thanks to (Trim) property of H.

Algorithm 2 Find solid cycles and add them to Z if possible.

Find a simple cycle of solid edges $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ not intersecting Z, all t_i distinct
if there is no such cycle, stop and report success.
if all the edges on the cycle are blue **then**
    if for some j there is no reverse edge $t_j \xleftarrow{p_j} t_{j+1} \in EH$ **then**
        return "Fail: no H-deadlock scheme"
    else if all edges $t_j \xleftarrow{p_j} t_{j+1}$ are blue **then**
        return "Fail: no H-deadlock scheme"
    end if
end if
$Z \leftarrow Z \cup \{t_1, \dots, t_k\}$
For every t_i remove from H all edges outgoing from t_i but those that are on the cycle.
if some solid process p has no edge in H **then**
    return "Fail: no H-deadlock scheme"
end if
repeat
    Apply Algorithm 1
until No more edges are removed from H

▶ **Lemma 42.** *Suppose (H, Z) satisfies the invariants Invariants 1–3 and H has property (Trim). If the execution of Algorithm 2 does not fail then the resulting H and Z also satisfy the invariants and (Trim). If the execution fails then there is no H-deadlock scheme.*

Proof. Suppose the algorithm finds a simple cycle $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ where all p_i are solid processes, and all t_i are distinct. By definition of a simple cycle, all p_i are distinct as well. If there is a H-deadlock scheme then it should assign either $t_i \xrightarrow{p_i} t_{i+1}$ or $t_i \xleftarrow{p_i} t_{i+1}$ to p_i.

We examine the cases when the algorithm fails. The first reason for failure may appear when all the edges on the cycle are blue. If for some j there is no reverse edge $t_j \xleftarrow{p_j} t_{j+1}$ in EH then a H-deadlock scheme, call it ds_H, should assign the edge $t_j \xrightarrow{p_j} t_{j+1}$ to p_j. In consequence, as ds_H has to give each t_i at most one outgoing edge, all the edges in the cycle should be in the image of ds_H. This is impossible as the cycle is blue. When there are reverse edges $t_i \xleftarrow{p_i} t_{i+1} \in EH$ for all i, algorithm fails if all of them are blue. Indeed, there cannot be an H-deadlock scheme in this case. The last reason for failure is when there is some solid process p and p-labeled edges were removed by the algorithm. These must be edges of the form $t_i \xrightarrow{p} t$ that are not on the cycle, for some $i = 1, \ldots, k$. Those edges cannot be taken in a deadlock scheme as it has to take the cycle in one direction or the other and thus cannot take other edges from nodes on that cycle. As a deadlock scheme cannot assign any edge to p, and p solid, there cannot be a deadlock scheme in that case.

If the algorithm does not fail then either the cycle $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ is not blue, or its reverse is not blue. Let (H', Z') be the values after execution of the algorithm. So $Z' = Z \cup \{t_1, \ldots, t_k\}$, and H' is H after removing edges in line 11. We show that the invariants hold.

For Invariant 2, we observe that for every lock in Z' there is exactly one outgoing edge in H'. So there is no solid edge from Z' to $H \setminus Z'$ if there was none from Z.

For Invariant 3, we extend our Z-deadlock scheme to Z'. We choose the cycle found by the algorithm or its reversal depending on which one is not blue. For every p_i we define $ds_{Z'}(p_i)$ to be the edge in the chosen cycle. We set $ds_{Z'}(p) = \perp$ for all $p \in \mathit{Proc}_{Z'} \setminus \mathit{Proc}_Z$ other than $p_1, \ldots, p_k$. We must show that such a p is necessarily fragile. Indeed, in this case p must have one of its locks t in Z, and the other one, t', in $Z' \setminus Z$. By Invariant 2, there is no solid edge from t to t' in H. In H' all edges from t' to t are removed. So p is fragile as the algorithm does not fail at line 12.

For Invariant 1 suppose there is a deadlock scheme on H'. Then it is also a deadlock scheme on H, as H' is a subgraph of H over the same set of locks. For the other direction take a deadlock scheme ds_H on H. By Lemma 23, as we showed that Invariant 2 is maintained, we can assume that ds_H is equal to $ds_{Z'}$ on Z'. We define a deadlock scheme $ds_{H'}$ on H'. If $ds_H(p) = \perp$ then $ds_{H'}(p) = \perp$. If the source edge of $ds_H(p)$ is in $H \setminus Z'$ then $ds_{H'}(p) = ds_H(p)$. This edge is guaranteed to exist in H'. If the two locks of p are both in Z' let $ds_{H'}(p) = ds_H(p) = ds_{Z'}(p)$. The remaining case is when $ds(p)$ is an edge $t \xrightarrow{p} t'$ with $t \in Z'$ and $t' \in H \setminus Z'$. In this case p is fragile as Z' has no solid arrows leaving it, and all solid arrows in $Z' \setminus Z$ stay in Z'. We can then set $ds_{H'} = \perp$. It can be verified that $ds_{H'}$ is a deadlock scheme. ◀

▶ **Lemma 43.** *If Algorithm 2 succeeds but does not increase Z or decrease H then (H, Z) satisfies three properties:*

H1 *H is trimmed.*

H2 *There is no cycle of solid edges intersecting $T \setminus Z$ in H.*

H3 *Every solid process has an edge in H.*

Proof. Since H was not modified, Algorithm 1 did not find any solo solid edge $t \xrightarrow{p} t'$ with other outgoing edges from t, hence property H1 is satisfied.

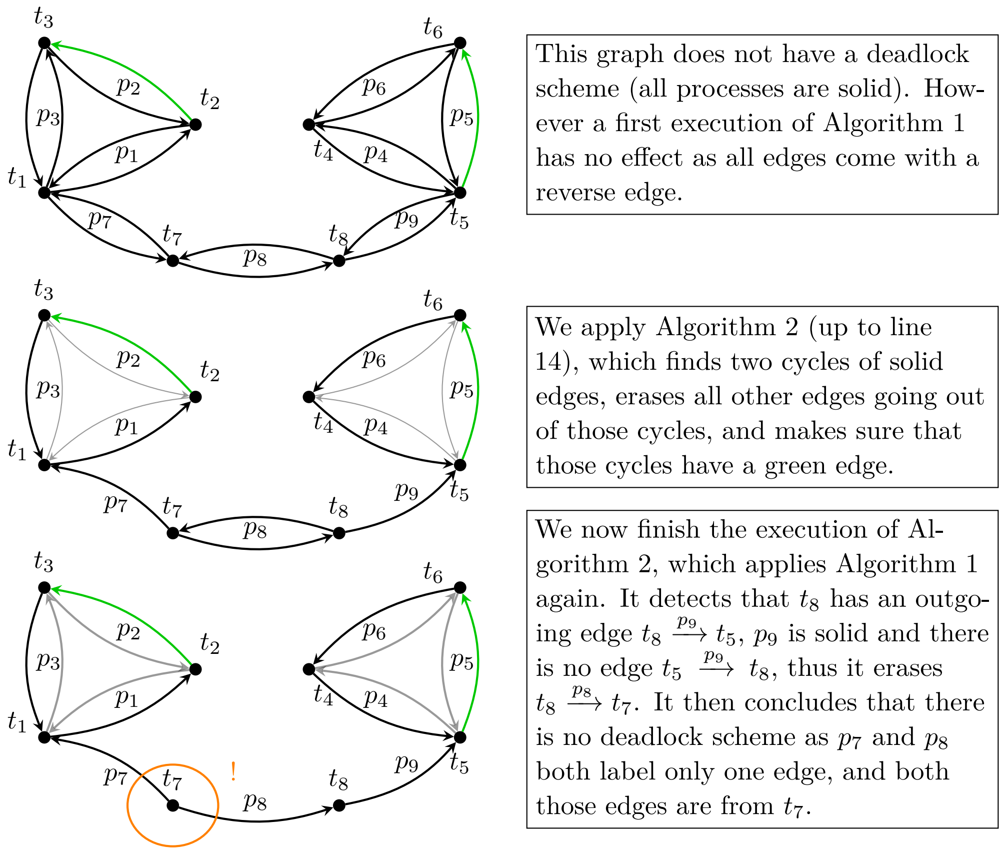

Figure 4 Illustration of Algorithm 1 and Algorithm 2.

By Lemma 42, Invariant 2 is satisfied, hence any cycle intersecting $T \setminus Z$ in H must be entirely in $T \setminus Z$. However if such a cycle existed then Algorithm 2 would not have stopped on line 2 and thus would have either failed or increased Z. There are therefore no such cycle intersecting $T \setminus Z$, hence property H2 is also satisfied.

If H3 was not satisfied then Algorithm 2 would have failed on lines 12-13. ◀

Since in the rest of the algorithm we increase Z and do not modify H, the three properties form the lemma will continue to hold. We will refer to them as H1-H3.

▶ **Definition 44.** *For a pair (H, Z), we define an equivalence relation between locks in T: $t_1 \equiv_H t_2$ if $t_1, t_2 \notin Z$ and there is a path of double solid edges between t_1 and t_2.*

Intuitively, once we have trimmed the graph and eliminated simple cycles of solid edges with Algorithm 2, the equivalence classes of $\equiv_H$ are "trees" made of double solid edges.

▶ **Lemma 45.** *If H satisfies property H1 and $t_1 \xrightarrow{p} t_2$ is in H for a solid process p then either the $\equiv_H$-equivalence class of t_1 is a singleton, or $t_1 \xleftarrow{p} t_2$ is in H, hence $t_2 \equiv_H t_1$.*

Proof. Suppose there exists t_3 such that $t_1 \equiv_H t_3$, then there is a double solid edge from t_1. By (Trim) property, there cannot be solo outgoing edges from t_1. ◀

▶ **Lemma 46.** *Suppose H satisfies properties H1 and H2. Let $t_1, t_2 \in T \setminus Z$. If $t_1 \equiv_H t_2$ then there is a unique simple path of solid edges from t_1 to t_2.*

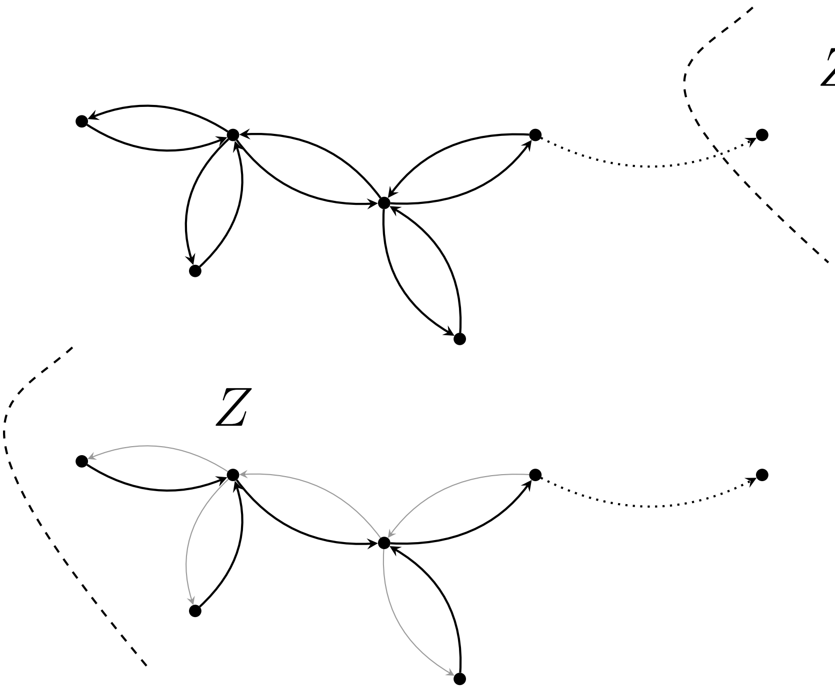

Figure 5 Illustration of Algorithm 3, with dotted fragile edges and full solid ones. The black edges in the second figure are the ones selected in the deadlock scheme when we extend them in the proof of the algorithm.

Proof. If $t_1 = t_2$ then any non-empty simple path of solid edges from t_1 to t_2 would contradict property H2, hence the empty path is the only simple path from t_1 to t_2.

If $t_1 \neq t_2$ then by definition of $\equiv_H$ there is a path of solid double edges from t_1 to t_2, hence there is a simple path from t_1 to t_2.

Suppose there exist two distinct simple paths from t_1 to t_2, then by Lemma 45 all the locks on those paths are in the equivalence class of t_1 and t_2. Hence as $t_1 \notin Z$, there is a cycle of double solid edges intersecting $H \setminus Z$, contradicting property H2. ◀

Our third algorithm looks for an edge $t_1 \xrightarrow{p} t_2$ with $t_1 \notin Z$ and $t_2 \in Z$, and adds the full $\equiv_H$-equivalence class C of t_1 to Z. This step is correct, as we can extend a Z-deadlock scheme to $(Z \cup C)$-deadlock scheme by orienting edges in C.

Algorithm 3 Extending Z with locks that can reach Z

---

while there exists $t_1 \xrightarrow{p} t_2 \in HE$ with $t_1 \notin Z$ and $t_2 \in Z$ **do**
    $Z \leftarrow Z \cup \{t \in T \mid t \equiv_H t_1\}$
end while

▶ **Lemma 47.** *Suppose H satisfies properties H1-H3, and (H, Z) satisfies Invariants 1–3. After an execution of Algorithm 3 H and Z also have all these properties, and H has no edges from $T \setminus Z$ to Z.*

Proof. Let (H', Z') be the pair obtained after execution of Algorithm 3. Observe that $H' = H$, hence Invariant 1 holds. For the same reason H1 and H3 are still satisfied. Furthermore, as Z can only increase, so H2 continues to hold.

Let Z_m be the value of Z when entering the loop, and Z_m the value of Z at the end of m-th iteration. So $Z_{m+1} = Z_m \cup \{t \in T \mid t \equiv_H t_1\}$, where $t_1 \xrightarrow{p} t_2$ is the edge found in the guard of the while statement. We verify that Z_{m+1} satisfies Invariants 2 and 3 if Z_m does.

For Invariant 2, Lemma 45 says that there are no outgoing solid edges from the $\equiv_H$-equivalence class of t_1, unless that class is a singleton. If it is a singleton, there are no outgoing solid edges from t_1 or $t_1 \xrightarrow{p} t_2$ is the only outgoing edge of t_1. In both cases, there are no solid edges from Z_{m+1} to $T \setminus Z_{m+1}$ in H.

For Invariant 3 we extend a Z_m-deadlock scheme to Z_{m+1}. So we are given ds_m and construct ds_{m+1}. If the two locks of some process q are in Z then $ds_{m+1}(q) = ds_m(q)$. We set $ds_{m+1}(q)$ to be the edge $t_1 \xrightarrow{q} t_2$ found by the algorithm. Let C be the equivalence class of t_1: $C = \{t \in T \mid t \equiv_H t_1\}$. By Lemma 46 there is a unique simple path from $t \in C$ to t_1. Let $t \xrightarrow{q} t'$ be the first edge on this path. We set $ds_{m+1}(q)$ to be this edge. We set $ds_{m+1}(q) = \perp$ for all remaining processes q.

We verify that ds_{m+1} is a Z_{m+1}-deadlock scheme. By the above definition every lock in C has a unique outgoing edge in ds_{m+1}, hence every lock in Z_{m+1} does. It is also immediate that the image of ds_{m+1} does not contain a blue cycle as it would need to be already in the image of ds_m (every lock has exactly one outgoing edge in ds_{m+1} and the path obtained by following those edges from an element of C leads to Z_m). It is maybe less clear that $ds_{m+1} \neq \perp$ for every solid $q \in Proc_{Z_{m+1}}$. Let q be a solid process in $Proc_{Z_{m+1}}$, and suppose ds_{m+1} is not defined by the procedure from the previous paragraph. If both of locks of q are in Z_m then $ds_{m+1}(q)$ must be defined because $ds_m(q)$ is. If $q = p$, the process labeling the transition chosen by the algorithm, then $ds_{m+1}(q)$ is defined. Otherwise both locks of q are in C. Say these are t and t'. If neither $t \xrightarrow{q} t'$ is on the shortest path from t to t_1, nor is $t \xleftarrow{q} t'$ on the shortest path from t' to t_1 then there must be a cycle in C. But this is impossible as we assumed that there are no cycles of solid edges intersecting $T \setminus Z$ (property H2) and $Z \subseteq Z_m$. Hence $ds_{m+1}(q)$ is defined, and ds_{m+1} is a Z_{m+1}-deadlock scheme.

All that is left to prove is that H has no edges from $T \setminus Z$ to Z, which is immediate as otherwise Algorithm 3 would not have stopped. ◀

Algorithm 4 Incorporating green cycles

---

if there exists a green cycle $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ with $t_1 \xrightarrow{p_k} t_1$ green **then**
  $Z \leftarrow Z \cup \bigcup_{i=1}^{k} \{t \mid t \equiv_H t_i\}$
end if

▶ **Lemma 48.** *Suppose H satisfies H1-H3, (H, Z) satisfies Invariants 1–3, and moreover there are no edges from $T \setminus Z$ to Z. After an execution of Algorithm 4 H satisfies H1-H3, and (H, Z) satisfies Invariants 1–3.*

Proof. Let (H', Z') be the pair obtained after execution of Algorithm 3. Observe that $H' = H$, hence Invariant 1 holds. For the same reason H1 and H3 are still satisfied. Furthermore, as Z can only increase, so is H2. It remains to verify Invariants 2 and 3.

Consider the green cycle found by the algorithm: $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$

For Invariant 2, Lemma 45 says that there are no outgoing solid edges from the $\equiv_H$-equivalence class of t_1, unless that class is a singleton. If it is a singleton, there are no outgoing solid edges from t_1 or $t_1 \xrightarrow{p} t_2$ is the only outgoing edge of t_1. In both cases, there are no solid edges from Z_{m+1} to $T \setminus Z_{m+1}$ in H.

For Invariant 3 we extend a Z-deadlock scheme ds_Z to Z'. For every lock $t \in Z' \setminus Z$ let j be the biggest index among $1, \ldots, k$ with $t \equiv_H t_j$. If $t = t_j$ then set $ds_{Z'}(p_j)$ to be the edge $t_j \xrightarrow{p_j} t_{j+1}$. Otherwise, take the unique path from t to t_j in the $\equiv_H$-equivalence class of the two locks; this is possible thanks to Lemma 46. If the path starts with $t \xrightarrow{p} t'$ then set $ds_{Z'}(p)$ to this edge. Then set $ds_{Z'}(p) = \bot$ for all remaining processes p.

We claim that $ds_{Z'}$ is a Z'-deadlock scheme. First, there is an outgoing $ds_{Z'}$ edge from every lock in Z' because of the definition. Moreover it is unique.

We need to show that $ds_{Z'}(p)$ is defined for every solid process p. This is clear if the two locks, t and t', of p are in Z. If both locks are not in Z then either $t \equiv_H t'$ or there is a solo solid edge between the two, say $t \xrightarrow{p} t'$. In the latter case this is the only edge from t, as H is trimmed. As the equivalence class of t is then a singleton, this must be an edge on the cycle and $ds_{Z'}(p)$ is defined to be this edge. Suppose $t \equiv_H t'$ and $ds_{Z'}(p)$ is not defined. Let j be the biggest index among $1, \dots, k$ such that $t \equiv_H t_j$. If neither $t \xrightarrow{p} t'$ is on the shortest path from t to t_j, nor $t \xleftarrow{p} t'$ is on the shortest path from t' to t_j then there must be a cycle in C. But this is impossible as we assumed that there are no cycles of solid edges intersecting $T \setminus Z_m$ in H (property H2). The remaining case is when one of the locks of p is in Z and another in $Z' \setminus Z$. There is no solid edge leaving Z by Invariant 2. There is no solid edge entering Z by the assumption of the lemma. So p is a solid process labeling no edge in H which contradicts H3.

The last thing to verify is that there is no blue cycle in $ds_{Z'}$. We first check that $ds_{Z'}$ contains $t_k \xrightarrow{p_k} t_1$. This is because t_k is necessary the last from its equivalence class. A blue cycle cannot contain locks from Z as there are no edges entering Z in $ds_{Z'}$. Let $t'_1 \xrightarrow{p'_1} t'_2 \dots \xrightarrow{p'_l} t'_{l+1} = t'_1$ be a hypothetical blue cycle in $Z' \setminus Z$ using transitions in $ds_{Z'}$.

Consider x such that $t'_1 \equiv_H t'_j$ for $j \le x$ but $t'_1 \not\equiv_H t'_{x+1}$. By definition of $ds_{Z'}$ we must have that t'_x is the last lock among $t_1, \dots, t_k$ equivalent to t'_1, say it is t_y. As each lock only has one outgoing transition in the image of $ds_{Z'}$, and as there is a path from t_y to t_k in that image, t_k must be on that cycle, and thus the green edge $t_k \xrightarrow{p_k} t_1$ as well, contradicting the assumption that this is a blue cycle. ◀

Algorithm 5 below is the complete algorithm as required by Proposition 24. Its correctness is stated in the next lemma.

Algorithm 5 Complete algorithm

```
H ← G_σ
Z ← ∅
repeat
    Apply Algorithm 1
until No more edges are removed from H
repeat                                              ▷ H is trimmed
    Apply Algorithm 2
until No more edges are removed from H
repeat                          ▷ From now on H satisfies properties H1-H3
    Apply Algorithm 3                              ▷ no edges from T \ Z to Z
    Apply Algorithm 4
until Z does not increase anymore
if there is a process p ∈ Proc \ Proc_Z that is not Z-lockable then
    return "Fail: σ is winning"
else
    return "σ is not winning"
end if
```

▶ **Lemma 49.** *Algorithm 5 fails if and only if* $(G_\sigma, \{L_p\}_{p\in Proc})$ *admits a deadlock scheme. The algorithm works in polynomial time.*

Proof. Let (H', Z') be the values at the end of the execution of the algorithm.

Suppose the algorithm fails. If it is before line 13 then using the previous lemmas and Invariant 1 we get that G_σ does not have a deadlock scheme. If the algorithm fails in line 14 then there exists a process p with one of its locks outside of Z and not Z-lockable. Suppose towards a contradiction that there is a deadlock scheme ds_H for H. It must have an edge from a lock of p that is not in Z, say from t. By definition, every lock with an incoming edge in ds_H must also have an outgoing edge in ds_H. Following these edges we get a cycle in H. During the last iteration of lines 9-12, Z was not increased, hence by Lemma 47 there are no edges from $T \setminus Z$ to Z. This cycle is therefore outside Z. It has to be a green cycle by definition of a deadlock scheme, which is a contradiction because Algorithm 4 did not increase Z in its last application.

If the algorithm succeeds then there is a Z-deadlock scheme, say ds_Z. It may still not be a deadlock scheme on G_σ because $Proc_Z$ may be a strict subset of $Proc$. We construct a deadlock scheme (Z, ds) for G_σ. First, we set $ds(p) = ds_Z(p)$ for all $p \in Proc$. Let us take $p \in Proc \setminus Proc_Z$, as the algorithm did not fail in lines 13-14, p is Z-lockable and thus we can map it to $\perp$.

Finally, this algorithm operates in polynomial time as all steps of all loops in the algorithms either decrease H or increase Z. Furthermore, the condition on line 13 is easily verifiable by checking in the lockset family $\{L_p\}_{p\in Proc}$ of σ whether there exists $B \in L_p$ such that $B \subseteq Z$. ◀

C Solving the exclusive case in PTIME

▶ **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 $t_2 \overset{p}{\mapsto} t_1$.*

Proof. Suppose there is a blue edge $t_1 \overset{p}{\hookrightarrow} t_2$, then there is a process p and a local run of $\mathcal{A}_p$ of the form $u = u_1(a_1, \mathtt{acq}_{t_1})u_2(a_1, \mathtt{rel}_{t_2})u_3(a_3, \mathtt{acq}_{t_2})$ respecting σ, with no $\mathtt{rel}_{t_1}$ in u_2 or u_3. Hence there is a point in the run at which p holds both locks.

It is not possible that there is always a release between two acquire operations in u, as then the run would never hold both locks. So there are two acquires in u with no release in-between. Thus there is a green edge between the first lock taken and the second one because 2LSS is exclusive. As $t_1 \overset{p}{\hookrightarrow} t_2$ is blue, the only possibility is that there is 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.*

Proof. By definition of G_σ, as $t_1 \overset{p}{\to} t_2$ is an edge, there exists a local σ-run u_p of p acquiring t_1. Consider the first acquire transition in the run $u_p = u(a, \mathtt{acq}_{t_i})u'$ for some $i \in \{1, 2\}$ and u containing only local actions. As our 2LSS is exclusive, this means that u makes p reach a configuration with only one outgoing transition acquiring t_i, and p not having any lock. Hence p is $\{t_i\}$-lockable and thus $\{t_1, t_2\}$-lockable. ◀

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

The left-to-right direction is handled by the lemma below.

▶ **Lemma 50.** *If all processes are BT_σ-lockable then σ is not winning.*

Proof. We construct a BT_σ-deadlock scheme for $(G_\sigma, Locks_\sigma)$ as follows: for all $t \in BT_\sigma$ we select an outgoing edge in BT_σ, say $t \xrightarrow{p_t} \bar{t}$ to some $\bar{t} \in BT_\sigma$, and a green one if possible. Such an edge always exists by definition of BT_σ. We define ds as $ds(p_t) = t \xrightarrow{p_t} \bar{t}$ for all $t \in BT_\sigma$, and $ds(p) = \perp$ for all other $p \in Proc$.

We show that ds is a BT_σ-deadlock scheme for $(G_\sigma, Locks_\sigma)$. Hence it is also just a deadlock scheme because BT_σ locks all processes.

Clearly for all $p \in Proc$, $ds(p)$ is either $\perp$ or a p-labeled edge within BT_σ. Furthermore as all processes are BT_σ-lockable, in particular the ones mapped to $\perp$ by ds are. It is also clear that all locks of BT_σ have an unique outgoing edge. Now suppose there is a blue cycle in $ds(Proc)$, then by Lemma 25 there is a reverse cycle of green edges between the same locks. This means all those locks have an outgoing green edge within BT_σ, which is a contradiction as we have chosen for ds green outgoing edges whenever possible. ◀

For the right-to-left direction we first prove an auxiliary lemma.

▶ **Lemma 51.** *If ds is a Z-deadlock scheme for $(G_\sigma, Locks_\sigma)$ then $Z \subseteq BT_\sigma$.*

Proof. Suppose there exists $t \in Z \setminus BT_\sigma$, then there exists p such that $ds(p) = t \xrightarrow{p} t'$ for some t'. By definition of BT_σ, there are no edges from $T \setminus BT_\sigma$ to BT_σ in G_σ, hence $t' \in Z \setminus BT_\sigma$. By iterating this process we eventually discover a proper cycle in G_σ outside of BT_σ, which is impossible as this cycle should be part of a direct deadlock component, and thus be included in BT_σ. ◀

▶ **Lemma 52.** *If some process p is not BT_σ-lockable then σ is winning.*

Proof. Suppose there exists p that is not BT_σ-lockable.

Towards a contradiction assume that σ is not winning, hence there is a Z-deadlock scheme ds for σ. As p is not BT_σ-lockable, it is not Z-lockable either, hence $ds(p)$ is not $\perp$. So $ds(p)$ must be an edge $t_1 \xrightarrow{p} t_2$ from G_σ and $t_1, t_2 \in Z$. By previous lemma $t_1, t_2 \in BT_\sigma$.

By Lemma 26, p is $\{t_1, t_2\}$-lockable, and therefore also BT_σ-lockable, yielding a contradiction. ◀

This completes the proof of Proposition 27.

▶ **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 $BT_{\mathcal{S}}$.*

Proof. One direction is easy: if all strategies make all processes able to acquire a lock from $BT_{\mathcal{S}}$ then there is no winning strategy. Let σ be a control strategy, and G_σ its lock graph and its SCCs. Note that $G_{\mathcal{S}}$ is a subgraph of G_σ, hence every SCC in G_σ is a superset of an SCC in $G_{\mathcal{S}}$. Observe that if an SCC in G_σ contains a direct semi-deadlock SCC of $G_{\mathcal{S}}$ then it is direct deadlock. Indeed, if an SCC in $G_{\mathcal{S}}$ is a direct semi-deadlock but not a direct deadlock then σ adds an edge $t_1 \xrightarrow{p} t_2$ to this SCC in G_σ. As t_1, t_2 are in that SCC of $G_{\mathcal{S}}$, there is a simple path from t_2 to t_1 not involving p. Hence, a direct semi-deadlock SCC becomes a direct-deadlock SCC. This implies $BT_{\mathcal{S}} \subseteq BT_\sigma$. Let $p \in Proc$, as there is a run of p acquiring a lock of $BT_{\mathcal{S}}$, either p is $BT_{\mathcal{S}}$-lockable (and thus BT_σ-lockable) or there is an edge labeled by p towards $BT_{\mathcal{S}}$, meaning that both locks of p are in BT_σ and thus that p is BT_σ-lockable by Lemma 26. As a consequence, all processes are BT_σ-lockable. We conclude by Proposition 27.

In the other direction we suppose that there exists a process p and a strategy σ_p forbidding p from acquiring a lock of $BT_{\mathcal{S}}$. We construct a strategy σ such that p is not BT_σ-lockable. This will show that σ is winning.

Let $FT = T \setminus BT_{\mathcal{S}}$ be the set of locks not in $BT_{\mathcal{S}}$. In $G_{\mathcal{S}}$, no node of FT can reach a direct semi-deadlock SCC. In particular, there is no direct semi-deadlock SCC in $G_{\mathcal{S}}$ restricted to FT. We construct a strategy σ such that, when restricted to FT, the SCC's of G_σ and $G_{\mathcal{S}}$ are the same.

Let us linearly order SCC components of $G_{\mathcal{S}}$ restricted to FT in such a way that if a component C_1 can reach a component C_2 then C_1 is before C_2 in the order.

We use strategy σ_p for p. For every process $q \neq p$ we have one of the two cases: (i) either there is a local strategy σ_q inducing only the edges that are already in $G_{\mathcal{S}}$; (ii) or every local strategy induces some edge that is not in $G_{\mathcal{S}}$. In the second case there are no q-labeled edges in $G_{\mathcal{S}}$, and for each of the two possible edges there is a strategy inducing only this edge.

For a process q from the first case we take a strategy σ_p that induces only the edges present in $G_{\mathcal{S}}$.

For a process q from the second case,

- If both locks of q are in $BT_{\mathcal{S}}$ then take any strategy for p.
- If one of the locks of q is in $BT_{\mathcal{S}}$ and the other in FT then choose a strategy inducing an arrow from the $BT_{\mathcal{S}}$ lock to the FT lock.
- If both locks of q are in FT then choose a strategy inducing an edge from a smaller to a bigger SCC.

Consider the graph G_σ of the resulting strategy σ. Restricted to FT this graph has the same SCCs as $G_{\mathcal{S}}$. Moreover, there are no extra edges in G_σ added to any SCC included in FT, and there are no edges from FT to $BT_{\mathcal{S}}$. As a result, we have $BT_{\mathcal{S}} = BT_\sigma$.

As p cannot acquire any lock from $BT_{\mathcal{S}}$, it is not $BT_{\mathcal{S}}$-lockable and thus not BT_σ-lockable either. ◀

D Nested-locking strategies

By abuse of language we will denote in this section a run u, not necessarily initial, as neutral if every lock acquired in u is also released within u.

▶ **Lemma 53.** *Every local run u that respects a nested-locking strategy has a unique stair decomposition.*

Proof. We set $u = u_1 \mathtt{acq}_{t_1} u_2 \mathtt{acq}_{t_2} \cdots u_k \mathtt{acq}_{t_k} u_{k+1}$ such that $\{t_1, \ldots, t_k\}$ is the set of locks held by p at the end of the run, and the distinguished $\mathtt{acq}_{t_i}$ are the last time these locks were taken in u. Consequently, there is no operation on t_i in $u_{i+1}, \ldots, u_{k+1}$.

Observe that u_{k+1} must be neutral because the process owns $\{t_1, \ldots, t_k\}$ at the end of u. If some u_i, $i \leq k$, is not neutral, then there exists $t \in T$ such that $t \notin \{t_1, \ldots t_i\}$ and p holds t after $u_1 \mathtt{acq}_{t_1} \cdots u_i \mathtt{acq}_{t_i}$. Then p has to release t at some point later in the run: if $t \notin \{t_1, \ldots, t_k\}$ then p does not hold it at the end, and if $t \in \{t_1, \ldots, t_k\}$ then $t \in \{t_{i+1}, \ldots, t_k\}$ and thus t is taken again later in the run. But this contradicts the nested-locking assumption, because t would be released before t_i, which has been acquired after t.

◀

▶ **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* $\preceq^p$*.*

Proof. Suppose σ is not winning, let w be a run leading to a deadlock. For all p let T^p_{owns} be the set of locks owned by p after w. Take u^p the local run of p in w. Since w leads to a deadlock every u^p is risky. For every p, consider the stair pattern $(T^p_{owns}, T^p_{blocks}, \preceq^p)$ of u^p. This way we ensure it is a pattern from $\mathcal{P}_p$.

We need to show that these patterns satisfy the requirements of the lemma. Since the configuration reached after w is a deadlock, every process waits for locks that are already taken so $T^p_{blocks} \subseteq \bigcup_{q\in Proc} T^q_{owns}$, for every process p, proving the first condition.

We have that T^p_{owns} is the set of locks that p has at the end of the run w. So the sets T^p_{owns} are pairwise disjoint.

For the last requirement of the lemma take an order $\preceq$ on T satisfying: $t \preceq t'$ if the last operation on t appears before the last operation on t' in w.

Let $p \in Proc$, let $u^p = u^p_1 \mathtt{acq}_{t^p_1} u^p_2 \mathtt{acq}_{t^p_2} \cdots u^p_k \mathtt{acq}_{t^p_k} u^p_{k+1}$ be the stair decomposition of u^p. As p never releases t^p_i, the distinguished $\mathtt{acq}_{t^p_i}$, is the last operation on t^p_i in the global run. Consequently, for all t we have $t^p_i \preceq t$ whenever t is used in $u^p_{i+1}\mathtt{acq}_{t^p_{i+1}} \cdots u^p_k \mathtt{acq}_{t^p_k} u^p_{k+1}$. As a result, $\preceq$ is compatible with all $\preceq^p$.

For the converse implication, suppose that there are patterns satisfying all the conditions of the lemma. We need to construct a run w ending in a deadlock. For every process p we have a stair pattern $(T^p_{owns}, T^p_{blocks}, \preceq^p)$ coming from a local σ-run u^p of p, with $u = u^p_1 \mathtt{acq}_{t^p_1} u^p_2 \mathtt{acq}_{t^p_2} \cdots u^p_k \mathtt{acq}_{t^p_k} u^p_{k+1}$ as stair decomposition. There is also a linear order $\preceq$ compatible with all $\preceq_p$. Let $\prec$ be its strict part. Let $t_1 \dots t_k$ be the sequence of locks from $\bigcup_p T_p$ listed in $\prec$ order (which is possible as the T_{owns} are disjoint and thus no lock appears twice in that sequence), and let $\{p_1, \dots, p_n\} = Proc$. We claim that we can get a suitable run w as $u^{p_1}_1 \dots u^{p_n}_1 w'$ where w' is obtained from $t_1 \dots t_k$ by substituting each t^p_i by $\mathtt{acq}_{t^p_i} u^p_{i+1}$.

All u^p_i are neutral, hence after executing $u^{p_1}_1 \dots u^{p_n}_1$ all locks are free. Let $t^p_i \in T_p$, suppose we furthermore executed all $\mathtt{acq}_{t^q_j} u^q_{j+1}$ with $t^q_j \prec t^p_i$. Then the set of non-free locks is $\{t^q_j \mid t^q_j \prec t^q_i\}$. As $\preceq$ is compatible with all $\preceq^p$, all locks t used in $\mathtt{acq}_{t^p_i} u^p_{i+1}$ are such that $t^p_i \preceq t$. Moreover, all t^q_j that were taken before are such that $t^q_j \prec t^p_i$, thus $\mathtt{acq}_{t^p_i} u^p_{i+1}$ only uses locks that are free and can therefore be executed.

As a result, w can be executed. It leads to a deadlock as $T^p_{blocks} \subseteq \bigcup_q T^q_{owns}$. ◀

▶ **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* $\mathcal{P}_p$*.*

Proof. For every process p we proceed as follows We extend the states of p to store in it the stair profile of the current run. This increases the number of states by the factor $|T|!2^{|T|^2}$. As the set of locks owned by p is now a function of the current state, this also allows us to eliminate all non-realizable transitions which acquire a lock that p owns or release one it does not have.

Then the risky nature and the stair pattern of a run u depend only on the state it ends in and the choice of transitions of the system from that state after executing u.

We focus on states with only outgoing transitions acquiring locks (including those with no outgoing transition). Those states force the runs entering them to be risky. The choices of transitions of the system in that state will then determine the stair pattern of each run entering it. We mark such states as bad if all choices of transitions of the system yield a stair pattern that is not in $\mathcal{P}_p$.

We iteratively delete all bad states and all their ingoing transitions, as we need to ensure that we never reach them. If we delete an uncontrollable transition then we mark its source state as bad as reaching that state would make the environment able to reach a bad state. As we deleted some transitions, we may have reduced the choices of the system in some states, therefore we check again all states and mark as bad all the ones in which no choice of transitions of the system yields a pattern in $\mathcal{P}_p$. We again delete all bad states, and so on.

At some point we reach a system with no bad state. If it is empty, then we conclude that there is no suitable strategy. If not, we construct a strategy by selecting for each state a set of outgoing transitions that either contains a transition not acquiring a lock or allows runs reaching this state to match a pattern of $\mathcal{P}_p$. This strategy ensures that all risky runs have a pattern in $\mathcal{P}_p$. ◀

▶ **Proposition 54.** *The deadlock avoidance control problem is* NEXPTIME*-hard.*

Proof. For the lower bound, we reduce the domino tiling problem over an exponential grid. In this problem, we are given an alphabet Σ, with a special letter b, an integer n (in unary) and a set D of dominoes, each domino d being a 4-tuple $(up_d, down_d, right_d, left_d)$ of letters of Σ. The output is whether there exists a function $til : \{0, \dots, 2^n - 1\}^2 \to D$ such that for all $x, y, x', y' \in \{0, \dots, 2^n - 1\}$,

- if $x' = x$ and $y' = y + 1$ then $up_{til(x,y)} = down_{til(x',y')}$,
- if $x' = x + 1$ and $y' = y$ then $right_{til(x,y)} = left_{til(x',y')}$,
- if $x = 0$ then $left_{til(x,y)} = b$
- if $x = 2^n - 1$ then $right_{til(x,y)} = b$
- if $y = 0$ then $down_{til(x,y)} = b$
- if $y = 2^n - 1$ then $up_{til(x,y)} = b$

This problem is well-known to be NEXPTIME-complete.

Let n, Σ, D, b be an instance of that problem. We construct a lock-sharing system as follows: We have three processes p, $\overline{p}$ and q. Process p will use locks $0^x_i, 1^x_i, 0^y_i$, 1^y_i, for $1 \leq i \leq n$, together with $t(d)$ for each domino $d \in D$, and an extra lock *lock*. Process p will use analogous locks but with a bar: $\overline{0^x_i}$, $\overline{1^x_i}$, $\overline{0^y_i}$, $\overline{1^y_i}$, $\overline{t(d)}$,$\overline{lock}$. Process q will use all the locks.

Let us describe process q. In the initial state the environment can choose between several actions: *equality*, *vertical*, *horizontal*, b_{left}, b_{right}, b_{up} and b_{down}. Each of these actions leads to a different transition system, but the principle behind all systems is the same. In the first phase, for each $1 \leq i \leq n$, the environment can choose to take either 0^x_i or 1^x_i, and then take either $\overline{0^x_i}$ or $\overline{1^x_i}$. In the second phase the same happens for y locks. After these two phases environment has chosen two pairs of n-bit numbers, call them $\#x, \#y$ and $\#\overline{x}, \#\overline{y}$. Where the three systems differ is how the choice of $\overline{x}$'s and $\overline{y}$'s is limited in these two phases. This depends on the first action done by the environment.

- If it is *equality* then $\#x = \#\overline{x}$ and $\#y = \#\overline{y}$.
- If it is *vertical*, then $\#x = \#\overline{x}$ and $\#y + 1 = \#\overline{y}$;
- If it is *horizontal*, then $\#x + 1 = \#\overline{x}$ and $\#y = \#\overline{y}$.
- If it is b_{left} (resp. b_{right}) then $\#x = 0$ (resp. $2^n - 1$).
- If it is b_{down} (resp. b_{up}) then $\#y = 0$ (resp. $2^n - 1$).

These constraints are easily implemented. For example, *equality* is checked by forcing the environment to take the same bit for $\overline{x}$ after choosing each bit for x (similarly for y).

In the third phase, process q has to take and then immediately release *lock* and then $\overline{lock}$ before it reaches a state *dominoes*.

Every state in the three phases before *dominoes* has a local loop on it, meaning that q cannot deadlock while being in one of these states.

In state *dominoes*, the system chooses to take two dominoes d and $\overline{d}$ such that:

- If the environment chose *equality* then $d = \overline{d}$
- If it chose *vertical* then $up_d = down_{\overline{d}}$
- If it chose *horizontal* then $right_d = left_{\overline{d}}$
- If it chose b_{left} (resp. $b_{right}, b_{up}, b_{down}$) then $left_d = b$ (resp. $right_d, up_d, down_d$).

Each choice leads to a different state $s_{d,\overline{d}}$. From there transitions force the system to take every lock $t(d')$ and $\overline{t(d')}$, except for $t(d)$ and $t(\overline{d})$ in order to reach a state *win* with a local loop on it and no other outgoing transitions.

We now describe process p. It starts by taking the lock *lock*, which it never releases. Then the environment chooses to take one of 0^x_i and 1^x_i and one of 0^y_i and 1^y_i for all $1 \leq i \leq n$. Finally, the system chooses a domino d and takes the lock $t(d)$ before reaching a state with no outgoing transitions.

Process $\overline{p}$ behaves identically, but uses locks with a bar.

We need to show that if there is a tiling $til : \{0, \dots, 2^n - 1\}^2 \to D$ then there is a winning strategy. The strategy for q is to respond with the correct tiles: if environment chooses $\#x$, $\#y$, $\#\overline{x}$, $\#\overline{y}$ the strategy chooses locks corresponding to d_1 and $\overline{d_2}$ with $d_1 = til(\#x, \#y)$ and $d_2 = til(\#\overline{x}, \#by)$. The strategy of p does the same but uses inverse encoding of numbers: considers 0 as 1, and 1 as 0. Similarly for $\overline{p}$.

Assume for contradiction that the strategy is not winning, so we have a run leading to a deadlock. First, observe that the environment must have q pass the *lock* phase before p and $\overline{p}$ start running, because all states before *lock* have a self-loop, so q cannot block there. If p or $\overline{p}$ starts before q has passed the *lock* phase, then q can never pass it as one of $lock, \overline{lock}$ will never be available again.

If q passed the *lock* phase then process p has no choice but to take *lock*, and then the remaining locks among x, y. Similarly for $\overline{p}$. At this stage strategy σ is defined so that the three processes will never take the same lock. So q cannot be blocked before reaching state *win*. Thus deadlock is impossible.

For the other direction, suppose there is a winning strategy σ for the system. Observe that the strategy σ_p for process p should decide which domino to take after the environment has decided what x and y locks to take. So σ_p defines a function $til : \{0, \dots, 2^n - 1\}^2 \to D$. Similarly $\sigma_{\overline{p}}$ defines $\overline{til}$.

We first show that $til(i, j) = \overline{til}(i, j)$ for all $i, j \in \{0, \dots, 2^n - 1\}$. If not then consider the run where environment chooses *equality* and then x, $\overline{x}$ to be the representations of i, and y, $\overline{y}$ to be representations of j. So q reaches state *dominoes*. Next the environment makes processes p and $\overline{p}$ to choose locks corresponding to dominoes $til(i, j)$ and $\overline{til}(i, j)$. The two processes p and q reach a deadlock state. Since these are two different dominoes, q cannot reach state *win* from any state $s_{d,\overline{d}}$. Hence there is a deadlock run, that we have assumed impossible.

Once we know that the strategies σ_p and $\sigma_{\overline{p}}$ define the same tiling function it is easy to see that in order to be winning when environment chooses *vertical*, *horizontal* or b_{left}, b_{right}, b_{down}, b_{up} actions, the tiling function should be correct.

◀

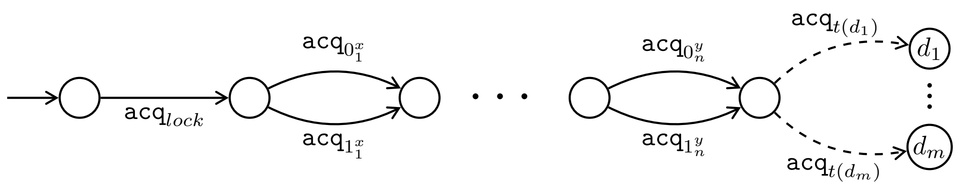

Figure 6 Transition system for process p (with $D = \{d_1, \dots, d_m\}$), dashed arrows are controlled by the system.

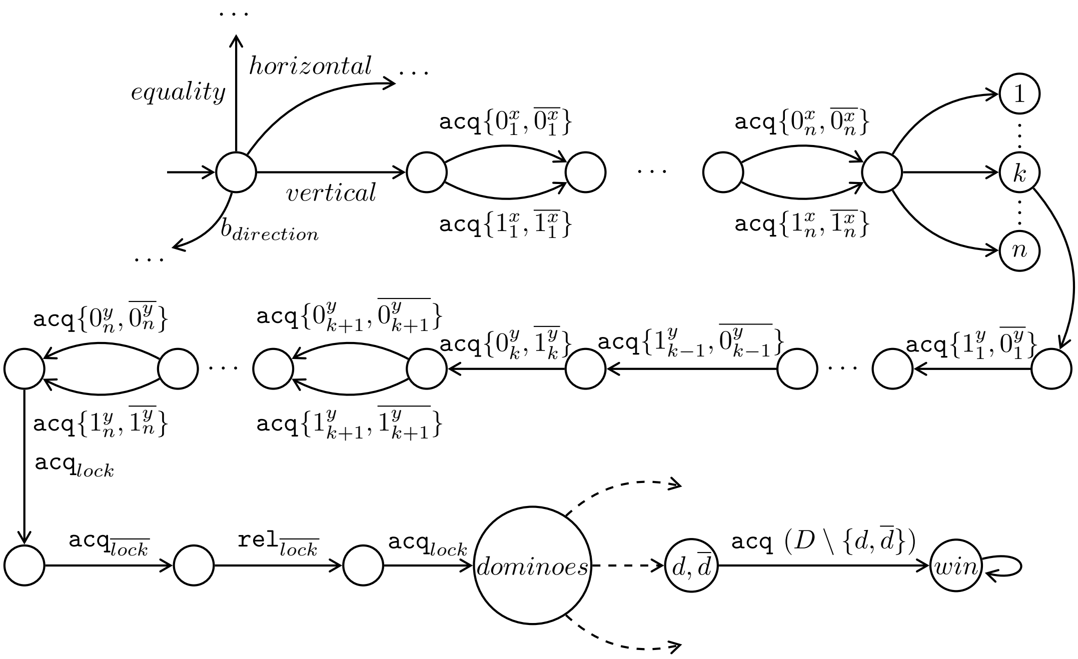

Figure 7 Transition system for process q, dashed arrows are controlled by the system, every state before *dominoes* has a self-loop that is not drawn and acq S means that there is a sequence of forced transitions with the operations acq$_t$ for each $t \in S$ (in some order). We only drew the subsystem used when the environment chooses *vertical*.

E Undecidability for unrestricted lock-sharing systems

E.1 Initial ownership of locks

In a lock-sharing system all locks are assumed to be initially free. We consider now the variant where some of the locks are initially owned by some processes.

The input is a lock-sharing system $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$ and an initial configuration $C_{init} = (init_p, I_p)_{p \in Proc}$ with pairwise disjoint sets $I_p \subseteq T$. The question is whether there exists a control strategy that guarantees that no run from C_{init} deadlocks.

It turns out that this generalization of the deadlock-avoidance control problem is not more difficult than our original problem, as stated in Lemma 35:

▶ **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.*

Proof. The system $\mathcal{S} = (\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$ with initial ownership $(I_p)_{p \in Proc}$ is trans-formed into a new system $\mathcal{S}_\emptyset$ with additional locks. The transformation introduces one extra lock per process, denoted k_p and called *the key of p*. Each process uses in addition to T_p the $|Proc|$ extra locks.

The transition system $\mathcal{A}_p$ of process p is extended with new states and transitions, which define a specific finite run called the *init sequence*. The new states and transitions can occur only during the init sequence. When a process p completes his init sequence in $\mathcal{S}_\emptyset$, he owns precisely all locks in I_p, plus the key k_p, and has reached his initial state $init_p$ in $\mathcal{A}_p$. After that, further actions and transitions played in $\mathcal{S}_\emptyset$ are actions and transitions of $\mathcal{S}$, unchanged. All the new actions are uncontrollable, thus there is no strategic decision to make for the controller of a process p until his init sequence is completed.

The init sequence.

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

1. First, p takes one by one (in a fixed arbitrary order) all 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 his key k_p and keeps it forever.

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

In order to prevent the init sequence to create extra deadlocks, every state used in the initialisation sequence is equipped with a local self-loop on the *nop* operation. This way, a deadlock may only occur if all processes have finally completed their init sequences.

Linking runs in $\mathcal{S}_\emptyset$ and $\mathcal{S}$.

When a process completes his init sequence, he has been until that point the sole owner of its initial locks:

▷ Claim 55. Let p be a process and $u_\emptyset$ a run of $\mathcal{S}_\emptyset$ such that the last action of $u_\emptyset$ is $\mathtt{acq}_{k_p}$ by process p. Let $t \in I_p$, then p is the only process to acquire t in $u_\emptyset$.

Proof. By contradiction, let $t \in I_p$ and $q \neq p$ and assume that $u_\emptyset$ factorizes as $u_\emptyset = u_0 \cdot (\mathtt{acq}_t, q) \cdot u_1 \cdot (\mathtt{acq}_{k_p}, p)$ (we abuse the notation and denote $(\mathtt{acq}_t, q)$ and $(\mathtt{acq}_{k_p}, p)$ the transitions where q and p respectively acquire t and k_p). Process p must take and release k_q before taking k_p, thus the transition $\delta = (\mathtt{acq}_{k_q}, p)$ occurs either in u_0 or in u_1. However δ cannot occur in u_0: the init sequence of p requires that p owns t permanently in the interval between the occurence of δ and the occurence of $(\mathtt{acq}_{k_p}, p)$, thus $(\mathtt{acq}_t, q)$ cannot occur in the meantime. Hence δ occurs in u_1. But this leads to a contradiction: since t is not an initial lock of q, process q is not allowed to acquire t during his init sequence, hence q has completed his init sequence in u_0. After u_0, q owns permanently k_q, but then it is impossible that $\delta = (\mathtt{acq}_{k_q}, p)$ occurs during u_1. ◀

There is a tight link between runs in $\mathcal{S}_\emptyset$ and runs in $\mathcal{S}$.

▷ Claim 56. Let $u_\emptyset$ be a global run in $\mathcal{S}_\emptyset$ in which all processes have completed their init sequences. There exists a global run u in $\mathcal{S}$ (with initial lock ownership $(I_p)_{p \in Proc}$) with the same local runs as $u_\emptyset$, except that the init sequences are deleted.

Proof. The proof is by induction on the number N of transitions in $u_\emptyset$ which are not transitions of the init sequence. In the base case $N = 0$, then $u_\emptyset$ is an interleaving of the init sequences of all processes and u is the empty run. Assume now $N > 0$. Let δ be the last transition played in $u_\emptyset$ which is not part of an init sequence, and $Z \subseteq \mathit{Proc}$ the set of processes that have not yet completed their init sequence when δ occurs. Then $u_\emptyset$ factorizes as

$$u_\emptyset = u'_\emptyset \cdot \delta \cdot u''_\emptyset$$

where $u''_\emptyset$ is an interleaving of infixes of the init sequences of processes in Z.

Assume first that $u''_\emptyset$ is empty. We apply the inductive hypothesis to $u'_\emptyset$, get a global run ν and set $u = \nu \cdot \delta$. Then u has the same local runs as $u_\emptyset$, after deletion of init sequences.

We now reduce the general case to the special case where $u''_\emptyset$ is empty. Let q be the process operating in δ and (a, op) the corresponding pair of action and operation on locks. Since δ is not part of an init sequence, then $q \notin Z$ and op is not an operation on one of the keys. Moreover, according to Claim 55, neither is op an operation on one of the initial locks of processes in Z. Thus (a, op) can commute with all transitions in $u''_\emptyset$ and become the last transition of the global run, while leaving the local runs unchanged, and we are back to the case where $u''_\emptyset$ is empty. ◀

We turn now to the proof of the theorem.

▷ Claim 57. The system wins in $\mathcal{S}_\emptyset$ if and only if it wins in $\mathcal{S}$ with initial lock ownership $(I_p)_{p \in \mathit{Proc}}$.

Since in $\mathcal{S}_\emptyset$ there is no strategic decision to make during the init sequence, the strategies in $\mathcal{S}_\emptyset$ are in a natural one-to-one correspondence with strategies in $\mathcal{S}$. For a fixed strategy we show that there is some deadlock in $\mathcal{S}$ if and only if there is some deadlock in $\mathcal{S}_\emptyset$.

If there is a deadlock in $\mathcal{S}$ then there is also one in $\mathcal{S}_\emptyset$, by executing first all init sequences, and then the deadlocking run of $\mathcal{S}$. The execution of all init sequences is in two steps: first each process p acquires its initial locks I_p and acquires and releases the keys $k_q, q \neq p$ of other processes. Second, each process p acquires (definitively) its key k_p.

Suppose now that there is a deadlocking run $u_\emptyset$ in $\mathcal{S}_\emptyset$. Observe first that all processes $p \in \mathit{Proc}$ have completed their init sequences in u, because all states used in this sequence have local nop self-loops. By Claim 56 there exists a global run u of $\mathcal{S}$ which has the same local runs as $u_\emptyset$ (apart from the init sequences). Since $u_\emptyset$ is deadlocking, so is u. ◀

E.2 Undecidability

In this section we show that the deadlock-avoidance control problem becomes undecidable if we do not limit the maximal number of locks that processes can use.

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

We reduce from the question whether a PCP instance has an infinite solution. Let $(\alpha_i, \beta_i)_{i=1}^m$ be a PCP instance with $\alpha_i, \beta_i \in \{0,1\}^*$. We construct below a system with three processes $P, \overline{P}, C$, using locks from the set

$$\{c, s_0, s_1, p, \overline{s}_0, \overline{s}_1, \overline{p}\}\,.$$

Process P will use locks from $\{c, s_0, s_1, p\}$, process $\overline{P}$ from $\{c, \overline{s}_0, \overline{s}_1, \overline{p}\}$, and C all seven locks.

Processes $P, \overline{P}$ are supposed to synchronize over a PCP solution with the controller process C. That is, P and C synchronize over a sequence $\alpha_{i_1}\alpha_{i_2}\dots$, whereas $\overline{P}$ and C

synchronize over a sequence $\beta_{j_1}\beta_{j_2}\ldots$. The environment tells C at the beginning whether she should check index equality $i_1 i_2 \cdots = j_1 j_2 \ldots$ or word equality $\alpha_{i_1}\alpha_{i_2}\cdots = \beta_{j_1}\beta_{j_2}\ldots$.

For the initial configuration we assume that P owns p, $\overline{P}$ owns $\overline{p}$ and C owns $c, s_0, s_1, \overline{s}_0, \overline{s}_1$.

We describe now the three processes $P, \overline{P}, C$. Define first for $b = 0, 1$:

$$u_P(b) \quad = \quad \mathtt{acq}_{s_b}\mathtt{rel}_p\,\mathtt{acq}_c\;\mathtt{rel}_{s_b}\mathtt{acq}_p\mathtt{rel}_c$$

$$u_{\overline{P}}(b) \quad = \quad \mathtt{acq}_{\overline{s}_b}\mathtt{rel}_{\overline{p}}\,\mathtt{acq}_c\;\mathtt{rel}_{\overline{s}_b}\mathtt{acq}_{\overline{p}}\mathtt{rel}_c$$

The automaton of $\mathcal{A}_P$ ($\mathcal{A}_{\overline{P}}$, resp.) allows all possible action sequences from $(u_P(0) + u_P(1))^\omega$ $((u_{\overline{P}}(0) + u_{\overline{P}}(1))^\omega$, resp.). If e.g. process P manages to execute a sequence $u_P(b_1)u_P(b_2)\ldots$ then this means that C, P synchronize over the sequence $b_1, b_2, \ldots$.

Process C's behavior for checking word equality consists in repeating the following procedure: she chooses a bit b through a controllable action, then tries to do $u_C(P, b)u_C(\overline{P}, b)$, with:

$$u_C(P, b) \quad = \quad \mathtt{rel}_{s_b}\,\mathtt{acq}_p\,\mathtt{rel}_c\;\mathtt{acq}_{s_b}\mathtt{rel}_P\,\mathtt{acq}_c$$

$$u_C(\overline{P}, b) \quad = \quad \mathtt{rel}_{\overline{s}_b}\mathtt{acq}_{\overline{p}}\,\mathtt{rel}_c\;\mathtt{acq}_{\overline{s}_b}\mathtt{rel}_{\overline{p}}\,\mathtt{acq}_c$$

For index equality C's behavior is similar: she chooses an index i and then tries to do $u_C(P, b_1)\ldots u_C(P, b_k)u_C(\overline{P}, b'_1)\ldots u_C(\overline{P}, b'_\ell)$, where $\alpha_i = b_1 \ldots b_k$, $\beta_i = b'_1 \ldots b'_\ell$.

The next lemma is the key property showing that the system deadlock-avoiding strategy if and only if the PCP instance has a solution.

▶ **Lemma 58.** *Assume that C owns $\{s_0, s_1, c\}$, P owns $\{p\}$, C wants to execute $u_C(P, b)$, and P wants to execute $u_P(b')$. Then the system deadlocks if and only if $b \neq b'$. If $b = b'$ then C and P finish executing $u_C(P, b)$ and $u_P(b)$, respectively, and the lock ownership is the same as before the execution.*

Proof. If, say, $b = 0$ and $b' = 1$ then C releases s_0 but P wants to acquire s_1, so that P deadlocks. Since C wants to acquire p as second step, she deadlocks, too. Process $\overline{P}$ will deadlock as well, because he is waiting for either $\overline{s}_0$ or $\overline{s}_1$.

Suppose now that $b = b'$, say with $b = 0$. Then there is only one possible run alternating between steps of $u_C(P, 0)$ and $u_P(0)$, until C finally acquires c. Then both C and P have finished the execution of $u_C(P, 0)$ and $u_P(0)$, respectively. Moreover, C re-owns $\{c, s_0, s_1\}$ and P re-owns $\{p\}$. ◀