DISTRIBUTED CONTROLLER SYNTHESIS FOR DEADLOCK AVOIDANCE

HUGO GIMBERT [a], CORTO MASCLE [b], ANCA MUSCHOLL [b], AND IGOR WALUKIEWICZ [a]

[a] Université de Bordeaux, CNRS, France
e-mail address: hugo.gimbert@labri.fr, igw@labri.fr

[b] Université de Bordeaux, France and MPI-SWS Kaiserslautern, Germany
e-mail address: cmascle@mpi-sws.org

[c] Université de Bordeaux, France
e-mail address: anca@labri.fr

Abstract. We consider the problem of distributed control for systems synchronizing over locks. The goal is to find a local controller for each of the processes so that global deadlocks of the system are avoided. Without restrictions this problem is shown to be undecidable, even for a fixed number of processes and locks. We identify two restrictions that help to recover decidability. The first one is that each process can use at most two locks. The control problem is shown to be Σ_2^P-complete in this case, and even in Ptime under some additional assumptions. The paradigmatic example of the dining philosophers satisfies these assumptions. The second restriction is the nested usage of locks. In this case the distributed control problem is shown to be NExpTime complete. The drinking philosophers problem falls in this case.

1. Introduction

Automatic 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 too big to handle. The other important point is that distributed synthesis produces by definition a distributed system, while a central controller may not be implementable on a given distributed architecture. Unfortunately, very few settings are known for which distributed synthesis is decidable, and those we know of require very high complexity.

Distributed synthesis was first formulated in a synchronous setting by Pnueli and Rosner [PR90]. 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 [KV01, MT01, FS05]. In addition, the complexity is non-elementary in the size of the pipeline. These negative results motivated later a strand of work around distributed controller synthesis in the setting of Zielonka automata, in particular synthesis with so

Key words and phrases: distributed synthesis, lock synchronisation, deadlock avoidance.

Preprint submitted to
Logical Methods in Computer Science

called causal memory. Here the problem was shown decidable for co-graph action alphabets [GLZ04], and for tree architectures of processes [GGMW13, MW14]. Yet the complexity can be again non-elementary, e.g. in the depth of the tree representing the acyclic architecture. Worse, it has been recently established that distributed synthesis with causal memory is undecidable for unconstrained process architectures [Gim22]. Distributed synthesis for (safe) Petri nets [FO17] has encountered a similar line of limited advances, and due to [Gim22], is undecidable in the general case, too, since it is inter-reducible to distributed synthesis for asynchronous automata [BFHH19]. This situation raised the question if there is any natural setting for distributed synthesis that covers some standard examples of distributed systems, and is manageable algorithmically.

In this work we consider distributed systems with a weaker synchronization mechanism, namely lock sharing. Here each process can take or release a lock from a pool of locks. Locks are a classic concept in distributed systems, and one of 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 deadlock. Note that local strategies are purely local: they do not involve any information exchange as in the case of synthesis with causal memory (Zielonka automata or Petri nets). In this sense the synthesis problem resembles the Pnueli and Rosner framework, but for the asynchronicity of processes.

For unrestricted lock-sharing systems we hit again an undecidability barrier, as for the models discussed above. Undecidability was known already for the verification of systems where each process is modeled as a pushdown automaton [KIG05], since unrestricted usage of locks allows for inter-process communication. Yet, we are able to find quite interesting restrictions making distributed control synthesis for lock-sharing systems decidable, and even algorithmically manageable. The first restriction is to limit the number of locks available to each process to two. The standard example is the dining philosophers problem, where each philosopher has two locks corresponding to the left and the right fork. It is important to note that we do *not* limit the *total number* of locks in the system. We show that for such systems the complexity of the synthesis problem is at the second level of the polynomial hierarchy. The problem gets even simpler when we restrict the local strategies such that they cannot block the process when all locks are available. We call such strategies *locally live*. In this case we obtain an NP-algorithm, and even a Ptime algorithm when the access to locks is *exclusive*. The latter means that once a process tries to acquire some lock it cannot switch to another action before getting it. In other words, a process that tries to get a lock is blocked as long as the lock is not available. The second restriction is nested lock usage. This is a very common restriction in concurrent programs [KG06], sometimes enforced syntactically by associating locks with program blocks. Nested lock usage simply says that acquiring and releasing locks should follow a stack discipline. Verification of concurrent programs with nested locks has been shown decidable in [KIG05, KG06], and this triggered further work on extensions of lock usage policies [Kah09, BCMV13, LMSW13]. In distributed computing, the drinking philosophers setting [CM84] is an example of nested lock usage. We show that in this case the distributed synthesis problem is Nexptime-complete, where the exponent in the algorithm depends only on the number of locks available to the process. A decision procedure for the verification of such systems, based on similar ideas on lock orderings, appeared already in [KIG05]. We study here a more general problem, namely distributed

control. Our results are stated for finite-state processes only, in order to keep the setting simple, but they hold for pushdown processes as well.

As mentioned above, we formalize the distributed synthesis problem as a control problem [RW89]. 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 [ELM$^+$16], which are commonly 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.

To summarize, the main results of our work are:

- Σ_2^P-completeness of the deadlock avoidance control problem for systems where each process has access to at most 2 locks (2LSS for short).
- An NP algorithm for 2LSS with locally live strategies.
- A Ptime algorithm for 2LSS with locally live strategies and exclusive lock access.
- A Nexptime algorithm and the matching lower bound for lock-sharing systems with nested lock usage.
- Undecidability of the deadlock avoidance control problem for systems with unrestricted access to locks (with fixed number of processes and locks).

Related work. Distributed synthesis is an old idea motivated by Church's synthesis problem [Chu57]. Actually, the logic CTL has been proposed with distributed synthesis in mind [CE81]. 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 synthesis model has been proposed by Pnueli and Rosner [PR89, PR90]. 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 [KV01, MT01, FS05]. 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 [MT01]. A furthergoing proposal was to consider only input-output specifications. A characterization, still very restrictive, of decidable architectures for this case is given in [GSZ09].

The asynchronous (Zielonka) automaton setting was proposed as a reaction to these negative results [GLZ04]. 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 [GLZ04], connectedly communicating systems [MTY05], and tree architectures [GGMW13, MW14]. There is also a weaker condition covering these three cases [Gim17]. This line of research suffered however from a very recent result showing undecidability in the general case [Gim22].

Distributed synthesis in the Petri net model, called Petri games, has been proposed recently in [FO17]. 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 [BFHH19], hence the undecidability result [Gim22] applies. The problem is EXPTIME-complete for one environment token and arbitrary many system tokens [FO17]. This case stays decidable even for global safety specifications, such as deadlock, but undecidable in general [FGHO22]. As a way to circumvent the undecidability, bounded synthesis has been considered in [Fin15, HM19], where the bound on the size of the resulting controller is fixed in advance. The approach is implemented in the tool ADAMSYNT [GHY21].

The control formulation of the synthesis problem comes from the control theory community [RW89]. 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 [Wal21] 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 [RW92, Tri04, Thi05, AW07].

We would like to mention two further results that do not fit into the main threads outlined above. In [WLK+09] 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 [BBB+20] 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 article. 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 article. 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 3.2 presents an NP algorithm for 2LSS with locally live strategies. Section 3.3 gives a PTIME algorithm for the exclusive case with locally live strategies. Next in Section 4 we consider systems with nested locks, and show that the problem is NEXPTIME-complete in this case. Finally, in Section 5 we prove that without any restrictions the problem is undecidable.

This paper is an extended version of [GMMW22].

To help the reader we use the LaTeX package `knowledge` that hyperlinks definitions with their usage.

2. PRELIMINARIES

A *lock-sharing system* is a parallel composition of processes sharing a pool of locks. Processes do not communicate, but they may acquire or release locks from the pool. Some transitions of processes are uncontrollable, meaning that the environment decides if such a transition is taken. The goal is to find a local strategy for each process so that the system never deadlocks. The challenge is that the strategies are purely local, in the sense that each process only knows its previous actions.

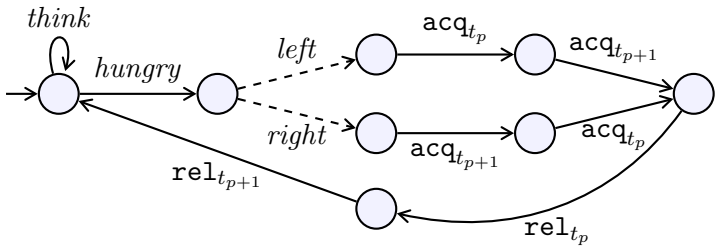


Figure 1: A dining philosopher p. Dashed transitions are controllable.

A *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 consist in acquiring ($\texttt{acq}_t$) or releasing ($\texttt{rel}_t$) some lock t from T_p, or doing nothing: $Op(T_p) = \{\texttt{acq}_t, \texttt{rel}_t \mid t \in T_p\} \cup \{nop\}$. Figure 1 gives an example. For simplicity we write action names in our examples only for *nop*, otherwise we just write the lock operation of the action.

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 and p owns no locks. A transition between two local configurations $(s, B) \xrightarrow{(a,op)}_p (s', B')$ exists when $\delta_p(s, a) = (op, s')$ and one of the following holds:

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

A *local run* $(a_1, op_1)(a_2, op_2) \dots (a_n, op_n)$ of $\mathcal{A}_p$ is a finite sequence over $\Sigma_p \times Op(T_p)$ such that there exists a sequence of local configurations $(init_p, \emptyset) = (s_0, B_0) \xrightarrow{(a_1,op_1)}_p (s_1, B_1) \xrightarrow{(a_2,op_2)}_p \dots (s_n, B_n)$. 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 local runs of $\mathcal{A}_p$. We call a local run *neutral* if it starts and ends with the same set of locks.

A *lock-sharing system* (LSS) $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$ is a set of processes together with a partition of actions between *controllable* actions from Σ^s and *uncontrollable* actions from Σ^e, and a set T of locks. We write $T = \bigcup_{p \in Proc} T_p$, 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 are disjoint: $S_p \cap S_q = \emptyset$ and $\Sigma_p \cap \Sigma_q = \emptyset$ for all $p \neq q$. The sets of locks are not disjoint, in general, since processes may share locks.

Example 2.1. The dining philosophers problem can be formulated as a control problem for a lock-sharing system $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$. Let $Proc = \{1, \dots, n\}$ and $T = \{t_1, \dots, t_n\}$ as the set of locks. For every $p \in Proc$, 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, p 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.

The semantics of such systems is *asynchronous*, as a step of computation is simply defined as one process taking a local transition: $C \xrightarrow{(a,op)} C'$ with $C = (s_p, B_p)_{p \in Proc}$ and $C' = (s'_p, B'_p)_{p \in Proc}$ if for some process p, $(s_p, B_p) \xrightarrow{(a,op)}_p (s'_p, B'_p)$ 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. Observe that any action name determines the process that executes it, since the Σ_p are disjoint. A global run w *determines a local run* of each process: $w|_p$ is the projection of w on Σ_p.

A *local strategy* σ_p says which actions p can take depending on its local run so far. Moreover, it cannot block environment actions. Formally, for every $u \in Runs_p$ define $out(u) \subseteq \Sigma_p$ as the set of actions that are possible after u. Then $\sigma_p : Runs_p \to 2^{\Sigma_p}$ is such that $\sigma_p(u) \subseteq out(u)$ provided that $(\Sigma^e \cap out(u)) \subseteq \sigma_p(u)$. 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 run u of p *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 of that process, there is no inter-process communication. 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 2.1. Suppose that both local strategies always say to take the *left* transition. So $hungry^1, left^1, \mathtt{acq}^1_{t_1}, \mathtt{acq}^1_{t_2}$ is a local run of process 1 respecting the strategy; similarly $hungry^2, left^2, \mathtt{acq}^2_{t_2}, \mathtt{acq}^2_{t_1}$ for process 2. (We use superscripts to indicate the process doing an action.) The global run $hungry^1, hungry^2, left^1, left^2, \mathtt{acq}^1_{t_1}, \mathtt{acq}^2_{t_2}$ respects the strategy. It deadlocks, since each philosopher needs a lock the other one owns.

Definition 2.2 (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, we get more efficient algorithms when we exclude strategies that can block a process by itself:

Definition 2.3 (Locally live strategy)**.** A local strategy σ_p for process p is *locally live* if every σ_p-run u of p can be prolonged: there is some $b \in \Sigma_p$ and $op \in Op(T_p)$ such that $u\,(b, op)$ is a σ_p-run, too. A strategy σ is locally live if each of its associated local strategies is so.

In other words, a locally live strategy guarantees that a process does not block if it runs alone according to σ_p. Back to Example 2.1: 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 transition. This is because with our definition, 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 2.4. We can assume w.l.o.g. that LSS are *lock-aware*: by this we mean that every process knows from its local state which locks it holds, and it never tries to acquire a lock that it already owns, or release a lock that it does not own. Note that enforcing lock-awareness does not compromise the complexity results when processes can access only a fixed number of locks. We will not use lock-awareness in Section 4, where a process can access arbitrarily many locks (in nested fashion).

Without any restrictions our synthesis problem is undecidable. The proof of the theorem below is in Section 5.

Theorem 2.5. *The deadlock avoidance control problem for arbitrary LSS is undecidable (even when the number of locks and processes is fixed).*

We propose then two interesting cases when the control problem becomes decidable.

In the first case each process accesses at most two different locks. In the following definition, we require each process to use exactly two locks, as it is more convenient to avoid case distinctions on the number of locks used by a process. This is not more restrictive as we can always add some dummy locks, which are never used.

Definition 2.6 (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 a 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 a 2LSS. Our first main result says that the control problem is decidable for 2LSS.

Theorem 2.7. *The deadlock avoidance control problem for 2LSS is Σ_2^p-complete.*

The second main result says that restricting to locally live strategies helps to obtain a quite tractable case:

Theorem 2.8. *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 get a Ptime algorithm under one more assumption:

Definition 2.9 (Exclusive systems)**.** A process p is *exclusive* if for every state $s \in S_p$: if s has an outgoing transition with some $\texttt{acq}_t$ operation then all outgoing transitions have the same $\texttt{acq}_t$ operation. A system is *exclusive* if all its processes are.

Example 2.10. The process from Figure 1 is exclusive, while the one from Figure 2 is not. The latter has a state with one $\texttt{acq}_{t_{p+1}}$ and one $\texttt{rel}_{t_p}$ 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 2.11. *The deadlock avoidance control problem for exclusive 2LSS is in* Ptime*, when strategies are required to be locally live.*

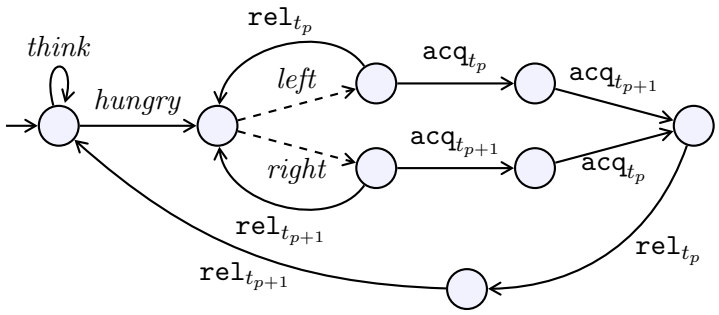


Figure 2: A flexible philosopher p. She can release a fork if the other fork is not available.

Without local liveness, the problem for exclusive 2LSS remains Σ_2^p-hard.

The second case we consider is a common restriction on the usage of locks:

Definition 2.12 (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 process is *nested-locking* if all its local runs are, and an LSS is nested-locking if all its processes are.

Note that none of the processes in Figures 1 and 2 are nested-locking. However, both can be made nested-locking by remembering in the local state in which order the locks were obtained. With this information one can easily determine if an LSS is nested-locking.

Theorem 2.13. *The deadlock avoidance control problem for nested-locking LSS is* NEXPTIME*-complete.*

3. Two locks per process

We describe how to solve the deadlock avoidance control problem for 2LSS, so for systems where every process uses at most two locks. We present the three results announced in the previous section, namely, Theorems 2.7, 2.8, and 2.11.

The general case, treated in Theorem 2.7, puts no restriction on strategies or on the system, besides being a 2LSS. The main idea is that each winning strategy can be decomposed into local strategies, each summarized by an object of polynomial size, called its behavior. We show that from a computational complexity perspective we cannot do better than guessing these behaviors to solve the problem.

The next case is when we require strategies to be locally live. With such strategies, a process can only block if all locks it asks for are taken forever. This simplifies the analysis and enables us to reason on a graph because of the two-locks restriction.

Finally, we consider the restriction of the deadlock avoidance problem to *exclusive* systems, still with locally live strategies. Here, whenever a process can execute an action acquiring a lock it is the only thing it can do. This means that a process gets blocked whenever it tries to get a certain lock that is not available. Recall that the system in Figure 2 is not exclusive, whereas the one in Figure 1 is so.

Throughout this section we fix a 2LSS $\mathcal{S} = ((\mathcal{A}_p)_{p\in Proc}, \Sigma^s, \Sigma^e, T)$ over the set of processes $Proc$. We also assume that the 2LSS is lock-aware (cf. Remark 2.4). We also fix a control strategy $\sigma = (\sigma_p)_{p\in Proc}$.

The three following subsections present the three cases.

3.1. **The general case of 2LSS.** We will use summaries of local runs through so-called *patterns*, that describe the most recent lock operations. We will see later that this information is sufficient to decide if the strategy is winning (Lemma 3.3). Informally, a *pattern of a local run* of process p in a 2LSS describes which of the four following situations are possible for p at the end of its run:

- p owns both locks;
- p owns no lock;
- p owns exactly one of its locks, say t, and either
  - its last operation on locks was $\texttt{acq}_t$; or
  - the last operation on locks was $\texttt{rel}_{t'}$ with $t \neq t'$.

Before defining patterns formally we introduce the runs for which we need them, which are runs that lead potentially to deadlocks:

Definition 3.1 (Risky run)**.** Consider a local σ-run u of a process p. We say that u is σ-*risky* if after executing u all transitions allowed by σ are $\texttt{acq}$ transitions[1]. We simply write risky when σ is clear from the context.

We write $\mathsf{Owns}_{p,\sigma}(u)$ for the set of locks owned by p after u, or simply $\mathsf{Owns}_p(u)$ when σ is clear from context. We write $\mathsf{Blocks}_{p,\sigma}(u) = \{t : \texttt{acq}_t \in \sigma_p(u)\}$, or simply $\mathsf{Blocks}_p(u)$ when σ is clear from context.

Note that if a σ-run u is risky and the strategy σ is locally live, then $\mathsf{Blocks}_p(u) \neq \emptyset$; if σ is not locally live then $\mathsf{Blocks}_p(u)$ can be empty. If the run is not risky then the process can do some local action or a release action.

We can now define patterns formally.

Definition 3.2 (Patterns)**.** Consider a **risky** local σ-run u of process p. We say that u has a *strong pattern* $\mathsf{Owns}_p(u) \Longrightarrow \mathsf{Blocks}_p(u)$ if $\mathsf{Owns}_p(u) \neq \emptyset$ and the last operation on locks in u is a release. Otherwise we say that u has a *weak pattern* $\mathsf{Owns}_p(u) \dashrightarrow \mathsf{Blocks}_p(u)$. We also write $\mathsf{Owns}_p(u) \longrightarrow \mathsf{Blocks}_p(u)$ if we do not specify if a pattern is strong or weak.

We say that σ *admits a pattern* $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p$ ($\mathsf{Owns}_p \Longrightarrow \mathsf{Blocks}_p$, $\mathsf{Owns}_p \dashrightarrow \mathsf{Blocks}_p$, resp.) for process p if there exists some risky σ-run u of p with $\mathsf{Owns}_p = \mathsf{Owns}_p(u)$, $\mathsf{Blocks}_p = \mathsf{Blocks}_p(u)$ and this kind of pattern (strong, weak, resp.).

We write $\mathbb{P}^\sigma_p$ for the set of patterns for p admitted by σ. We write $\mathbb{P}^\sigma = (\mathbb{P}^\sigma_p)_{p \in Proc}$ and denote $\mathbb{P}^\sigma$ as the *behavior* of σ.

We will refer to patterns of process p as $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p$, in order to stress the name of the process, and we always assume that $\mathsf{Owns}_p \cap \mathsf{Blocks}_p = \emptyset$. Since in a 2LSS any process uses two locks, a strong pattern $\mathsf{Owns}_p \Longrightarrow \mathsf{Blocks}_p$ for p is such that $\mathsf{Owns}_p = \{t_1\}$, and Blocks_p is either $\{t_2\}$ or $\emptyset$, where t_1, t_2 are the two locks used by p. For example, the 2LSS in Figures 1 and 2 admit only weak patterns. Consider now the 2LSS in Figure 3. If the strategy of process p_i takes only the lower branch, then its patterns are $\emptyset \dashrightarrow \{x_i\}$, $\{x_i\} \dashrightarrow \{\overline{x_i}\}$ and $\{x_i\} \Longrightarrow \emptyset$. If the strategy allows both branches then we add another strong pattern, $\{\overline{x_i}\} \Longrightarrow \emptyset$.

The next lemma characterizes winning strategies in terms of patterns.

[1] A particular case is where after u no transitions are possible at all.

Lemma 3.3. *Let $\sigma = (\sigma_p)_{p\in Proc}$ be a strategy and $\mathbb{P}^\sigma = (\mathbb{P}^\sigma_p)$ its behavior. Then σ is **not** winning if and only if for every p there is some pattern $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p$ in $\mathbb{P}^\sigma_p$ such that all conditions below hold:*

- $\bigcup_{p\in Proc} \mathsf{Blocks}_p \subseteq \bigcup_{p\in Proc} \mathsf{Owns}_p$,
- *the sets Owns_p are pairwise disjoint,*
- *there exists a total order $<$ on T such that for all p, if p admits a strong pattern $\{t\}\Longrightarrow\mathsf{Blocks}_p$ then $t < t'$, where t' is the other lock used by p.*

Proof. Suppose that σ is not winning, let u be a global σ-run ending in a deadlock, and for each process p let u_p be the corresponding local run.

For every p, the local run u_p has to be risky, otherwise u_p could be extended into a longer run consistent with σ. Thus u_p has a pattern $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p$ in $\mathbb{P}^\sigma_p$.

We check that these patterns meet all requirements of the lemma. Clearly as we are in a deadlock, the only actions available to each process acquire locks that are already taken, hence the first condition is satisfied. Furthermore, no two processes can own the same lock, implying the second condition. Finally, let $<$ be a total order on locks compatible with the order in u between the last operation on each lock, that is: $t < t'$ if the last operation on t in u is before the last one on t'. If one of t, t' is untouched throughout the run then the order is taken arbitrarily.

Consider a process p using locks t, t' and such that u_p has a strong pattern $\{t\}\Longrightarrow\mathsf{Blocks}_p$. So u_p is of 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 $t < t'$ since the last action on t is before the last action on t'.

We now prove the other direction of the lemma. Suppose that for each p there is a pattern $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p$ in $\mathbb{P}^\sigma_p$ such that those patterns satisfy all three conditions of the lemma. Let $<$ be a total order on locks witnessing the third condition.

By definition, for all p there exists a risky local run u_p with $\mathsf{Owns}_p = \mathsf{Owns}_p(u_p)$ and $\mathsf{Blocks}_p = \mathsf{Blocks}_p(u_p)$. We show now the existence of a global run u with $u_p = u|_p$ for every $p \in Proc$. We start by executing one by one, in some arbitrary order, all the u_p such that $\mathsf{Owns}_p = \emptyset$. After executing each such run, all locks are free, hence we can execute the next one. At the end all locks are still free.

For all p such that $\mathsf{Owns}_p = \{t\}$ and $\mathsf{Owns}_p \dashrightarrow \mathsf{Blocks}_p$ is weak, we can write u_p as $u_1^p(a, \mathtt{acq}_t)u_2^p$ with u_1^p neutral and u_2^p not containing any operation on locks. We can execute u_1^p, which again leaves all locks free as it is neutral.

Next we consider all the processes p where u_p has a strong pattern $\{t_p\}\Longrightarrow\mathsf{Blocks}_p$. We execute all runs u_p according to the order $<$. This is possible, as for each such p we have $t_p < t'_p$, where t'_p is the other lock used by p. The order $<$ guarantees that before executing u_p all locks $t \geq t_p$ are free. In particular, since t_p and t'_p are free, we can execute u_p.

At this point all locks are free except for locks t_p of processes p with a strong pattern $\{t_p\}\Longrightarrow\mathsf{Blocks}_p$. We now come back to the u_p with weak patterns. We execute the remaining parts of u_p, namely $(a, \mathtt{acq}_t)u_2^p$ as above. As u_2^p contains no operation on locks, we only need t to be free to execute this run. As all Owns_q are disjoint, and all locks taken at that point belong to some other Owns_q, t is free, hence all such runs can be executed.

Finally, the remaining runs u_p are the ones such that $\mathsf{Owns}_p = \{t, t'\}$ contains both locks of p. As all Owns_p are disjoint, both t, t' are free, hence u_p can be executed.

We have executed all local runs, therefore we reach a configuration where all processes need some lock from $\bigcup_{p\in Proc} \mathsf{Blocks}_p$ to keep running, and all locks in $\bigcup_{p\in Proc} \mathsf{Owns}_p$ are taken. As $\bigcup_{p\in Proc} \mathsf{Blocks}_p \subseteq \bigcup_{p\in Proc} \mathsf{Owns}_p$, we have reached a deadlock. □

Thanks to Lemma 3.3, in order to decide if there is a winning strategy for a given system it is enough to come up with a set of patterns $\mathbb{P}_p$ for each process p and show two properties:

- there exists a strategy σ such that $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$ for each process p;
- the sets of patterns $\mathbb{P}_p$ do not meet the conditions given by Lemma 3.3.

Note that in the first condition we only require an inclusion because by the previous lemma, the less patterns a strategy allows, the less likely it is to create a deadlock.

We start by showing that given a set of patterns for each process, we can check the first condition in polynomial time.

Lemma 3.4. *Given a behavior $(\mathbb{P}_p)_{p\in Proc}$, it is decidable in* PTIME *whether there exists a strategy σ such that for every p we have $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$.*

Proof. First of all recall that we only need to check for each p that there exists a local strategy σ_p that does not allow any risky run of p with pattern not in $\mathbb{P}_p$.

Let $p \in Proc$ and $\mathcal{A}_p = (S_p, \Sigma_p, T_p, \delta_p, init_p)$ be its transition system. Recall that we assume that $\mathcal{A}_p$ is lock-aware. We can do a bit more: in a state where p owns lock t_1, we store an additional bit of information saying whether p released its other lock t_2 since the last acquisition of t_1. This way, the risky nature of a local run and its pattern depend only on the state in which the run ends and the 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 $\{t_1\}\Longrightarrow\emptyset$.

A local state is called bad if all its outgoing transitions have acquire operations, and there is no subset of outgoing transitions that includes all uncontrollable such transitions and that yields only patterns in $\mathbb{P}_p$. Otherwise, the state is called good.

Clearly, a strategy σ satisfies $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$ iff all states reached by σ_p-runs are good.

To know whether there exists a local strategy σ_p such that all its patterns are in $\mathbb{P}_p$ we proceed as follows. We iteratively delete bad states and all their ingoing transitions. If one of those transitions is uncontrollable 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 $init_p$, we conclude that there is no suitable local strategy. Otherwise the subsystem we obtain has only good states, and it corresponds to a strategy σ_p as desired. □

Proposition 3.5. *The deadlock avoidance control problem for 2LSS is decidable in Σ_2^{P}.*

Proof. The algorithm first guesses a set of patterns $\mathbb{P}_p$ for each process p. Note that the overall size of $\mathbb{P}$ is polynomial in $|Proc|$. By Lemma 3.4, we can then check in polynomial time if there exists a strategy $\sigma = (\sigma_p)_{p\in Proc}$ with σ_p admitting only patterns in $\mathbb{P}_p$. By Lemma 3.3 we can determine in coNP whether σ is winning.

For the correctness of the algorithm observe that if there exists a winning strategy σ then it suffices to guess its behavior $\mathbb{P}^\sigma$. Conversely suppose the algorithm guessed a behavior not meeting the requirements of Lemma 3.3. Then the strategy obtained by Lemma 3.4 is winning. □

Theorem 2.7. *The deadlock avoidance control problem for 2LSS is Σ_2^p-complete.*

Proof. The upper bound follows immediately from Proposition 3.5.

For the lower bound we reduce from ∃∀-SAT. Suppose that we are given a formula in 3-disjunctive normal form $\bigvee_{k=1}^{s} \alpha_k$, so each α_k is a conjunction of three literals $\ell_1^k \wedge \ell_2^k \wedge \ell_3^k$

over a set of variables $\{x_1, \dots, x_n, y_1, \dots, y_m\}$. The question is whether the formula $\varphi = \exists x_1 \dots \exists x_n \forall y_1 \dots \forall y_m, \bigvee_{k=1}^{s} \alpha_k$ is true.

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

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

For each $1 \leq i \leq n$ we have a process p_i for each existentially quantified variable, as depicted in Fig. 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 each universally quantified variable we have a process q_j, $1 \leq j \leq m$, in which the environment has to take y_j or $\overline{y_j}$, and then it blocks.

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

The environment can block all processes $p(\alpha_k)$ with the last type of processes. For each clause α_k and each literal ℓ of α_k there is a process $p(\alpha_k, \ell)$. There the process has to acquire t_k and then ℓ before entering a state with a self-loop. In order to block all processes $p(\alpha_k)$, each t_k has to be taken by a process $p(\alpha_k, \ell)$ for some literal ℓ of α_k. For process $p(\alpha_k, \ell)$ to be blocked, lock ℓ has to be taken before, by some p_i or q_j.

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 a valuation, a lock remaining free corresponds to the literal being true.

If the formula φ is true, then the system chooses the valuation of the x_i's in order to make φ true. As soon as processes p_i, q_j have reached their final state, we also have a valuation for the y_j's. At this point there is at least one clause α_k true, so with all its literals $\ell_1^k, \ell_2^k, \ell_3^k$ true. Observe that among the 4 processes $p(\alpha_k)$ and $p(\alpha_k, \ell_1^k)$, $p(\alpha_k, \ell_2^k)$, $p(\alpha_k, \ell_3^k)$ at least one can reach its self-loop, namely the one that acquires t_k first. Hence, the system does not deadlock. Note also that no winning strategy here can be locally live, because of processes p_i and q_j.

Otherwise, if the formula φ is not true, then for each choice of the system for the x_i's, the environment can chose afterwards a suitable valuation of the y_j's that falsifies φ ("afterwards" means that we look at a suitable scheduling of the acquire actions). For such a valuation, for every α_k there is some literal ℓ^k of α_k that is false. Consider the scheduling that lets $p(\alpha_k, \ell^k)$ acquire t_k first. Since t_k is taken, this implies that $p(\alpha_k, \ell^k)$ is blocked. Also, $p(\alpha_k)$ is blocked because of t_k. The other two processes $p(\alpha_k, \ell)$ with $\ell \neq \ell^k$ are also blocked because of t_k. So overall the entire system is blocked. □

3.2. **Locally live strategies.** We now consider the case of 2LSS with locally live strategies. Such a strategy ensures that no process blocks when running alone. Hence a process can only block if all its available transitions need to acquire a lock, but all these locks are taken. This restriction prevents a construction like the one used to obtain the lower bound of Theorem 2.7.

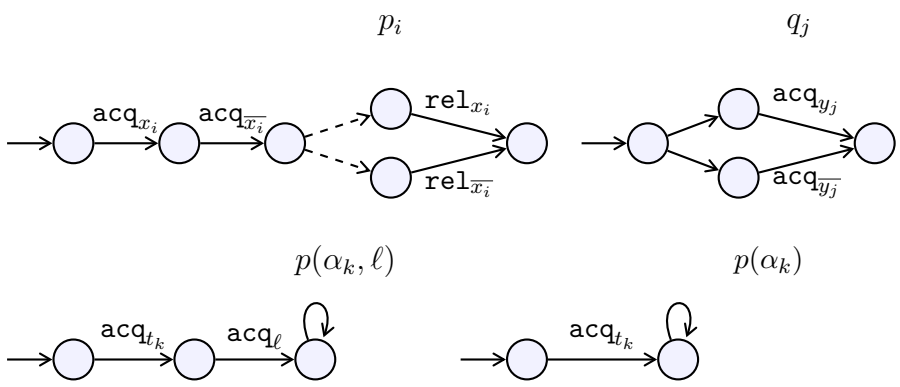


Figure 3: Processes used in Theorem 2.7. Transitions of the system are dashed. All unlabeled transitions carry *nop* as operation. Processes p_i and q_j handle existentially and universally quantified variables, resp.; process $p(\alpha_k, \ell)$ handles literal ℓ in clause α_k, and process $p(\alpha_k)$ handles clause α_k.

In the last subsection we were guessing a behavior of a strategy and then checking in coNP if the condition from Lemma 3.3 does not hold. Here we show that this check can be done in Ptime.

The argument is quite lengthy and requires a precise analysis of the graph representing the guessed behavior. We represent a behavior as a lock graph $G_{\mathbb{P}}$, with vertices corresponding to locks and edges to patterns. Then, thanks to local liveness, instead of Lemma 3.3 we get Lemma 3.10 characterizing when a strategy is not winning by the existence of a subgraph of $G_{\mathbb{P}}$, called (full) deadlock scheme. The main body of the proof is a polynomial time algorithm to decide the existence of full deadlock schemes.

As we are in a locally live framework, some patterns of local runs are impossible. We do not have patterns of the form $\mathsf{Owns}_p \to \emptyset$ as a local run can block only because it requires some locks that are taken. This leaves two possible types of patterns, $\{t_1\} \to \{t_2\}$ and $\emptyset \to \mathsf{Blocks}_p$ for some non-empty $\mathsf{Blocks}_p \subseteq \{t_1, t_2\}$. The set of patterns of the first type defines a graph: an edge labeled by p from t_1 to t_2 represents the pattern $\{t_1\} \to \{t_2\}$ of process p. Recall that this corresponds to a local run ending in a situation when p holds t_1 and all actions need to acquire t_2. The second type of patterns will be incorporated later in form of fragile processes.

We define *weak* and *strong* patterns and cycles, as well as *solid* and *fragile* processes. We are from the point of view of the controller: we want to obtain strong patterns and solid processes, as they make deadlocks less likely.

Definition 3.6 (Lock graph $G_{\mathbb{P}}$)**.** For a behavior $\mathbb{P} = (\mathbb{P}_p)_{p \in Proc}$, we define a labeled graph $G_{\mathbb{P}} = \langle T, E_{\mathbb{P}} \rangle$, called *lock graph*, whose nodes are locks and with two types of edges, weak or strong. Edges are labeled by processes.

There is a *weak edge* $t_1 \overset{p}{\dashrightarrow} t_2$ in $G_{\mathbb{P}}$ whenever there is a weak pattern $\{t_1\} \dashrightarrow \{t_2\}$ in $\mathbb{P}_p$. There is a *strong edge* $t_1 \overset{p}{\Longrightarrow} t_2$ whenever there is a strong pattern $\{t_1\} \Longrightarrow \{t_2\}$ in $\mathbb{P}_p$ **and** there is no weak pattern $\{t_1\} \dashrightarrow \{t_2\}$ in $\mathbb{P}_p$. We write $t_1 \xrightarrow{p} t_2$ when the type of the edge is irrelevant.

A path (resp. cycle) in $G_{\mathbb{P}}$ is called *simple* if all its edges are labeled by different processes. A cycle is *weak* if it contains some weak edge, and *strong* otherwise.

The next definition provides some notions for patterns of the form $\emptyset \longrightarrow \mathsf{Blocks}_p$.

Definition 3.7 (solid/fragile)**.** For a behavior $\mathbb{P} = (\mathbb{P}_p)_{p \in Proc}$, a process p is called *solid in* $\mathbb{P}$ (or just solid, if $\mathbb{P}$ is clear from the context) if there is no pattern of the form $\emptyset \longrightarrow \mathsf{Blocks}_p$ in $\mathbb{P}_p$; otherwise it is called *fragile in* $\mathbb{P}$ (or just fragile).

A process p is called *Z-fragile* if there is some pattern $\emptyset \to B$ in $\mathbb{P}_p$ with $B \subseteq Z$. Note that a process is fragile if and only if it is Z-fragile for some $Z \subseteq T$.

A *solid edge* of $G_{\mathbb{P}}$ is one that is labeled by a solid process. A *solid cycle* is one that only has solid edges.

What the previous definition says is that a solid process needs to take a lock to be blocked, whereas a fragile one can be blocked without owning a lock. So we take into account only solid processes in the deadlock schemes defined next:

Definition 3.8 (Z-deadlock scheme)**.** Consider a behavior $\mathbb{P} = (\mathbb{P}_p)_{p \in Proc}$, and the associated lock graph $G_{\mathbb{P}}$. Let $Z \subseteq T$ be a set of locks. We set $Proc_Z$ as the set of those processes that can access only locks in Z.

A *Z-deadlock scheme for* $\mathbb{P}$ is a partial function $ds_Z : Proc_Z \dot{\to} E_{G_{\mathbb{P}}}$ such that all conditions below are satisfied:

(1) For all $p \in Proc_Z$, if $ds_Z(p)$ is defined then it is a p-labeled edge of $G_{\mathbb{P}}$.
(2) If $p \in Proc_Z$ is solid then $ds_Z(p)$ is defined.
(3) For all $t \in Z$ there exists a unique $p \in Proc_Z$ such that $ds_Z(p)$ is an outgoing edge of t.
(4) The subgraph of $G_{\mathbb{P}}$ restricted to $ds_Z(Proc_Z)$ does not contain any strong cycle.

A *deadlock scheme* for $\mathbb{P}$ is a Z-deadlock scheme for $\mathbb{P}$ for some set Z.

The idea underlying the previous definition is that a Z-deadlock scheme witnesses a way to reach a configuration in which all locks of Z are taken, and all processes from $Proc_Z$ are blocked. Each solid process from $Proc_Z$ is mapped to an edge telling which lock it holds in the deadlock configuration and which one it needs in order to advance.

For every lock in Z there is a unique outgoing edge in ds_Z, corresponding to the process owning that lock. Note that this implies that the subgraph induced by ds_Z is a union of cycles, with some non-branching paths going into these cycles.

The fourth condition excluding strong cycles is required as to be able to schedule the local runs according to the edges of the deadlock scheme into a global run.

A Z-deadlock scheme is not a full witness for deadlock because fragile processes are missing. The next definition takes care of fragile processes. Note that $ds_Z(p)$ is always undefined if $p \notin Proc_Z$.

Definition 3.9 (Full deadlock scheme)**.** A *full Z-deadlock scheme* for a behavior $\mathbb{P}$ is a Z-deadlock scheme ds_Z for $\mathbb{P}$ for some $Z \subseteq T$ such that for every process $p \in Proc$ either $ds_Z(p)$ is defined, or p is Z-fragile. A *full deadlock scheme* for $\mathbb{P}$ is a full Z-deadlock scheme for $\mathbb{P}$, for some set $Z \subseteq T$.

We now prove an analogous result to Lemma 3.3: a strategy is not winning if and only if its lock graph admits a full deadlock scheme. The existence of a winning strategy will be established by non-deterministically guessing a behavior, verifying that there exists a strategy respecting it, computing the corresponding lock graph and then checking that it has no full deadlock scheme. The most involved step is the last one.

Lemma 3.10. *Consider a locally live control strategy σ and $\mathbb{P}^\sigma = (\mathbb{P}^\sigma_p)_{p \in Proc}$ the behavior of σ. The strategy σ is **not** winning if and only if there is a full deadlock scheme for $\mathbb{P}^\sigma$.*

Proof. Suppose that σ is not winning. Then by Lemma 3.3, there exist patterns $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p \in \mathbb{P}_p^\sigma$, one for each p, such that:

- $\bigcup_{p\in Proc} \mathsf{Blocks}_p \subseteq \bigcup_{p\in Proc} \mathsf{Owns}_p$,
- the sets Owns_p are pairwise disjoint,
- there exists a total order $\leq$ on T such that for all p, if $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p$ is a strong pattern of the form $\{t\}{\Longrightarrow}\mathsf{Blocks}_p$ then $t \leq t'$ where t, t' are the two locks used by p.

Let $Z = \bigcup_{p\in Proc} \mathsf{Owns}_p$. For every process $p \in Proc_Z$, define $ds_Z(p)$ as $t_1 \xrightarrow{p} t_2$ if $\mathsf{Owns}_p = \{t_1\}$ and $\mathsf{Blocks}_p = \{t_2\}$. Note that $ds_Z(p)$ is undefined if $\mathsf{Owns}_p = \emptyset$.

Moreover, there are no other possible cases above, as σ is locally live and thus Blocks_p cannot be empty.

We show that ds_Z is a full Z-deadlock scheme for $\mathbb{P}^\sigma$ by checking the four conditions from Definition 3.8. The first condition holds by definition of ds_Z. For the second condition let $p \in Proc_Z$ and suppose p is solid. Thus, Owns_p is not empty, hence $ds(p)$ is defined. For the third condition let $t \in Z$. As Z is the disjoint union of the sets Owns_p there exists a unique $p \in Proc_Z$ such that $t \in \mathsf{Owns}_p$, so a unique edge $ds(p)$ outgoing from t. For the last condition note that for all strong edges $t{\overset{p}{\Longrightarrow}}t'$ the pattern $\mathsf{Owns}_p{\Longrightarrow}\mathsf{Blocks}_p$ must be strong as well, hence $t \leq t'$. As $\leq$ is a total order on locks, there cannot be any strong cycle.

Finally, suppose that $p \notin Proc_Z$ or $ds(p)$ is undefined. In both cases $\mathsf{Owns}_p = \emptyset$, thus p is Blocks_p-fragile, and hence Z-fragile as $\mathsf{Blocks}_p \subseteq Z$. As a consequence, ds is a full Z-deadlock scheme for $\mathbb{P}^\sigma$.

For the other direction, suppose we have a full Z-deadlock scheme ds for $\mathbb{P}^\sigma$, for some set Z of locks. For each process $p \in Proc$ we can find a pattern $\mathsf{Owns}_p \longrightarrow \mathsf{Blocks}_p \in \mathbb{P}_p^\sigma$ as follows:

- If $ds(p)$ is undefined or $p \notin Proc_Z$ then p is Z-fragile. In this case we choose $\mathsf{Blocks}_p \subseteq Z$ such that $\emptyset \longrightarrow \mathsf{Blocks}_p \in \mathbb{P}_p^\sigma$ and set $\mathsf{Owns}_p = \emptyset$.
- If $ds(p) = t_1 \xrightarrow{p} t_2$ then there exists a pattern $\{t_1\} \longrightarrow \{t_2\} \in \mathbb{P}_p^\sigma$ with $\{t_1, t_2\} \subseteq Z$. We set $\mathsf{Owns}_p = \{t_1\}$ and $\mathsf{Blocks}_p = \{t_2\}$.

We check now the conditions of Lemma 3.3.

As all locks of Z have exactly one outgoing edge in $ds(Proc_Z)$, and as all Owns_p with $p \notin Proc_Z$ or $ds(p)$ undefined are empty, the sets Owns_p are pairwise disjoint. Moreover, $\bigcup_{p\in Proc} \mathsf{Blocks}_p \subseteq Z \subseteq \bigcup_{p\in Proc} \mathsf{Owns}_p$.

It remains to check the last condition. Consider a strong pattern $\mathsf{Owns}_p{\Longrightarrow}\mathsf{Blocks}_p$ with $\mathsf{Owns}_p = \{t\}$. Since σ is locally live we have that $\mathsf{Blocks}_p = \{t'\}$, where t, t' are the two locks used by p. As $ds(Proc_Z)$ does not contain any strong cycle, we can pick a total order $\leq$ on locks such that for every strong edge $t_1{\overset{p}{\Longrightarrow}}t_2$ belonging to $ds(Proc_Z)$, we have $t_1 < t_2$. In particular, $t < t'$, which finishes the proof. □

From now on we fix a behavior $\mathbb{P}$ and its lock graph $G_\mathbb{P}$. We will show how to decide if there is a full deadlock scheme for $\mathbb{P}$ in Ptime. For this we need to be able to certify in Ptime that there is no Z-deadlock scheme for $\mathbb{P}$, as in Definition 3.8. Our approach will be to eliminate edges from $G_\mathbb{P}$ and try to construct a Z-deadlock scheme on increasingly larger sets Z of locks. We will show that this process either yields a set Z that provides a full deadlock scheme for $\mathbb{P}$, or it fails, and in this case there is no full deadlock scheme for $\mathbb{P}$.

The next lemma provides a condition that allows to extend a Z-deadlock scheme towards a full deadlock scheme for $\mathbb{P}$, if one exists. This lemma is a basic ingredient to construct a Z-deadlock scheme for increasingly larger sets Z of locks.

Lemma 3.11. *Let $Z \subseteq T$ be such that there is no solid edge from Z to $T \setminus Z$ in $G_{\mathbb{P}}$. Suppose that $ds_Z : Proc_Z \dot{\to} E$ is a Z-deadlock scheme for $\mathbb{P}$. If there exists some full deadlock scheme for $\mathbb{P}$ then there is one which is equal to ds_Z over $Proc_Z$.*

Proof. Suppose that ds is a full deadlock scheme for $\mathbb{P}$, so ds is a B-deadlock scheme for some $B \subseteq T$ such that for every $p \in Proc$ either $ds(p)$ is defined or p is B-fragile in $\mathbb{P}$. We construct a $(B \cup Z)$-deadlock scheme ds' which is equal to ds_Z over $Proc_Z$. Then we show that ds' is a full $(B \cup Z)$-deadlock scheme for $\mathbb{P}$.

For every process $p \in Proc$, set $ds'(p)$ as:

- $ds_Z(p)$ if $p \in Proc_Z$,
- $ds(p)$ if $p \notin Proc_Z$ and p does not label any edge of $G_{\mathbb{P}}$ from Z to $T \setminus Z$.

First we check that ds' is a $(B \cup Z)$-deadlock scheme. The first condition of a deadlock scheme is satisfied by construction. Recall that we assume that there are no solid edges from Z to $T \setminus Z$. In particular, all processes p such that $ds'(p)$ is undefined are fragile, so the second condition is satisfied as well. By definition of Z-deadlock scheme there is a unique outgoing edge of ds_Z from every lock in Z. A lock $t \in B \setminus Z$ has exactly one outgoing edge in $ds(Proc)$, and this edge in conserved in ds'. Thus, the third condition is satisfied, too. Finally, there cannot be any strong cycle in $ds'(Proc)$ as there are none within Z, nor in $B \setminus Z$, and there are no edges from Z to $T \setminus Z$ in ds'.

It remains to show that ds' is a full $(B \cup Z)$-deadlock scheme for $\mathbb{P}$. Let $p \in Proc$ be an arbitrary process. We make a case distinction on the locks of p. The first case is when both locks are in Z. If p is solid then $ds'(p) = ds_Z(p)$ is defined. If p is fragile then it is Z-fragile, so also $(B \cup Z)$-fragile. The second case is when one lock is in $B \setminus Z$ and the other one in $B \cup Z$. If p is solid then $ds(p)$ must be defined because ds is a full B-deadlock scheme. We must have $ds'(p) = ds(p)$ as there are no solid edges from Z to $T \setminus Z$. If p is fragile then p is B-fragile, so also $(B \cup Z)$-fragile. The final case is when one lock of p is not in $B \cup Z$. Since ds is a full B-deadlock scheme, p must be B-fragile, so also $(B \cup Z)$-fragile. □

Recall that we have fixed a behavior $\mathbb{P}$, and that $G_{\mathbb{P}} = (T, E_{\mathbb{P}})$ is its lock graph. We will describe in the following several polynomial-time algorithms operating on a subgraph $H = (T, E_H)$ of $G_{\mathbb{P}}$, so $E_H \subseteq E_{\mathbb{P}}$, and a set Z of locks.

We will say that H has a *deadlock scheme* to mean that there is a deadlock scheme using only edges in H. The notion of full is the same as for $G_{\mathbb{P}}$.

Each of the four algorithms introduced below will either eliminate some edges from H or extend Z, while maintaining the following three invariants:

Invariant 1. *$G_{\mathbb{P}}$ has a full deadlock scheme for $\mathbb{P}$ if and only if H does.*

Invariant 2. *There are no solid edges from Z to $T \setminus Z$ in H.*

Invariant 3. *There exists a Z-deadlock scheme for $\mathbb{P}$ in $G_{\mathbb{P}}$.*

Invariant 1 expresses that the edges we removed from $G_{\mathbb{P}}$ to get H were not essential for finding a full deadlock scheme for $\mathbb{P}$. Invariant 2, along with Lemma 3.11, will guarantee that we can always extend a Z-deadlock scheme to a full one, if one exists. Invariant 3 maintains the existence of a Z-deadlock scheme, while Z is growing.

Our algorithm will extend Z as much as possible while maintaining the three invariants. In the end we either obtain a full Z-deadlock scheme for $\mathbb{P}$, or a Z-deadlock scheme that is not full, but cannot be extended anymore. In the second case we show that no full deadlock scheme exists.

We may also at some point observe contradictions in the edges of H that exclude the existence of any full deadlock scheme for H, in which case we can conclude immediately thanks to Invariant 1.

We start with $H = G_{\mathbb{P}}$ and $Z = \emptyset$. All invariants are clearly satisfied.

Our first two algorithms will analyze solid edges in H, since any Z-deadlock scheme is defined over solid processes. The first algorithm will possibly remove some edges, and the second one will look for cycles and possibly enlarge Z. The third algorithm will extend Z by locks that can reach it. Finally, the fourth algorithm will also add to Z weak cycles that are outside of Z.

Definition 3.12 (Double and solo solid edges)**.** Consider a solid process p. We say that there is a *double solid edge* $t_1 \overset{p}{\leftrightarrow} t_2$ in H if both $t_1 \overset{p}{\to} t_2$ and $t_1 \overset{p}{\leftarrow} t_2$ exist in H. We say that $t_1 \overset{p}{\to} t_2$ in H is a *solo solid edge* if there is no $t_1 \overset{p}{\leftarrow} t_2$ in H.

Algorithm 1 below looks for a solo solid edge $t_1 \overset{p}{\to} t_2$ in H and erases all other outgoing edges from t_1. It will be proven correct exploiting the following property:

> ($\star$) If $t_1 \overset{p}{\to} t_2$ is a solo solid edge in H, then any deadlock scheme ds_H in H is such that $ds_H(p) = t_1 \overset{p}{\to} t_2$.

The argument behind Property ($\star$) is that a deadlock scheme needs to map every solid process to one of the two possible edges of the lock graph. So if there is only one (remaining) edge labeled by p, this edge is needed and cannot be deleted.

We repeat this algorithm until no edges are removed. If some call of the algorithm fails then there can be no full deadlock scheme for $\mathbb{P}$ in H. Otherwise the resulting H satisfies the property:

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

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

Algorithm 1 Trimming the graph for one solo solid edge

Look for $t \in T \setminus Z$ with a solo solid edge $t \overset{p}{\to} t' \in E_H$ and some other outgoing edges.
if there is no such edge then stop and report success.
for every edge $t \overset{q}{\to} t'' \in E_H$ from t with $q \neq p$ **do**
    if q is solid and $t \overset{q}{\leftarrow} t'' \notin E_H$ **then**
        return "H has no deadlock scheme for $\mathbb{P}$"
    else
        delete $t \overset{q}{\to} t''$ from E_H
    end if
end for

Lemma 3.13. *Suppose (H, Z) satisfies Invariants 1 to 3. If Algorithm 1 fails then H has no full deadlock scheme for $\mathbb{P}$. After a successful execution of the algorithm all the invariants are still satisfied. If a successful execution does not remove any 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 outgoing solo solid edges. In this case it is impossible to find a full deadlock scheme in H, because of Property $(\star)$ above and since a deadlock scheme has exactly one outgoing edge from each lock.

Finally, if the algorithm succeeds but H' is smaller than H, we must show that all the invariants on page 16 hold. Since the algorithm does not change Z, Invariants 2 and 3 continue to hold. For Invariant 1 we use Property $(\star)$ and the fact that a deadlock scheme has a unique outgoing edge from each lock to conclude that any full deadlock scheme in H is also a full deadlock scheme in H'. For the other direction, a full deadlock scheme in H' is also full in H, as H' is a subgraph of H with the same set of vertices. □

Algorithm 2 below searches for simple cycles formed by solid edges and eventually adds them to Z. If such a cycle is weak then it can be added to Z. If the cycle is strong, it may still be the case that its reversal is weak (see p_1, p_2, p_3 in Figure 4). 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 $t_i \xleftarrow{p_i} t_{i+1}$ (which is solid by definition, since p_i is so). If the reversed cycle is also strong then there is no H-deadlock scheme. Otherwise, it is weak and it can be added to Z. We will show that the result still satisfies the invariants thanks to property (Trim).

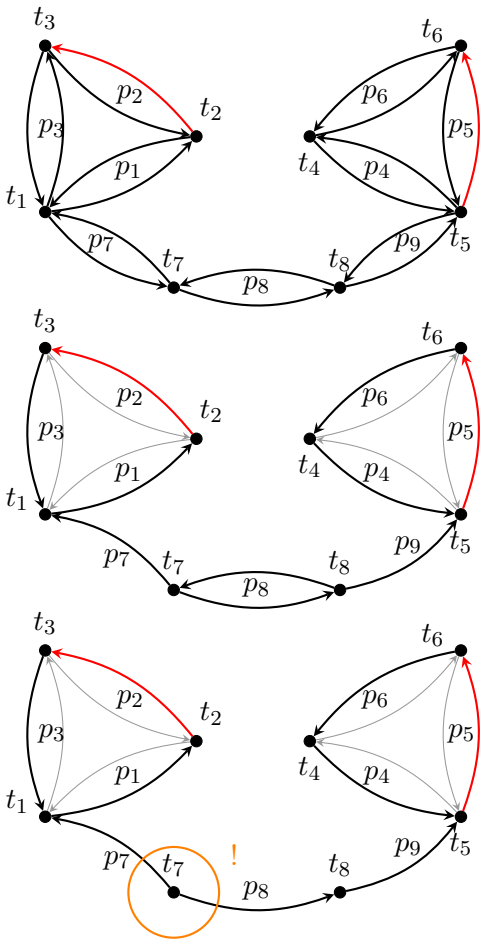
Algorithm 2 Find a simple solid cycle.

Look for 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 and with all t_i distinct.
if there is no such cycle, stop and report success.
if all the edges on the cycle are strong **then**
    if for some j there is no reverse edge $t_j \xleftarrow{p_j} t_{j+1} \in E_H$ **then**
        return "H has no deadlock scheme for $\mathbb{P}$"
    else if all edges $t_j \xleftarrow{p_j} t_{j+1}$ are strong **then**
        return "H has no deadlock scheme for $\mathbb{P}$"
    end if
end if
$Z \leftarrow Z \cup \{t_1, \ldots, t_k\}$
For every t_i remove from E_H all edges outgoing from t_i except for $t_i \xrightarrow{p_i} t_{i+1}$.
if some solid process p has no edge in H **then**
    return "H has no deadlock scheme for $\mathbb{P}$"
end if
repeat
    Apply Algorithm 1
until no more edges are removed from H

Figure 4 presents a case where Algorithm 2 detects an inconsistency in the solid edges, proving the non-existence of a deadlock scheme.

Lemma 3.14. *Suppose (H, Z) satisfies the Invariants 1 to 3 and H is trimmed. If the execution of Algorithm 2 does not fail then the resulting H and Z also satisfy all invariants and (Trim). If the execution fails then H has no full deadlock scheme for $\mathbb{P}$.*

Proof. Suppose that 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



This graph does not have a full deadlock scheme (all processes are solid, weak edges are displayed in red). However a first execution of Algorithm 1 has no effect as all edges are double.

We apply Algorithm 2, which finds solid cycles, erases all other edges going out of those cycles, and makes sure that those cycles are weak.

We now apply Algorithm 1 again. It detects that $t_8 \xrightarrow{p_9} t_5$ is a solo solid edge and it erases the other outgoing edge $t_8 \xrightarrow{p_8} t_7$. It then concludes that there is no full deadlock scheme as t_7 has two outgoing solo solid edges.

Figure 4: An example of application of Algorithm 2.

as well. If there is a full deadlock scheme for H 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, because p_i is solid.

We examine the cases when the algorithm fails. The first reason for failure may appear when all the edges on the cycle are strong. If for some j there is no reverse edge $t_j \xleftarrow{p_j} t_{j+1}$ in E_H then a full deadlock scheme for H, call it ds_H, should assign the edge $t_j \xrightarrow{p_j} t_{j+1}$ to p_j, because the edge is solo solid (recall Property $(\star)$). As a consequence, as ds_H has to give each t_i at most one outgoing edge and all edges of the cycle are solid, all the edges in the cycle should be in the image of ds_H. But this is forbidden by the last condition in the definition of deadlock scheme, as the cycle is strong.

When there are reverse edges $t_i \xleftarrow{p_i} t_{i+1} \in E_H$ for all i, the algorithm fails if all of them are strong. Indeed, there cannot exist any full deadlock scheme for H in this case either, because either the cycle or its reverse would need to be in the image of ds_H, but both are strong.

The last reason for failure is when there is some solid process p and all the 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 $i = 1, \ldots, k$ and $p \neq p_i$. Those edges cannot belong to a deadlock

scheme as it has to contain the cycle in one direction or the other and thus cannot contain other outgoing edges from that cycle. As a deadlock scheme cannot assign any edge to p, and p is solid, there cannot exist any full 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 weak, or its reverse is. Thanks to Lemma 3.13, we only need to show that all three invariants hold after line 11. Let (H', Z') be the values at that point. So $Z' = Z \cup \{t_1, \ldots, t_k\}$, and H' is H after removing edges in line 11. We show now that all invariants on page 16 continue to hold.

For Invariant 2, we observe that thanks to (Trim) for every lock in Z' there is exactly one outgoing edge in H'. So there is no solid edge from Z' to $T \setminus Z'$ as there was none from Z to $T \setminus Z$ and the only solid edge of H' outgoing from t_i is $t_i \xrightarrow{p_i} t_{i+1}$.

For Invariant 3, we extend our Z-deadlock scheme to a Z'-deadlock scheme: we choose the cycle found by the algorithm or its reversal, depending on which one is weak. For every p_i we define $ds_{Z'}(p_i)$ to be the edge in the chosen cycle. For all $p \in Proc_{Z'} \setminus Proc_Z$ other than $p_1, \ldots, p_k$, $ds_{Z'}(p)$ is undefined. We must show that such any such p is fragile. If both locks used by p are among the $\{t_1, \ldots, t_k\}$ then p must be fragile because Algorithm 2 does not fail at line 12. The other case is where p has one lock t in Z, and the other, t' in $Z' \setminus Z$. If p was solid, then given that the algorithm does not fail at line 12, there must be some (solid) edge labeled by p in H'. However, by Invariant 2 for H, an edge from t to t' cannot be solid. Moreover, the edge from t' to t is removed at line 11. Therefore, p is fragile.

For Invariant 1 suppose that H' has a full deadlock scheme for $\mathbb{P}$. Then this is also a full deadlock scheme for H as well, as H' is a subgraph of H over the same set of locks. For the other direction consider a full B-deadlock scheme ds_H in H, for some $B \subseteq T$. By Lemma 3.11, as we showed that Invariant 2 is maintained for Z', we can assume that $Z' \subseteq B$ and ds_H is equal to $ds_{Z'}$ on $Proc_{Z'}$. We define a deadlock scheme $ds_{H'}$ for H' as follows. If $ds_H(p)$ is undefined then $ds_{H'}(p)$ is undefined, too. Otherwise, if the source vertex of $ds_H(p)$ is not in Z' then $ds_{H'}(p) = ds_H(p)$. This edge is guaranteed to exist also in H' because only some edges outgoing from the t_i were removed. 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 $ds_H(p)$ is an edge $t \xrightarrow{p} t'$ with $t \in Z'$ and $t' \notin Z'$. Note that t, t' are both in B. If $t \in Z$ then this would contradict Condition 3 in the definition of deadlock scheme, as $p \notin Proc_Z$. Hence $t = t_i$ for some i, and p is fragile as the only solid edge leaving t_i in H' is $t_i \xrightarrow{p_i} t_{i+1}$. We let $ds_{H'}(p)$ be undefined in this case, and Condition 2 of deadlock scheme is satisfied.

We establish now that $ds_{H'}$ is a full B-deadlock scheme in H'. All we need to check is that any process p with $ds_{H'}(p)$ undefined is B-fragile. If $ds_H(p)$ was already undefined then we get that p is Z-fragile, so B-fragile as well. If $ds_H(p)$ was defined, but $ds_{H'}(p)$ is not, then since both locks of p are in B and p is fragile, we obtain that p is B-fragile. This concludes the proof. □

Lemma 3.15. *If Algorithm 2 succeeds but does not increase Z nor decrease H then (H, Z) satisfies three properties:*

H1 : *H is trimmed.*
H2 : *H has no solid cycle that intersects $T \setminus Z$.*
H3 : *Every solid process has an edge in H.*

Proof. Property H1 is satisfied because H was not modified by Algorithm 1.

Invariant 2 is satisfied by Lemma 3.14, hence any solid simple cycle intersecting $T \setminus Z$ in H must lie entirely in $T \setminus Z$. Moreover, it is easy to see that if there is some solid cycle in H

intersecting $T \setminus Z$, then there exists also a simple one. In this case Algorithm 2 would not have stopped in line 2, and thus would have either failed or increased Z. There is therefore no solid cycle intersecting $T \setminus Z$ in H, hence property H2 is also satisfied.

Finally, Property H3 is satisfied because Algorithm 2 did not fail at line 12-13. □

The next algorithms will not modify H anymore and only increase Z. Therefore, all three properties stated in the previous lemma will continue to hold.

Definition 3.16. Given a pair (H, Z) consisting of a subgraph H of $G_{\mathbb{P}}$ and a set $Z \subseteq T$ of locks we define the following equivalence relation on T: $t_1 \equiv_H t_2$ if $t_1, t_2 \in T \setminus Z$ and there is a path of double solid edges in H 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 (c.f. Lemma 3.18 below) with no outgoing edges (except for singletons, c.f. Lemma 3.17).

Lemma 3.17. *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 $t_1 \equiv_H t_2$.*

Proof. If the $\equiv_H$-equivalence class of t_1 is not a singleton then $t_1 \notin Z$ and there is a double solid edge from t_1. By property H1, there cannot be any outgoing solo solid edge from t_1, so $t_1 \xleftarrow{p} t_2$ must be in H, too. □

Lemma 3.18. *Suppose that H satisfies properties H1 and H2. Let $t_1, t_2 \in T \setminus Z$. If $t_1 \equiv_H t_2$ then H has a unique simple path of solid edges from t_1 to t_2.*

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 double solid edges from t_1 to t_2, hence there is such 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 3.17 all the locks on those paths are in the $\equiv_H$-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 will be shown correct by showing that a Z-deadlock scheme extends to a $(Z \cup C)$-deadlock scheme by orienting edges in C towards Z, as displayed in the example in Figure 5.

Algorithm 3 Extending Z by locks that can reach it.

1: **while** there exists $t_1 \xrightarrow{p} t_2 \in E_H$ with $t_1 \notin Z$ and $t_2 \in Z$ **do**
2: $\quad Z \leftarrow Z \cup \{t \in T \mid t \equiv_H t_1\}$
3: **end while**

Lemma 3.19. *Suppose that H satisfies properties H1, H2 and H3, and (H, Z) satisfies Invariants 1 to 3. After executing Algorithm 3, the resulting H and Z also satisfy all these properties, and H has no edges from $T \setminus Z$ to Z.*

Proof. Let (H', Z') be the pair obtained by applying Algorithm 3. Invariant 1, and properties H1 and H3 continue to hold because $H' = H$. Also property H2 continues to hold, because $Z \subseteq Z'$.

It remains to show that Invariant 2 (no solid edges from Z to $T \setminus Z$) and Invariant 3 (existence of Z-deadlock scheme) are preserved.

Let Z_{m+1} be the value of Z at the end of the 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 3.17 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 ds_m to a Z_{m+1}-deadlock scheme ds_{m+1}. If the two locks of some process q are both in Z_m then $ds_{m+1}(q) = ds_m(q)$. We set $ds_{m+1}(p)$ to be the edge $t_1 \xrightarrow{p} t_2$ found by the algorithm, so here $t_1 \in Z_{m+1} \setminus Z_m$ and $t_2 \in Z_m$. Let C be the $\equiv_H$-equivalence class of t_1: $C = \{t \in T \mid t \equiv_H t_1\}$. By Lemma 3.18 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 let $ds_{m+1}(q)$ be undefined for all remaining processes q.

We verify now that ds_{m+1} is a Z_{m+1}-deadlock scheme. By construction every lock in C has a unique outgoing edge in ds_{m+1}, hence every lock in Z_{m+1} does so. It is also immediate that $ds_{m+1}(Proc_{Z_{m+1}})$ does not contain a strong cycle as it would need to be already the case for ds_m and Z_m: every lock of C 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 remains to show that ds_{m+1} is defined for every solid process $q \in Proc_{Z_{m+1}}$. Suppose by contradiction that $ds_{m+1}(q)$ is not defined by the procedure. If both 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. In the remaining case both locks of q, say t, t', are in C. 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 solid cycle in C. But this is impossible as we assumed that there are no solid cycles 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 what 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. □

Our last algorithm looks for weak cycles in the remaining graph. If it finds one, it adds to Z not only all locks in the cycle but also their $\equiv_H$-equivalence classes.

Algorithm 4 Incorporating weak cycles

1: **if** there exists a weak cycle $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ with $t_k \xrightarrow{p_k} t_1$ weak and $t_i \notin Z$ for some i, **then**
2: $\quad Z \leftarrow Z \cup \bigcup_{i=1}^{k} \{t \mid t \equiv_H t_i\}$
3: **end if**

Lemma 3.20. *Suppose H satisfies H1, H2 and H3, (H, Z) satisfies Invariants 1 to 3, and moreover there are no edges from $T \setminus Z$ to Z. After an execution of Algorithm 4, H still satisfies H1, H2 and H3, and the resulting (H, Z) satisfies Invariants 1 to 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 $Z \subseteq Z'$, so is H2. It remains to verify Invariants 2 and 3.

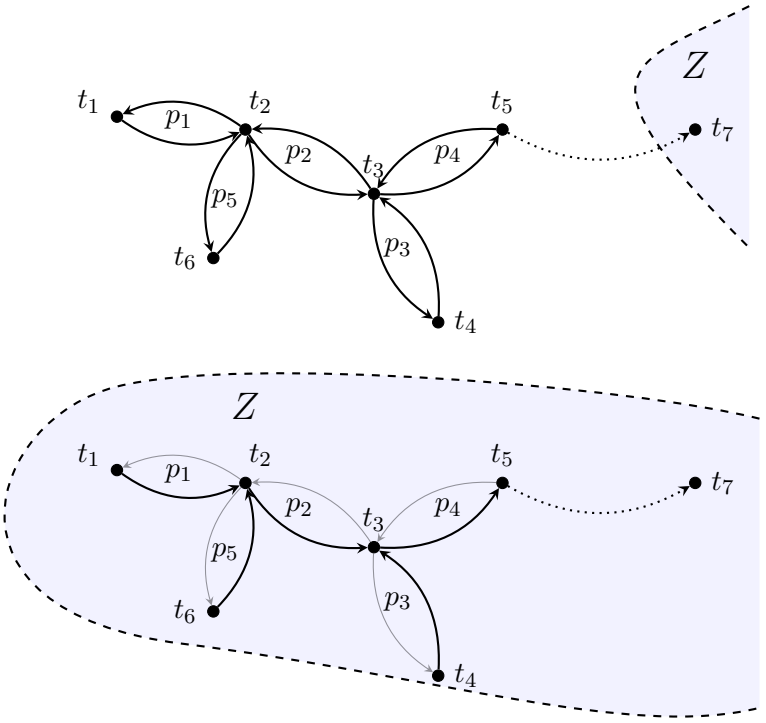


Figure 5: Illustration of Algorithm 3. A deadlock in Z can be extended to all these processes by orienting all edges/processes towards Z (black arrows in the bottom part).

Consider the weak cycle found by the algorithm $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$, and note that $t_i \notin Z$ for all i because H has no edges from $T \setminus Z$ to Z. Let $Z' = Z \cup \bigcup_{i=1}^{k} \{t \mid t \equiv_H t_i\}$ as in line 2.

Towards showing Invariant 2 consider some t_i on the cycle. Lemma 3.17 says that there are no outgoing solid edges from the $\equiv_H$-equivalence class of t_i, unless that class is a singleton. If this class is a singleton, there are no outgoing solid edges from t_i or $t_i \xrightarrow{p} t_{i+1}$ is the only outgoing edge of t_i. In both cases, there are no solid edges from Z' to $T \setminus Z'$ in H.

For Invariant 3 we extend a Z-deadlock scheme ds_Z to a Z'-deadlock scheme $ds_{Z'}$. 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 3.18. If the path starts with $t \xrightarrow{p} t'$ then set $ds_{Z'}(p)$ to this edge. For all remaining processes p we let $ds_{Z'}(p)$ be undefined.

We show now that $ds_{Z'}$ is a Z'-deadlock scheme. First, note that there is an outgoing $ds_{Z'}$ edge from every lock in Z' by definition, and this edge is unique.

Next we 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 in $Z' \setminus Z$ 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 $\equiv_H$-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 now that $t \equiv_H t'$ 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 solid cycles intersecting $T \setminus Z$ in H (Property H2). The last case is when one of the locks of p is in Z and the other 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 for a Z'-deadlock scheme is that there is no strong cycle in $ds_{Z'}(Proc_{Z'})$. We first check that $ds_{Z'}(Proc_{Z'})$ contains $t_k \xrightarrow{p_k} t_1$. This is because t_k is necessarily the last one from its $\equiv_H$-equivalence class. A strong 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 strong cycle in $Z' \setminus Z$ using transitions in $ds_{Z'}$.

Consider x such that $t'_1 \equiv_H t'_j$ for $j \leq x$ 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 weak edge $t_k \xrightarrow{p_k} t_1$ as well, contradicting the assumption that this is a strong cycle. □

We conclude with our complete algorithm (if one of our sub-algorithms returns a result, then the entire algorithm stops):

Algorithm 5 Algorithm to check the existence of a full deadlock scheme for $\mathbb{P}^\sigma$

1: $H \leftarrow G_{\mathbb{P}^\sigma}$
2: $Z \leftarrow \emptyset$
3: **repeat**
4: apply Algorithm 1
5: **until** no more edges are removed from H
6: **repeat** ▷ H is trimmed
7: apply Algorithm 2
8: **until** no more edges are removed from H
9: **repeat** ▷ from now on H satisfies properties H1, H2 and H3
10: apply Algorithm 3 ▷ no edges from $T \setminus Z$ to Z
11: apply Algorithm 4
12: **until** Z does not increase anymore
13: **if** there is a process $p \notin Proc_Z$ that is not Z-fragile **then**
14: **return** "σ is winning"
15: **else**
16: **return** "σ is not winning"
17: **end if**

Lemma 3.21. *Algorithm 5 terminates in polynomial time, and return "σ winning" if and only if no full deadlock scheme for $\mathbb{P}^\sigma$ exists.*

Proof. Let $\mathbb{P} = \mathbb{P}^\sigma$. Suppose that the algorithm fails before reaching the end. If this happens before line 13 then using Lemma 3.13, Lemma 3.14 and Invariant 1 we obtain that $G_{\mathbb{P}}$ does not have any full deadlock scheme for $\mathbb{P}$. If the algorithm fails at line 14 then there exists

a process $p \notin Proc_Z$ that is not Z-fragile. Suppose towards a contradiction that H has a full deadlock scheme ds_H, and assume that ds_H is a B-deadlock scheme for some $B \subseteq T$. By Lemma 3.11 we can assume that $Z \subseteq B$ and ds_H is equal to ds_Z on $Proc_Z$. Observe that one of the locks of p must belong to B, by the definition of full deadlock scheme. So there must exist some outgoing edge from a lock of p, say t, in $ds_H(Proc_B)$. Since $ds_Z(p)$ was undefined, and ds_H, ds_Z coincide on $Proc_Z$, the lock t cannot belong to Z.

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 the image of ds_H. During the last iteration of lines 9-12, Z did not increase, hence by Lemma 3.19 there are no edges from $T \setminus Z$ to Z. This cycle is therefore outside Z. It has to be a weak 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 reaches the end then by Invariant 3 we know that a Z-deadlock scheme for $\mathbb{P}$, say ds_Z, exists. We construct a full deadlock scheme (Z, ds) in $G_{\mathbb{P}}$ as follows. First, we set $ds(p) = ds_Z(p)$ for all $p \in Proc_Z$. For $p \notin Proc_Z$, as the algorithm did not fail at lines 13-14, p is Z-fragile, and we let $ds(p)$ undefined.

Finally, this algorithm runs 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 behavior $(\mathbb{P}^\sigma_p)_{p \in Proc}$ of σ whether there exists $\emptyset \longrightarrow B \in \mathbb{P}_p$ such that $B \subseteq Z$. □

Theorem 2.8. *The deadlock avoidance control problem for 2LSS is in* NP *when strategies are required to be locally live.*

Proof. We start by guessing a behavior $\mathbb{P} = (\mathbb{P}_p)_{p \in Proc}$ such that no $\mathbb{P}_p$ contains any pattern of the form $\mathsf{Owns}_p \longrightarrow \emptyset$. Its size is polynomial in the number of processes. We can check in polynomial time that there exists a strategy respecting the patterns in $\mathbb{P}$ by Lemma 3.4. Note that if there is one, by the requirement we made on $\mathbb{P}$ it must be locally live.

If yes, then we compute the lock graph $G_{\mathbb{P}}$ for $\mathbb{P}$ and check if there is a full deadlock scheme for $\mathbb{P}$ in polynomial time by Lemma 3.21.

By Lemma 3.10, this algorithm answers yes if and only if the system has a locally live strategy that avoids deadlocks. More formally, if there exists a winning, locally live strategy σ then it suffices to guess the behavior $\mathbb{P}^\sigma$ and the Algorithm 5 will return "σ is winning". For the other direction, assume that we guess a behavior $\mathbb{P}$ such that no pattern $\mathsf{Owns}_p \longrightarrow \emptyset$ belongs to $\mathbb{P}_p$, for any p. Assume also that Lemma 3.4 tells that there exists some strategy σ such that $\mathbb{P}^\sigma_p \subseteq \mathbb{P}_p$ for every p. If Algorithm 5 returns "σ is winning" then by Lemma 3.21 we know that there is no full deadlock scheme for $\mathbb{P}$, so there cannot be any for $\mathbb{P}^\sigma$ either. □

3.3. **Exclusive 2LSS.** In this section we study exclusive 2LSS. These systems enjoy enough properties to be able to decide the deadlock avoidance control problem with locally live strategies in polynomial time.

Recall that in an exclusive system, if a state has an outgoing $\mathtt{acq}_t$ transition, then all its outgoing transitions are labeled with $\mathtt{acq}_t$. So in such a state the process is necessarily blocked until t becomes available.

Behaviors of exclusive systems have some special properties, see Lemma 3.24. First, whenever a strategy has a strong pattern $\{t_1\} \Longrightarrow \{t_2\}$ for a process p, it also allows a reverse weak pattern $\{t_2\} \dashrightarrow \{t_1\}$. This will imply that the strong cycle condition in our deadlock

schemes can be satisfied automatically, because any strong cycle can be replaced by a reverse cycle of weak edges. Second, all processes that have some pattern are fragile.

The above observations simplify the analysis of the lock graph. First, we get a much simplified NP argument (Proposition 3.25). This allows us to eliminate guessing and obtain a PTIME algorithm (Proposition 3.29).

Throughout this section we fix an exclusive 2LSS $\mathcal{S}$, and consider only locally live strategies. As we have seen in the previous section, whether or not a strategy σ is winning is determined by its behavior $\mathbb{P}^\sigma$. More precisely, σ is winning if and only if $\mathbb{P}^\sigma$ does not admit a full deadlock scheme, see Lemma 3.10. In this section we show that the latter property can be decided in PTIME for exclusive 2LSS.

Definition 3.22. We call a behavior $\mathbb{P}$ *exclusive* if

- whenever $\mathbb{P}_p$ contains $\{t_1\}\Longrightarrow\{t_2\}$ then it contains either $\{t_1\}\dashrightarrow\{t_2\}$ or $\{t_2\}\dashrightarrow\{t_1\}$, and
- whenever $\mathbb{P}_p$ contains $\{t_1\} \longrightarrow \{t_2\}$ then p is $\{t_1, t_2\}$-fragile in $\mathbb{P}_p$.

Remark 3.23. Assume that we have a strong cycle $t_1 \overset{p_1}{\Longrightarrow} t_2 \overset{p_2}{\Longrightarrow} \cdots \overset{p_k}{\Longrightarrow} t_{k+1} = t_1$ in the lock graph $G_{\mathbb{P}}$ of an exclusive behavior $\mathbb{P}$. Then by definition of strong edges, every p_i has pattern $\{t_i\}\Longrightarrow\{t_{i+1}\}$ but not $\{t_i\}\dashrightarrow\{t_{i+1}\}$. Then by definition of exclusive behavior, they all have a pattern $\{t_{i+1}\}\dashrightarrow\{t_i\}$, hence there is a weak cycle $t_1 = t_{k+1} \overset{p_k}{\dashrightarrow} \cdots \overset{p_2}{\dashrightarrow} t_2 \overset{p_1}{\dashrightarrow} t_1$.

Lemma 3.24. *If σ is a locally live strategy in an exclusive 2LSS and $\mathbb{P}^\sigma = (\mathbb{P}_p)_{p\in Proc}$ is its behavior, then $\mathbb{P}^\sigma$ is exclusive.*

Proof. Consider the first statement. Suppose there is a strong pattern $\{t_1\}\Longrightarrow\{t_2\}$ in $\mathbb{P}_p$, then there exists a local σ-run of p of the form

$$u = u_1(a_1, \mathtt{acq}_{t_1})u_2(a_1, \mathtt{rel}_{t_2})u_3(a_3, \mathtt{acq}_{t_2})\,,$$

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. In consequence, there must be two acquires in u with no release in-between. As the process is exclusive, the state from which the second lock is taken is such that all outgoing transitions take this lock. Thus there is a weak pattern from the first lock taken to the second one. For the second statement, suppose that $\{t_1\} \longrightarrow \{t_2\}$ is in $\mathbb{P}_p$. Thus there exists a σ-run of p making it acquire t_1, so there must be some σ-run u of the form $u = u_1(a, \mathtt{acq}_{t_i})u_2$ for some $i \in \{1, 2\}$ and u_1 containing only local actions. As $\mathcal{S}$ is exclusive, this means that u_1 makes p reach a configuration where all outgoing transitions acquire t_i, and p owns no lock. Since σ is locally live this means that u_1 has the pattern $\emptyset \longrightarrow \{t_i\}$, so p is $\{t_i\}$-fragile, hence also $\{t_1, t_2\}$-fragile. □

Now consider a decomposition of the lock graph $G_{\mathbb{P}}$ of a given behavior $\mathbb{P}$ into strongly connected components (SCC for short). An SCC of $G_{\mathbb{P}}$ is a *direct deadlock* if it contains a simple cycle. A *deadlock SCC* is a direct deadlock SCC or an SCC from which a direct deadlock SCC can be reached.

Figure 6 illustrates these concepts: the left graph has a direct deadlock SCC formed by the three locks at the top. The two remaining locks form a deadlock SCC, because there is a path towards a direct deadlock SCC. Observe that the two locks at the bottom are not a direct deadlock SCC because there is only one process between the two locks and thus no simple cycle within the SCC.

Let $B_{\mathbb{P}} \subseteq T$ be the set of all locks appearing in some deadlock SCC of $G_{\mathbb{P}}$.

Proposition 3.25. *Consider an exclusive behavior $\mathbb{P}$. There is a full deadlock scheme for $\mathbb{P}$ if and only if all processes in Proc are $B_{\mathbb{P}}$-fragile.*

The proof of Proposition 3.25 follows from the lemmas below. In all these lemmas we assume that $\mathbb{P}$ is an exclusive behavior.

Lemma 3.26. *If all processes are $B_{\mathbb{P}}$-fragile then there is a full deadlock scheme for $\mathbb{P}$.*

Proof. We construct a deadlock scheme for $G_{\mathbb{P}}$ as follows. For all direct deadlock SCCs we select a simple cycle inside. By Remark 3.23 and Lemma 3.24, this cycle is weak or has a reverse weak cycle. We select a direction in which the cycle is weak, and for all t in the cycle we set p_t as the process labeling the edge outgoing from t in the cycle.

While there is some edge $t \xrightarrow{p} t'$ in $G_{\mathbb{P}}$ such that p_t is not yet defined but $p_{t'}$ is, we set $p_t = p$. When this ends we have defined p_t for all locks $t \in B_{\mathbb{P}}$. We define ds as $ds(p_t) = t \xrightarrow{p_t} t'$ for all $t \in B_{\mathbb{P}}$. For all other $p \in Proc$, $ds(p)$ is undefined.

We show now that ds is a full deadlock scheme for $\mathbb{P}$. Clearly, for all $p \in Proc$, if $ds(p)$ is defined then it is a p-labeled edge of $G_{\mathbb{P}}$. Furthermore, as all processes are $B_{\mathbb{P}}$-fragile, in particular all processes p with $ds(p)$ undefined are $B_{\mathbb{P}}$-fragile. It is also clear that all locks of $B_{\mathbb{P}}$ have a unique outgoing edge. Finally, by construction we ensured that ds has no strong cycle. □

Lemma 3.27. *Any full Z-deadlock scheme for $\mathbb{P}$ is such that $Z \subseteq B_{\mathbb{P}}$.*

Proof. Suppose that ds_Z full Z-deadlock scheme for $\mathbb{P}$. If there is some $t \in Z \setminus B_{\mathbb{P}}$, then there exists p such that $ds_Z(p) = t \xrightarrow{p} t'$, for some $t' \in Z$. By definition of $B_{\mathbb{P}}$, there are no edges from $T \setminus B_{\mathbb{P}}$ to $B_{\mathbb{P}}$ in $G_{\mathbb{P}}$, hence $t' \in Z \setminus B_{\mathbb{P}}$. By iterating this process we eventually find a simple cycle in $G_{\mathbb{P}}$ outside of $B_{\mathbb{P}}$, which is impossible, as this cycle should be part of a direct deadlock SCC, and thus included in $B_{\mathbb{P}}$. □

Lemma 3.28. *If some process p is not $B_{\mathbb{P}}$-fragile then there is no full deadlock scheme for $\mathbb{P}$.*

Proof. Suppose there exists p that is not $B_{\mathbb{P}}$-fragile. Towards a contradiction assume that there is some full Z-deadlock scheme ds_Z for $\mathbb{P}$, for some Z.

As p is not $B_{\mathbb{P}}$-fragile, then by Lemma 3.27 it is not Z-fragile either. Hence, $ds(p)$ is an edge $t_1 \xrightarrow{p} t_2$ in $G_{\mathbb{P}}$, with $t_1, t_2 \in Z$, and thus $t_1, t_2 \in B_{\mathbb{P}}$. By Lemma 3.24, p is $\{t_1, t_2\}$-fragile, and therefore also $B_{\mathbb{P}}$-fragile, yielding a contradiction. □

This concludes the proof of Proposition 3.25.

Deciding the existence of a winning strategy for exclusive systems. Until now we have assumed that we were given a strategy σ, and we described how to check if it is winning, by constructing $B_{\mathbb{P}}$ and checking that every process is $B_{\mathbb{P}}$-fragile, where $\mathbb{P} = \mathbb{P}^{\sigma}$. Now we want to decide if there is any winning strategy. We use the insights above, but we cannot simply enumerate all exclusive behaviors, as they are exponentially many.

We say below that a strategy σ for p *induces* the edge $t_1 \xrightarrow{p} t_2$ if σ_p admits the pattern $\{t_1\} \longrightarrow \{t_2\}$.

For every process p and every set of edges between two locks of p we can check if there is a strategy for p inducing only edges within this set, as a consequence of Lemma 3.4.

We call an edge $t_1 \xrightarrow{p} t_2$ *unavoidable* if it is induced by every locally live strategy of p.

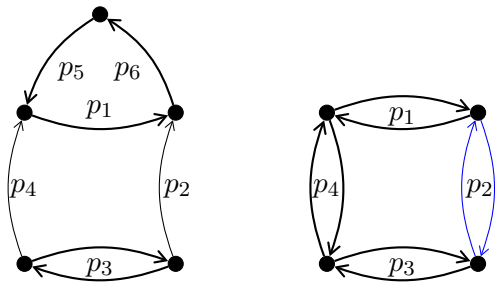


Figure 6: Semi-deadlock SCCs: the blue double edge is not in G_u, but every strategy of the system will induce one of those two edges.

Let G_u be the graph whose nodes are locks and whose edges are the unavoidable edges. We will compute a set of locks B_u in a similar way as $B_{\mathbb{P}}$ in the previous section except that we will use slightly more general basic SCCs of G_u.

A *direct semi-deadlock SCC* of G_u is either a direct deadlock SCC, or an SCC containing only double edges, with two locks t_1 and t_2 such that for some process p using t_1 and t_2, every strategy for p induces at least one edge between t_1 and t_2. Then a *semi-deadlock SCC* of G_u is either a direct semi-deadlock SCC or an SCC from which a direct semi-deadlock SCC can be reached.

Let B_u be the set of locks appearing in semi-deadlock SCCs.

In the graph on the right of Figure 6 the black edges are in G_u, the double blue ones are not, but indicate that every strategy σ of process p_2 induces one of the two blue edges in $G_{\mathbb{P}^\sigma}$. The four locks do not form a direct deadlock SCC of G_u as there is no simple cycle (without the blue edges, which do not belong to G_u). However they do form a direct semi-deadlock SCC, as p_2 will induce an edge no matter its strategy, forming a simple cycle.

Proposition 3.29. *There is a winning strategy for deadlock avoidance iff there exists some process p and a local strategy σ_p that prevents p from acquiring a lock from B_u.*

Proof. One direction is easy: if all strategies make all processes acquire a lock from B_u then there is no winning strategy. Let σ be a strategy, $\mathbb{P} = \mathbb{P}^\sigma$ its behavior and $G_{\mathbb{P}}$ its lock graph. Note that G_u is a subgraph of $G_{\mathbb{P}}$, hence every SCC in $G_{\mathbb{P}}$ is a superset of an SCC in G_u. Observe that if an SCC in $G_{\mathbb{P}}$ contains a direct semi-deadlock SCC of G_u then it is a direct deadlock SCC. Indeed, if an SCC in G_u is a direct semi-deadlock but not a direct deadlock one then σ adds one of edges between the locks of p, say edge $t_1 \xrightarrow{p} t_2$, to this SCC in $G_{\mathbb{P}}$. As t_1, t_2 are in that SCC of G_u, 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 $B_u \subseteq B_{\mathbb{P}}$.

Let $p \in \mathit{Proc}$, as there is a σ-run of p acquiring a lock of B_u, either p is B_u-fragile (and thus $B_{\mathbb{P}}$-fragile) or there is an edge labeled by p towards B_u, meaning that both locks of p are in $B_{\mathbb{P}}$ and thus that p is $B_{\mathbb{P}}$-fragile by Lemma 3.24. As a consequence, all processes are $B_{\mathbb{P}}$-fragile. We conclude by Proposition 3.25.

In the other direction we suppose that there exists a process p and a strategy σ_p forbidding p to acquire any lock of B_u. We construct a strategy σ such that p is not $B_{\mathbb{P}}$-fragile. This will show that σ is winning by Proposition 3.25.

Let $F_u = T \setminus B_u$ be the set of locks not in B_u. By definition of B_u, in G_u no node of F_u can reach a direct semi-deadlock SCC. In particular, there is no direct semi-deadlock

SCC in G_u restricted to F_u. We construct a strategy σ such that, when restricted to F_u, the SCCs of $G_{\mathbb{P}}$ and G_u are the same, where $\mathbb{P} = \mathbb{P}^\sigma$.

Let us linearly order the SCC of G_u restricted to F_u 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_u; or (ii) every local strategy induces some edge that is not in G_u. In the second case there are no q-labeled edges in G_u, and for each of the two possible edges there is a local strategy inducing only this edge.

For a process q from the first case we take a local strategy σ_q that induces only the edges present in G_u.

For a process q from the second case,

- If both locks of q are in B_u then take any local strategy for q.
- If one of the locks of q is in B_u and the other in F_u then choose a strategy inducing an edge from the lock in B_u to the lock in F_u.
- If both locks of q are in F_u then choose a strategy inducing an edge from a smaller to a bigger SCC of G_u.

In the last case, both locks cannot be in the same SCC of G_u: As they are in F_u, this would have to be an SCC with no simple cycles, i.e., a tree of double edges. But then the existence of q implies that this is a direct semi-deadlock SCC, which contradicts the fact that those locks are in F_u.

Consider the graph $G_{\mathbb{P}}$ of the resulting strategy σ. Restricted to F_u this graph has the same SCCs as G_u. Moreover, there are no extra edges in $G_{\mathbb{P}}$ added to any SCC included in F_u, and there are no edges from F_u to B_u. As a result, we have $B_u = B_{\mathbb{P}}$. As p acquires no lock from B_u, it is not B_u-fragile and thus not $B_{\mathbb{P}}$-fragile either. □

Theorem 2.11. *The deadlock avoidance control problem for exclusive 2LSS is in* PTIME*, when strategies are required to be locally live.*

Proof. First we need to compute the unavoidable edges. An edge $t_1 \xrightarrow{p} t_2$ is avoidable iff there exists some locally live strategy σ_p that does not admit the pattern $\{t_1\} \longrightarrow \{t_2\}$. Recall that we assume that $\mathcal{A}_p$ is lock-aware. Then the above means that we look for a locally live strategy σ_p that avoids all states in $\mathcal{A}_p$ where p owns t_1 and needs to acquire t_2. This question reduces to a usual safety game.

Next we have to determine which SCCs of G_u are a direct deadlock, which amounts to check the existence of a simple cycle. Observe that an SCC does not contain such a cycle iff it is a tree of double edges, which is easy to check in PTIME. Knowing whether an SCC is a direct semi-deadlock or a semi-deadlock can also be done in PTIME.

Finally we have to check the condition from Proposition 3.29, so the existence of a locally live strategy σ_p that prevents p to take a look from B_u. As above, this amounts to a safety game. □

4. NESTED LOCKS

We consider now nested-locking LSS, in which the system has to ensure that locks are acquired and released in a stack-like manner. So a process can release only the last lock it has acquired.

Throughout the section the action associated with each operation is omitted, to simplify the presentation.

Local runs of nested-locking LSS have a natural decomposition into staircases:

Definition 4.1. A *stair decomposition* of a local run u is of the form

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

where $u_1, \ldots, u_{k+1}$ are neutral runs, and no u_i uses locks from $\{t_1, \ldots, t_{i-1}\}$.

Lemma 4.2. *Every nested-locking local run u 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 the process, call it p, at the end of the run u, and the distinguished $\mathtt{acq}_{t_i}$ are the last acquisitions of these locks in u. All properties are immediate. □

We now define patterns of risky local runs that will serve as witnesses of reachable deadlocks, in a similar manner as in Definition 3.2.

Definition 4.3. Consider a local risky σ-run u of process p, and its stair decomposition $u = u_1 \, \mathtt{acq}_{t_1} u_2 \, \mathtt{acq}_{t_2} \cdots u_k \, \mathtt{acq}_{t_k} u_{k+1}$. We associate with u a *stair pattern* $(\mathsf{Owns}_p, \mathsf{Blocks}_p, \preceq^p)$, where $\mathsf{Owns}_p = \{t_1, \ldots, t_k\}$, Blocks_p is the set of locks requested by outgoing transitions allowed by σ in the state reached by u, and $\preceq^p$ is the smallest partial order on T_p satisfying:

> For all $1 \leq i \leq k$ and all $t \in T_p$, if the last operation on t in u is after the last $\mathtt{acq}_{t_i}$ then $t_i \preceq^p t$.

A *behavior* of σ is a family of sets of stair patterns $(\mathbb{P}_p^\sigma)_{p \in Proc}$, where $\mathbb{P}_p^\sigma$ is the set of stair patterns of local risky σ-runs of p.

Example 4.4. Consider the local run displayed in Figure 7. It is nested-locking and risky, hence we can define its stair pattern $(\{t_1, t_2, t_4\}, (t_2 < t_1 < t_3 < t_5 < t_4), \{t_3, t_5\})$. This pattern describes the set of locks held at the end, the order on the last operations on each lock appearing in the run, and the set of locks that can be acquired at the end.

Lemma 4.5. *A control strategy σ with behavior $(\mathbb{P}_p^\sigma)_{p \in Proc}$ is **not** winning if and only if for every $p \in Proc$ there is some stair pattern $(\mathsf{Owns}_p, \mathsf{Blocks}_p, \preceq^p) \in \mathbb{P}_p^\sigma$ such that:*

- $\bigcup_{p \in Proc} \mathsf{Blocks}_p \subseteq \bigcup_{p \in Proc} \mathsf{Owns}_p$,
- *the sets Owns_p are pairwise disjoint,*
- *there exists a total order $\preceq$ on the set of all locks that is compatible with all $\preceq^p$.*

Proof. Suppose σ is not winning, and let w be a run leading to a deadlock. For all p let Owns_p be the set of locks owned by p after w. Let $u^p = w|_p$ be 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 $(\mathsf{Owns}_p, \mathsf{Blocks}_p, \preceq^p)$ of u^p. By definition, this is a pattern from $\mathbb{P}_p^\sigma$.

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 $\bigcup_p \mathsf{Blocks}_p \subseteq \bigcup_p \mathsf{Owns}_p$, proving the first condition. Moreover, the sets Owns_p are pairwise disjoint.

For the last requirement of the lemma consider some 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 $u^p = u_1^p \, \mathtt{acq}_{t_1^p} u_2^p \, \mathtt{acq}_{t_2^p} \cdots u_k^p \, \mathtt{acq}_{t_k^p} u_{k+1}^p$ be the stair decomposition of u^p. As p never releases t_i^p, the distinguished $\mathtt{acq}_{t_i^p}$, is the last operation on t_i^p in the global run. Consequently, for all

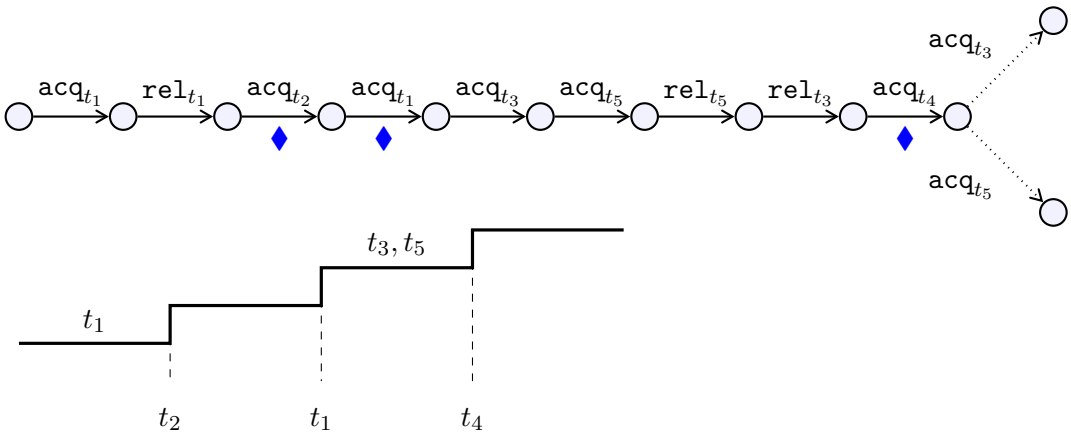


Figure 7: Example of a nested-locking local run. The dotted arrows are the available transitions at the end of the run. Blue diamonds mark transitions taking a lock that is not released later in the run. The lower part shows the stair pattern of this run (without the Blocks part). Steps represents the points at which a lock is taken and not released later. On each step we write the set of locks used in the corresponding section of the run.

t we have $t_i^p \preceq t$ whenever t is used in $u_{i+1}^p \mathtt{acq}_{t_{i+1}^p} \cdots u_k^p \mathtt{acq}_{t_k^p} u_{k+1}^p$. Hence, $\preceq$ 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 $(\mathsf{Owns}_p, \mathsf{Blocks}_p, \preceq^p)$ coming from a local σ-run u^p of p, with $u^p = u_1^p \mathtt{acq}_{t_1^p} u_2^p \mathtt{acq}_{t_2^p} \cdots u_k^p \mathtt{acq}_{t_k^p} u_{k+1}^p$ as stair decomposition. There is also a linear order $\preceq$ compatible with all $\preceq^p$. Let $\prec$ be its strict part. Let $t_1, \ldots, t_k$ be the sequence of locks from $\bigcup_p \mathsf{Owns}_p$ listed according to $\prec$. Let $\{p_1, \ldots, p_n\} = \mathit{Proc}$. We claim that we can get a suitable global run w as $u_1^{p_1} \ldots u_1^{p_n} w'$ where w' is obtained from $t_1 \ldots t_k$ by substituting each t_i^p by $\mathtt{acq}_{t_i^p} u_{i+1}^p$. Observe that every t_j from the sequence $t_1 \ldots t_k$ corresponds to exactly one t_i^p, as the sets $\mathsf{Owns}_{p_1}, \ldots, \mathsf{Owns}_{p_n}$ are disjoint.

All u_i^p are neutral, hence after executing $u_1^{p_1} \ldots u_1^{p_n}$ all locks are free. Let $t_i^p \in T_p$, suppose furthermore that all $\mathtt{acq}_{t_j^q} u_{j+1}^q$ with $t_j^q \prec t_i^p$ have been executed after $u_1^{p_1} \ldots u_1^{p_n}$. Then the set of taken locks is $\{t_j^q \mid t_j^q \prec t_i^p\}$. As $\preceq$ is compatible with all $\preceq^p$, all locks t used in $\mathtt{acq}_{t_i^p} u_{i+1}^p$ are such that $t_i^p \preceq t$. Moreover, since all t_j^q that were taken before are such that $t_j^q \prec t_i^p$, the run $\mathtt{acq}_{t_i^p} u_{i+1}^p$ uses only locks that are free and can therefore be executed.

To sum up, w can be executed. It ends in a deadlock as $\bigcup_p \mathsf{Blocks}_p \subseteq \bigcup_p \mathsf{Owns}_p$. □

Lemma 4.6. *Given a nested-locking LSS $\mathcal{S}$, a process $p \in \mathit{Proc}$ and a set of patterns $\mathbb{P}_p$, we can check in polynomial time in $|\mathcal{A}_p|$ and $2^{|T| \log(|T|)}$ whether there exists a strategy σ with $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$.*

Proof. Fix a process p. We extend the states of p to keep track of the set of locks held by p as well as the order $\preccurlyeq$ induced by the stair pattern of the run seen so far (as in Definition 4.3). This increases the number of states by the factor $|T|! \cdot 2^{|T|}$.

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.

Consider a state s where all outgoing transitions have a lock acquisition as operation. Thanks to the previous paragraph, s determines the set of locks $\mathsf{Owns}(s)$ and an order $\prec_s$ such that every local run ending in s has a pattern $(\mathsf{Owns}^s, B, \prec_s)$, where B depends on the choices a strategy for p makes in s. We mark s bad if none of these possible patterns is in $\mathbb{P}_p$.

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 because reaching that state would make the environment able to reach a bad state. If this process marks the initial state bad then there is no local strategy with patterns included in $\mathbb{P}_p$. Otherwise, we look for new bad states as in the previous paragraph. Indeed, a state may satisfy the conditions of the previous paragraph after removing some of its outgoing transitions, for example a transition not accessing locks. If some new state is marked bad then we repeat the whole procedure.

When this double loop stabilizes and if the initial state is not marked bad, then the remaining transitions form a strategy for p with all patterns in $\mathbb{P}_p$. □

Proposition 4.7. *The deadlock avoidance control problem is decidable for nested-locking lock-sharing systems in non-deterministic exponential time.*

Proof. First we apply the first step of Lemma 4.6 so that every state encodes which locks are taken, and in which order. In this way we ensure that every release is applied in nested manner.

The decision procedure for the existence of a winning strategy guesses a behavior $\mathbb{P}_p$ for each process p. The size of the guess is at most $2^{2|T|} \cdot |T|!$, hence $2^{O(|T| \log(|T|)}$. Then it checks if there exist local strategies yielding subsets of those behaviors. This takes exponential time by Lemma 4.6. If the result is negative then the procedure rejects. Otherwise, it checks if some condition from Lemma 4.5 does not hold. It it finds one then it accepts, otherwise it rejects.

Clearly, if there is a winning strategy then the procedure can accept by guessing the family of behaviors corresponding to this strategy. For these behaviors the check from Lemma 4.6 does not fail, and one of the conditions of Lemma 4.5 must be violated.

Conversely, if the decision procedure concludes that there exists a winning strategy, then let $(\mathbb{P}_p)_{p \in Proc}$ be the guessed family of behaviors. We know that there exists a strategy σ with behaviors $(\mathbb{P}'_p)_{p \in Proc}$ such that $\mathbb{P}'_p \subseteq \mathbb{P}_p$ for all $p \in Proc$. Furthermore, as there are no patterns in $(\mathbb{P}_p)_{p \in Proc}$ satisfying the requirements of Lemma 4.5, there cannot be any in the $\mathbb{P}'_p$ either. Hence σ is a winning strategy. □

Theorem 2.13. *The deadlock avoidance control problem for nested-locking LSS is* NEXPTIME*-complete.*

Proof. The upper bound is given by Proposition 4.7. For the lower bound, we reduce from 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 question is whether there exists a mapping $t : \{0, \dots, 2^n - 1\}^2 \to D$ representing a valid tiling of the grid, i.e. such that for all $x, y, x', y' \in \{0, \dots, 2^n - 1\}$:

- if $x' = x$ and $y' = y + 1$ then $up_{t(x,y)} = down_{t(x',y')}$
- if $x' = x + 1$ and $y' = y$ then $right_{t(x,y)} = left_{t(x',y')}$
- if $x = 0$ then $left_{t(x,y)} = b$
- if $x = 2^n - 1$ then $right_{t(x,y)} = b$
- if $y = 0$ then $down_{t(x,y)} = b$
- if $y = 2^n - 1$ then $up_{t(x,y)} = b$

The above problem is well-known to be NEXPTIME-complete.

Let n, Σ, D, b be an instance of the tiling problem. We construct a LSS as follows: We have three processes p, $\overline{p}$ and q. Process p uses locks from $\{0_i^x, 1_i^x, 0_i^y, 1_i^y \mid 1 \leq i \leq n\}$, together with a lock t_d for each domino $d \in D$, and an extra lock called simply ℓ. Process $\overline{p}$ will use similar locks but with a bar: $\overline{0_i^x}$, $\overline{1_i^x}$, $\overline{0_i^y}$, $\overline{1_i^y}$, $\overline{t_d}$, $\overline{\ell}$. Process q will use all the locks of p and $\overline{p}$.

Let us describe process q represented in Figure 9. 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 the systems is the same. In the first phase, for each $1 \leq i \leq n$, the environment can choose to take either lock 0_i^x or 1_i^x, and then take either $\overline{0_i^x}$ or $\overline{1_i^x}$. In the second phase the same happens for y locks. After these two phases the 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. $\#x = 2^n - 1$).
- If it is b_{down} (resp. b_{up}) then $\#y = 0$ (resp. $\#y = 2^n - 1$).

All these constraints are easily implemented. For example, after *equality* the environment must take the same bits for $\overline{x}$ as for x (similarly for y).

In the third phase, process q has to take and then immediately release locks ℓ and $\overline{\ell}$, before it reaches a state called *dominoes*. Note that every state in the three phases before *dominoes* has a 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 has chosen *equality* then $d = \overline{d}$.
- If it has chosen *vertical* then $up_d = down_{\overline{d}}$.
- If it has chosen *horizontal* then $right_d = left_{\overline{d}}$.
- If it has chosen 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 called *win* with a local loop on it and no other outgoing transitions.

We now describe process p represented in Figure 9. It starts by taking the lock ℓ, which it never releases. Then the environment chooses to take one of 0_i^x and 1_i^x and one of 0_i^y and 1_i^y 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 $t : \{0, \ldots, 2^n - 1\}^2 \to D$ then there is a winning strategy. The strategy for q is to respond with the correct tiles: if the environment chooses

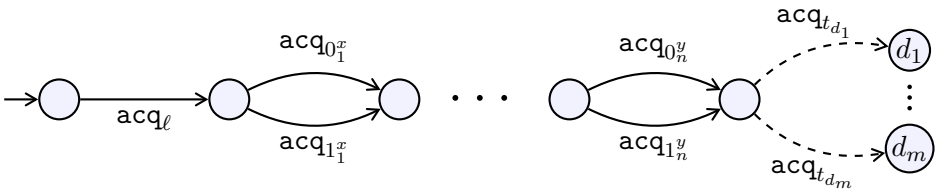


Figure 8: Transition system for process p for the proof of Theorem 2.13 (with $D = \{d_1, \ldots, d_m\}$). Dashed arrows are controlled by the system.

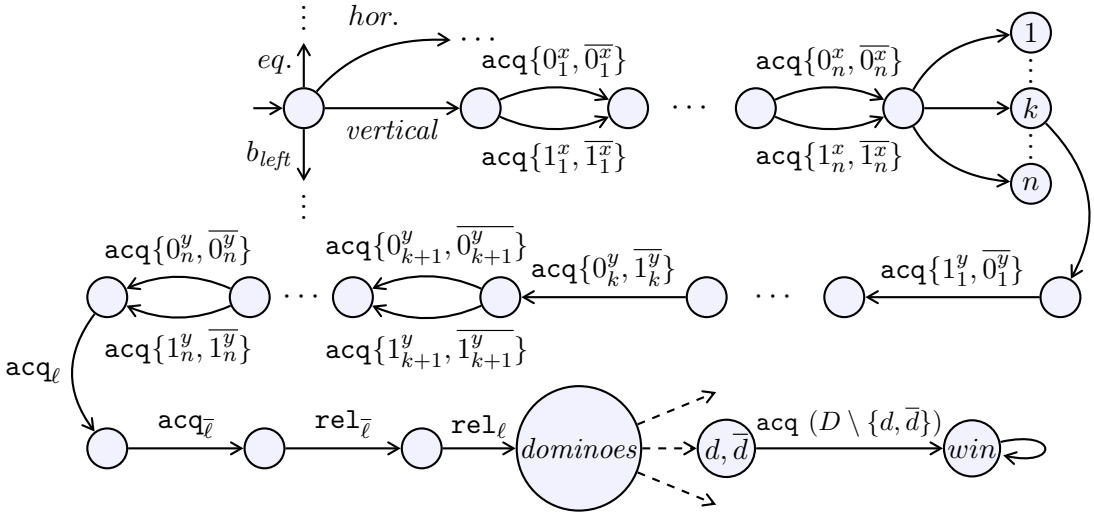


Figure 9: Transition system for process q in the proof of Theorem 2.13. Dashed arrows are controllable, every state before *dominoes* has a self-loop (not drawn) and $\mathtt{acq}\ S$ means a sequence of forced transitions with the operations $\mathtt{acq}_t$ for each $t \in S$ (in some order). For simplicity only the *vertical* case is shown.

$\#x$, $\#y$, $\#\overline{x}$, $\#\overline{y}$ the strategy chooses locks corresponding to d_1 and $\overline{d_2}$ with $d_1 = t(\#x, \#y)$ and $d_2 = t(\#\overline{x}, \#\overline{y})$. 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 process q go through state *dominoes* before p and $\overline{p}$ start running, because all states before *dominoes* have a self-loop, so q cannot block there. If either p or $\overline{p}$ starts before q has reached *dominoes*, then q can never reach it, as one of the locks $\ell, \overline{\ell}$ will never be available again.

If q reached state *dominoes* then process p has no choice but to take ℓ, and then the remaining locks among x, y. Similarly for $\overline{p}$. At this stage the 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 decides which domino to take after the environment has

decided which x and y locks to take. So σ_p defines a function $t : \{0, \ldots, 2^n - 1\}^2 \to D$. Similarly $\sigma_{\overline{p}}$ defines $\overline{t}$.

We first show that $t(i,j) = \overline{t}(i,j)$ for all $i, j \in \{0, \ldots, 2^n - 1\}$. If not then consider for example the run where the environment chooses *equality* and then x, $\overline{x}$ to be the representations of i, and y, $\overline{y}$ to be representations of j. Suppose we have a run where process q reaches state *dominoes*, and assume that q's strategy tells to go to state $(d, \overline{d})$. Next the environment makes processes p and $\overline{p}$ reach the states where they chose their dominoes, $t(i,j)$ and $\overline{t}(i,j)$ respectively. The two processes p and $\overline{p}$ then reach a deadlock state. Since we assumed that $t(i,j) \neq \overline{t}(i,j)$, process q cannot reach state *win* from any state $s_{d,\overline{d}}$. Hence we have a deadlock run, a contradiction.

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 the environment chooses one of the actions *vertical*, *horizontal* or b_{left}, b_{right}, b_{down}, b_{up}, the tiling function must be correct. □

5. Undecidability in the general case

In this section we show that the deadlock avoidance control problem is undecidable. With a more involved proof we show undecidability using only 4 locks per process in [GMMW23]. The case of 3 locks per process remains open. In this section we present a lightweight proof, where processes use a larger (but still fixed) number of locks.

Theorem 2.5. *The deadlock avoidance control problem for arbitrary LSS is undecidable (even when the number of locks and processes is fixed).*

We begin by showing undecidability under the assumption that processes can already hold some locks in the initial configuration. We then reduce this problem to the deadlock avoidance control problem, where all processes start holding no lock.

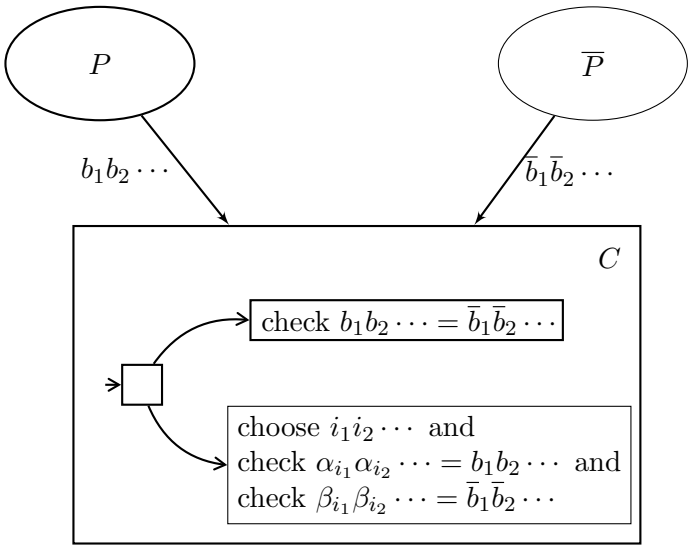


Figure 10: High-level view of the undecidability proof.

5.1. **Deadlock avoidance with initialization.** The input for the deadlock avoidance control problem with initialization 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 $I_p \subseteq T_p$. The question is whether there exists a strategy that guarantees that no run from C_{init} yields a global deadlock. It turns out that this generalization of the deadlock avoidance control problem is not more difficult than our original problem, as we will later see in Lemma 5.4.

Theorem 5.1. *The control problem for LSS with initial configuration and at most 7 locks per process is undecidable.*

The proof of Theorem 5.1 follows a well-known schema. We reduce from the question whether a PCP instance has an infinite solution.

Two processes P and $\overline{P}$ send independently sequences of bits $b_1, b_2, \dots$ and $\overline{b}_1, \overline{b}_2, \dots$ to process C. The environment asks C either to check that $b_1 b_2 \cdots = \overline{b}_1 \overline{b}_2 \dots$ or choose a sequence of indices $i_1, i_2, \dots$ and check that $\alpha_{i_1}\alpha_{i_2}\cdots = b_1 b_2 \cdots$ and $\beta_{i_1}\beta_{i_2}\cdots = \overline{b}_1\overline{b}_2\cdots$. Since P and $\overline{P}$ do not know what is being checked, they have to send sequences of letters and indices that satisfy both conditions, i.e., an infinite PCP solution. The difficulty here is that P and $\overline{P}$ use only locks to communicate.

Formally, let $(\alpha_i, \beta_i)_{i=1}^m$ be a PCP instance with $\alpha_i, \beta_i \in \{0,1\}^*$. We construct 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 $\overline{P}$ locks from $\{c, \overline{s}_0, \overline{s}_1, \overline{p}\}$, and C all seven locks. For the initial configuration we assume that $I_p = \{p\}$, $I_{\overline{P}} = \{\overline{p}\}$ and $I_C = \{c, s_0, s_1, \overline{s}_0, \overline{s}_1\}$.

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

$$\begin{aligned} u_P(b) &= \mathtt{acq}_{s_b}\mathtt{rel}_p\,\mathtt{acq}_c\,\mathtt{rel}_{s_b}\mathtt{acq}_p\mathtt{rel}_c \\ u_{\overline{P}}(b) &= \mathtt{acq}_{\overline{s}_b}\mathtt{rel}_{\overline{p}}\,\mathtt{acq}_c\,\mathtt{rel}_{\overline{s}_b}\mathtt{acq}_{\overline{p}}\mathtt{rel}_c \end{aligned}$$

The automaton $\mathcal{A}_P$ ($\mathcal{A}_{\overline{P}}$, 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 $u_P(b_1)u_P(b_2)\dots$ then this will mean that C, P synchronize over the sequence $b_1, b_2, \dots$, as we show below.

Process C's behavior for checking word equality consists in repeating the following procedure: she chooses a bit $b \in \{0,1\}$ through a controllable action, then tries to execute $u_C(P,b)\,u_C(\overline{P},b)$, where:

$$\begin{aligned} u_C(P,b) &= \mathtt{rel}_{s_b}\,\mathtt{acq}_p\,\mathtt{rel}_c\,\mathtt{acq}_{s_b}\mathtt{rel}_p\,\mathtt{acq}_c \\ u_C(\overline{P},b) &= \mathtt{rel}_{\overline{s}_b}\mathtt{acq}_{\overline{p}}\,\mathtt{rel}_c\,\mathtt{acq}_{\overline{s}_b}\mathtt{rel}_{\overline{p}}\,\mathtt{acq}_c \end{aligned}$$

For index equality C's behavior is similar: she chooses an index $i \in \{1, \dots, m\}$ and then tries to do $u_C(P,b_1)\dots u_C(P,b_k)u_C(\overline{P},b'_1)\dots u_C(\overline{P},b'_r)$, where $\alpha_i = b_1\dots b_k$, $\beta_i = b'_1\dots b'_r$.

Let us now prove that there is a winning strategy if and only if there is an infinite solution to the PCP instance.

We start by formalizing the intuition that the sequences $u_P(b)$ and $u_C(P,b)$ make the processes P, C synchronize over bit b.

Lemma 5.2. *Let ρ be a finite global run between two configurations γ and γ', such that the sequence of operations of C in ρ is $u_C(P,b)$, and C holds $s_0, s_1, \overline{s}_0, \overline{s}_1$ in γ . Then the sequence of operations executed by P in ρ is $u_P(b)$ and $\overline{P}$ stays idle in ρ. Furthermore C holds $s_0, s_1, \overline{s}_0, \overline{s}_1$ in γ'.*

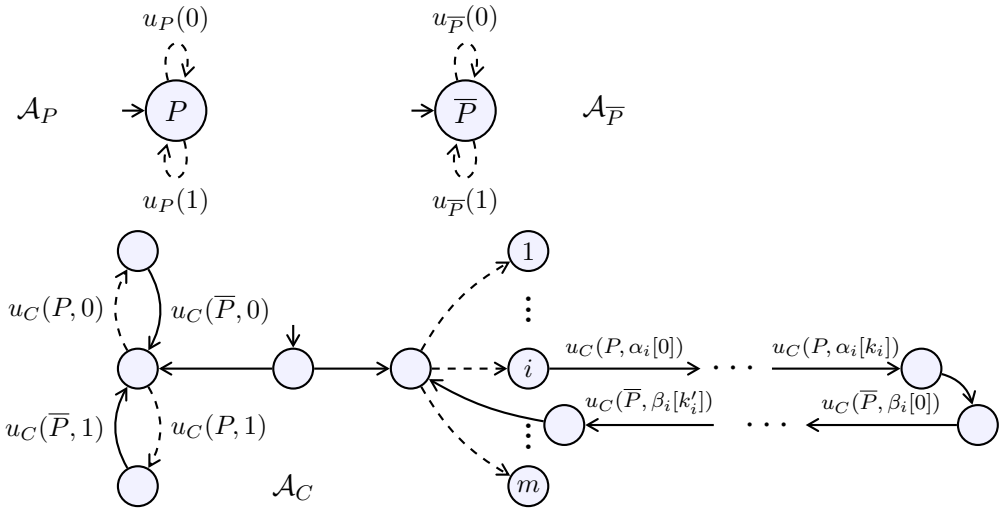


Figure 11: The system used for the undecidability proof. The letters $\alpha_i[0], \ldots, \alpha_i[k_i]$ and $\beta_i[0], \ldots, \beta_i[k_i']$ are defined so that $\alpha_i[0] \cdots \alpha_i[k_i] = \alpha_i$ and $\beta_i[0] \cdots \beta_i[k_i'] = \beta_i$. Dashed transitions are controlable.

Proof. Let us start with $\overline{P}$. At the start it cannot be holding $c, \overline{s}_0$ or $\overline{s}_1$ as C holds all of them. This implies that it is in its initial state, and not in one of the loops $u_{\overline{P}}(0)$ or $u_{\overline{P}}(1)$. Therefore its next action can only be to acquire $\overline{s}_0$ or $\overline{s}_1$. As those locks are never released by C in $u_C(P, b)$, $\overline{P}$ has to stay idle.

It is easy to see from C's sequence of actions that in γ' it holds $s_0, s_1, \overline{s}_0, \overline{s}_1$.

Concerning P, for the same reason it has to be in its initial state in γ. From γ, process C releases s_b and acquires p, meaning that P has started executing the $u_P(b)$ loop, acquired s_b and released p and is waiting for c. Then C releases c and acquires s_b, which means that P has taken c and released s_b, and is waiting for p. Finally C releases p and acquires c, which implies that P has acquired p and released c, and is stuck in its initial state as C holds both s_0 and s_1. Therefore, P has executed precisely $u_P(b)$. □

Assume that there is a winning strategy for the problem with initialization. We can observe that P has no incentive to allow both $u_P(0)$ and $u_P(1)$ at any point, since this leaves the choice to the environment. On the other hand, if P disallows both choices, then he keeps p forever, thus C will eventually be blocked as it needs to acquire p infinitely often. Hence the lock c will be held indefinitely by C. Then $\overline{P}$ will also be blocked since it needs to acquire c infinitely often. As a consequence, we can assume that P uses a strategy that allows exactly one of $u_P(0), u_P(1)$ each time.

Therefore, the strategy of P boils down to choosing a sequence of bits $b_0 b_1 b_2 \cdots$ and executing $u_P(b_0)u_P(b_1)u_P(b_2)\cdots$. Similarly, $\overline{P}$ chooses a sequence of bits $\overline{b}_0\overline{b}_1\overline{b}_2\cdots$ and executes $u_{\overline{P}}(\overline{b}_0)u_{\overline{P}}(\overline{b}_1)u_{\overline{P}}(\overline{b}_2)\cdots$. Also, if the environment makes C verify word equality, then C chooses a sequence of bits $b_0''b_1''b_2''\cdots$ and executes $u_P(b_0'')u_{\overline{P}}(b_0'')u_P(b_1'')u_{\overline{P}}(b_1'')\cdots$. Otherwise, C chooses a sequence of indices $i_0 i_1 i_2 \cdots$ and executes an interleaving of $u_P(b_0')u_P(b_1')u_P(b_2')\cdots$ with $b_0'b_1'\cdots = \alpha_{i_0}\alpha_{i_1}\cdots$ and $u_{\overline{P}}(\overline{b}_0')u_P(\overline{b}_1')u_P(\overline{b}_2')\cdots$ with $\overline{b}_0'\overline{b}_1'\cdots = \beta_{i_0}\beta_{i_1}\cdots$.

Let us now observe the relations between those sequences. First of all note that if any process gets blocked forever, then so do the other two, by Lemma 5.2. Thus a winning strategy should ensure that all processes run forever. To do so, by Lemma 5.2, the case of checking word equality implies that we should have $b_0 b_1 \cdots = b_0'' b_1'' \cdots = \overline{b}_0 \overline{b}_1 \cdots$.

Moreover, the case of index equality imposes that $b_0 b_1 \cdots = \alpha_{i_0} \alpha_{i_1} \cdots$ and $\overline{b}_0 \overline{b}_1 \cdots = \beta_{i_0} \beta_{i_1} \cdots$. As a result, we must have $\alpha_{i_0} \alpha_{i_1} \cdots = \beta_{i_0} \beta_{i_1} \cdots$, hence the PCP instance has an infinite solution.

Let us now show the other direction. Suppose there are indices $i_0 i_1 \cdots$ such that $\alpha_{i_0} \alpha_{i_1} \cdots = \beta_{i_0} \beta_{i_1} \cdots$. Let $b_0 b_1 \cdots = \alpha_{i_0} \alpha_{i_1} \cdots$ A winning strategy is to make P and $\overline{P}$ choose that same sequence of bits $b_0 b_1 \cdots$. If C has to check word equality, it chooses the sequence $b_0 b_1 \cdots$, otherwise it chooses indices $i_0 i_1 \cdots$. In the following lemma we say that a process wants to execute a sequence of operations if those are the operations of the next transitions chosen by its strategy.

Lemma 5.3. *Assume that C owns $\{s_0, s_1, c, \overline{s}_0, \overline{s}_1\}$, P owns $\{p\}$, C wants to execute $u_C(P, b)$, P wants to execute $u_P(b)$ and $\overline{P}$ wants to execute $u_{\overline{P}}(b')$. Then C and P finish executing $u_C(P, b)$ and $u_P(b)$ without encountering a global deadlock, $\overline{P}$ stays idle, and the lock ownership is the same as before the execution.*

Proof. Locks $\overline{s}_0, \overline{s}_1$ will never be released in the sequence we describe, thus $\overline{P}$ has to stay idle.

It suffices to observe that the environment has no choice for the sequence of operations. At first every lock is taken, and C is the only process which can release one (s_b), so it does. Then process P is the only one which can move, by taking s_b, and then releasing p, and so on. Eventually P will have executed $u_P(b)$ and C will have executed $u_C(P, b)$. □

By construction of the strategy, no matter if the environment chooses to check word equality or index equality, the sequence of operations of C is an interleaving of the sequences $u_C(P, b_0) u_C(P, b_1) \cdots$ and $u_C(\overline{P}, b_0) u_C(\overline{P}, b_1) \cdots$. The sequences of operations on P and $\overline{P}$ are respectively $u_P(b_0) u_P(b_1) \cdots$ and $u_{\overline{P}}(b_0) u_{\overline{P}}(b_1) \cdots$. As a consequence of this lemma, we obtain that the system cannot reach a global deadlock.

We have shown that there is a winning strategy if and only if the PCP instance has an infinite solution.

5.1.1. *Removing the initialization.* We aim to prove the following lemma:

Lemma 5.4. *There is a polynomial-time reduction from the deadlock avoidance control problem for lock-sharing systems with initialization to the control problem where all locks are initially free. The reduction adds one process and $|Proc| + 1$ new locks in total.*

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 transformed into a new system $\mathcal{S}_\emptyset$ with one extra process and additional locks. The transformation introduces one extra lock for each process p, denoted k_p and called *the key of p*. The extra process is called q and also has a key k_q. Each process $p \in Proc$ uses in addition to T_p the locks k_p and k_q.

The automaton $\mathcal{A}_q$ of q consists of a sequence of states connected with uncontrollable transitions where q first acquires k_q, then acquires and releases each k_p in some arbitrary, fixed order. Additionally, every state except the last one has an uncontrollable *nop* self-loop.

This is to make sure that q must execute the full sequence in any run leading to a global deadlock.

The automaton $\mathcal{A}_p$ of process p is extended by 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 IS_p 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 k_q.
(3) Finally, p acquires its key k_p and reaches the initial state $init_p$ of $\mathcal{A}_p$.

In addition, an uncontrollable *nop* self-loop labels every state of this sequence (except for $init_p$). The uncontrollable self-loops on every state of IS_p guarantee that a deadlock may occur only after all processes have fully completed their init sequences.

Linking runs in $\mathcal{S}_\emptyset$ and $\mathcal{S}$. We establish that there is a winning strategy in one system if and only if there is one in the other. The reason for this is that essentially, in order to reach a deadlock in $\mathcal{S}_\emptyset$ the environment is forced to execute the init sequences of all processes and then continue with an execution of $\mathcal{S}$.

Claim 1. *If there is a winning strategy in $\mathcal{S}_\emptyset$, then there is one in $\mathcal{S}$.*

Proof. Let $\sigma' = (\sigma'_p)_{p \in Proc \cup \{q\}}$ be a winning strategy in $\mathcal{S}_\emptyset$. We define a strategy $\sigma = (\sigma_p)_{p \in Proc}$ in $\mathcal{S}$ by letting $\sigma_p(u) = \sigma'_p(IS_p\, u)$ for every local run u of p in $\mathcal{S}$. Since all transitions in IS_p are uncontrollable, σ_p is well-defined.

Suppose by contradiction that there is a σ-run leading to a global deadlock in $\mathcal{S}$. We can execute the first two steps of IS_p for each process p, one by one, then let q execute all its transitions (acquire k_q, then acquire and release each k_p). At this point q is deadlocked. Finally we execute the third step of IS_p, for each process (acquire k_p). We can then execute the σ-run leading to a global deadlock in $\mathcal{S}$, which also leads to a global deadlock in $\mathcal{S}_\emptyset$. This contradicts the assumption that σ' is winning. □

Claim 2. *If there is a winning strategy in $\mathcal{S}$, then there is one in $\mathcal{S}_\emptyset$.*

Proof. Let $\sigma = (\sigma_p)_{p \in Proc}$ be a winning strategy in $\mathcal{S}$. We define $\sigma' = (\sigma'_p)_{p \in Proc}$ such that $\sigma_p(u) = \sigma'_p(IS_p\, u)$ for every local run u of p in $\mathcal{S}$.

Suppose by contradiction that we have a σ'-run leading to a global deadlock in $\mathcal{S}_\emptyset$. As every state along the init sequence has a *nop* self-loop, they must all have executed their init sequence in full. Similarly, q must have entirely executed its sequence of operations. Each p must hence have executed steps (1) and (2) of IS_p, and this before q has taken k_q. On the other hand, each $p \in Proc$ must have taken k_p after q has taken and released it.

As a consequence, there is a point in the run at which each p has taken all locks in I_p, but none of them has reached $init_p$ yet. Consider the rest of the run from that point and remove every action from the init sequences and from q. We obtain a σ-run of $\mathcal{S}$ leading to a global deadlock. This contradicts the assumption that σ is winning. □

The two claims above prove that there is a winning strategy in one system if and only if there is one in the other. This concludes the reduction. □

We obtain Theorem 2.5 from Theorem 5.1 and Lemma 5.4.

6. Conclusions

Motivated by a recent undecidability result for distributed control synthesis of Zielonka automata [Gim22] we have considered a simpler model, for which the problem has not been investigated yet. With hindsight it is strange that the well-studied model using lock synchronization has not been considered in the context of distributed synthesis. One reason may be the non-monotone nature of the synthesis problem: for a less expressive class of systems the problem is not 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 article 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 3.4 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 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 2.8 is tight. The algorithm for verifying if there is a deadlock in a given lock graph, Algorithm 5, 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 [LR81, Lyn96]. 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.

Acknowledgements. We thank the *LMCS* reviewers for the thorough reading and their numerous and helpful comments.

References

[AW07] André Arnold and Igor 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.

[BBB+20] Béatrice Bérard, Benedikt Bollig, Patricia Bouyer, Matthias Függer, and Nathalie 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, Brussels, Belgium, September 21-22, 2020, volume 326 of *EPTCS*, pages 33–49, 2020. To appear in Information and Computation. doi:10.4204/EPTCS.326.3.

[BCMV13] Rémi Bonnet, Rohit Chadha, P. Madhusudan, and Mahesh Viswanathan. Reachability under contextual locking. *Log. Methods Comput. Sci.*, 9(3), 2013. doi:10.2168/LMCS-9(3:21)2013.

[BFHH19] Raven Beutner, Bernd Finkbeiner, and Jesko 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.

[CE81] Edmund M. Clarke and E. Allen 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.

[Chu57] Alonzo Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, volume I, pages 3–50. Cornell Univ., Ithaca, N.Y., 1957.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984. doi:10.1145/1780.1804.

[ELM+16] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier 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.

[FGHO22] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger 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, February 14-19, 2022, Göttingen, Germany (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.

[Fin15] Bernd 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.

[FO17] Bernd Finkbeiner and Ernst-Ruediger Olderog. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.*, 253:181–203, 2017.

[FS05] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *LICS'05*, pages 321–330. IEEE Computer Society, 2005.

[GGMW13] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor 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.

[GHY21] Manuel Gieseking, Jesko Hecking-Harbusch, and Ann 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, Luxembourg City, Luxembourg, March 27 - April 1, 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.

[Gim17] Hugo Gimbert. On the control of asynchronous automata. In *FSTTCS'17*, volume 30 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Gim22] Hugo Gimbert. Distributed asynchronous games with causal memory are undecidable. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/lmcs-18(3:30)2022.

[GLZ04] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 275–286. Springer, 2004.

[GMMW22] Hugo Gimbert, Corto Mascle, Anca Muscholl, and Igor Walukiewicz. Distributed controller synthesis for deadlock avoidance. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 125:1–125:20. Schloss

Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ICALP.2022.125, doi:10.4230/LIPICS.ICALP.2022.125.

[GMMW23] Hugo Gimbert, Corto Mascle, Anca Muscholl, and Igor Walukiewicz. Distributed controller synthesis for deadlock avoidance. *CoRR*, abs/2204.12409, 2023. URL: https://doi.org/10.48550/arXiv.2204.12409, arXiv:2204.12409, doi:10.48550/ARXIV.2204.12409.

[GSZ09] Paul Gastin, Nathalie Sznajder, and Marc Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, June 2009.

[HM19] Jesko Hecking-Harbusch and Niklas 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, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 369–386. Springer, 2019. doi:10.1007/978-3-030-31784-3_22.

[Kah09] Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 27–36, 2009. doi:10.1109/LICS.2009.45.

[KG06] Vineet Kahlon and Aarti 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.

[KIG05] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, page 505–518, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11513988_49.

[KV01] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *LICS'01*, pages 389–398. IEEE, 2001.

[LMSW13] Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Contextual locking for dynamic pushdown networks. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 477–498. Springer, 2013. doi:10.1007/978-3-642-38856-9_25.

[LR81] Daniel Lehmann and Michael 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.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MT01] 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.

[MTY05] P. Madhusudan, P. S. Thiagarajan, and Shaofa Yang. The MSO theory of connectedly communicating processes. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 201–212. Springer, 2005.

[MW14] Anca Muscholl and Igor Walukiewicz. Distributed synthesis for acyclic architectures. In *FSTTCS'14*, volume 29 of *LIPIcs*, pages 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proc. ACM POPL*, pages 179–190, 1989.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS'90*, pages 746–757. IEEE Computer Society, 1990.

[RW89] Peter J.G. Ramadge and Walter M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989.

[RW92] Karen Rudie and W. Murray Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Trans. on Automat. Control*, 37(11):1692–1708, 1992.

[Thi05] John G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005.

[Tri04] Stavros Tripakis. Undecidable problems in decentralized observation and control for regular languages. *Information Processing Letters*, 90(1):21–28, 2004.

[Wal21] Igor 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.

[WLK+09] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott 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, Savannah, GA, USA, January 21-23, 2009*, pages 252–263. ACM, 2009. doi:10.1145/1480881.1480913.